IMAGE-BASED GRAPH VISUALIZATION

OZAN ERSOY

Simplified Visualization of Large Graphs

Cover: Edge bundled graph with a radial layout produced by SBEB.

Image-Based Graph Visualization
Ozan Ersoy
Supervised by Prof. dr. Alexandru C. Telea
PhD thesis Rijksuniversiteit Groningen
ISBN 978-90-367-6345-5 (printed version)
ISBN 978-90-367-6344-8 (electronic version)

RIJKSUNIVERSITEIT GRONINGEN

IMAGE-BASED GRAPH VISUALIZATION

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. E. Sterken,
in het openbaar te verdedigen op
vrijdag 25 oktober 2013
om 14.30 uur

door

OZAN ERSOY

geboren op 07 november 1978
te Kütahya, Turkije

Promotor:	Prof. dr. A.C. Telea
Beoordelingscommissie:	Prof. dr. J. Doellner
	Prof. dr. S. Diehl
	Prof. dr. G. Melancon

The greatest value of a picture
is
when it forces us to notice
what we never expected to see.

John W. Tukey. Exploratory Data Analysis. 1977.

CONTENTS

1 INTRODUCTION

Software maintenance covers 80% of the cost of modern software systems, of which over 40% represent software understanding [164, 35]. Although many visual tools for software understanding exist, most know very limited acceptance in the IT industry. Key reasons for this are limited scalability of visualizations and/or dataset sizes, long learning curves, and poor integration with software analysis or development toolchains, as strongly voiced by several researchers [143, 31, 99, 223].

Software visualization (SoftVis) uses information visualization (InfoVis) techniques to create interactive displays of software structure, behavior, and evolution. Recent trends in SoftVis include scalable Infovis techniques such as treemaps, icicle plots, bundled graph layouts, table lenses, parallel coordinates, multidimensional scaling, and dense pixel charts to increase the amount of data shown to the user at a single time [43]. Software visualization tools such as Rigi [95], VCG [148], aiSee [2], Mondrian [108], sv3D [116], CodeCity [212, 213], and SeeSoft [51] are well known in academic circles. Many more such tools appear on a yearly basis, as illustrated by the material published in the proceedings of *e.g.* ACM SOFTVIS, IEEE VISSOFT, MSR, ICPC, WCRE, ICSE, and CSMR conferences.

Visual analytics (VA) integrates graphics, visualization, interaction, and data collection and analysis to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [220, 188]. These fields share many similarities with software maintenance in terms of *data* (large databases, structured text, and graphs), *tasks* (sensemaking by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization). VA stresses tool integration, as opposed to pure data mining or fact extraction (whose main focus is scalability) or information visualization (Infovis, mainly focused on presentation). As such, VA is a promising model for building effective and efficient *software visual analysis* (SVA) tools. However, just as for the established VA context, building efficient and effective SVA tools which gain wide

acceptance in both academic and industrial contexts, is challenging.

Software systems contain large and complex sets of *dependencies* between their components, such as call and inheritance graphs, and data flow and type dependency graphs. Analyzing such dependencies is arguably one of the most important tasks of software maintenance processes such as reverse engineering and reengineering. A good understanding of such data supports decisions for code refactoring, removing code clones, identification of design patterns, and debugging [43, 93, 99].

However, understanding large sets of dependencies is challenging. Visualization is a method of choice, given the inherent difficulty for understanding large, abstract graphs. Although numerous methods are being proposed for visualizing dependency graphs in the information visualization (InfoVis), software visualization, (SoftVis), and graph drawing (GD) communities, it is still unclear how such methods are received by software practitioners in the field, and how they compare when one must accomplish tasks in program comprehension. Secondly, it is not clear how such graph visualization methods can be improved in order to assist the program understanding tasks outlined above more effectively.

1.1 GRAPH EDGE BUNDLING

The visual depiction and understanding of large graphs, such as composed by the dependencies mentioned in the previous section, is an open challenge to both SoftVis and InfoVis communities. Efficient and effective visualization methods that address this task have a wider relevance than program comprehension, as they are applicable to numerous other application fields, such as network analysis, life sciences, social sciences, and geosciences [207].

As the number of nodes and edges of a graph increases, node-link graph visualizations become challenged by *clutter*, *i.e.* unorganized groups of nodes and edges onto small screen areas. Clutter impairs tasks such as finding the nodes that a given edge (or edge set) connect, and at a higher level, understanding the coarse-scale graph structure. Several approaches exist to reduce clutter in graph visualizations. First, the graph can be simplified prior to visualization, *e.g.* by extracting structures such as spanning trees or strongly connected components. Secondly, the layout of nodes and/or edges can be adjusted. Both

methods can be applied globally, based on clutter estimation metrics, or locally, based *e.g.* on user interaction [219, 216].

When node positions encode information, they should not be changed. Also, clutter is related most often to edge crossings [137, 76]. One of the main directions for reducing clutter is *edge bundling*, or the geometrical grouping of edges that follow close paths. Edge-bundling layouts (EBLs) exist for general graphs [39, 80, 133], circular layouts [68], hierarchical digraphs [78], and parallel coordinates [117, 225]. Bundling typically starts with a given set of node positions, either present in the input data, or computed using a layout algorithm. Edges found to be close in terms of graph structure, geometric position of their endpoints, data attributes, or combinations thereof, are drawn as tightly bundled curves. This trades clutter for overdraw and produces images which are easier to understand and/or better emphasize the graph structure. Edge bundles can be rendered using various effects such as blending or shading [80, 103]. Edge bundling algorithms exist for both compound (hierarchy-and-association) [78] and general graphs [80, 39, 133, 103, 70, 150].

1.2 CONTRIBUTIONS OF THIS THESIS

In this thesis, we take a new look at edge bundling layouts (EBLs) as a method for reducing the clutter in large graphs, with software understanding as an application area. Within this context, we address the following research questions:

1. Is edge bundling an effective instrument for the understanding of large graphs as compared to more classical node-link graph visualization techniques?

2. How can we design edge bundling techniques which computationally scale to handle large graphs?

3. How can we complement edge bundling techniques with rendering and interaction techniques to simplify the resulting images, and also add more contextual information, for a better understanding?

We approach the above questions as follows.

In Chapter 3, we look at the problem of understanding call graphs extracted from software systems which have a hierarchical structure, or so-called *compound graphs*. We describe a tooling pipeline that covers static code analysis, extraction of call

relations and hierarchy data, and visualization. For visualization, we compare state-of-the-art node-link visualization techniques [9, 10] with hierarchical edge bundles (HEBs), the best known bundling method available to the date [78]. The comparison outlines that, in the described context of program comprehension, HEBs clearly outperform classical node-link layouts.

In Chapter 4, we take the challenge of edge bundling layouts a step further. We detail the design, implementation, and evolution of a software visual analytics (SVA) toolset for program comprehension from early research prototypes to a commercial toolset used in the IT industry. The described SVA toolset uses HEB visualizations as one of its key components. We describe the usage of our SVA toolset in several contexts. The conclusions of our study strengthen and refine the insights obtained in Chapter 3: HEB visualizations have a high potential to be accepted by end-users as effective and useful program comprehension instruments, if they are seamlessly embedded in an end-to-end SVA toolset that combines data mining and visual analysis.

In Chapter 5, we focus on one perceived limitation of EBLs discovered during our studies presented in Chapters 3 and 4: the difficulty of understanding bundles in overlapping regions. We present a novel technique for the visualization of the coarse-scale structure of an EBL which clarifies edge clutter caused by bundle overlaps and assists the task of finding nodes connected by a bundle. The main novel element in our technique is using an *image-based* approach: A given EBL is simplified, and next rendered, using a sequence of operations which occur only in image space. This offers a way to continuously simplify an EBL after it has been computed, and thus yields a smoothly varying sequence of progressively simplified views of the underlying graph.

In Chapter 6, we take the image-based approach from Chapter 5 a step further. We show how the image processing operations used earlier for EBL simplification (distance fields and skeletons) can be effectively and efficiently used for the actual construction of an EBL starting from an unbundled input graph. The proposed method has several advantages as compared to existing EBLs: computational efficiency, generation of organic-like bundle shapes, and most importantly a way to analyze the robustness of the proposed method, by using underlying properties of the used image processing operators.

In Chapter 7, we further exploit the results obtained in Chapter 6. Specifically, we show how we can construct EBLs of gen-

eral graphs using only a density map of the input graph. This further simplifies the EBL construction by removing the main technical requirements of SBEB – graph clustering, image segmentation, and skeleton computation – while keeping its key idea of grouping bundles towards the local maxima of edges densities. We also present an efficient GPU implementation of our EBL method. At a conceptual level, we show that our EBL is practically and conceptually identical to mean shift image segmentation [34], which opens new ways for understanding the properties of edge bundling techniques.

Chapter 8 moves the focus to dynamic graphs. We present two types of techniques for visualizing dynamic graphs using edge bundles. The first technique considers *streaming graphs*, *i.e.* temporally ordered, unstructured, edge-sequences with start and end lifetime moments. For this use-case, we show how the EBL technique proposed in Chapter 7 can be naturally extended to smoothly bundle time-dependent graphs. The second technique considers *graph sequences*, *i.e.* a discrete set of graphs between which higher-level correspondences can be inferred. For this use-case, we exploit additional edge-correspondence information to further highlight events of interest such as the appearance, change, and disappearance of edge groups. We demonstrate our method both on software graphs and graphs describing air traffic information.

Chapter 9 introduces MoleView, a framework for interactive exploration of large element-based plots, which are sets of discrete data elements, each with several data and/or position (layout) attributes. Examples thereof are EBLs, (multidimensional) scatter plots, and images. We show how we can extend well-known semantic lenses with a range-based attribute filter to select a 'data layer' at a user-defined point, *i.e.* a set of data elements falling within the lens' position and attribute filter values. Next, we present several dynamic re-layouting techniques that smoothly push these elements away from the lens to obtain a smooth transition between the original and detailed dataset. Finally, we extend the semantic lens concept for the task of exploring a dataset by the smooth animated interpolation between two completely different layouts of the same data, using as example the exploration of two-dimensional scalar images.

We conclude this thesis in Chapter 10 by comparing the research questions outlined above in this section against the visualization methods presented in the subsequent chapters. Overall, our observation is that image-based techniques are a promising avenue for enhancing the efficiency and effectiveness of

edge bundling layouts, and that future work on this topic can open several new directions for a more widespread application and usage of such techniques for the exploration of large graphs both in a program comprehension context and beyond.

2 RELATED WORK

We divide related work in the context of this thesis into several subsections, as follows.

Fact extraction (see Sec. 2.1) focuses on the process that mines relational data, and its attributes, from the raw datasets of interest to the visual analysis. In our context that focuses on software visual analytics, fact extraction consequently focuses on static and dynamic analysis of software code bases and their related artifacts.

Node-link graph visualization (see Sec. 2.2.1) focuses on general graph visualization techniques aimed at displaying large graphs, and their related attributes, with a focus on the most used such techniques in a software visualization context, and their challenges when visualizing large graphs.

Edge bundling methods (see Sec. 2.2.2) focuses on a specific subclass of solutions for managing scalability in the context of visualizing large graphs: edge bundling. This is also the main focus of the work presented in this thesis.

Rendering graphs (see Sec. 2.2.3) overviews the final part of a graph visualization pipeline. We focus here on techniques such as shading, transparency, texturing, and color mapping which are used in this step of the visualization pipeline to map a computed layout, and additional data attributes, to a final image.

Time-dependent graphs (see Sec. 2.3) outlines the challenges involved in the visual analysis of graphs which encode information that changes in time. In particular, we outline here distinctions between various types of graphs, such as streaming and sequence graphs.

Finally, we overview the class of *interaction* techniques used in the context of visual analysis of large graphs (see Sec. 2.4).

We should stress, upfront, that the review of related work presented in this chapter should not be seen as a comprehensive coverage of all techniques related to large-scale graph visualization. Such a review would require an entire volume by itself. In contrast, our focus here is to focus on existing research work that relates to our main investigation topic – the usage of edge bundling techniques for the visual analysis of large graphs. As such, we limit our discussion in this chapter to existing techniques which are related to edge bundling, on the one hand,

and to the visualization of large graphs on software understanding, on the other hand.

2.1 FACT EXTRACTION

Fact extraction, in the context of software understanding, covers the gathering of information from source code, binaries, and source control management (SCM) systems such as CVS, Subversion, CM/Synergy, Git, or ClearCase. In the following, we denote such sources of information by the generic term software *databases*. For the work in this thesis, the actual type of information source is not important, as long as it provides (a) attributed graph data; and (b) the size of the provided graphs is large enough so that the clutter and scalability changes mentioned in Chapter 1, and discussed further in the current chapter, are present.

In our context, we are mainly interested in the extraction of *relational* information from software databases. From the perspective of software understanding, these are classified into various types of graphs. Common types include dependency, call, and control flow graphs produced by static syntactic and semantic analysis [14, 112, 181, 119, 118]; program slice graphs produced by similar types of analysis [191, 21]; and code duplicates or code clones extracted by clone detectors [87, 131, 89, 85]. Attributes include software quality metrics, *e.g.* code size, complexity, cohesion, and coupling extracted by white-box code analysis engines [106], and attributes pertaining to the evolution of software, such as change moments, commit logs, and change requests, extracted by repository mining techniques [43, 120].

Two aspects are important for the fact extraction part of the visual software understanding pipeline: data modeling (Sec. 2.1.1) and data mining (Sec. 2.1.2), as follows.

2.1.1 *Data modeling*

From a visual analysis perspective, we need to represent the facts, or data, extracted from a software repository. A generic model for such data is the so-called *compound attributed graph*. Such a graph $G = (V, E)$ is a set of nodes, or vertices, V and edges E. Nodes V model software artifacts, ranging from high-level ones such as folders, files, (sub)packages, and libraries, to low-level ones such as classes, functions and methods, and individual code lines or symbols. Edges E model relationships be-

tween the extracted artifacts. These are further subdivided into *containment* and *association* relations.

Containment relations are directed edges which describe the structure of the software, *i.e.* model inclusion. For instance, a containment relation $c = (n_1, n_2)$ captures the fact that the software entity n_1 (*e.g.*, a class) contains, or includes, the software entity n_2 (*e.g.*, a method definition). Containment relations most usually create trees, with the entire system under study at the root, and the finest-grained extracted artifacts as leaves. However, in the most general case, containment relations can also be structured as directed acyclic graphs (DAGs) – consider, for example, the case of programming languages such as C# where a class declaration can be contained in several files.

Association relations are directed or undirected edges which typically describe all extracted relations which cannot be classified as containment. Examples of such directed relations are function calls (a relation $c = (n_1, n_2)$ models that function n_1 calls function n_2); symbol usage-to-declaration links (a relation $u = (d, s)$ models that the symbol s has the declaration d); and execution sequencing (a relation $s = (n_1, n_2)$ models that the statement n_2 executes after the statement n_1). Examples of undirected associations are code clones, or code duplicates (a relation $c = (n_1, n_2)$ models the fact that the source code in artifact n_1 is (very) similar to the source code in artifact n_2).

Attributes can be associated to both nodes and edges in the above model. In the most general case, a node or edge has a set $A = \{(k_i, v_i)\}$ of key-value pairs (k_i, v_i). Here, k_i is an identifier describing the name of a certain attribute, *e.g. function name*, *element type*, or *lines of code*. The elements v_i encode the values of the named attributes. These can be numeric (*e.g* amount of lines of code or complexity); nominal or unordered categorical (*e.g.* type of a syntactic construct); ordinal or ordered categorical (*e.g.* code safety ranked on a scale *Low, Average, High*), or text (*e.g.* the fully-qualified name of a function). In this model, there is no restriction that all nodes have the same number of attributes, nor that all nodes have to have the same set of keys or attribute names.

Although the above model for software relational datasets can generally capture all the information extracted from software datasets, several challenges exist, as follows:

- *Variability:* To fully describe software datasets, several separate containment and association relation types are needed. For instance, a software system admits several types

of hierarchies, *e.g.* a *storage* (folder-file) hierarchy and a *syntactic* (namespace-symbol) hierarchy. Similarly, several types of associations exist (calls, symbol usage-definition, type inheritance). Visualizing several such hierarchies *and* association types simultaneously is very hard [108, 95];

- *Efficiency:* Efficiently storing, querying, and editing compound attribute graphs containing hundreds of thousands of nodes, edges, and attributes is difficult. Although several implementations exist that are optimized for specific types of data and query operations, such as syntax trees [14, 11, 119, 118, 23], a general solution is still missing;

- *Interchange:* Effective construction and usage of software analysis pipelines involves the composition of several tools [99, 145]. This can practically happen only if data exchange formats exist for compound attributed graphs. Many formats have been proposed in software analysis, *e.g.* GXL [77], FAMIX [189, 128], and SourceML [33]. However, the widespread acceptance of these formats, as well as their ability to model the full spectrum of compound attributed graphs produced by software fact extraction, is still limited.

2.1.2 *Data mining*

From a practical perspective, we need techniques and tools that are able to populate the compound attributed graph introduced in Sec. 2.1.1 with actual facts (relationships and attributes) from a given software database. We call this operation *data mining*. The graph such constructed will be next the input to our visual analysis.

Strictly speaking, the challenges of data mining form a separate concern which is not subject to our work, as our focus is on the visualization part of the software understanding pipeline. However, to better outline the type of data and challenges which our subsequent visualizations will be subjected to, we overview here several data mining tools which we have concretely used for extracting static structure and associations (Sec. 2.1.2.1), metric attributes (Sec. 2.1.2.2), and clone associations (Sec. 2.1.2.3) for our compound attributed graphs.

2.1.2.1 *Static structure and associations*

Structure (containment) and association relations are most frequently extracted from source code using so-called *static analyzers*. In our work, we have focused mainly on source code written in the C and C++ programming languages. The main reasons hereof where (a) the fact that, at the inception of this work, C and C++ code bases were dominant in the open-source arena; (b) such code bases were significantly larger (millions of lines of code) than code bases written in other programming languages such as Java or C#; and (c) the inherently more complex syntactic structure of C++ provided us with more complex containment and association graphs. C++ programs are particularly interesting for software visualization, as they have a deeper hierarchical structure (folders, files, namespaces, classes, nested classes, methods), whereas C program hierarchical structure is limited to folders, files, and functions. Moreover, object-oriented code is supposed to be more modular than classical procedural code, so a good visualization may be able to emphasize the presence (or absence) of such modularity.

Well-known static analyzers include LLVM [111], ROSE [139], Cppx [109], Columbus [61], Eclipse CDT [49], Elsa [119] (for C/C++), Recoder [115] (for Java), Reflector [142] (for C# and .NET), and ASF+SDF (a meta-framework with language-specific front-ends) [197]. Static analyzers can be further divided into *lightweight* ones, and *heavyweight* ones. *Lightweight* extractors, *e.g.* SrcML [33], SNIFF+, GccXML, and MCC, do only partial parsing and type-checking using a subset of the target language grammar and semantics and trade fact completeness and accuracy for speed and simplicity producing only a fraction of the entire static information. *Heavyweight* extractors, *e.g.* DMS [14], ASF+SDF [196], CPPX [109], ROSE [139], oink [119, 118], Columbus [61], and SolidFX [181] perform (nearly) full syntactic and semantic analysis at higher cost. For call data extraction from C and specifically C++, a heavyweight extractor is mandatory, as we need full semantic (type) information, as well as a full implementation of the C++ lookup rules, to be able to correctly link calls to function declarations and those further to function definitions, for all types of functions including constructors, destructors, and operators [23, 181]. Heavyweight extractors can be further classified into strict ones, based on a compiler parser which halts on lexical or syntax errors, *e.g.* CPPX and, up to a large extent, LLVM; and tolerant ones, based on fuzzy parsing or Generalized Left-Reduce (GLR) grammars, *e.g.* Colum-

BUS, OINK or SOLIDFX. All extractors are typically run in batch mode, and produce an annotated syntax graph. This graph is further filtered to extract the desired types of compound attributed graphs presented in Sec. 2.1.1.

In our work, we have used both lightweight extractors (GCCXML) and heavyweight ones (SOLIDFX, OINK). In the end, heavyweight extractors proved more suitable, both in terms of the completeness and correctness of the delivered graphs, for our subsequent visualization purposes. As such, all examples presented further in this thesis which use software graphs extracted by static analysis were created using the above-mentioned heavyweight extractors. Further on, we have also investigated the visualization of compound attributed graphs extracted from other programming languages, specifically Java and C#. For these, we have used the heavyweight extractors Recoder and Reflector.

2.1.2.2 *Software metrics*

Metric tools include CodeCrawler [105], Understand [149], and Visual Studio Team System (VSTS). Insights in software evolution and software quality metrics are given by Mens *et al.* [120], Lanza *et al.* [106], and Fenton *et al.* [60]. For our specific visualization purpose, the exact selection and type of software metrics being used is less important, as our main goal is to demonstrate how such metrics can be scalably and effectively visualized. As such, we have limited ourselves to the usage of the best known quality metrics, such as code size, complexity, fan-in, fan-out, and coupling. We extract these metrics directly from the syntactic structure already provided by the heavyweight extractors we use (see Sec. 2.1.2.1). If desired, other metrics can be directly replaced in our discussion on visualization in the following chapters.

2.1.2.3 *Code duplicates*

Code duplicates, or code clones, are source code fragments in a given code base which share large similarities [131]. Several techniques and tools exist in the program understanding literature which help finding code clones. Baxter *et al.* extract abstract syntax trees from the code, determine a hash code from the entire tree structure, and compare same-hashcode trees using a bottom-up matching algorithm [13]. Jiang *et al.* compute fixed-length vector descriptors of syntax tree nodes, recording the number of occurrences of each node type, and hash similar

subtrees based on the Euclidean distance between vectors [85]. Koschke *et al.* use a suffix token tree approach, comparing syntax trees by serializing the tree node types to strings, thereby combining the speed of string approaches with the precision of tree-based approaches [100]. Wahler *et al*. use an XML-based syntax trees and database queries to find code clones as frequent item-sets [210]. Ducasse *et al.* advocate a string-based clone detection, thereby removing the need for heavyweight parsers [46]. Ekoko and Robillard proposed a method to track code clones across several versions of a code base, by reusing the SimScan clone detector atop of a lightweight clone representation combining structural and lexical clone information [45]. Clone detection methods are also implemented in widely-used clone detection software, such as the well-known CCfinder tool [88].

Just as for software metrics, the exact choice of a clone detector is not critical for our software visualization aims, as long as this detector can extract compound graphs, and can handle large code bases written in the programming language of our choice (C and C++). To this end, in our work, we have extracted compound graphs with containment relations describing the software hierarchy and association relations describing clones using the SolidSDD clone detector [161]. SolidSDD implements a slightly modified version of the CCfinder algorithm [88], and can analyze code bases of millions of lines written in C and C++ in a few minutes on a desktop PC.

2.2 COMPOUND GRAPH VISUALIZATION

Once a compound attributed graph is extracted, *e.g.* by the software analysis techniques mentioned in Sec. 2.1, the next step in a typical analysis pipeline is to *visualize* this graph. Concrete tasks to be addressed by such a visualization include the overall assessment of the connectivity pattern (created by association edges) between several parts of a software system (represented by subtrees induced by containment edges); the structural comparison of two graphs to find similar and different subgraphs; and the discovery of specific structure-and-attribute patterns present in the graph [78, 36, 43].

Creating a visual representation of a compound attributed graph G can be seen as a two-step process:

1. *Layout:* Given G, the layout can be seen as a function $L : G \to \mathbb{R}^n$, which associates to each node, or vertex, $v \in G$

and edge $e \in G$ a spatial representation in $\mathbb{R}^n$. In practice, $n \in \{2, 3\}$, *i.e.* we create two- or three-dimensional visual representations of G.

2. *Rendering:* Given a layout $L(G)$ of a graph, rendering associates concrete visual representations with each laid out node $L(v \in G)$ and edge $L(e \in G)$. Examples include setting the shape, color, transparency, and texture of the nodes and edges.

The layout *vs* rendering distinction is important: Layout is a purely geometric operation that tells *where* nodes and/or edges are to be placed in 2D or 3D, but does not specify (or constrain) *how* these are to be drawn. In contrast, rendering is a purely screen-space operation that tells how nodes and edges are to be drawn at a given 2D or 3D position, but does not specify this position. Making this distinction enables us to further classify and analyze the existing graph visualization techniques (Secs. 2.2.1-2.2.3).

The more information (containment and association edges and node and edge attributes) such a visualization is able to display *at a single time*, the more general are the tasks it can support. For instance, visualizing only the call relations of a software system is of limited use, if our questions are of the type "which are the dependencies between two subsystems". Similarly, visualizing only the structure of the same system does not allow us to reason about modularity. Finally, showing no attributes on a structure-and-association visualization does not allow distinguishing between various types of associations, such as calls and inheritance relations.

However, when more information is "pushed" into such a visualization, the resulting image can easily get overloaded and hard to understand. As such, the key challenge we find for graph visualization in the context of software understanding, is the design of suitable layout and rendering techniques that (a) can depict the highest amount of available graph information; and (b) produce understandable images.

2.2.1 *Node-link Layouts*

Historically, *node-link layouts* are the first type of graph layouts. The key aspect of these methods is representing the edges as straight-line segments between their corresponding node positions, a visual convention which is well understood, and consid-

ered intuitive, by most end users [72]. Apart from this aspect, node-link layouts differ in terms of how the graph nodes are spatially embedded in 2D or 3D.

Several methods exist in the literature for laying out compound graphs [125]. SHriMP Views and similar methods show containment as nested boxes and associations using straight lines atop of the nesting [166, 17, 140]. Variations hereof are well known and used in software visualization, as shown by several toolsets, *e.g.* Rigi [95, 190], CodeCrawler [105], VCG [148] and SoftVision [172]. Although intuitive, such methods have scalability limitations. For large systems, association relations, mapped to straight lines, tend to clutter the nested layout, as any two elements in the hierarchy can be connected. For a more extensive discussion of clutter in information visualization, we refer to [53]. ArcTrees lay out containment as nested rectangles and associations as curved arcs connecting the elements [125]. However, they have similar association edge cluttering problems as SHriMP views. Curved edges showing associations can also be overlaid on treemaps [59], having however the same cluttering issues. Matrix views remove the clutter by showing associations as an adjacency matrix and hierarchy as tree views or icicle plots along the matrix edges [195]. A hybrid technique that combines the lack of clutter of matrix layouts and intuitiveness of node-link layouts is presented by Henry and Fekete [75]. However, matrix views (and their variants) are less intuitive than node-link diagrams and also are less effective in visually showing modularity, *i.e.* if associations (calls) from a subsystem are mainly directed at a few other subsystems [72, 195].

For very large graphs, optimizations of both the layout algorithms and graph data management are essential to usability. One system providing these is the graph visualization framework Tulip, which offers a wide range of search, layout, visualization, and interaction features, as well as high scalability for graphs of hundreds of thousands of elements [9, 10]. Even though less known in the software visualization community, Tulip is well-known in the InfoVis community, and is arguably one of the most sophisticated node-link graph visualization frameworks available. Further references on classical node-link layouts can be found in Tollis *et al.* [192].

Having recognized the clutter challenge posed by large graphs in node-link layouts, several solutions have been studied. Edge routing algorithms are one of the earliest attempts to reduce clutter in graphs. Edge routing algorithms intend to increase the readability of the graphs by minimizing edge lengths and

amount of edge bendings while maintaining a low number of edge crossings and avoiding node-edge overlaps [168, 44, 47]. Graph simplification techniques reduce clutter by simplifying the graph prior to layout *e.g.* by grouping strongly connected nodes and edges into so-called metanodes, followed by using classical node-link layouts for visualization. Several simplification methods exist, *e.g.* [1, 4]. Graph simplification is attractive as it reuses existing node-link layouts out of the box, but can be sensitive to simplification parameters, which further depend on the type of graph being processed. Furthermore, simplification does not allow a *continuous* treatment of the graph: The simplification events yield a set of discrete graphs rather than a smooth exploration scale [103]. Also, simplification typically changes node positions (collapse to metanodes), which can be undesirable *e.g.* when positions encode information.

A different solution to the clutter problem – graph bundling – is discussed in the next section.

2.2.2 *Edge bundling layouts*

As the number of nodes and edges of a graph increases, node-link layouts can produce significant visual clutter, which shows up as overlapping edges or nodes. Clutter impairs tasks such as finding the nodes that a given edge (or edge set) connect, and at a higher level, understanding the coarse-scale graph structure. In our software visualization context, for instance, answering questions such as "which are the main association connections between subsystems" can be severely impaired by cluttering.

A fundamentally different solution to the visual clutter problem from graph simplification (Sec. 2.2.1) is *edge bundling*. In layout terms, an edge bundling layout (EBL) spatially groups edges $e_i \in E$ for a graph $G(V, E)$ using a metric $d(e_i, e_j)$ that models closeness in either graph space, layout space, or both. Edges $e_i = \{p_{ij}\}_{j=1}^{N}$, where $N = |e_i|$, are discretized into points p_{ij} which are next positioned so as to minimize d.

At a high level, EBLs can be seen as a *simplification* of the degrees of freedom along which a layout is created: While a straight-line node-link layout will lay out each edge independently, in the direction prescribed by the positions of its end nodes, EBLs reduce the number of possible directions for edges by grouping similar edges into a bundle. As we shall see later on in Ch. 7, this amounts to a sharpening, or simplification, of both the directional and spatial density distributions of straight-

line edges. As such, EBLs trade clutter for *overlap*: Similar edges are routed close to, or atop of, each other. Less individual edges are visible. In contrast, the coarse graph structure becomes more visible. If related nodes are laid out close to each other, the task of finding coarse-level connections between groups of nodes, *e.g.* finding high-level dependencies between subsystems in a software system, becomes easier than with node-link layouts.

Hierarchical edge bundles (HEBs) are arguably the first well-known EBL algorithm used to lay out large compound digraphs [78, 36]. Containment is compactly shown as a circular icicle plot. Associations are drawn as splines, routed to follow the containment hierarchy. When the analyzed software system exhibits modularity (many edges exist that connect subsystems represented by containment-edge subtrees), the edges get 'bundled' together, making it possible to see this modularity. Visual edge clutter is next interpreted as a sign of limited modularity. HEBs have been used in visualizing call graphs in various applications [36]. However, as the authors mention themselves, a study on the effectiveness of this method for large-scale software systems, as compared to other dependency visualizations, is still to be done [78]. In an attempt to fill this gap, we present such a comparison study in Ch. 3.

Dickerson *et al.* merge edges by reducing non-planar graphs to planar ones [42]. Although this technique preceeds HEBs [78], it is limited to graphs whose nodes can be reordered, using a circular layout, to reduce edge crossings. Gansner and Koren bundle edges in a circular node layout similar to [78] using area optimization metrics [68]. Dwyer *et al.* use curved edges in force-directed layouts to minimize crossings, which implicitly creates bundle-like shapes [48]. Force-directed edge bundling (FDEB) creates bundles by attracting control points on edges close to each other, and generalizes HEBs to graphs which do not have a hierarchical structure (containment edges) [80]. Still, FDEB is implicitly constrained by the node layout, *i.e.*, for meaningful bundles to emerge, we need that related nodes are placed close to each other. This is a general constraint of all EBL methods.

FDEB can be significantly optimized using multilevel clustering techniques such as the MINGLE method [70]. Flow maps produce a binary clustering of nodes in a directed graph representing flows to route curved edges along [133]. Control meshes are used by several authors to route curved edges, *e.g.* [138, 224]. Other recent methods include geometric-based edge bundling (GBEB) [39], which uses Delaunay diagrams, and 'wind-

ing roads' (WR), which uses boundaries of Voronoi diagrams of the node positions for 2D [103] and 3D [104] edge layouts.

Overall, EBLs have gained significant attention in the last years, both in program understanding and in the more general Infovis field. However, EBLs are still faced with a number of important questions and challenges:

- *Scalability:* Computing EBLs for large graphs which do not have hierarchical information is computationally expensive;
- *Clutter:* Even though EBLs reduce small-scale clutter produced by straight-node line drawings, at a higher level, they create significant amounts of overdraw, by placing edges atop of each other. As such, disambiguating close or overlapping bundles, *i.e.* seeing which nodes these connect, can be hard [68, 81]. Bundles are typically implicit: it is hard to exactly say which are the main bundles in an EBL and what sub-graphs these relate, since bundles do not have a distinct visual identity;

In Chapters 5-7, we will show how the above issues can be alleviated.

2.2.3 *Rendering*

Classical rendering of straight-line node-link layouts involves drawing lines for edges, and various types of glyphs for the nodes [95, 172]. Node and edge attributes can be next encoded in the size, color, transparency, shape, and texture of the drawn nodes and edges. The same types of techniques can also be used for EBLs, *e.g.* color interpolation along edges for edge directions [78, 39]; and transparency or hue for local edge density, *i.e.* the importance of a bundle, or for edge lengths [103].

Although these rendering techniques work well for moderately-sized graphs (under roughly 1000 nodes and/or edges), they create visual clutter for larger graphs. The key reason hereof is that, in such graphs, nodes and/or edges overlap. As such, rendering nodes and/or edges independently will cause artifacts, depending *e.g.* on the drawing order of these elements.

Several techniques exist for reducing visual clutter in the rendering pass of large graphs. An early technique in this direction, graph splatting, convolves nodes and (optionally) edges of a node-link layout with a Gaussian filter into a height or intensity map [199]. Dense edge regions, which can cause clutter

in node-link renderings, show up as compact high-value splats. The filter width controls the scale at which overlap is perceived. An enhancement of the basic technique is presented by Niels *et al.* for the visualization of graphs describing spatial movement of vessels [215]. However producing simplified views, and eliminating local clutter, splatting makes it hard to follow edges. Also, the filter width needs careful tuning to avoid creating disconnected, thus misleading, splats. Shaded cushions are effective for showing hierarchies, and have been used for rectangular and Voronoi treemaps [200, 12] and icicle plots and edge bundles [177]. Image-space blending of bundled edges can be used to emphasize both the local edge density and outlier edges (which run in opposite direction from the main direction in a bundle) [78].

Image-space techniques such as the ones mentioned above have several important advantages. Firstly, they reduce the amount of local clutter by essentially performing a low-pass filter on the rendered graph. This can also achieve a continuous simplification of the drawn graph, where more information is shown when more drawing space is available. Secondly, image-based techniques can be efficiently implemented in graphics hardware (GPUs), and thereby achieve the high performance necessary for visualizing large graphs.

However, so far, image-based techniques have been used mainly as a postprocessing step to the actual graph layout and graph rendering. In Chapter 5, we show how EBLs can be simplified using image-based techniques so that the most salient bundles become visible. This addresses the clutter issue mentioned at the end of Sec. 2.2.2. In Chapters 6 and 7, we take this process a step further, and show how the EBL process itself can be formulated as an image-processing operation. This addresses the scalability issue mentioned at the end of Sec. 2.2.2.

2.3 TIME-DEPENDENT GRAPHS

As mentioned earlier, graphs can be either static or time-dependent. In our context, examples of the former are structure-and-dependency graph extracted using static analysis from a given release of a code base. Examples of the latter are graphs that describe the structure-and-dependency of the relationships present in all versions of a code base, such as present, for instance, in a software repository.

Given our focus on edge bundling, the main question in this respect is whether EBL methods can be used to effectively and efficiently depict time-dependent graphs. We outline the related work on this topic below.

2.3.1 *Types of dynamic graphs*

Dynamic graphs can be organized in two categories, as follows.

Streaming graphs are defined as graphs $G = (V, E)$ on a vertex-set V and edge-set E, where edges

$$e \in E = \{n_{start}(e) \in V, n_{end}(e) \in V, t_{start} \in \mathbb{R}, t_{end} \in \mathbb{R}\} \tag{2.1}$$

are defined by their start and end nodes n_{start} and n_{end}, and lifetime $[t_{start}, t_{end} > t_{start}]$. A weak form of Eqn. 2.1 can be used to model streaming graphs where only an ordering of the t_{start} and t_{end} values is specified, rather than absolute values. Streaming graphs occur naturally in cases when an entire graph is not known in advance, *e.g.* events collected from live data sources [3].

Graph sequences are defined as ordered sets of graphs $G^i = (V^i, E^i)$ which typically capture snapshots of the structure of a system at N moments $1 \leqslant i \leqslant N$ in time. We further call a graph G^i in such a sequence a *keyframe*. In contrast to streams, edges are explicitly grouped in keyframes, and additional semantics can be associated with each such keyframe. Following this, sequences may contain so-called correspondences

$$c : E^i \to \{\{e_{corr} \in E^{i+1}\}, \varnothing\} \tag{2.2}$$

Here, $c(e \in E^i)$ yields an edge $e_{corr} \in E^{i+1}$ which logically corresponds to e (if such an edge exists), or the empty set (if no such edge exists). Correspondences model edge-pairs in consecutive keyframes that are related from an application perspective, *e.g.* caller-callee relations between the *same* function definitions in consecutive revisions of a software system.

2.3.2 *Dynamic graph visualization*

Visualizing dynamic graphs has a long history. Methods can be divided into two classes, as follows.

Unfolding the time dimension along a spatial one, *e.g.* using the "small multiples" approach [24], has led to many dynamic graph visualizations. In graph drawing, specific solutions are known for planar straight-line graphs [22]. In software visualization, TimelineTrees [27], TimeRadarTrees [26], TimeArcTrees [177], and CodeFlows [177] lay out a graph along a 1D space, *e.g.* circle or line, and juxtapose several instances thereof on an orthogonal axis to show the graph evolution. Although reducing clutter by not using a node-link drawing metaphor, such methods are visually not highly scalable, nor are they very intuitive, especially for long time series containing complex event dynamics.

Producing an *animation* of the graph's evolution is a second way to understand dynamic graphs. Several techniques generate incremental node-link drawings that show the graph evolution by optimizing a cost function that combines classical static-graph-drawing aesthetic criteria with maximizing the layout stability of unchanging graph parts [65, 56, 62, 82]. Animation can be preferable to small-multiples in conveying dynamic patterns, especially for long repetitive time series [194]. Such methods, however, may suffer from visual clutter, due to the underlying node-link metaphor.

2.3.3 *Bundling dynamic graphs*

It seems appropriate to use the EBL metaphor to visualize the (simplified) structure of dynamic graphs. Indeed, if EBL methods succeed in showing the coarse-level structure of a static graph, they can arguably also be effective in accomplishing the same task for dynamic graphs.

Pioneering work in this area has been recently presented by Nguyen *et al.*, who cut a streaming graph into a set of graphs using a sliding time-window, and visualize each such graph using existing edge-bundling methods [80, 78]. Edge similarity, or compatibility, is enhanced to take into account temporal coherence. Given a stable edge-bundling layout, this method can produce animations of bundled graphs with spatial and temporal continuity.

This approach can be improved in several directions: scalability (number of edges handled), ensuring a high spatio-temporal continuity of the produced animations where large-scale and long-life structures are stable over time and display space, and using the correspondence information present in graph sequenc-

es. In Chapter 8, we present two edge bundling methods for streaming and sequence graphs which incorporate the above-mentioned improvements.

2.4 INTERACTION

Apart from the concerns of constructing a graph layout (see Secs. 2.2.1 and 2.2.2) and next rendering the constructed layout (see Sec. 2.2.3), *interaction* is a major component of any graph visualization method. Essentially, interaction solves the problem of presenting additional information (in a limited spatial area) which the underlying EBL and rendering methods being used cannot show due to either the inherent limitations of these methods, or the size of the visualized graphs.

Related work in interaction for large graph visualization falls within several areas, as follows.

2.4.1 *Magic lenses*

The Magic Lens [19] introduced the idea of locally modifying a screen region based on a user-selected operator. Originally used for modifying the graphics appearance and/or editing the properties of shapes at a focal point, the Magic Lens was subsequently extended to allow more complex operations such as complex effect compositing and interactive lens parameter editing [20]. Tangible magic lenses extended the base concept to allow users to 'slice' through, or zoom in, layered 2D or 3D datasets by interactively moving a 3D tracked physical planar object (the lens) which is either rigid [162] or flexible [113]. Non-linear projection was added to magic lenses to deform 3D scenes as if seen through a cylindrical or spherical lens, working fully in image space, *i.e.* without access to the actual 3D scene [221].

2.4.2 *Semantic lenses, focus and context, and deformation*

The dust and magnet technique allows users to de-clutter large scattered plots by placing several data-attribute-driven 'magnets' in the display space and moving data points close to them based on the points' attributes [222]. This metaphor is somewhat similar to the preset controller [201] which is, however, used for the inverse operation of synthesizing data values based on the distance of a cursor to several data-attributed presets. The bundled graph visualization presented in [79] for compar-

ing software hierarchies proposes a circular and a line-based lens which allow users to interactively select a bundle of interest by drawing and/or brushing over the displayed graph. However, no deformation is used here: focus+context is reached by color-based highlighting the selected edges.

In a different context, Niels *et al.* visualize vessel movements (trajectories) on a geographical map using a blending technique which groups close trajectories into smoothly shaded shapes [215]. Overdraw is eliminated as the dataset is shown as a continuous shaded map. A simple form of semantic lens is used to emphasize specific trajectories, *e.g.* slow moving ships, by tuning the shading and blending parameters. However, spatial deformation is not used to declutter trajectories, since position data is deemed too important to be altered.

Deformation techniques are used for visualizing large datasets by locally changing the underlying spatial layout of the data elements in order to dedicate more space to important data elements than to less important elements. Many variations have been proposed from the original fisheye view [67]. For data tables, the table lens locally distorts the Cartesian cell layout to give more space to specific table rows or columns [141]. For node-link layouts, techniques include local edge deformations, or re-layouts, such as the EdgeLens and its variations [219], and selective edge hiding based on attributes at the position of a user-specified focus point. The local edge lens and bring-neighbors lens of [193] are variations of EdgeLens which remove edges between nodes within a focus zone (lens) and pull nodes connected to nodes-in-focus within the lens, respectively. Edge plucking allows the user to explicitly drag groups of edges away to clarify cluttered zones and/or specify nodes or edges to be left unmoved [218, 217]. However effective, edge plucking requires a certain amount of manual effort. Link sliding and 'bring & go' techniques [122] assist the exploration of node-link diagrams by constraining the user-controlled focus point along a given path in a snap-to-edge manner and moving nodes connected to a node of interest close to that point. Fisheye techniques have also been proposed for trees [193, 69].

2.4.3 *Interaction in EBL layouts*

For EBLs, the visualization can be simplified by creating additional empty space. However, overdraw, or edge congestion, makes interactive selection of specific edges difficult [218]. Since

many edges overlap, local interaction techniques such as edge plucking are less applicable here below bundle level. We address this issue at several levels. In Chapter 5, we present the 'digging lens', a technique that helps separating bundles which spatially overlap at a given location to allow one to see and/or select bundles obscured due to the inherent overdraw. In Chapter 9, we present a more general interaction technique for EBL layouts that generalizes semantic lenses to work on combined position and data attributes rather than on position or data only, as present in most existing lens applications; generalizes the lens from a fixed or parameterized shape (as present in existing work) to arbitrary 2D shapes which are interactively specified by the user via direct painting; uses animation to continuously deform elements within the lens, for any 2D lens shape; and generalizes the deformation to interpolate between two different spatial layouts of a given dataset, apart from repelling elements based on distance to a focal point.

3

COMPARISON OF NODE-LINK AND HIERARCHICAL EDGE BUNDLING LAYOUTS

ABSTRACT: *In Chapter 2, we have outlined that edge bundling is a popular technique to display large graphs. However, to our knowledge, there is only little evidence for the effectiveness of edge bundling techniques in program understanding, as compared to more classical node-link metaphors. In this chapter, we introduce an informal user study that compares HEB [78, 36], probably the best known edge bundling method, and several classical node-link layouts provided in the Tulip graph visualization framework [9, 10]. As task, we focus on the comprehension of very large dependency graphs mined from several open-source C/C++ software projects. We present supporting evidence for the added value of edge bundling techniques, and also several enhancements to the basic bundling technique which we found useful.*

3.1 INTRODUCTION

Software systems contain large and complex sets of dependencies between their components, such as call and inheritance graphs, and data flow and type dependency graphs. Analyzing such dependencies is arguably one of the most important tasks of maintenance processes such as reverse engineering and reengineering. A good understanding of such data supports decisions for code refactoring, removing code clones, identification of design patterns, and debugging.

However, understanding large dependency sets is challenging. Visualization is a method of choice, given the inherent difficulty for understanding large, abstract graphs. Although numerous methods are being proposed for visualizing dependency graphs in the information visualization (InfoVis), software visualization (SoftVis), and graph drawing (GD) communities, it is still unclear how such methods are received by software practitioners in the field, and how they compare when one must accomplish tasks in program comprehension.

In this chapter, we focus on a subset of these activities, and look at the problem of understanding call graphs extracted from software systems which have a hierarchical structure. As we aim to understand the effectiveness of such methods in practice, several aspects are relevant besides the visualization method chosen, *e.g.* the availability of a robust method to extract the call

graphs; the perfect integration of data extraction and visualization [99]; and the scalability of the entire pipeline to real-world systems of hundreds of KLOC.

We describe here an entire tooling pipeline that covers static code analysis, extraction of calls and hierarchy data and their attributes, and visualization. Our focus here is on C/C++ code bases. For this, we implemented a standalone call graph extractor based on the OINK framework [130], one of the most complete, robust, and scalable open-source static analyzers for C/C++. Our call graph extractor enhances the OINK framework with several analyses important for call graph extraction, such as linking declarations to definitions across multiple translation units, and detecting the potential set of called candidates for virtual functions and function pointers. Besides calls, our extractor also delivers hierarchy data (folders, files, classes, methods) and various attributes thereof, such as the call type (static, virtual, by pointer or reference), and details over the function definitions (signature data, access rights, and source code location). The extracted data is saved in several easily importable formats.

For visualization, we needed a scalable, understandable, and easy to use method. As a candidate, we considered the hierarchical edge bundling (HEB) technique, which was very well received in both the InfoVis and SoftVis communities [78, 36]. However, a main question is: How does this technique compare with classical, more accepted, techniques such as node-link diagrams (NLDs)? Such a comparison lacks, and is needed, for large-scale graphs, as the author of the HEB technique also points out. To this end, we performed a study that compares our own implementation of the HEB which adds several enhancements we found useful, and several classical NLD layouts provided in the Tulip graph visualization framework [9, 10].

For this comparison and also to test our entire pipeline, we analyzed several large software systems written in C, C++, and a mix of the two, such as *bison*, *Mozilla Firefox*, and the OINK static analysis framework itself. The analyses were done by developers experienced in software engineering in general and C/C++ in particular, but had no knowledge of the analyzed systems. They had to answer several questions solely based on the two visualizations. We compared the results with the aim of drawing conclusions on the two types of visualizations.

Overall, we can describe our work using the 5-dimensional model of Marcus *et al* [116]: our *task* is to analyze how two different visual metaphors support the visual understanding of call relations in large source code bases; the *audience* includes soft-

ware developers, designers, and architects; the *target* is a graph containing attributed call and hierarchy data; the *medium* consists of two different visualization tools, the Tulip framework and our own enhanced HEB method; finally, the *representation* consists of various types of node-link diagrams and the hierarchical edge bundle metaphor.

This chapter is structured as follows. Section 3.2 presents our call dependency extractor for C/C++. Sections 3.4.1,3.5 and [130] present the results obtained when visualizing call graphs extracted from the bison, Mozilla Firefox, and OINK open-source code bases. In this part, we also introduce the various enhancements we added to the original HEB technique. Section 3.7 discusses the results found in this study. Section 3.8 concludes the chapter.

3.2 CALL DATA EXTRACTION

For our goal, we need a call graph extractor able to accurately detect the various types of function calls occurring in C and C++ source code bases, and is also scalable for real-world systems. After analyzing the available options, we choose to build such a tool atop of the OINK static analysis framework. OINK includes a full-fledged C/C++ parser based on GLR technology. Parsing produces possibly ambiguous abstract syntax trees (ASTs), which are next disambiguated and merged by a semantic analysis pass in a single annotated syntax graph (ASG). The semantic analyzer implements the full C/C++ lookup rules, operator overloading disambiguation, type conversions, and all other operations that determine the type of a symbol and associate it with its declaration. The output of OINK is an ASG of the program, *i.e.* an AST annotated with type information. A major advantage of OINK is that it is open source, and also has a fine-grained API to investigate the produced ASG. This allowed us to implement a call graph extractor atop of the basic static analyzer with relative ease, as follows.

3.2.1 *Location of calls and definitions*

The first step is to locate the constructs denoting function calls. This is easy, as OINK provides a visitor by which we can find all AST node types denoting function calls. These are 'classical' function calls, constructors, destructors, and operators (including conversions and `new` operators). OINK will also provide im-

plicit function calls that do not appear as explicit syntax in the program, *e.g.* calls of base class constructors in derived class constructors and calls of destructors of local stack objects when a scope is exited. Such information has equal importance to 'classical' function calls in refactoring analyses.

Second, we locate all function definitions, *i.e.* functions having a body. This is equally easy using the AST visitor of OINK. As for calls, all types of function definitions are located, including inline functions and template functions.

Third, for each function call and function definition, we locate its *declaration*. For function definitions, this is trivial, as a definition is its own declaration. For function calls, the type information in the ASG output by the semantic analysis provides us with the unique declaration of the called function within its translation unit.

3.2.2 *Linking*

C/C++ programs consist of multiple translation units assembled by a linker which links function declarations from a unit with the corresponding (unique) definitions provided by another unit. We implemented this step atop of OINK as follows. For each translation unit, we save the definitions and declarations of all externally visible functions (*i.e.*, functions that do not have static linkage) in a temporary file. Next, we scan all declarations without definitions in all such files and match them to definitions. This step is massively simplified as OINK provides APIs to check that two function signatures match, according to the full specification of the C/C++ languages. Function declarations for which no definition is found, *e.g.* because they are implemented in binary system libraries, are left unmatched.

3.2.3 *Special cases*

The output of the linking step is a program-wide call graph whose nodes are function calls and function definitions (or declarations, for functions having no definitions) and whose edges are the calls. However, some complications exist. In C/C++, functions can be called via pointers, and C++ has virtual functions. In such cases, the previously described method would only find the declarations of the called functions, but not their definitions. We can provide more specific information, as follows. For functions called via pointers, we construct a set of can-

didates over the entire program which could be the targets of the respective call. This involves all function definitions whose signature matches the call and which do not have static linkage. For virtual C++ functions, we do the same, this time considering all public virtual methods declared in each class hierarchy. This yields a *conservative* set of candidates for each call via pointers or virtual functions. Although such candidate sets may seem to be overly large, they are quite small in practice (5..15 functions), as signature variability and static linkage limit the number of potential candidates. It is possible to further restrict this set by using more sophisticated data flow analyses. The OINK framework provides APIs that could be used to implement this, albeit with more effort.

3.2.4 *Hierarchy*

Apart from function calls, we also extract a program hierarchy. This contains nodes that describe the containment of function definitions (or declarations when no matching definitions were found), and has several levels: directories, files, namespaces, classes, and methods, as well as 'free' (file-scope) global functions. Constructing this hierarchy is easy, since OINK provides for each AST node its exact source code location.

Overall, the output of our entire analysis is a program-wide compound digraph containing calls and containment relations. Besides this, we also save data attributes for each node, *e.g.* its name, function details (method, signature, location in the code, access specifiers) and call details (static, virtual, by pointer, and whether the call is exact or determined via our conservative analysis outlined above). Producing such a graph from a given code base is easy: one can simply run an existing makefile, substituting the compiler's name with our extractor, with no further changes. The resulting graph is the input for the visualizations described next.

3.3 METHODOLOGY

To compare the two types of visualizations we target, *i.e.* node-link diagrams (NLD) and hierarchical edge bundles (HEB), we proceeded as follows. First, we extracted several call graphs from increasingly large systems, as described in Sec. 3.2, among which we mention the *bison* parser generator, the OINK C/C++ static analysis framework, and the *Mozilla Firefox* browser. Next,

five developers with no prior knowledge on the analyzed systems were introduced to the NLD and HEB visualization methods to be used, and were given the opportunity to use these systems for a few days, on small datasets, until they were comfortable with their operation. Next, the developers used the NLD and HEB visualizations to answer a number of generic questions on the analyzed systems, *e.g.*: which are the main components in the system; how these components communicate with each other; whether the system is highly modular or not; where is dead code (uncalled functions); and how is the use of polymorphic interfaces (*i.e.* function pointers and virtual functions) spread over the system. Next, several specific questions were asked, *e.g.*: which interfaces (*i.e.* sets of functions declared in the same component) does a specific component call, or provide; and where is a given interface used. The answers, as well as additional comments and remarks on the operations performed to achieve the answers, and the ease-of-use of the respective visualizations, were recorded. A sixth person with detailed knowledge on the analyzed systems performed the study separately and also checked the answers of the other five. Finally, conclusions were drawn using the analyzed answers.

3.4 CASE STUDY 1: THE *bison* PARSER

3.4.1 *Node-link visualizations*

The first type of visualization we analyzed is the classical node-link visualization. Nodes are function definitions or containers (directories, files, classes) and edges show calls. For visualization, we used the Tulip framework for several reasons. First, Tulip provides a wide range of functions including many node-link layouts, search and select functions, interactive navigation, and visual customization of colors, shapes, textures, and labels. Second, Tulip is highly memory and speed optimized for very large graphs. Last but not least, all operations are directly accessible via a well-documented user interface (menus, dialogs), making it usable with zero programming effort. This is essential for us, as we assume our users are programmers who want to quickly investigate a large call graph, and have no time or experience to develop their own visualization code.

Figure 3.1 shows several snapshots produced using the NLD layouts of Tulip on the *bison* call graph (868 functions, 5535 calls). From the recorded procedure, we saw that all users first aimed

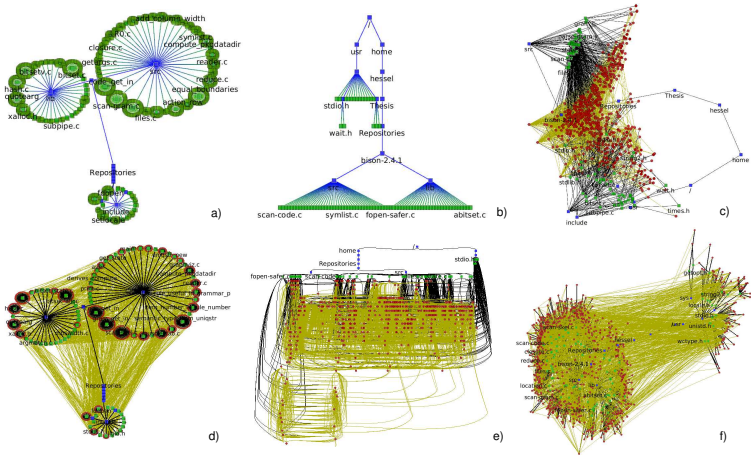

Figure 3.1: Visualizations of the *bison* call graph using Tulip: hierarchy only using bubble trees (a) and directed trees (b); force-directed layouts of hierarchy and calls using HDE embedder (c); hierarchy and calls using bubble trees (d) and dendrograms (e); and GEM (f).

at obtaining a simple hierarchy view, the reason being to get an idea of the system size, number of layers, and which are the largest subsystems. Images (a) and (b) in Fig. 3.1 show the two layouts which were found best for this task: the bubble tree layout, which arranges child subtrees in a circle around their parent node [73] and the classical directed tree layout. For this and the other systems analyzed, the bubble tree layout was found easier to comprehend, as it yields layouts with good aspect ratios, and also lets one compare the relative sizes of subsystems quite easily (circle size versus length of a row of nodes in the tree layout).

The next step was to bring the calls in the picture. For this, the first attempt was to add them to the existing hierarchy visualizations. Figure 3.1 d shows the complete compound graph with the bubble tree layout. Calls are drawn as thin yellow (light) lines, containments are drawn as thick (dark) black lines. Node colors and shapes show their type: directories (blue, squares), files (green, squares), and functions (red, circles). As suspected upfront, the result is quite cluttered. At this scale, the only conclusion that was drawn is that the system is quite tightly connected; its three main subsystems `lib`, `src` and `include`, *i.e.* the top-left, top-right, and bottom large circles respectively, are all strongly interacting. Another observation achieved with this

view is that functions are not uniformly spread over files: some green squares are surrounded by many red circles. These are files containing many functions, whereas others have only one or a few such circles. These are files containing few used functions, *e.g.* the `include` subsystem.

Alternative types of tree layouts provided by Tulip were explored to show both hierarchy and calls, as well as various layout parameter settings. Most of them did not produce useful results, due to the high clutter caused by the call edges. For example, Fig. 3.1 e shows a dendrogram layout overlaid with call edges drawn as splines. It might be argued that this layout is useful in comparing call depths between different subsystems, by looking at the height of the red dot sequences (functions) in the lower part of the image. However, showing the actual call edges is not useful, as they produce just clutter.

Further, several force-directed layouts were tried out. Figures 3.1 c and f show the compound graph drawn using the HDE embedder [74] and GEM [63] layouts of Tulip. These are optimized versions of the original publications, which add several heuristics and speed enhancements to deliver higher quality in less computational time (see [9, 10] for details). The HDE layout is able to pull the hierarchy nodes (directories and files, shown in blue, respectively green) apart from the function nodes (shown in red, in the middle). For example, we see the files in the `src` directory being isolated in the upper-left part of the image. However, the function nodes, strongly connected by many calls, form a cluster in the middle which is not understandable. Figure 3.1 f shows a layout using an enhanced version of the well-known GEM spring embedder. This layout is able to pull apart the `include` subsystem, which contains system functions used by the *bison* core, but cannot separate well the *lib* and *src* subsystems, as these are tightly coupled.

Overall, the bubble layout was considered to be the best for the generic comprehension tasks, as it exhibits a stable, regular node placement pattern. For the specific comprehension tasks (see Sec. 3.3), the built-in search-by-attribute-value and path highlighting functions of Tulip were used. Although these functions are easily accessible via Tulip's GUI, the high visual clutter caused by the dense call pattern in *bison* severely limits the effectiveness of the node-link visualizations. Here again, the bubble tree performed best. The reason seems to be the fact that this layout strongly emphasizes the hierarchy, which is used as a visual guide when analyzing specific call relations.

3.4.2 *Hierarchical edge bundling visualizations*

For the second type of visualization, we used SolidSX [159], our own implementation of the HEB method with several enhancements, described next[1]. The design of SolidSX is minimalist: all operations are available within a single interface, the main visualization. There are no extra buttons or menus except pop-ups. All operations are accessible with the smallest number of mouse clicks possible. The main HEB idea is simple: hierarchy is drawn as a set of concentric rings divided in sectors, each sector being the container of inner ring sectors corresponding to it; calls are drawn as splines between their corresponding ring sectors; splines are further bundled according to the containment hierarchy, as described in [78].

Figure 3.2 a is an overview of the same *bison* call graph. Several points were made when comparing this image with the NLD layouts in Fig. 3.1. Showing containment as concentring rings was very easy to understand. The fact that node labels are, at least on the larger rings, readable was seen as a great advantage compared to the NLD label display. Although great effort was done in Tulip to eliminate label overlaps and provide an automatic level-of-detail control of the label size, this was not seen as highly effective. Labels still overlap call edges, and the level-of-detail feature makes labels pop in and out the view depending on the zoom level in a disturbing way.

We enhanced the concentric ring design to display attributes. Each node in SolidSX's input graph can have any number of data attributes, stored as (name,value) pairs, the values being string, numerical, or boolean. We map these values to node colors. A pop-up widget displays all different attribute names present in the input (Fig. 3.2 a top-right). Attributes can be sorted by name or number of different values they take in the input data. Simply moving the mouse over the listed attributes (brushing) changes the colormapped attribute. For numerical and boolean attributes, we use a simple blue-to-red colormapping based on range. Strings are mapped based on alphabetical ordering. Overall, one can see which attributes are available, and quickly change the one shown to compare different attributes over the same dataset, with one single mouse click and mouse stroke. An identical mechanism is provided for edge attributes, which are mapped to edge colors.

1 SolidSX is available for academic or commercial users from `www.solidsourceit.com/products`

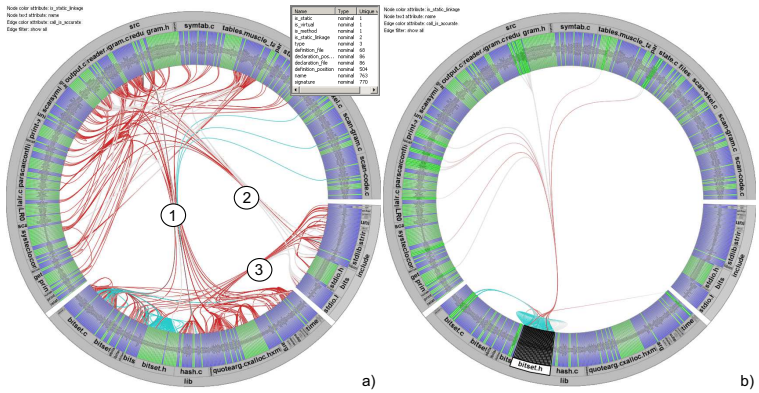

Figure 3.2: Visualizations of the *bison* call graph using SolidSX: entire system (a); selected subsystem with most function-pointer calls (b).

Looking at the overview of *bison* in Fig. 3.2 a, we quickly locate the main interactions between the three subsystems: `src-lib` (1), `src-include` (2), and `lib-include` (3). Due to the bundling effect, visual clutter is much smaller than in the NLD visualizations in Fig. 3.1. It was easy to find the many functions which do not get called: these are the innermost circle segments which have no edges connected to them. Doing this task in the NLD visualizations was only possible using Tulip's search functions, but not the images.

Assessing the usage of 'polymorphic' interfaces[2] was easy, by coloring edges based on call type. In Fig. 3.2 a, static function calls are red, and pointer calls are blue. Clearly, the pointer calls are a minority. Many such calls exist in the `bitset.h` file (below in the image). We added to SolidSX the ability to show only calls related to a given node, by clicking on that node. When clicking on `bitset.h`, the file and all its contained functions are displayed in black outline (Figure 3.2 b). We see that this file has many blue edges going to itself, two blue edges going to `bitset.c` to its left, and a few red edges going to various other parts of the system. This is interpreted as follows: `bitset.h` provides many function declarations, which have equivalent signatures (the loop-like edges atop `bitset.h`); these are only called via pointers; there are only few clients who call such function pointers (the red edges going outwards from `bitset.h`); and there are only two function definitions, in `bitset.c`, which can implement these interfaces. This is precisely the type of infor-

2 By this, we mean C functions called via pointers in *bison*

mation stored in the candidate sets of the pointer-call analysis (Sec. 3.2).

Adding color to the function definitions in SolidSX brings additional insight. In Fig. 3.2, we show the static linkage attribute of a function. Green indicates `static` functions, while blue shows functions visible by a linker. Interestingly, all function declarations in `bitset.h` are static. Hence, access to these 'polymorphic' features of *bison* can only be done via pointers to them.

3.5 CASE STUDY 2: MOZILLA FIREFOX

In this second example, we analyzed the Mozilla Firefox code base. Given space limitations, we will only discuss two plugins of the entire system. Figure 3.3 a,b show the entire call graphs of the *libgklayout* plugin (11817 functions, 21167 edges), visualized using SolidSX and Tulip's GEM layout. Directory nodes are drawn blue, files are yellow, classes are green, and functions are blue. Static calls (edges) are red, virtual calls are cyan. At this scale, the GEM layout is clearly not able to disentangle the calls. However, the HEB layout is reasonably easy to read, due to the edge bundling and edge aggregation. For example, we see that almost all virtual calls are directed at a few functions in a single file, *nsCOMPtr.h*, outlined in black in the upper-left of Fig. 3.3 a. The virtual calls are only visible as a blue spot in the GEM layout (Fig. 3.3 b).

This figure illustrates also a further enhancement we added to the basic HEB idea. In SolidSX, we allow users to show or hide entire hierarchy layers by simple mouse operations. Hidden layers, usually the top-level ones in the system, are drawn as very thin outer rings, as opposed to the regular visible layers, which are thick. Hiding layers saves screen space for the inner layers in deep hierarchies. Still, one can see the color of the hidden (thinly drawn) layers, and count them, thereby getting a cue of how deep one is in the hierarchy. The width of the hidden rings, regular rings, and leaf-node (functions) ring can be also controlled explicitly by the user, if desired. A zoom-in on a small sector of Fig. 3.3 a is shown in Fig. 3.4. We see here 10 hidden hierarchy layers which take up only the space needed by a single layer in the big picture.

Fig. 3.3 c,d show a much smaller plugin, *libembed* (677 nodes, 936 edges). At this scale, both the NLD and HEB layouts perform comparatively. The users detected here quite easily, in both

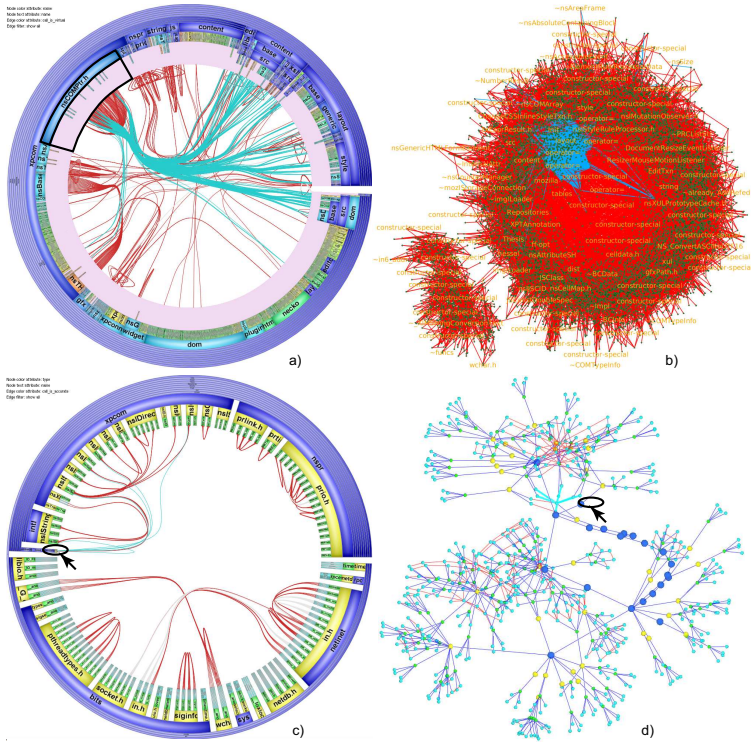

Figure 3.3: Call graphs of Mozilla plugins: *libgklayout* (a,b) and *libembed* (c,d). Color emphasizes virtual calls.

images, that this plugin contains only a single virtual function (marked by a circle and arrow in the images), called 7 times. This figure shows yet another enhancement of the original circular layout: Instead of rendering all nodes on the same level as contiguous segments on the same circle, we leave gaps between neighbor segments which correspond to nodes which do not have the same parent. In other words, contiguous circle segments indicate siblings, and gaps separate subtrees. This view helps emphasizing the software's hierarchical tree structure, at the expense of a small space trade-off.

3.6 CASE STUDY 3: THE OINK FRAMEWORK

In this third and last example, we analyzed the OINK C/C++ static analysis framework itself. OINK contains around 350 KLOC written mainly in C++, with small parts in C, developed over 6 years by a team of 10 people. The architecture of OINK is

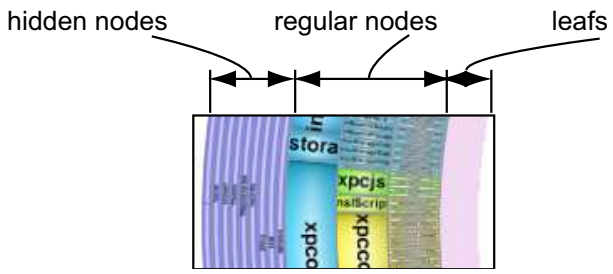

Figure 3.4: Zoom-in on Fig. 3.3 a illustrating the hierarchy hiding.

quite elaborate. It consists of a lexer (implemented using *flex*), a GLR parser (implemented using the *elkhound* parser-generator library[118]), an *ast* class library for the over 180 C/C++ grammar nodes, and a semantic analyzer (*elsa*). Our expert programmer, who worked for over 2 years on OINK development, stated that the lexer, parser generator, and AST class library are rather modular and reusable subsystems, in line with the intentions of the OINK developers to make these reusable for a family of languages; however, the semantic analyzer is a much more complex subsystem, with tight couplings throughout the entire system. The question was if this kind of insight could be obtained by the other users using only dependency visualizations.

The OINK call graph, extracted as described in Sec. 3.2, has 23497 function definitions, 242132 calls, and 2060371 attribute values. This is *two* orders of magnitude larger than all systems visualized so far with the HEB method [36]. At this scale, all NLD layouts in Tulip break down - some only produce fully cluttered images, some abort with no result. Since showing all these calls at once may be sometimes too much information even for the HEB layout, we added support in SOLIDSX to allow to navigate the input graph by hierarchy layers. Clicking on nodes allows expanding or collapsing a node. Collapsed nodes aggregate all their calls from/to outside nodes n_i and display a single thick edge per node n_i. If the attributes of all aggregate edges have the same value, then this value is used to color the edge, else the edge is colored gray. Figures 3.5 a-c show the call dependencies of the entire OINK system at file level (a), class level (b), and method level (c). Only three clicks are needed to produce these three views - each click further expands a deeper hierarchy level. Directory nodes are drawn blue, files are yellow, classes are green, and functions are blue.

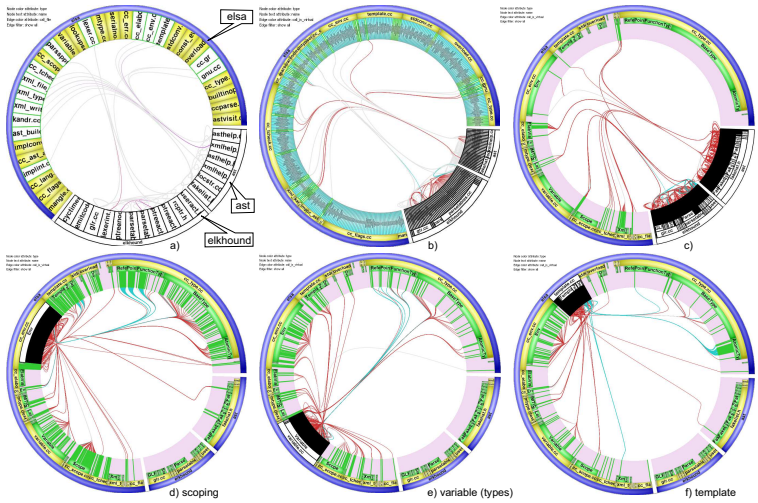

Figure 3.5: OINK framework: multilevel visualization of calls on the level of files (a), classes (b), and functions (c); Main semantic analysis subsystems: the scoping environment (d), variables (e), and template analysis code (f). Selected subsystems are shown in black.

In Fig. 3.5 a, the *ast* and *elhound* systems, visible by their white background and thick black borders, were selected by clicking as explained in Sec. 3.4.2. We see that these systems have few calls from the analyzer's core, *elsa*. This suggests a potentially good separation of these three subsystems. The next image, Fig. 3.5 b, shows the entire system one level deeper, *i.e.* at class level. The innermost ring is predominantly blue, which means function definitions (blue) are directly contained in implementation files. The few green spots denote private implementation classes, which are thus only sparsely used in this system. The next image, Fig. 3.5 c zooms one level deeper, showing all functions in the system. As the number of functions in this case is far larger than the amount of available pixels, we chose a simple solution: we render only the nodes involved in the relations with the selected nodes, and render the remainder of the nodes in gray (see Fig. 3.4 for a detail zoom-in corresponding to the image in Fig. 3.3 a). These are shown in green on the inner circle in Fig. 3.5 c. Here, we see that, although the two selected subsystems *ast* and *elkhound* have very strong internal cohesion (many self edges), their communication with the system's core (*elsa*) is indeed quite limited. This is a good sign for modularity.

Figure 3.5 c also shows the relative sizes of OINK's components. Files containing many functions occupy a larger part of the circular layout. We see that these are the files of the semantic analyzer: the scoping environment (`cc_scope.cc`), the template analysis code (`template.cc`), the type system classes (`variable`, `cc_type.cc`). To analyze how modular the semantic analyzer is, we select its components by clicking (see Fig. 3.5 d-f). We now see not only that these are large, but also have much more connections with large parts of the entire system than the *ast* and *elkhound* subsystems - compare the amount of green segments on the innermost ring and number of edges in Figs. 3.5 d-f with those in Fig. 3.5 c. This finding correlates with the expert programmer's experience: OINK is modular with respect to the *ast* and *elkhound* parser generator, but its semantic analyzer is over half of its code, and a tightly coupled one for that matter.

Finally, to assess the polymorphism of the OINK code base, we use again edge coloring. In Fig. 3.5, red denotes static function calls, and blue denotes virtual calls. We see relatively few virtual calls - this is in line with the OINK design documentation, which stresses a minimal use of virtual methods for optimal performance.

3.7 DISCUSSION

3.7.1 *Usability comparison*

We distilled several points from the reports provided by the five users in this study. All users strongly agreed that the HEB layout is overall vastly superior to node-link diagrams (NLDs) for navigating call-and-hierarchy graphs larger than a few hundred nodes, for virtually all considered tasks, because:

1. HEBs show more data on the same amount of screen space
2. edges in HEBs are much less cluttered
3. hiding/showing nodes changes HEBs less than NLDs
4. the circular layout draws parent nodes naturally larger
5. HEBs show more node labels with less clutter than NLDs
6. interaction in HEB is always near-real-time, while some NLDs take long to compute

However, some advantages of NLDs were mentioned too:

1. NLDs allow more freedom in manual layout editing
2. NLDs make it easier to follow a path than the HEB
3. the HEB circular layout places sometimes unrelated nodes close to each other

For our tasks of interest, the advantages of HEB compensated the advantages of NLDs. Although not rigorously timed, we noticed users of HEBs being 3..5 times faster in accomplishing the same task than when using NLDs, the average task in HEB being 1..3 minutes. The search and select functions of both tools used are comparable in effectiveness and simplicity, so the difference can be attributed to the visualization part. For instance, obtaining a view as Figs. 3.2 or 3.5 takes around 1.5 minutes and around 10-15 mouse clicks, including loading the data. Obtaining a similar image in Tulip takes around 5 minutes and a few tens of clicks and selections. In both cases, we used no custom application presets.

3.7.2 *Performance comparison*

Both Tulip and SolidSX are highly engineered for performance, which is important for graphs of hundreds of thousands of elements and attributes. For example, the oink dataset (Sec. 3.6) takes 178 MB to store in Tulip and 395 MB in SolidSX. The difference is explained by Tulip's special memory management which uses custom bit-level allocation to limit memory waste [8]. All Tulip tree-like layouts are of comparable, near-real-time, performance as the HEB layout. The HDE embedder and GEM are considerably slower, taking *e.g.* about 2 minutes on the relatively small *bison* dataset (Sec. 3.4.1) on a 2.8 GHz PC.

3.7.3 *Threats to validity*

For our comparison of visualization methods for call-and-hierarchy data, the following points are important. First, we only compared a limited number of NLD layouts with the HEB layout. Other layouts, *e.g.* SHriMP-like ones, could perform better than those studied here. There are, however, reasons to believe the opposite. SHriMP-like layouts are effective in showing containment, but they do not scale well in number of associations. These tend to occlude the containment drawing, and also are hard to distinguish among themselves (see *e.g.* [106, 172]).

Such layouts are effective for top-level architecture-like views, but not for massive call graphs. However, we could not test all possible NLD layouts in existence. The choice for Tulip was explicitly done from an end-user perspective: choose a scalable, documented, user-friendly, highly optimized NLD visualization tool, compare it with a HEB implementation sharing the same features, and see which one is better accepted by users.

3.7.4 *Availability*

The entire toolset, including the C/C+ call-and-hierarchy extractor, the SolidSX visualization tool, and the extracted call graphs in Tulip and SQL formats, are available from the authors upon request. Additional components to our toolset, not discussed here, include plug-ins to automatically extract dependencies, syntactic information, and metrics from Visual C++ projects and .NET assemblies.

3.8 CONCLUSIONS

We have presented a study that compares the usage of node-link diagram (NLDs) and hierarchical edge bundle (HEB) layouts for the visualization of large call-and-hierarchy graphs of software systems. To perform this, we have constructed a fully automatic pipeline for extracting call graphs from C/C++ programs, including a call static analyzer, and an enhanced implementation of the HEB method for navigating very large graphs. The study points out an important advantages of the enhanced HEB method for typical comprehension tasks involving call-and-hierarchy data, and demonstrates the applicability of such methods for the understanding of large, real-world, programs.

We are currently working on extending our call-and-hierarchy visualization with additional views to support investigation of additional graphs, *e.g.* class hierarchies, usage of types, and data flow, as well as visualizing multiple attributes in a single view, *e.g.* static type information, type matching, and source code metrics. It is also interesting to study how some of the perceived advantages of NLD layouts could be merged with the HEB views to obtain a visualization that combines the benefits of both methods.

The main conclusion from this study is that edge bundled layouts (EBLs) are a promising tool for exploring large structure-and-dependency (compound) graphs, and surpass the classical

straight-line node-link layouts in terms of readability. However, we also discovered that EBLs pose some interpretation challenges, such as visually following overlapping bundles. In Chapter 5, we present a method which alleviates these overlap problems by proposing a simplified way to render edge bundles.

This chapter is based on:

Alexandru Telea, Hessel Hoogendorp, Ozan Ersoy, and Dennie Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*, 81-88 (2009).

4

THE SOLID* TOOLSET FOR SOFTWARE VISUAL ANALYTICS

ABSTRACT: *In Chapter 3, we have shown that graph bundling is an effective technique for typical program comprehension tasks. However, to be successful in practical contexts, such as for maintenance activities in the IT industry, a visualization technique has to be embedded in a complete toolchain, with fact extraction, program analysis, data filtering and selection, and multiple views that address multiple concerns. In this chapter, we present the architecture of an end-to-end software visual analytics (SVA) system in which graph bundling is an essential component. Software visual analytics (SVA) tools combine static program analysis and fact extraction with information visualization to support program comprehension. We illustrate the toolset's usage for constructing software visualizations with examples in education, research, and industrial contexts. We next discuss several design choices which we made during the tool's evolution path from research prototype to integrated product. Our discussion provides a context for assessing the additional requirements that a software visualization application based on edge bundling has to comply with in order to be accepted in practice.*

4.1 INTRODUCTION

As mentioned in Chapter 1, software maintenance covers 80% of the cost of modern software systems, of which over 40% represent software understanding [164, 35]. Although many visual tools for software understanding exist, most know very limited acceptance in the IT industry. Key reasons for this are limited scalability of visualizations and/or dataset sizes, long learning curves, and poor integration with software analysis or development toolchains, as strongly voiced by several researchers [143, 31, 99, 223].

Visual analytics (VA) integrates graphics, visualization, interaction, and data collection and analysis to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [220, 188]. These fields share many similarities with software maintenance in terms of *data* (large databases, structured text, and graphs), *tasks* (sensemaking by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization). VA stresses tool integration, as opposed to pure data mining or fact extraction (whose main focus is scalability) or information visualization (Infovis, mainly focused on presentation). As such, VA is a promising model for building effective and efficient software visual analysis (SVA) tools. However, building SVA tools for software

comprehension is particularly challenging, as developers have to master static analysis, fact extraction, graphics, information visualization, and user interaction design technologies.

In Chapter 3, we have described a preliminary comparison of the effectiveness of node-link layouts and edge bundling layouts (EBLs) in the context of program understanding. The key conclusion was that EBLs show strong advantages, in terms of readability and visual scalability, as opposed to node-link layouts. However, for EBLs to be accepted as (part of) integral program comprehension solutions in the IT industry, they have to be complemented by additional tools and mechanisms, such as fact extraction, quality metrics computation, and additional visualization techniques.

In this chapter, we present our experience in building SVA tools for software maintenance. We outline the evolution path from a set of research prototypes to a commercial toolset used by many end-users in the IT industry. Our toolset supports static analysis, quality metrics computation, clone detection, and state-of-the-art Infovis techniques such as table lenses, cushion treemaps, and dense pixel charts. EBLs form a key element of our toolset. The toolset addresses several use-cases, of which we focus here on two: visual analysis of program structure and code duplication. These use-cases can be combined to support tasks such as assessing system quality and planning refactoring.

The contributions of this chapter are as follows:

- describe our toolset with respect to the ease of installation, usage, and applicability to program comprehension;
- detail the design decisions and evolution path of a SVA toolset for program comprehension from research prototypes into an actual product;
- present the lessons learned in developing our toolset in research and industrial contexts, with focus on efficiency, effectiveness, acceptance, and experienced pitfalls;
- present evidence for our design decisions based on actual toolset usage.

In the context of the current thesis, we argue that the construction and usage of our SVA toolset (in which EBLs are a key visual element) extend and support the hypothesis outlined in Chapter 3 that EBLs are an effective and efficient instrument for visual program comprehension in practice. However, the lessons learned from our SVA toolset construction presented

here also stress the point that EBLs, by themselves, cannot be a solution to program comprehension in practice: For them to work effectively, they have to be complemented (and integrated with) carefully designed techniques for data extraction, interaction, and additional visualizations.

The structure of this chapter is as follows. Section 4.2 details our toolset's architecture. Section 4.3 details two tools in our toolset which offer fact extraction and visualization of software structure, metrics, and code duplicates, and outlines its installation, usage, and extensibility. Section 4.4 shows the use of our toolset in an industrial software assessment case. Section 4.5 discusses the lessons learned in developing efficient, effective, and accepted SVA tools. Finally, section 4.6 concludes the chapter.

4.2 SVA PROGRAM COMPREHENSION TOOLSET: ARCHITECTURE

The research group in which the current thesis was completed was involved, in the past decade, in the construction of over 20 SVA tools for software requirements, architecture, behavior, source code, structure, dependencies, and evolution. These tools were used in academic classes, research, and industry, in groups from a few to tens of users. Latest versions of these tools have formed the basis of SolidSource, a company specialized in software visual analytics [160]. Table 4.1 outlines the most important tools in this collection, with binaries and source code available [169]. Evolution of this toolset highlights several aspects relevant to the path of effective academic-to-widely-used tool development: architecture and design decisions, choice of techniques and software components, and effective presentation aspects. We next detail the data and visualization architecture of this toolset (Secs. 4.2.1 and 4.2.2). The toolset's architecture evolution from research prototypes to products is discussed further in Sec. 4.5.9.

4.2.1 *Data architecture*

Our toolset uses a simple dataflow architecture (Fig. 4.1). Raw input data comes as unversioned source code or versioned files stored in SCM systems. From raw data, we extract several *facts*: syntactic and semantic structure, static dependency graphs, and source code duplication. The data architecture used to manage these facts is detailed next.

Tool	Targeted data types	Visual techniques	Analysis techniques	Drawing	Data storage	Users
SoftVision (2002) [183]	software architecture	node-link layouts (2D and 3D)	none (visualization tool only)	Open Inventor	text files (RSF format)	10..20
CSV (2004) [112]	source code syntax and metrics	pixel text, cushions	C++ static analysis (gcc based parser)	Open Inventor	plain text and XML	10..20
CVSscan[206] (2005)	file evolution	dense pixel charts annotated text	CVS fact extraction (authors & line-level changes)	OpenGL	text files and SQLite	20..30
CVSgrab (2006) [204]	project evolution	dense pixel charts, cushions	CVS/SVN fact extraction (project-level changes)	OpenGL	text files and SQLite	30..50
MetricView (2006) [186]	UML diagrams and quality metrics	2D node-link layouts, table lenses	C++ lightweight static analysis (class diagram extraction)	Open Inventor	UML files (XMI format)	50..80
MemoView (2007) [121]	dynamic logs (memory allocations)	table lenses, cushions, timelines	C runtime instrumentation (libc malloc/free logging)	OpenGL	binary files (own format)	5..10
SolidBA (2007) [182]	build dependencies build cost metrics	table lenses, 2D node-link layouts	C++ dependency mining, and automated refactoring	OpenGL	SQLite	15..25
SolidFX (2008) [181]	reverse engineering	pixel text, annotations, table lenses	C/C++ heavyweight static analysis	OpenGL	binary files and SQLite	50..75
SolidSTA (2008) [160]	file and project-level evolution	dense pixel charts, cushions, timelines	CVS/SVN/Git fact extraction and source code metrics	OpenGL	SQLite	200..250
SolidSX (2009) [160]	structure, metrics, associations	HEB views, treemaps, table lenses, cushions	.NET, C++, Java lightweight static analysis	OpenGL	SQLite	200..250
SolidSDD (2010) [160]	code duplicates, structure, metrics	HEB views, treemaps, table lenses, pixel text	C, C++, Java, C# configurable syntax-aware clone detection	OpenGL	SQLite	100..150

Table 4.1: Software visual analytics tools - evolution history (Sec. 4.5.9). Tools discussed in this paper are in bold. The first six tools are research prototypes. The latter five tools (Solid*) are commercial products.

Relational and attribute data is stored into a SQLite database [163] whose entries point to flat text files (for actual source code) and binary files (for complete syntax trees, see 4.3.2.1). Fact extraction is implemented by specific tools: parsers and semantic analyzers for source code, and clone detectors for code duplication (see Sec. 4.3).

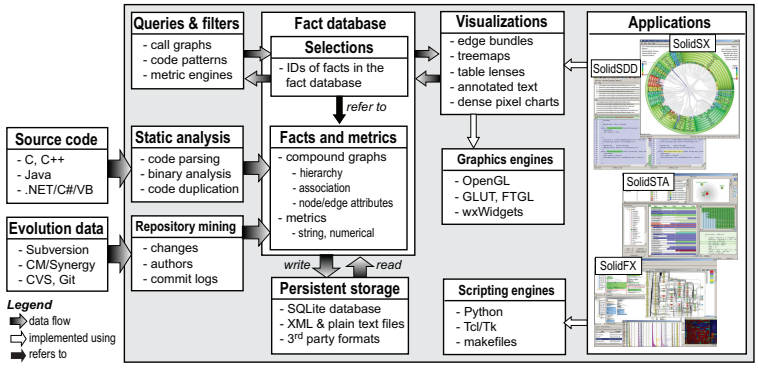

Figure 4.1: Toolset architecture (see Section 4.2).

Besides facts, our database stores two other elements: selections and metrics. *Selections* are sets of facts or other selections. They support the iterative data refinement in the so-called visual sensemaking cycle of VA [220, 188]. Selections are created either by tools, *e.g.* extraction of class hierarchies or call graphs from annotated syntax graphs (ASGs), or interactively by users. Selections have unique names by which they are referred by the tools and under which they are persistently saved. *Metrics* are numerical, ordinal, categorical, or text attributes. Metrics can be added to facts or selections by tools, *e.g.* complexity, fan-in, fan-out, cohesion, and coupling, or set as annotations by users, *e.g.* marking certain classes in a UML view as being 'unsafe'.

Using an SQL database to query large attributed relational data can pose efficiency problems if this requires multiple-table joins [90, 203]. To alleviate this, we adopted the following schema (see Fig. 4.2):

- each node, containment (hierarchy) or association edge, and selection, has a unique ID;
- a *hierarchy* table: each row stores a containment edge, listed as (parent, child) node IDs;
- an *association* table: each row stores an association edge, listed as (from, to) node IDs;
- a *node attribute* table: each row stores all attributes (metrics) $a_1, \ldots, a_n$ of a given node as n columns;
- an *edge attribute* table: each row stores all attributes (metrics) $a_1, \ldots, a_n$ of a given edge as n columns; different edge types, *e.g.* calls, uses, includes, are modeled by adding an edge-type attribute;
- two *selection* tables per selection, for the node IDs and edge IDs of the selected facts, respectively; additional attributes (metrics) of the selected facts in the context of a specific selection can be added as extra columns.

This schema can store any compound (hierarchy-and-association) attributed graph: ASGs, class hierarchies, call graphs, or clone relations. New association types can be added to the database without changing its schema, since types are stored as attributes. The example in Fig. 4.2 (bottom) shows this for a simple program. Hierarchy consists of a file `main.cc` with two functions `main()` and `run(Foo)`, and a class `Foo` with a method

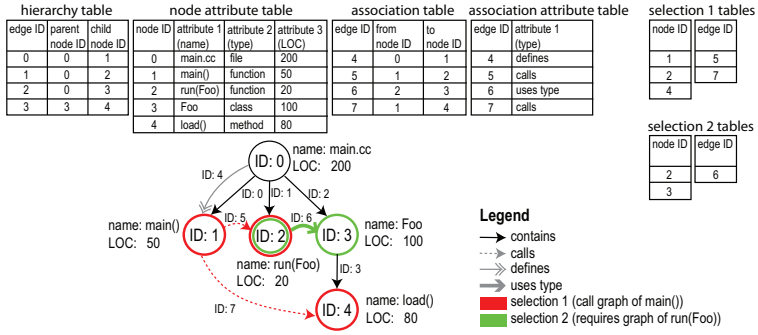

Figure 4.2: Database schema (top) for a compound attributed graph (bottom) and two selections: the call graph of main() and the 'requires' graph of run(Foo).

load(). Associations capture call; define; and 'uses type' relations (*e.g.* the fact that run(Foo) uses the type Foo), modeled as edge 'type' attributes. Nodes have two attributes: name and lines-of-code size (LOC). Two selections exist: the call graph of main() (red), and the 'requires' graph of run(Foo) (green).

On-the-fly the computed data, *e.g.* selections, metrics, visualization and layout properties, and annotations, imply the creation and/or editing of additional selection tables or attribute columns. Using a separate table for each selection optimizes speed and memory consumption, as in a typical analysis scenario tens or even hundreds of different selections of variable size are created and deleted dynamically. Missing values are naturally accommodated by the SQL database. Storing hierarchy data as a separate table from the associations enables efficient traversals of the compound graph for *e.g.* level-of-detail rendering and interactive selection. For instance, finding the children of a node uses just the hierarchy table. If we had containment and associations edges in the same table, distinguished *e.g.* by a special 'type' attribute in the association attribute table, finding children would need to query *both* the association table and association attribute table. Rendering and selection have to be as fast as possible to maximize visual fluency, so we opted for a separate hierarchy table solution. The same special treatment of hierarchy relations is used also by other graphics or graph layout toolkits, *e.g.* OpenInventor [208] and Graphviz [7].

The above simple model scales well to fact databases of hundreds of thousands of facts with tens of metrics per fact [81, 205]. Multiple hierarchies can be added as multiple hierarchy tables. Simple queries and filters can be directly written in SQL. Struc-

tural queries, *e.g.* connected components or reachability, are efficient, as they require just iterating over the node and association tables. For example, rendering the graph in Fig. 4.3 (4K nodes, 15K associations) takes subsecond time on a commodity PC.

Selections are the glue that allows composing different tools. All analysis (*e.g.* filters, queries, or transformations) and visualization components in our toolkit read selections, and optionally create selections (see Fig. 4.1). Hence, selections are the only interface that tools use to communicate with each other. In this way, existing or new tools can be composed either statically or at run-time with no configuration costs. To ensure consistency, each tool decides internally whether (and how) to execute its function on a given input selection.

4.2.2 *Visualization architecture*

Visualizations display selections and allow users to interactively navigate, pick elements, and customize the visual aspect. Since they only receive selection names as inputs, they 'pull' their required referred data on demand, *e.g.* a source code viewer opens the files in its input selection to render the text. Tools can freely decide to cache data internally to reduce traffic with the fact database, if desired. This decision is completely transparent at the architecture level.

Our current toolset offers several visualizations, as follows. *Table lenses* show large tables by zooming out the table layout and drawing cells as pixel bars scaled and colored by data values [141]. Subpixel sampling techniques allow rendering tables up to hundreds of thousands of rows on a single screen [173, 169]. *Hierarchically bundled edges* (HEBs) compactly show compound (structure and association) software graphs, *e.g.* containment and call relations, by bundling association edges along the structure [78]. *Squarified cushion treemaps* show software structure and metrics for up to tens of thousands of elements on a single screen [155, 169]. We also provide classical views, such as metric-annotated code text, tree browsers, customizable color maps, legends, annotations, timelines, details-on-demand (tooltips), and text-based search tools. The above visualizations are illustrated by the two tools described next in Sec. 4.3.

A single treemap, HEB, or table lens can show the correlation of just a few attributes. We augment this by the *multiple correlated views* technique. Besides the 'input selection', which contains the data to render, each view has a 'user selection', which

holds the data interactively selected in that view. Views that share input selections show the same data. Views sharing user selections highlight user-picked data in different contexts. All components are synchronized by an Observer pattern on selections; when data is modified by user interaction or by analysis engines, all toolset components update automatically.

4.3 TOOLSET HIGHLIGHTS: SOLIDSX AND SOLIDSDD

We next present our toolset installation (Sec 4.3.1) and describe two tools of the toolset: the SolidSX structure analyzer (Sec. 4.3.2) and the SolidSDD clone analyzer (Sec. 4.3.3).

4.3.1 *Toolset Installation and First Usage Steps*

SolidSX and SolidSDD are provided as self-contained installers, freely available for research [160] on Windows XP and later editions. Installation takes a few minutes and mouse clicks, once an install location on the host system is provided. SolidSDD, which uses SolidSX internally (Sec. 4.3.3), installs SolidSX or uses a previously installed copy of SolidSX. No scripting, third-party package installation, environment setting, or compilation are needed. Manuals and sample datasets are also installed.

After installation, starting to use both tools is simple. To explore software structure and metrics with SolidSX, one only needs to load *.bsc* symbol files (for Visual C++), the root of a code base (for Java code), or any .NET assemblies (for C# or VB). An example tutorial of using SolidSX is provided separately [175]. To explore code clones with SolidSDD, one needs to load the root of a source code base (C, C++, Java, or C#) and click the clone detection button. For example, the scenario in Sec. 4.3.3 can be replicated as follows: (1) install SolidSDD; (2) download the source code to analyze [209]; (3) point SolidSDD to the root of the downloaded code; (4) start the clone detection; (5) visualize the detected clones.

4.3.2 *SolidSX: Structural Analysis*

The SolidSX (Software eXplorer) tool supports the analysis of software structure, dependencies, and metrics. Several built-in parsers are provided: Recoder for Java source and bytecode [115], Reflector for .NET assemblies [142], and Microsoft's free parser for Visual C++ *.bsc* symbol files. Built-in filters refine parsed

data into a compound attributed graph with folder-file-class-method and namespace-class-method hierarchies; calls, symbol usage, inheritance, and package or header inclusion dependencies; and basic metrics, *e.g.* code and comment size, complexity, fan-in, fan-out, and symbol source code location.

4.3.2.1 *Static Analysis: Ease of Use Considerations*

For .NET/VB/C#, Java, and Visual C++, static analysis is completely automated. The user only needs to input a root directory for code and, for Java, optional class paths. For Java, Recoder, a relatively less known analyzer [115], is close to ideal, as it delivers heavyweight information at 100 KLOC/second on a typical PC computer. For .NET, the Reflector lightweight analyzer is fast, robust, and simple to use. The same holds for Microsoft's *.bsc* symbol file parser.

C/C++ static analysis beyond Visual Studio is much harder. Setting up C/C++ analysis without a tight analyzer integration with a build system is complex and time consuming. Language dialect, files to analyze, include and library paths, facts to export, and handling of analysis errors must be explicitly specified if there is no build system, *e.g.* project file or makefile, to take these from. An example hereof is the integration of SolidSX with our separate SolidFX C/C++ static analyzer [181]. SolidFX scales to millions of lines of code, covers several dialects (*e.g.* gcc, C89/99, ANSI C++), handles incorrect and incomplete code, has a preprocessor, and integrates with the `gcc` and Visual C++ build systems via compiler wrapping [61]. Still, certain options such as platform defines and headers cannot be inferred from build systems and must be manually specified. Also, compiler wrapping requires a working build system on the target machine, which is not always the case. Other heavyweight C/C++ analyzers *e.g.* Columbus [61] or Clang [111] have the same issues. We also considered using lightweight C/C++ analyzers, *e.g.* CPPX [109], gccxml, and MC++. We found that these deliver massively incorrect information, due to simplified preprocessing and name lookup implementations. Finally, we considered using the built-in C/C++ parsers of Eclipse CDT, KDevelop, QtCreator, and Cscope [16]. While better in correctness, such parsers depend in complex ways on their host IDEs and do not have well-documented APIs, so cannot be embedded into third-party tools. Extended discussions with Roberto Raggi, the creator of KDevelop and QtCreator, confirmed this point.

4.3.2.2 *Structure Visualization*

SolidSX offers four views (Fig. 4.3): tree browsers, table lenses of node and edge metrics, treemaps, and HEB layouts [78]. All views have carefully designed *presets* which allow using them with no extra customization. All views show selections from the fact database created by static analysis (Sec. 4.2.2). User selections, created interactively or by queries, effectively 'link' facts shown in different views, which enables one to easily correlate structure, dependencies, and metrics along different viewpoints.

Figure 4.3 illustrates this on a C# system of around 45 KLOC (a graphical number puzzle program, whose source code is available in the tool distribution). The radial HEB view shows function calls over system structure, with caller edge ends blue and callee edge ends gray. Node colors show McCabe's complexity on a green-to-red colormap. We can now correlate complexity with system structure: We see that the most complex functions (warm colors) are in the module and classes top-left in the HEB view. The table lens shows several function-level code metrics, sorted on decreasing complexity. This shows how different metrics correlate with each other. Linked HEB-table lens selections support queries such as "what are the metrics of this module?" or "where are the most complex functions located?" The treemap view shows a flattened system hierarchy (modules and functions only), with functions ordered top-down and left-to-right in their parent modules on code size, and colored on complexity. The visible 'hot spot' shows that complexity correlates well with size. Constructing the entire scenario, including the static analysis, takes about 2 minutes and under 20 mouse clicks.

SolidSX brings several visual additions atop of Holten's HEB layout [78]. Luminance textures, added to nodes, enhance the hierarchical structure. When nodes are collapsed and/or expanded, the layout is smoothly animated between the initial and final views, which reduces the visual change and helps users maintain their mental map (for an illustration, see the actual tool in use). Multiple edges between collapsed nodes are replaced by a single edge. This edge's color shows the aggregated value (min, max, or average) of the collapsed edges' attributes, as indicated by user preferences. Adjacent nodes which are under a few pixels in width are replaced by gray, untextured bars. This indicates that the view cannot show the full dataset and also maintains a high frame-rate regardless of dataset size, since

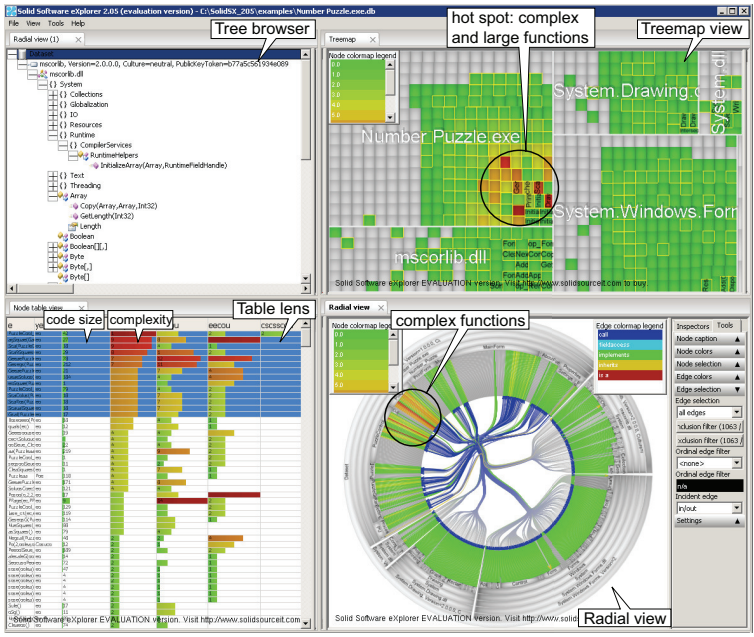

Figure 4.3: SolidSX views (tree browser, treemap, table lens, radial HEB).

the number of rendered nodes never exceeds the actual view size divided by the minimal node size. A similar technique is used for rendering cells in the table view [173]. Finally, attribute-based color mapping allows quickly locating elements of interest in a large visualization, such as highly complex elements.

4.3.2.3 *Toolchain Integration*

Similarly to Koschke [99], we noticed that *integration* in accepted toolchains is a key acceptance factor when using our tools in industrial applications [182, 205, 152, 181]. We address integration by a *listener* mechanism. The SVA tool listens for command events sent asynchronously as Windows system messages, *e.g.* load a dataset, zoom on some subset, change view parameters, and also sends user interaction events (*e.g.* user has selected some data). This allows integrating SolidSX in any third-party tool(chain) by building thin wrappers which read, write, and process such events. No changes to our tool's code are needed. For example, we integrated SolidSX in Visual Studio by writing a thin plug-in of around 200 LOC which translates between the IDE and SolidSX events. Selecting and browsing code in the two

tools is now in sync. The open SQLite database format further simplifies data integration.

4.3.3 *SolidSDD: Clone Inspection*

Code duplication (or clone) detection is an important tool in software maintenance. Although many clone detectors exist, few show the clone information in easy to understand ways. Simple clone lists do not show how clones are spread over a system's structure [87]. Node-link clone views inherit the limited scalability and visual clutter of force-directed graph layouts [85]. Adjacency matrices, showing clones between file pairs [88], are not immediately intuitive for software engineers, and do not show clone relations at several levels-of-detail, *e.g.* function, file, and folder.

To address these visualization issues, the SolidSDD (Software Duplication Detector) was developed. Clone detection uses the same algorithm as CCfinder [89], configurable by clone length (in statements), identifier renaming (allowed or not), size of gaps (inserted or deleted code fragments in a clone), and whitespace and comment filtering. From clones detected in an input C, C++, Java, or C# code base, we store a *compound duplication graph* in our fact database. Nodes are cloned code fragments. Edges indicate clone relations. Hierarchy is added either from the code directory structure or from a user-supplied dataset (*e.g.* from static analysis). Nodes and edges have metrics, *e.g.* percentage of cloned code, number of distinct clones, and whether a clone includes identifier renaming or not. Metrics are aggregated bottom-up using the hierarchy information (see Sec. 4.3.2.2).

We use the compound graph to visualize clones with two different views, as follows (see Fig. 4.4). Our test dataset is the well-known Visualization Toolkit code base [209]. On VTK version 5.4 (2420 C++ files, 668 C files, 2660 headers, 2.1 MLOC in total), SolidSDD found 946 clones in 280 seconds on a 3 GHz PC with 4 GB RAM, using the default tool settings. The first view (Fig. 4.4 a,b,e) is the SolidSX tool described in Sec. 4.3.2. Figure 4.4 a shows the code clones atop of the system structure. Edges show aggregated clone relations between files: two files are connected when they share at least one clone. Node colors show the percentage of cloned code in a subsystem on a green-to-red (0..100%) colormap. Edge colors show percentage of cloned code in the clones represented by an edge. Fig-

ure 4.4 a shows that the VTK system has many intra-system clones (edges connecting files in the same folder) but also some inter-system clones (edge connecting files in different folders).

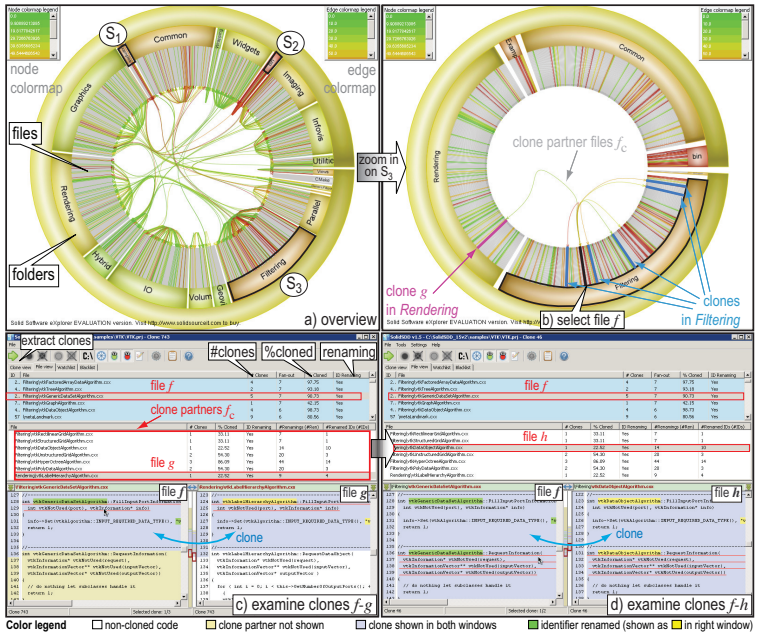

Figure 4.4: SolidSDD clone visualization using the HEB view (top) and text view (bottom) (see Sec. 4.3.3).

Three subsystems have high clone percentages (red tints in Fig. 4.4 a): *examples* (S_1), *bin* (S_2) and *Filtering* (S_3). Browsing these files we saw that clones in *examples* and *bin* are in tutorial code and test drivers, arguably created by copy-and-paste. Clones in *Filtering*, a core subsystem of VTK, are more interesting. In Fig. 4.4 b, we zoom on *Filtering* and select a file f (marked in black) which has over 50% cloned code: vtkGenericDataSetAlgorithm. This highlights the files f_c with which f shares clones, called the *clone partners* of f. We find five clone partners of f in the same *Filtering* subsystem (vtkStructuredGridAlgorithm, vtkDataObjectalgorithm, vtkUnstructuredGridAlgorithm, vtkHyperOctreeAlgorithm, vtkPolyDataAlgorithm, indicated by light blue in Fig. 4.4 b), and one in the *Rendering* subsystem (g, vtkLabelHierarchyAlgorithm, purple). Each such file contains a separate class, as standard in VTK. When writing these, developers probably copied and pasted code between sibling classes. Given VTK's guidelines to keep its subsystems in-

dependent *and* maximize code reuse, the clone g could be refactored by moving the common algorithm part to a superclass.

Figure 4.4 c shows the *text view* of SolidSDD. The top light-blue table shows all files with percentage of cloned code, number of clones, and presence of identifier renaming. Sorting this table allows *e.g.* finding files with the most clones or highest cloned code percentage. This view is linked with SolidSX via the message mechanism (Sec. 4.3.2.3), so selecting the file f in the HEB view (Fig. 4.4 c, black) highlights it in this table (in red). The table below (Fig. 4.4 c, middle panel) shows the clone partner files f_c of f. Here we find the file g which shares clones with f but is in another subsystem. We select g and use the two text views (Fig. 4.4 b, bottom panels) to study all clones between f and g. The left view shows the first selected file (f) and the right view the selected clone partner (g). Scrolling of these views is synchronized to easily compare corresponding code fragments. Text is color-coded: non-cloned code (white), code in f which is cloned in g (light blue), renamed identifier pairs (green in left view, yellow in right view), and clone code in f whose clones are in some other file $h \neq g$ (light brown). The last color allows us to navigate from f to other clone partners h: Clicking on light brown code in the left view (f) in Fig. 4.4 c replaces the file g in the right view by the clone partner h, and also selects h in the clone partner list view. Fig. 4.4 d shows this. We now notice that code in f which is part of the clone $f - g$ (light blue in Fig. 4.4 c) is included in the clone $f - h$ (light blue in Fig. 4.4 c).

4.4 TOOLSET APPLICATIONS

We have used our SVA toolset in both research [81, 178] and the industry [182, 180, 181]. We next describe several such use-cases and highlight strengths and limitations of our proposal.

4.4.1 *Toolset Usage in Education*

We used our SVA toolset in academic courses in two different contexts, as follows.

Tool end-user context: In the first case, 3rd year BSc students at the University of Groningen, the Netherlands, used SolidSX for the course Software Quality Assurance and Testing (SQAT) [175] (30 students per year on average during 2008-2011) to explore a 3500 LOC C++ image processing program [174, 171] in order to

design test cases using white-box testing and also assess the program's modularity, complexity, portability, and testability. Facts were provided by the Visual C++ .bsc parser (Sec. 4.3.2). No programming was required. After an 8-week course, students worked one month on the assignment. Additionally, the students could use the Visual Studio Team System (VSTS) tool to explore the code. Feedback gathered from mandatory course evaluations indicated that the students found the installation, learning, and usage of SolidSX very easy, although they had no previous experience in static analysis or software visualization. Table 4.2 shows tool-related points from this course evaluation, ranked on a 5-point scale (very limited, limited, neutral, good, very good), averaged for 87 students. The listed positive and/or negative observations aggregate comments given under a free-text heading concerning additional comments. The HEB view was found effective for assessing the modularity of a previously unknown code base, which replicates previous similar findings [78, 81]. On the negative side, they noted that SolidSX works on a too 'abstract' level, *i.e.* focuses on generic entities and relations, whereas white-box testing or code exploration tasks use more specific concepts *e.g.* implementation inheritance, dependency via type usage, or dataflow graphs.

Aspect	**Score**	**Observations**
Ease of installation	4.8	No installation problems; installation is easy
Quality of documentation	4.0	Manual needs more step-by-step examples
Scalability	4.2	Tool requires a modern computer with a fast (OpenGL accelerated) graphics card
Fact extraction	3.6	C/C++ support outside Visual Studio is limited
Visualization	4.4	Views have generic, hard to interpret, labels and annotations More customizable colormaps are required
Interaction	4.0	Selecting small elements can be hard in zoom-out mode
Effectiveness (for general comprehension)	3.9	Tool gives a good overview of a program's structure and dependencies It is a nice addition compared to classical IDE text-only views and queries
Effectiveness (for white-box testing)	2.2	Tool is too generic; needs customized wizards that should address specific questions
Usability (general)	4.0	Tool is easy to use and error-tolerant An undo function is however missing and would be very useful

Table 4.2: SolidSX tool evaluation in education.

Tool developer context: In the second case, 4^{th} year MSc students at the same university used SolidSX for the course Software Maintenance and Evolution (SME) (20 students per year on average during 2008-2011) to develop a visual exploration application of changing dependencies in a Subversion (SVN) repository. Hence, in contrast to the first use-case presented ear-

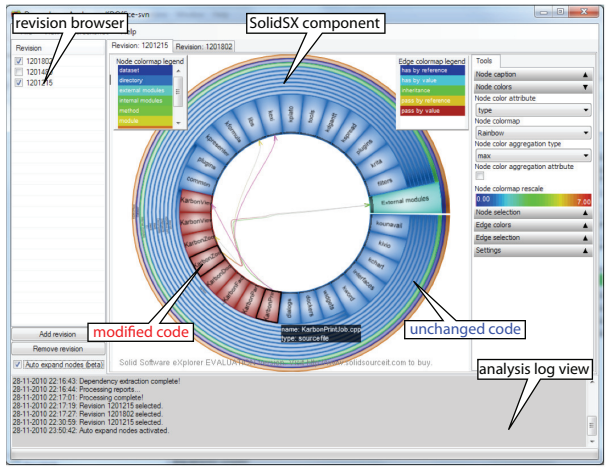

Figure 4.5: Subversion-repository dependency evolution browser tool interface (see Sec. 4.4.1).

lier in this section, where SolidSX was used out of the box by *end users*, in this second context, SolidSX was used as a component to *develop* a new SVA application. The repository chosen for exploration was KOffice, which has mainly C/C++ code (over 10000 files, 3500 versions, over a 8 year period) [97]. Repository access used the SharpSvn C# library [170]. From each revision, hierarchy and dependencies (inheritance, type usage, include relations, and function calls) were extracted with the lightweight CCCC analyzer [110], which uses a C/C++ parser built with the ANTLR parser generator. The resulting compound graph was stored in a SQLite database (Sec. 4.2.1). An exploration tool allowing the selection of the repository and revisions of interest, dependency extraction, and visualization of changing dependencies was built by the students using SolidSX as a starting point. One such tool resulting from a student project is shown in Fig. 4.5. Here, the HEB plot reuses the SolidSX tool; all other interface elements are custom-developed by the students for this specific project. Node colors indicate amount of code changed on a blue-to-red colormap. The student project along with manual, documentation, binaries, and C# source code is available for inspection [57]. The students were able to build atop SolidSX to develop their dependency evolution visualization without access to the tool's source code. The student tool was written in C#, while SolidSX itself is entirely written in C++ and Python. Key to this (re)use of SolidSX were the open SQLite data model

(Sec. 4.2.1) and the message-based mechanism that allows 'driving' SolidSX from any application (Sec. 4.3.2.3).

4.4.2 *Toolset Usage in Developing New Research*

Apart from education, we also used SolidSX in to develop new visualization research. One such example is Image-based edge bundles (IBEB), a technique for the simplified display of large compound graphs [178], presented next in Chapter 5. Briefly put, IBEB adds several image-processing techniques atop of the HEB view in order to explicitly group edges into salient shaded bundles. This shows a (software) system's structure on a coarse level of detail. Like in the educational context (Sec. 4.4.1), the SQLite data model and messaging interface were key to easy development. However, in this case we also extended SolidSX by adding new graphical elements to the radial plot (for details, see Ersoy and Telea [178]), rather than embedding it in a larger application. Figure 4.6 shows some results: The left image shows a set of software dependencies visualized with the standard SolidSX tool. The right view shows the same dataset, with dependencies grouped by IBEB into shaded bundles.

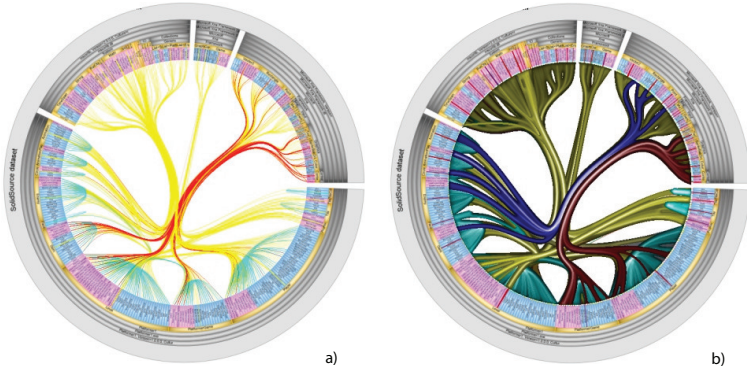

Figure 4.6: Left: HEB view of a compound software graph. Right: Simplified view of the same graph with the IBEB method built atop of SolidSX (Sec 4.4.2).

4.4.3 *Industrial Usage: Post-Mortem Assessment of a Software Project*

We now describe an application where several of our tools were combined in an industrial use-case. A major automotive company developed an embedded software stack of 3.5 MLOC of C

code in 15 releases over 6 years with three developer teams in Western Europe, Eastern Europe, and Asia. Towards the end, it was seen that the project could not be finished on schedule and that new features were hard to add. Management was not sure what went wrong. The main questions were: was the failure caused by bad architecture, coding, or management; and how to follow up - start from scratch or redesign the existing code. An external consultant team had *one week* to perform a post-mortem analysis using our toolset, and only the code repository as data source. For full details, we refer to Voinea and Telea [205].

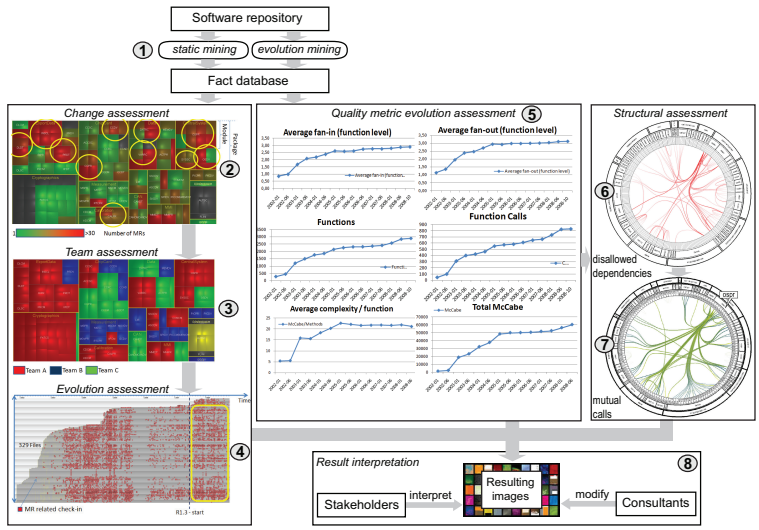

Figure 4.7: Data collection, hypothesis forming, and result analysis for a post-mortem software assessment (Sec. 4.4). Arrows show the order of the executed steps.

The approach involved the classical VA steps: data acquisition, hypothesis creation, refinement, (in)validation, and presentation (Fig. 4.7). Change requests (CRs), commit authors, static quality metrics, and call and dependency graphs were extracted from a CM/Synergy repository into our SQLite fact database. For fact extraction, we used our C/C++ analyzer SolidFX [181]. Similar heavyweight analyzers *e.g.* Clang or Columbus could be used as well. All further visual analyses were done using SolidSX and SolidSDD. Next, we examined the distribution of CRs over project structure. Several folders with many open CRs emerged (red treemap cells in Fig. 4.7 (2)). These correlate well with team structure: the 'red' team owns most CRs (3). To further see if this is a problem, we viewed the CR distribution over

files over time. In the table lens in Fig. 4.7 (4), files are shown as gray lines vertically stacked on age (oldest at bottom), and CRs are red dots. The gray area's shape shows little code size increase in the last project third (outlined in yellow), but many red dots over *all* files in this phase. These are CRs involving old files that were never closed. When seeing this image, the managers instantly recalled that the 'red' team (located in Asia) had lasting communication problems with the European teams, and added that it was a mistake to assign many CRs to this team.

We next analyzed the evolution of several quality metrics: fan-in, fan-out, number of functions and function calls, and average and total McCabe complexity. The graphs in Fig. 4.7 (5) show that these metrics increase little in the second project half. Hence, missed deadlines were not caused by code size or complexity explosion. Yet, the average complexity per function is high, which implies difficult testing. This was further confirmed by the project leader.

To find possible refactoring problems, we analyzed the project structure with SolidSX. Fig. 4.7 (6) shows forbidden dependencies, *i.e.* modules that interact bypassing interfaces. Fig. 4.7 (7) shows modules related by mutual calls, which violate the product's desired strict architectural layering. We decided to look at these specific structural problems after completing the project intake (the first day out of the one week total project duration). During this session, the two architects involved stated that their product should comply with strict interface-based communication and architectural layering. They were unsure if these desiderates were respected, and mentioned that problems in these areas would highly likely affect the ease of refactoring. The two views in Fig. 4.7 (6,7) suggest difficult step-by-step refactoring and also difficult unit testing. Again, these findings were in line with the impressions of the involved stakeholders.

Finally, to get more insight on the refactoring cost, we performed a code duplication analysis using SolidSDD. Finding duplicates can help in several ways. First, if a duplicated code block is refactored, then it makes sense to refactor all its duplicates too. Secondly, the code modifications caused by insight found during testing and debugging should be done consistently across duplicated code. We found little cross-file duplication in this code stack, which supports the case for relative low-cost refactoring (for full details, we refer to [205]).

The consultants in this project were familiar with the used tools (SolidSX and SolidFX), but did not (need to) have access to the tools' source code. Still, they succeeded in finding satisfac-

tory answers for the management questions in a few days and on a large code base. No modifications were done to SolidSX or SolidFX. Since SolidFX can export its dependency graphs directly into the shared SQLite database, tool communication was automatic. Besides tool installation, the only instrumentation effort required was for the extraction of evolution information (CRs, file change moments, and authors) from the CM/Synergy repository, for which we used the standard *ccm* client. This last step took approximately 4 hours of the entire project duration of one week. Post-study discussions outlined that important success factors were the easy installation of the tools, scalability, and the common (SQLite) database format that made data interchange between all involved tools very simple.

4.5 DISCUSSION

Based on our SVA tool building experience, we next address several questions of interest[1]. We use herein the concept of a tool *value model* [135]: A SVA tool is *useful* if it delivers high *value* with minimal *waste* to its stakeholders, which can be developers, testers, project managers, or consultants [184, 185]. Hence, the answers to the above-mentioned questions strongly depend on the users' views on value and waste, as follows.

4.5.1 *Should academic tools be of commercial quality?*

We see two main situations. If tools are used *purely* to test new algorithms or ideas (*e.g.* the IBEB visualization in Sec. 4.4.2), large investments in tool infrastructure are seen as waste. If tools are used in case studies (*e.g.* Sec. 4.4.1) or in real-life projects (*e.g.* Sec. 4.4.3), then usability is key to acceptance and success [99, 151, 180]. Hence, we believe that academic tools intended for other users than their own creators should not compromise on *critical* usability, *i.e.* interactivity, scalability, and robustness. However, effort critical for the latter *adoptability* phase, *e.g.* manuals, installers, how-to's, supporting many input/output formats, rich GUIs, can be limited.

An excellent overview of the path from a research tool to a commercial product is given by Bessey *et al.* [18] for the Coverity bug-finding tool. In our experience with SolidSX and SolidSDD, we noticed several points in line with Bessey *et al.*, as follows:

1 The questions are from the WASDeTT 2010 call for papers (www.info.fundp.ac.be/wasdett2010)

- industrial users tend to cluster into detractors and promoters, *e.g.* 'why would (visual) program comprehension help my job?' *vs* 'this is a great tool, no questions asked';
- early adoption in large organizations having code bases of several MLOC [182, 181] does cost significant promotion effort which cannot be easily bypassed;
- the path of least intrusion in a company's established practices *e.g.* via compiler wrapping (makefiles, build process, coding standards) is the most successful;
- countless variations of platforms, language dialects, and build systems pose hard problems to a (visual) tool acceptance;
- simple visualizations (albeit sometimes too simple) are easier accepted than subtler correlations of multiple variables which may be hard to understand.

Still, some differences exist between SVA tools and bug-checking tools such as Coverity. First, SVA is meant to provide *insight*, which for good or bad, is harder to quantify than a bug list. In other words, it is easier to measure the added value of introducing a bug-checking tool, *e.g.* as the ratio of the number of bugs found to the tool's ownership and usage costs, than to quantify how much insight a SVA tool has given. Secondly, we limited ourselves, on purpose, to support code bases where analysis is easily done with minimal configuration effort (see Sec. 4.3.2.1). This makes our proposal less generic than Coverity's but also reduces tool set-up costs. Thirdly, SVA tools are mainly aimed at individual developers and/or consultants. Hence, issues such as standard compliance with a company's policy, deployment effort, and pricing are less relevant. This further lowers the hurdles for acceptance of SVA tools.

4.5.2 *How to integrate and combine independently developed tools?*

This is a highly challenging question as both the integration degree required and the tool heterogeneity vary widely in practice. For SVA tools, such as the ones described in this chapter or the ones mentioned in Sec. 2.2.1, we have seen that the following patterns provide good returns on investment, in increasing order of difficulty/effort:

Dataflow: Tools communicate by reading and writing data files in standardized formats, *e.g.* SQL (tables), XML and GXL (attributed graphs) [77], and FAMIX and XMI (architecture models) [189]. This allows creating dataflow-like tool pipelines, like the excellent Cpp2Xmi UML diagram extractor using Columbus and GraphViz [98] or the SQuAVisiT toolset [198].

Shared databases: Tools communicate by reading and writing a single shared fact database which stores code, metrics, and relations. Data is typically stored as a combination of text files (code), XML (lightweight structured data), and proprietary binary formats for large datasets *e.g.* ASGs or execution traces. This model is used by CDT (Eclipse), Intellisense (Visual Studio) , and our toolset. In contrast to dataflows, shared databases support the finer-grained data access needed for real-time data browsing (SolidSX, SolidSDD) or symbol queries (Eclipse, Visual Studio). If a shared messaging system is used along a shared database, like for all tools above, tools automatically synchronize their views upon data changes. This is essentially the model-view-controller pattern, where the database is the model, the visualization tools are the views, and the message listener is the controller.

Common API: Tools use a single API to access shared data and also to execute operations. Although a common API does not necessarily mean a shared tool code base, the former typically implies the latter. API examples for SVA tools are Eclipse, Visual Studio, CodeCrawler [105] and Moose [128] and, at a lower level, the Prefuse and Mondrian Infovis toolkits [136, 108]. Common APIs allow a finer grained action composition than dataflow and shared databases. However, APIs are more restrictive to use in practice, as they add non-negligible API learning costs (for tool builders) and maintenance costs (for API providers).

A thorough discussion of interoperability in the context of reverse engineering tools is provided by Kienle ([93], Sec. 3.2.2). Although our context is somewhat different (SVA tools), most, if not all, the requirements and solutions surveyed by Kienle are very similar to ours, see *e.g.* dataflow *vs* ToolBus [41], and shared databases and common APIs *vs* the CORUM model [92]. In contrast to some reverse engineering toolsets, however, we chose not to explicitly store application or task-dependent schema models. This reflects our practical observation, in line with Kienle's analysis, that schema design (and reuse) is hard in prac-

tice; and our additional observation that visualization exploration scenarios often need to change viewpoints on the data dynamically, *i.e.* in the middle of an exploration, which makes predefined schemas less effective.

4.5.3 *What are the lessons learned and pitfalls in building tools?*

SVA tool building is a design activity. A critical success factor is creating visual and interaction models that optimally fit the users' mental map [202]. Within space limitations, we outline the following points:

2D vs 3D: Software engineers are used to 2D visualizations, so they will accept these much easier than 3D ones [187]. There is no single case, that we are aware of, where a 3D visualization was better accepted than a 2D one. As such, we abandoned earlier work in 3D visualizations [183] and focused on 2D visualizations only.

Interaction: Too much interaction and user interface options confuse even the most patient users. A good solution is to offer problem-specific minimalist interaction paths or wizards, and hide the complex options under an 'advanced' tab. This design is visible in the user interfaces of both SolidSX and SolidSDD.

Scalable integration of analysis and visualization is mandatory for SVA acceptance [99, 185]. However, achieving this is hard. Over 50% of our toolset code (which is over 700 KLOC in total) is dedicated to efficient data manipulation. SQLite performs well up to a few hundred thousands facts (Sec. 4.2). Heavyweight parsers, *e.g.* SolidFX, Clang, or Columbus create ASGs of millions of facts, roughly 10..15 facts per LOC. These are stored in a custom binary format which optimizes search speed to space ratio [181, 23]. The SQL database is used as a 'master' component which points to such special storage schemes. Using XML, although favored by several tool designs [71, 109], is simply not scalable enough for fine-grained fact databases or projects over a few hundred KLOC.

4.5.4 *What are effective techniques to improve the quality of academic tools?*

Quality of SVA tools depends on usability, which has different definitions depending on the tool's context. For example, research tools aimed at quickly testing new ideas should maximize API simplicity. Prefuse and Mondrian are good examples [136, 108]. In contrast, tools for software engineers in the field, or used in education, should maximize end-user effectiveness. This further implies uncluttered, scalable, and responsive displays, and tight integration for quick analysis-visualization sensemaking loops. Recent Infovis advances have massively improved the first points. However, integration remains hard, as it implies large engineering efforts which do not directly map to high-impact research results.

4.5.5 *What is needed to build an active community of developers and users?*

SVA tools rely upon techniques traditionally built in two separate communities: software analysis and information visualization. The two groups overlap via the software visualization community. The *OSS community* has invested considerable effort in SVA tool building, see *e.g.* the hundreds of plug-ins available for Eclipse, QtCreator, or KDevelop. However, most recent work in this area appears to target software analysis. Visualization is still centered around tables and node-link layouts. Although treemaps, table lenses, adjacency matrices, and HEB layouts have been proven to be more scalable and effective in the Infovis community, these techniques are still relatively unknown to OSS developers. The *commercial* community can serve as a strong catalyst: standardized open APIs of well-accepted tools, such as Visual Studio 2010's Intellisense semantic database, can give a widespread and quick impact to successful SVA tools. Unfortunately, many tool vendors provide limited tool-integration APIs, or even provide no such APIs, arguably to limit disclosure of internal architecture details [185].

4.5.6 *Are there any useful tool building patterns for software engineering tools?*

We see the following elements present in most SVA tools we have studied:

Architecture: The dataflow and shared database models are the widest used composition patterns.

Visualization: 2D dense-pixel views *e.g.* table lenses, HEBs, treemaps, and annotated text are highly scalable, thus suitable for large static analysis datasets. Node-link layouts work well for relatively small relational datasets of less than roughly a few hundred elements, in which position and shape encode specific meaning like in (UML) diagrams (see further Sec. 4.5.7). Shaded cushions, originally used for treemaps [155], are simple to implement, fast, scalable, and effective for conveying structure atop of complex layouts. Also, we did not notice so far requests from users of our toolset to include other types of visualizations *e.g.* node-link layouts, adjacency matrices, parallel coordinates, or 3D plots. This does not imply that such additional visualizations are not useful for specific use-cases, but it does support the hypothesis that general structure-and-metric comprehension of large code bases at a fine grained level is well supported by the techniques currently present in our toolset.

Integration: Combining separately developed analysis and visualization tools is still an open challenge. Although a message mechanism (Sec. 4.3.2.3) has limitations, *e.g.* cannot shared state, it is simple and allows keeping the software stacks of the tools to integrate independent.

Static analysis granularity: Lightweight analyzers are considerably simpler to build, deploy, and (re)use, are faster, and deliver sufficient details for visualization (Sec. 4.3.2.1). When visualization requirements increase, as it happens with a successful tool, so do the requirements on its input analyzer. Extending static analyzers is, however, not an incremental process: For providing more features, one typically needs switching to a completely new analyzer. Using a simple fact database schema helps this switching as the analyzer and visualization are weakly coupled. Adding new analyzers to our toolset, *e.g.* for additional languages, only requires setting up a small Python script (under 30 lines) which invokes the respective analyzer and populates the

standard SQLite database (Sec. 4.2.1) with the extracted facts. An example hereof is a Visual Basic parser, developed independently and written in Visual Basic itself, which we recently added to SolidSX [102].

4.5.7 *How to compare or benchmark such tools?*

SVA tools can be benchmarked by lab, class, or field user studies, or using them in actual IT projects. One can either compare several tools against each other [52, 152, 205] or test a tool *vs* a requirements set [94, 151]. *Technical* aspects *e.g.* speed, scalability, or analysis accuracy can be measured using *de facto* standard datasets from the ACM SOFTVIS, IEEE Vissoft, and IEEE MSR conference 'challenges', *e.g.* the Mozilla Firefox, Azureus, JUnit, or JHotDraw code bases. Measuring a tool's end-to-end *usefulness* is much harder as this is highly context specific. Still, side-by-side tool comparison can be used. For example, Figure 4.8 shows four SVA tools: Ispace, CodePro Analytix, SonarJ, and SolidSX. The first three are well-known in the Java community. We compared all tools for their effectiveness in supporting an industrial corrective maintenance task where participants were actual developers from the IT industry [153, 185]. Several of our design decisions, *e.g.* using dense-pixel layouts and bundled edges, are direct results of this study. Also, this study showed that integration, ease of installation, scalability, and the compact uncluttered dense-pixel layouts of SolidSX were perceived as important value factors by developers involved in program comprehension.

Other useful ways to gather qualitative feedback are 'piggybacking' the tool atop of an accepted toolchain (*e.g.* Eclipse or Visual Studio) and using community blogs for getting user sentiment and success (or failure) stories. From our experience, we noticed that this technique works for both academic and commercial tools.

4.5.8 *What particular languages and paradigms are suited to build tools?*

For SVA tools, our experience advocates a minimal set of proven technologies, as follows:

Graphics: OpenGL, possibly augmented with simple pixel shaders, is by far the most portable, easiest to code and deploy, and

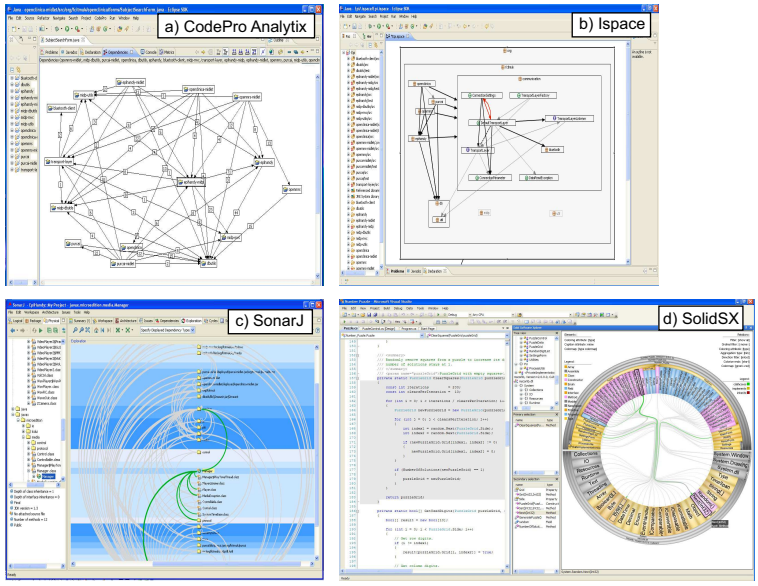

Figure 4.8: Four SVA tools for structure-and-association software analysis compared (see Sec. 4.5.7).

fastest, graphics solution. This experience is shared by other researchers, *e.g.* Holten [78].

Core: Scalability to millions of data items, relations, and attributes can only be achieved in programming languages like C, C++, C#, or Delphi. Over 80% of all our analysis and application code is C++. Although Java is a good candidate, its performance and memory footprint are, from our experience, still not on par with compiled languages. Recently, we reimplemented almost the entire core in C# (.NET 4), using the Windows Presentation Foundation (WPF), with promising results: Speed is 80% as compared to the original C++ implementation. Additionally, WPF offers easy ways to render visualizations to offscreen canvases, which makes embedding our results into *e.g.* web pages or PDF documents trivial. However, deployment becomes slightly more complex, as users have now to install the .NET 4 framework on their platform.

Scripting: Flexible configuration can be achieved in lightweight interpreted languages. The best candidate we found in terms of robustness, speed, portability, and ease of use was Python. Tcl/Tk (which we used earlier [183]) and Smalltalk (used by

[105]) are other candidates. From our experience, however, Smalltalk and Tcl/Tk require more effort for learning, deploying, and optimization. Lua [114] brings simplicity and flexibility for applications that do not require the richer features provided by Python's third-party libraries. For SVA tools, we distinguish two different uses for scripting. Generic SVA *frameworks* such as Rigi [190], Prefuse [136], Mondrian [108], or Moose [128] promote scripting to a first-class citizen: Users have to script to construct specific visualizations. This makes such frameworks very generic, but also less effective by end users not interested in, or not having the time for, programming. In contrast, coarser-grained tools, like ours, use scripting for (internal) configuration but expose all their functionality via built-in GUI options. This inherently limits the customizability of such tools, but makes them easier and faster to use for predefined tasks.

4.5.9 *Evolution from research prototype to product*

As already mentioned, our SVA toolset has evolved from an initial set of visualization research prototypes, starting with an early Rigi-like generic visualization system (SoftVision [183], see further Tab. 4.1), to an actual product toolset, via an iterative design process. We next discuss several aspects of this evolution.

Design decisions: During our tool design activities, several technical design refinements took place, as follows. First, using a simple, dynamically typed database with a minimal schema (Sec. 4.2.1) turned out to be simpler, and easier to maintain, than specialized object-oriented data models (see *e.g.* SoftVision [183] or SolidFX [181]). While the former model uses a fixed set of four data tables (see Sec. 4.2.1), the latter models need hierarchies of hundreds of classes, each being specialized for a different graph type, *e.g.* ASTs or call graphs. Secondly, using plain OpenGL for rendering is faster, both as development and runtime speed, than the more complex scene graph models of toolkits such as OpenInventor [208] used in SoftVision [183], CSV [112], MetricView [186], and SolidBA [182]. In scene graphs, the content to render is first stored explicitly, and next rendered. This works best for complex imagery that is often rendered but rarely changed. In SVA tools, views change often as the user interacts with the data, *e.g.* expand or col-

lapse nodes, so updating a scene graph is much slower than direct content rendering by OpenGL. Thirdly, our earlier tools used node-link layouts such as Sugiyama-style or spring embedders, based on the Graphviz and VCG engines [7, 148] (SoftVision [183], MetricView [186], and SolidBA [182]). However, no such engine can currently produce uncluttered layouts for graphs over a few hundred nodes. Hence, we restricted our later tools to HEB layouts, treemaps, and table lenses, which are always scalable and clutter-free. Fourthly, we experimented with data storage using binary files (SolidFX [181]) and XML files (MetricView [186], MemoView [121]). The SQLite model, used in CVSscan, CVSgrab, SolidBA, SolidSX, and SolidSDD proved to be much simpler to maintain and up to two orders of magnitude faster and more compact than XML. Finally, we kept unchanged those design elements which proved successful during our toolset evolution, such as selections and observers (introduced by SoftVision). Transition moments between these design decisions are explained further below.

Evolution phases: Our toolset evolution can be divided into four main phases:

1. *Inception:* In this phase, several SVA prototypes were created, with the main goal of exploring novel visualization and interaction technique (SoftVision, CSV, CVSscan). In this phase, a wide mix of visualization and interaction designs (*e.g.* node-link layouts, dense pixel displays, cushions, table lenses, linked views) and implementation technologies (*e.g.* C/C++ *vs* Python, OpenGL *vs* Open Inventor, SQLite *vs* plain file storage formats), were explored. The main drivers were the speed of creating new visualizations and the visualization scalability, with limited focus on reusability, genericity, or ease of deployment. The outcome of this phase was a selection of 'winner' visualization-and-interaction designs and implementation elements, *e.g.* OpenGL, cushions, table lenses, and C++, which were detected to be visually and computationally scalable. In this phase, testing was done on relatively small code bases (up to a few hundreds of entities and relationships and a few thousands of lines of code), and involved mainly researchers, students, and OSS code bases.

2. *Consolidation:* In this phase, additional research prototypes were created for different exploration tasks (CVSgrab, MetricView, and MemoView). The main focus of this phase

was to extend the options for SVA exploration to new types of datasets and tasks, *i.e.* entire repositories (CVSgrab), MetricView (UML diagrams and metrics) and program traces (MemoView). The design choices obtained as outcome from the previous phase were followed in this development. As they were confirmed to be effective in terms of desired scalability by the actual usage of the tools, these design choices were kept fixed until the later product refinement phase (see further below). Tool testing involved both large-scale OSS code bases (*e.g.* VTK and wxWidgets) and a few commercial code bases. For the first time, we involved IT professionals in using our tools, and collected informal feedback on usability and effectiveness, which further led to our focus on ease of deployment and configuration, and tool interoperability, addressed in the next phase. The decision to extend the tool scope to IT professionals was taken on an *ad hoc* basis, given our involvement in academy-industry joint projects. GUI design converged to using the wxWidgets toolkit, as opposed to several variants in the inception phase (FLTK, Tcl/Tk, and Windows MFC). The datasets used in this phase were significantly larger than for the inception phase, *e.g.* file evolution information from an entire repository (CVSgrab) as opposed to a single file evolution (CVSscan), and traces of hundreds of thousands of samples (MemoView). As such, efficient storage and retrieval proved critical. SQLite emerged as a natural candidate, given the usage of SQL fact databases in other SVA tools. For the specific cases which required node-link layouts (MetricView), Graphviz and VCG emerged as best candidates in terms of ease of use, genericity, and layout quality.

3. *Initial products:* In this phase, the interactive visual metaphors and implementation techniques tried and tested in the first two phases were used as basis to create the first versions of products (SolidFX, SolidBA, and SolidSTA). This phase focused on three main additions: First, new static and repository analysis components were added, thereby extending the visualization capabilities of the earlier tools with analysis functions (SolidFX extends CSV, SolidSTA extends CVSgrab and CVSscan). Secondly, a first unified fact database model, based on SQLite, was created (see Sec. 4.2.1), in order to make tool interoperability possible. Thirdly, various layers were created to facilitate

tool deployment, in terms of installers (NSIS-based [129]) and front-ends for automating static analysis. The technical outcome of this phase was a first version of the overall SVA toolset architecture presented in this paper.

4. *Refined products:* In this phase, a major refactoring of the visual functionality took place. Visualization components, so far scattered into different class libraries in the existing tools, were centralized in a single implementation, which led to SolidSX. The treemap and HEB components, pioneered by other research tools [155, 78], were also added from scratch. The message-based interface (Sec. 4.3.2.3) was developed, which allowed incorporating interactive visualization as a 'service' into analysis tools. Analysis-wise, we extended our initial scope on C/C++ with Java, C#, and .NET static analysis (SolidSX), and clone detection (SolidSDD). Deployment-wise, this phase added manuals and licensing mechanisms, which turned our toolset into a first version of true off-the-shelf end-user products. This enabled us also to conduct larger user evaluations with both IT professionals [153] and students (Sec. 4.4.1). Implementation-wise, we migrated from C++, OpenGL, and wxWidgets to .NET and WPF. This decision was taken due to perceived shorter development time in C#/.NET, experienced by our development team in the context of other ongoing projects, and the richer GUI options of .NET *vs* wxWidgets, including offscreen and web rendering, the latter which was required by several customers.

4.6 CONCLUSIONS

In this chapter, we have presented our experience in developing software visual analytics (SVA) tools, starting from research prototypes and ending with a commercial toolset. During this iterative design process, the presented toolset has converged from a wide variety of techniques to a relatively small set of proven concepts: a single shared fact database with a simple schema, implemented in SQL, which allows tool composition by means of shared fact selections; a small number of scalable Infovis techniques such as hierarchically bundled edge layouts, table lenses, annotated text, timelines, and dense pixel charts; control flow composition by means of lightweight message-based adapters as multiple linked-views in one or several independently devel-

oped tools; tool customization by means of Python scripts; and efficient core tool implementation using C/C++ and OpenGL.

We illustrated our toolset by means of two of its most recent members: SolidSX for visualization of program structure, dependencies, and metrics; and SolidSDD for visualization of code clones. We outlined the added value of combining several tools in typical visual analysis scenarios by means of simple examples, academic usage in research and education, and an industrial post-mortem software assessment case. Finally, from the experience gained in this development process, we addressed several questions relevant to the wider audience of academic tool builders.

Future work in the direction of improving our toolset for the general task of program comprehension in the IT industry covers extending our toolset at several levels *e.g.* lightweight zero-configuration C/C++ parsing; dynamic analysis for code coverage and execution metrics; and integration with IDEs beyond Visual Studio, *e.g.* Eclipse and KDevelop. Combining HEB layouts and annotated code text in a single scalable view to allow easy navigation from source code to structure is a second promising direction of work.

From the perspective of this thesis, two points are important to be outlined. First, HEB layouts, by themselves, do not form a solution. To become one, they have to be complemented by additional mechanisms for data storage, data filtering, fact extraction, automatic configuration, and additional views. Developing these mechanisms requires significant effort: Looking at our SVA toolset, the HEB implementation accounts for roughly 10% of the source code. This is important to remember for studies which wish to evaluate the effectiveness of HEBs in industrial program comprehension. Secondly, we have however seen that HEBs form a key ingredient to the success of our SVA toolset. As such, our conclusion is that HEB methods *can* be an effective program comprehension instrument in the real world, as long as they are seamlessly integrated in a production-grade toolset.

This chapter is based on:

Dennie Reniers, Lucian Voinea, Ozan Ersoy, and Alexandru Telea. The Solid* Toolset for Software Visual Analytics of Program Structure and Metrics Comprehension: From Research Prototype to Product. *Science of Computer Programming*, Elsevier (2012).

5 IMAGE-BASED EDGE BUNDLES

ABSTRACT: *In the previous chapters, we have shown that graph bundling is a potentially useful technique for understanding large graphs in program comprehension ranging from academic to industrial contexts. However, we also have shown that graph bundling techniques tend to generate complex visual structures when applied to very large graphs. In this chapter, we present a new approach aimed at simplifying the visual structure of bundle visualizations. For this, we combine an aggregation, or clustering, of the graph data with an image-based technique that renders clustered edges as simple compact shaded shapes, at a user-selected level of detail. We show how our shapes can be generated efficiently and automatically from a given bundled layout using a combination of image processing techniques such as distance transforms, splatting, and skeletonization. Next, we show how luminance, saturation, hue, and shading can be set to encode edge density, edge types, and edge similarity. Finally, we add brushing and a new type of semantic lens to help navigation where local structures overlap. We illustrate the proposed method on several real-world graph datasets with a focus on program comprehension.*

5.1 INTRODUCTION

As discussed in Chapters 3 and 4, hierarchically bundled edge (HEB) layouts are an effective instrument for visualizing large compound graphs for program comprehension, both in academic research and industrial contexts.

Separately, as we have outlined in Chapter 2, node-link layouts can produce significant visual clutter, which shows up as overlapping edges or nodes. Clutter impairs tasks such as finding the nodes that a given edge (or edge set) connect, and at a higher level, understanding the coarse-scale graph structure. As we have seen in Chapter 3, such clutter is also present in EBL visualizations, albeit in a different form: While, for node-link layouts, we cannot distinguish the coarse-level structure of the graph, in EBL layouts it is hard to disambiguate between several bundles which overlap at a given spatial position.

Several approaches exist to reduce clutter in graph visualizations. First, the graph can be simplified prior to visualization, *e.g.* by extracting structures such as spanning trees or strongly connected components. Secondly, the layout of nodes and/or edges can be adjusted. Both methods can be applied globally,

based on clutter estimation metrics, or locally, based *e.g.* on user interaction [219, 216].

When node positions encode information, they should not be changed. Also, clutter is related most often to edge crossings [137, 76]. Recent research targets clutter reduction and structure emphasis by geometrically grouping, or bundling, edges that follow close paths. Edge-bundling layouts (EBLs) exist for general graphs [39, 80, 133], circular layouts [68], hierarchical digraphs [78], and parallel coordinates [117, 225].

In this chapter, we approach the goal of visualizing the coarse-scale structure of an EBL and clarifying edge clutter caused by bundle overlaps. Given a bundling layout, which we do not change, we hierarchically cluster edges seen as similar from the viewpoint of the layout and, optionally, underlying attribute data. Next, we construct simple shapes that encode both geometric attributes of clusters (form, position, topology) and underlying edge data (spatial density and attributes). We render these shapes with an image-based technique that maps their attributes to shading and color on one or more scales. While keeping EBL advantages, our simplified visualization clarifies coarse-scale bundle overlaps by explicitly drawing each bundle as a separate shape, and assists the task of finding nodes connected by a bundle. The simplification level is user controlled. Finally, we add interaction to further clarify overlaps in desired areas and to offer details on demand.

This chapter is structured as follows. Section 5.2 introduces our EBL simplification technique. Section 5.3 presents several results focusing mainly structure-and-dependency graphs from software domain. Section 5.4 discusses our proposal. Section 5.5 concludes the chapter with future work directions.

5.2 METHOD

We aim to simplify a bundled edge visualization by emphasizing the coarse-level bundle structure to help users to visually trace such bundles to the nodes they connect. For this, we make bundles a first-class visualization object using splatting and shaded cushions, hence the name of our method: Image-Based Edge Bundles (IBEB). We use a six-step approach, as follows (see also Fig. 5.1).

1. We apply a given edge bundling layout (Sec. 5.2.1).

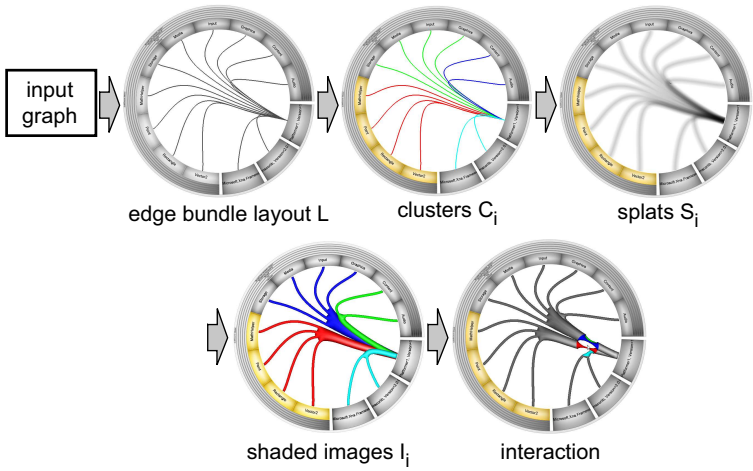

Figure 5.1: Image-based edge bundle (IBEB) visualization pipeline

2. We explicitly group laid out edges into a cluster hierarchy, using a distance that reflects edge positions and data attributes (Sec. 5.2.2).

3. We choose a set of clusters from the hierarchy at a user-selected level of detail. For each cluster, we create a compact shape around its edges (Sec. 5.2.3).

4. For each shape, we construct a cushion-like shading profile that also encodes data attributes (Sec. 5.2.4).

5. We render all shapes in a suitable order to minimize occlusion (Sec. 5.2.5).

6. We add a new semantic lens method to help visual exploration (Sec. 5.2.7).

These steps are detailed next.

5.2.1 *Layout*

We start with an edge bundling layout $L : G \rightarrow \mathbb{R}^2$ for the input graph $G(V, E)$. The next steps (Sec. 5.2.2 and further) are fully independent on this layout. The only assumptions made are that

1. each edge $e_i \in E$ is mapped to a set of points $p_{ij} \in \mathbb{R}^2$; different edges can have different amounts of points;

2. the layout does create edge bundles;

As an example, we use the HEB layout [78]. Yet, we use absolutely no hierarchical information beyond the layout. Other bundling layouts can be readily used (Sec. 5.3).

5.2.2 *Clustering*

As a pre-processing step to produce our simplified visualization, we explicitly group related edges. Each edge $e = \{p_j\}_{j=1}^{|e|}$ is modeled as a feature vector $v = \{x_1, y_1, \ldots, x_N, y_N, t_1, \ldots, t_T\} \in \mathbb{R}^{2N+T}$. The first 2N elements of v are regularly sampled points along the polyline $\{p_j\}$. N should be large enough to capture complex edge shapes. $N \in [50, 100]$ gives good results on different EBLs and datasets, in line with [80, 78, 68]. Some layouts do not encode semantic edge similarity into positions (assumption 2, Sec. 5.2.1): The HEB groups edges solely on their ends' hierarchy position; the FDEB uses solely edge points' positions. In some cases, *e.g.* visualizing a software system graph, we want to distinguish edge types (*e.g.* inheritance, call, uses) [81, 205]. To separate edges of different types $t \in \mathbb{N}$, we add $v_{2N+1} = t$. Multiple type dimensions can be encoded in $t_1, \ldots, t_T$, although in our experiments so far we have used a single type component ($T = 1$).

Next, we cluster all edges e_i with a well-known clustering framework for gene data [40]. Intuitively, we replace genes by our vectors v. We have tested several algorithms: Hierarchical bottom-up agglomerative (HBA) using full, centroid, single, and average linkage; and k-means clustering, both with Euclidean and statistical correlation (Pearson, Spearman's rank, Kendall's τ) distances. HBA with average or full linkage and Euclidean distance $d(v, w) = \sum_{i=1}^{N+T} \|v_i - w_i\|^2$ give the best results, *i.e.* clusters with edges being close both geometrically and typewise. To keep edges of different types separated, we bias $t_j \in v$ with a large value $k = \max_{e,e' \in E} \sum_{i=1}^{N} d(e, e')$. Similar techniques are used to handle gene components with different semantics, which also allows users to set weights to the different feature vector components [40]. However, mixing positions and types in one distance metric could in some cases lead to undesired results, *e.g.* having one kind of data dominate the other, depending on the values of N, T, and value ranges of position and type attributes. If we want to allow that only edges of the same type get clustered together, we define $d(v, w) =$

$\sum_{i=1}^{N} \|v_i - w_i\|^2$ if $v_j = w_j, \forall j \in [N+1, N+T]$, else $d(v,w) = k$. Implementing this in [40] is straightforward.

HBA delivers a dendrogram $T = \{C\}$ with the edge set E as leaves and distances $d(C)$ decreasing from root to leaves. We now select a partition $P = \{C_i\}$ of E so that $\bigcap_{C_i,C_j \in P} = \oslash$ and $\bigcup_{C_i \in P} = E$. For example, a similarity-based P contains all clusters with a $d(C) < d_{user}$ below a user-given value. Larger d_{user} values give more numerous, and more similar, clusters. Smaller d_{user} values give less, more dissimilar, clusters. Other methods can be used, *e.g.* select P for a given cluster count.

We stress that the clustering method choice is not the core of this chapter, but only a tool to create explicit edge groups. Any clustering can be used, as long as it groups edges logically related from an application viewpoint *and* spatially close. For example, the ink-minimizing clustering in [68] is a good option if the aim is to generate tight bundles which never cross and use a circular layout. The hierarchical clustering in [32], although proposed for tensor fibers, may also deliver good results. Also, it is very important to note that our partition is just a single level, or 'cut', in the graph, which we subsequently visualize.

5.2.3 *Shape construction*

Given a user-selected partition P (Sec. 5.2.2), we now construct a shape to visualize each edge set $C = \{e_i\} \in P$. Due to bundling *and* clustering, e_i typically follow a small set of directions (paths).

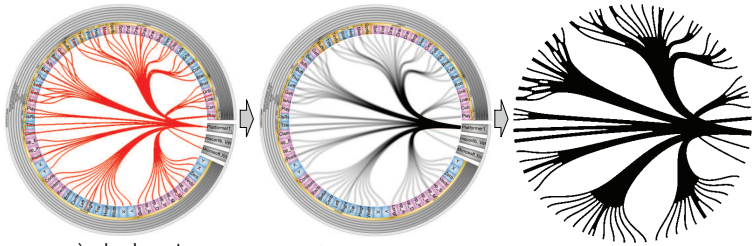

Figure 5.2: Shape construction. Edge bundles (a) are splatted into a density image (b), next thresholded into a binary shape (c).

We use splatting to show bundles in a compact way (Fig. 5.3). We convolve each edge $e \in C$ with a kernel k which linearly decreases from a maximum K to zero at a distance δ from the edge, and accumulate results, similar to [199]. For this, we sample k in

a 64x64 pixels alpha texture and additively blend textured polygons along all $p_i \in e \in C$ (`GL_SRC_ALPHA`, `GL_ONE`). We tried both radial and linear profiles for k (Fig. 5.3 bottom-right). Radial profiles are splatted centered at p_i. Linear profiles are splatted on two polygon strips built by offsetting edge segmenst $p_i p_{i+1}$ in vertex normal directions $\mathbf{n}_i$,$-\mathbf{n}_i$ with δ, like stream ribbons in flow visualization. Linear profiles are better: they allow freely choosing the edge resolution (number of p_i) and splat size δ, while these values must be carefully tuned for radial profiles to avoid splatting gaps.

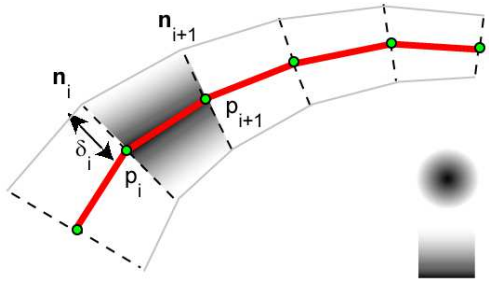

Figure 5.3: Splatting algorithm details

Splatting yields an edge density $D(x) = \sum_{p \in e, e \in C} k(p - x)$ (Fig. 5.2 b). Next, we threshold D to obtain a binary shape I (Fig. 5.2 c)

$$I(x) = \begin{cases} 1, & D(x) \geqslant \tau \\ 0, & D(x) < \tau \end{cases} \tag{5.1}$$

For illustration simplicity, Fig. 5.2 shows a single cluster (the tree root). In practice, we create one shape I_i for each cluster C_i in the user-selected partition P. Each I_i is, by construction, compact, and surrounds the edge bundle(s) in C_i, with a maximal offset $\delta(K - \tau)/K$. In practice, we always set $\tau = 0.7K$ and $K = 0.2$. The quantity δ is user-controlled, ranging between 1% and 5% of the viewport (see Sec. 5.2.4).

Additionally, we modulate δ to thin shapes half-way between their ends. For this, we use, at each point $p_i, i \in [1, N]$, a value $\delta_i = \delta\left(\epsilon\left|\frac{i-N/2}{N/2}\right| + 1 - \epsilon\right)$, *i.e.* shrink shapes from δ at their ends to $(1-\epsilon)\delta$ in the middle. Good values for ϵ range around 0.5, which was used for the examples in this paper. Shrinking reduces bundle overlaps, as we shall see next in Sec. 5.2.5.

5.2.4 *Shading*

For each binary image I created from clustered edge bundles, we now create a shaded shape that compactly conveys the underlying bundle structure. Following the original bundle metaphor, we want to encode several aspects in a shape:

- *bundling:* The shape should suggest the branching structure of a set of bundled curves in a simplified way;
- *structure:* Finer-level groups of edges, or even individual edges, should be visible;
- *density:* High edge-density regions should be visible. These are cues for strong couplings, relevant to many applications;
- *data:* The shape should be able to encode bundle attributes, *e.g.* edge types.

For this, we generalize rectangular shaded cushions [200] to our more complex shapes I, as follows. We compute the skeleton $Sk(I)$ of each shape I. $Sk(I)$ is a 1D structure locally centered with respect to the shape's boundary ∂I

$$Sk(I) = \{x \in I | \exists p \in \partial I, q \in \partial I, p \neq q, \|x - p\| = \|x - q\|\}$$

Next, we compute a shading profile

$$H = \frac{1}{2}\left[\min\left(\frac{DT(\partial I)}{DT(Sk)}, 1\right) + \max\left(1 - \frac{DT(Sk)}{DT(\partial I)}, 0\right)\right] \quad (5.2)$$

where $DT(\partial I)$ and $DT(Sk)$ are the distance transforms of the boundary ∂I and skeleton Sk respectively. We compute both DT and Sk using the implementation described in [179]. For any shape topology or geometry, H smoothly varies between 0 on ∂I and 1 on $Sk(I)$, as shown for a different application in [146]. Figure 5.4 b,c show Sk and H (the latter on a blue-to-red colormap) of the shape given by splatting Fig. 5.4 a.

We now set the hue, saturation, value, and transparency h, s, v, a at each pixel of I using the profile H, splatting density D (Sec. 5.2.3), and edge types, following the aims listed earlier in this section. We set $v = H^{\alpha}$, with $\alpha = 0.5$. This darkens shapes close to their border and brightens them close to the skeleton. The factor 0.5 smooths out H (Eqn. 5.2), creating a look akin to classical shaded cushions [200]. Next, we map edge types to hue

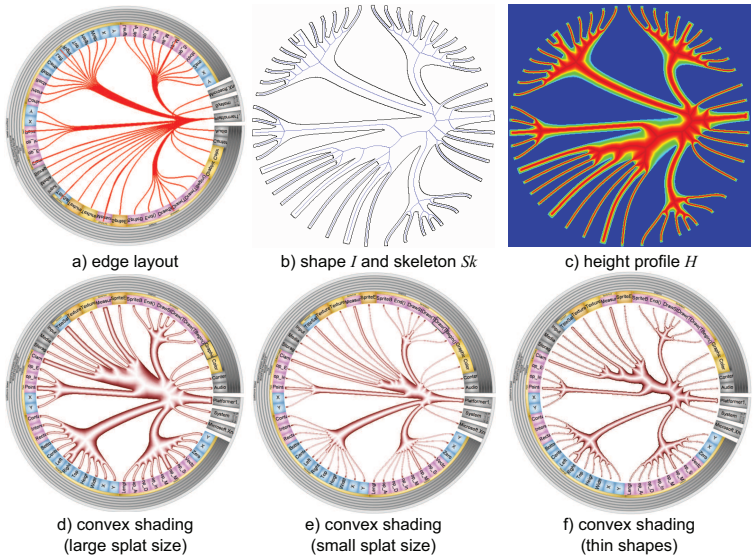

Figure 5.4: Shading pipeline (Sec. 5.2.4). Edges in a cluster (a) and their binary shape I and skeleton Sk (b) and shading profile H (c). Convex shading with shape thickness as function of the splat size (d,e) and shading profile thresholding (f).

h. Two options were explored: each shape has a different hue, or hues map edge types. The second option is relevant when clusters contain only same-type edges (Sec. 5.2.2). Finally, we use s and a to create different visual styles (Table 5.1).

The *convex* style renders opaque shapes dark and saturated at the border and bright and white in the middle (Fig. 5.4 d). In contrast to Phong shading H as a true height signal, as in [200, 25], this style emphasizes the skeletal structure (branching pattern). We see now the effect of the splat size δ (Sec. 5.2.3). Higher

Style	s	a
Convex	1-H	1
Density-luminance	$1 - HD$	1
Density-saturation	HD	1
Cores	H	$1 - H^3$
Outline	0	$1 - HD$

Table 5.1: Shape shading styles (see Secs. 5.2.4,5.2.5)

values yield thicker, simpler, shapes (Fig. 5.4 d). Smaller values yield thinner shapes with individual edges better visible (Fig. 5.4 e). We can further emphasize a bundle's branching structure by using $\max[0, (H - H_{min})/(1 - H_{min})]$ instead of H in Table 5.1. H's isolines continuously change from the shape's boundary to its skeleton, being halfway at $H = 0.5$ (see Fig. 5.4 c and [146]). $H_{min} = 0.5$ yields shapes which are thinner and also further emphasize the bundle structure, as in Fig. 5.4 f.
The last four shading styles in Table 5.1 are effective when visualizing several clusters, as discussed next.

5.2.5 *Rendering*

For a given clustering partition P, we now render one shape I for each cluster in back to front order, *i.e.* sorted on shape size (foreground pixel count $|I|$). Placing small shapes in front of larger ones reduces occlusions and makes small bundles visible.

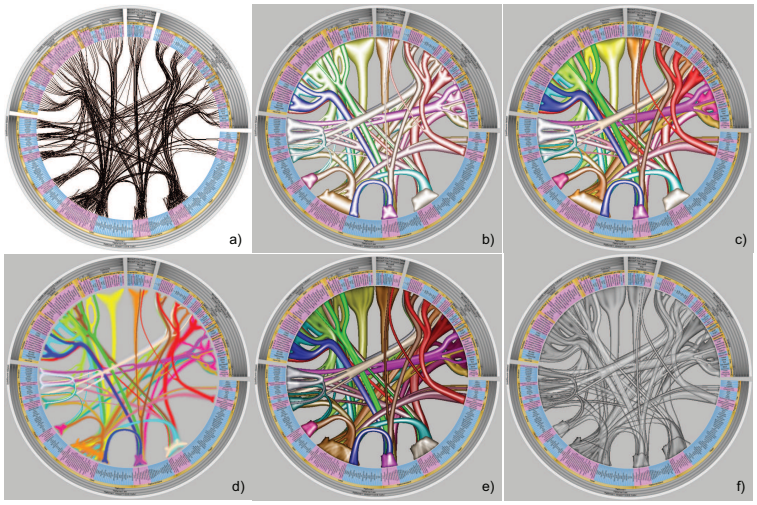

Figure 5.5: Rendering styles: convex shapes (b), density-luminance (c), density-saturation (d), bi-level (e), and outlines (f).

Fig. 5.5 illustrates this. Image (a) shows a dependency graph of 419 nodes and 988 relations extracted from a C# software system, laid out with the HEB. Nodes are .NET assemblies, packages, classes, and methods. Several bundles show up, but it is hard to determine (even with interaction) which subsystems they connect. Overlaps make it hard to visually follow a bundle end-to-end. Image (b) shows the result of our method, on a level-

of-detail with 18 clusters, using the convex style (Sec. 5.2.4). For illustration only, clusters were given different random hues from a hand-crafted colormap. Using a gray rather than white background emphasizes the coarse-scale bundles. Image (c) presents the density-luminance style (Table 5.1). Brightness emphasizes clusters with many edges. Fig. 5.5 d serves the same goal, but uses saturation: High-density areas are colorful, low-density areas are gray. Image (d) shows the *cores* style. Areas close to bundle skeletons are opaque, the rest is transparent. This reduces overlaps and stresses graph structural aspects, similar in aims to the opacity bands in clustered parallel coordinates [66].

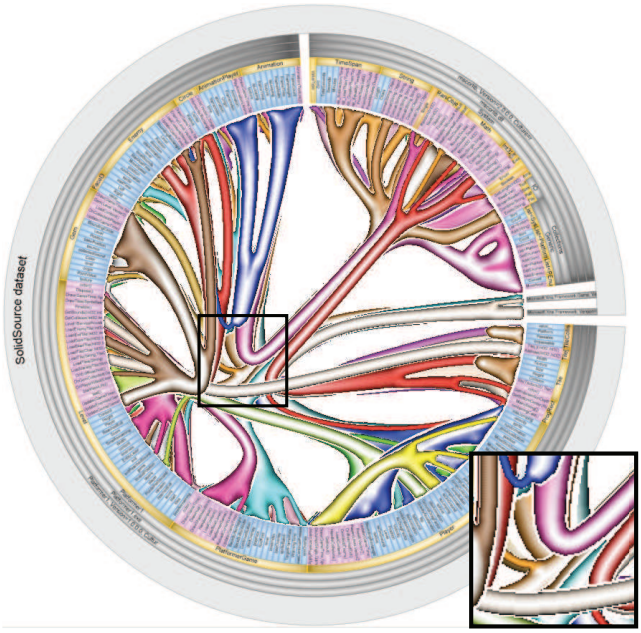

Figure 5.6: Bundle visual separation using halos

To better visually separate overlapping bundles, we can use a halo effect conceptually similar to the technique presented in [58] for tensor fibers. For every bundle shape I, we create a white, opaque border of fixed size σ (a few pixels) around I. Doing this is simple: all pixels x in the halo band are characterized by $DT(\partial I) \leqslant \sigma$. Since DT is computed in pixel space, the halos will be the same width σ, in pixels, regardless of the bundles' widths. If we desire halos of width proportional to the bundles thicknesses, we can use H instead of DT. An advantage of using H is that halos are guaranteed to never 'erase' very thin bundles completely. Figure 5.6 shows H-based halos, with a zoomed-in

detail in the inset. Halos are most effective for images showing a limited number of bundles. Denser images, *e.g.* Fig. 5.5, benefit less from halos as these always take a certain amount of screen space.

Figure 5.5 f shows the *outline* style. Here, we modulate alpha, to create transparent outlined tubes (Table 5.1). To reduce clutter caused by transparency, we use grayscale images. Although less salient than the previous styles, outlines are a useful visual cue of overall structure, especially when combined with interaction techniques.

Finally, we explored the possibility to add more visual detail to a bundle. For a user-chosen level d_{min} and partition $P = \{C\}$, we first compute H as in Sec. 5.2.4. Next, we re-partition C (Sec. 5.2.2) for a higher $d'_{min} = \mu d_{min}$, where $\mu = 1.2$ gives good results. Third, we add the profiles H' of each C' in its refined partition P'_i, scaled to a lower range $[0, h]$, to the coarse-scale H_i. We normalize the result $H + \sum_{C' \in P'} hH'$ and use it for shading (Sec. 5.2.4). Finer-scale bundles create luminance ridges within their parent clusters. From discussions with the users, we noted that bi-level images are perceived as more suggestive than single-level ones, as the second level acts as a detail texture suggesting the bundled edges, and also eliminate the undesired luminance peaks created by skeleton branches reaching to the corners of the bundle shapes (compare Figs. 5.5 (c) and (e)). However, our thin and long shapes preclude adding more levels to actually show bundle hierarchies like *e.g.* in cushion treemaps.

5.2.6 *Directional bundles*

We can modify IBEB to also generate textures which show direction, as follows (see also Fig. 5.7).

1. Given a bundle shape I, computed by distance thresholding as shown in Sec. 5.2.3, we define an arc-length parameterization $u : \partial I \to [0, 1]$ of its boundary I. Such a parameterization is actually already computed by the distance-transform and skeletonization technique we use in our method [179, 167];

2. We define a periodic sawtooth-like function $s : [0, 1] \to [0, 1]$ as

$$u(x \in [0, 1]) = x \bmod L \tag{5.3}$$

where L is the period of the sawtooth signal, or length of one of its pulses. Good values for L are in the range of $0.02..0.05$, *i.e.* a small fraction of the length of the boundary ∂I;

3. Given any point $\mathbf{x} \in I$, we now compute the luminance at x as

$$H'(\mathbf{x}) = H(\mathbf{x})s(H(\mathbf{x}) + \lambda u(FT(\mathbf{x}))), \tag{5.4}$$

where $FT(\mathbf{x})$ is the so-called *feature transform* of ∂I evaluated at point $\mathbf{x}$ defined as

$$FT(\mathbf{x}) = \underset{\mathbf{y}\in\partial I}{\operatorname{argmin}} \|\mathbf{x}-\mathbf{y}\|, \tag{5.5}$$

and H is the original cushion-like shading pattern defined by Eqn. 5.2;

4. Finally, we use the shading signal H' instead of our original signal H in rendering the bundles, as described earlier in this chapter.

Let us explain the results. Our shading profile H' equals the sawtooth (dark-bright-dark) signal u along the shape's boundary ∂I. As we advance inside the shape, *i.e.* H increases, this sawtooth pattern gets shifted parallel to the boundary with an angle between the pattern and boundary controlled by the parameter λ. For instance, when $\lambda = 1$, the created V-shape patterns form an angle of 45 degrees with the boundary (Fig. 5.7). Setting λ to negative values locally inverts the direction of the patterns. The created patterns meet precisely at the local center of the shape, *i.e.* on its skeleton, since H is locally normalized between 0 on the boundary and 1 on the skeleton.

The results shown in Fig. 5.7 show that this technique can generate directional V-like patterns which smoothly follow the shapes of the bundles, meet in the shape's local center, and also naturally split and merge around bundle junctions. However, we should stress that this technique is just a first step in the direction of a complete solution for visualizing directions for edges in a bundle. The key problem, yet to be solved, is how to set the value of the parameter λ so it conveys the *local* direction of edges in a bundle. Two possibilities exist here. First, we can precompute a *single* (dominant) direction for an entire bundle. If this is possible (and desirable for the type of insight to be conveyed in a concrete application), then λ can be globally set as a

Figure 5.7: Directional edge bundles: V-like patterns show the dominant edge direction in a bundle.

fixed value for the entire bundle, and we can apply the above algorithm with the effects illustrated in Fig. 5.7. However, there may be cases when this edge-direction aggregation cannot be done or does not make sense, *e.g.* in cases when a bundle contains edges with different directions along its several branches. In such cases, setting the *local* value of λ as a function of the local distribution of edge directions is a more complex challenge. We leave such challenges for future work.

5.2.7 *Interaction*

By construction, EBLs favor edge overlaps, so occlusion cannot be fully avoided. We alleviate this by several interaction techniques. First, we use classical brushing to render pixel-thin edges in the shape under the mouse. This shows the nodes linked by a given bundle, even if only a small part of the bundle is visible. Clicking on a shape brings it to front, sends it to back, or hides it. This helps bringing bundles of interest into focus.

We add a new interaction tool to further explore overlapping bundles: the digging lens. Given a focus point x (the mouse

pointer), and a pixel p within the lens radius R, $\|p - x\| < R$, we upper threshold the profiles $H(p)$ with $H_{min} = t[1 - (\|p - x\|/R)^2]$ for all visible shapes, where $t = 0.8$ gives the maximal thinning in the lens center. This smoothly shrinks shapes closer to the lens center, along the idea shown in Fig. 5.4 f (Sec. 5.2.4). We set the shapes' saturation to 1 in the lens and 0 outside. As the lens moves, shapes inside it get thinner (thus have less overlap) and also colorful (thus easy to focus on without distraction from outside shapes). As the user moves the mouse inside the lens, we automatically bring to front the shapes touched by the mouse.

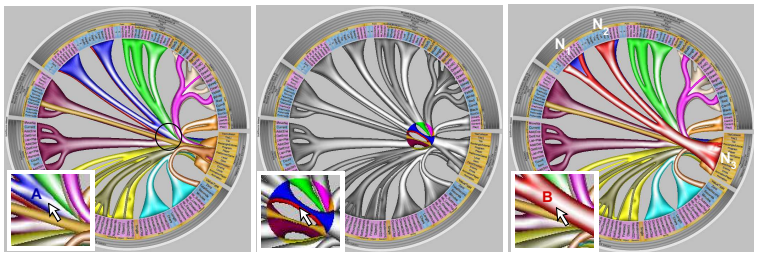

Figure 5.8: The digging lens is used to interactively explore areas where shapes overlap. Insets show zoomed-in details.

Figure 5.8 shows the digging lens. At the thin circle location (a), we see bundle overlaps. This cue triggers further exploration. For example, we want to see what is behind the blue bundle (A, inset). Activating the lens (by pressing Control) shows eight clusters, made distinct by shrinking and coloring (b). Moving the mouse over *e.g.* the red bundle (B, see inset) brings it to front, so we now see that it connects the node groups N_1,N_2 and N_3 (c). The entire process takes a few seconds and requires one key and one mouse click. Although useful, the digging lens cannot fully handle all possible overlaps: Where long bundles of same thickness overlap nearly completely, the lens will shrink them equally, and thus not reveal the hidden bundles. The lens is effective in places where bundles overlap but have slightly different directions and/or thicknesses.

5.3 RESULTS

Figure 5.9 shows the IBEB applied to the software dependency graph from Sec. 5.2.5. As a use-case, we consider analyzing *type usage*, *i.e.* inheriting from a class or using its type (functionality) in client code. This is one of the hardest kinds of dependencies

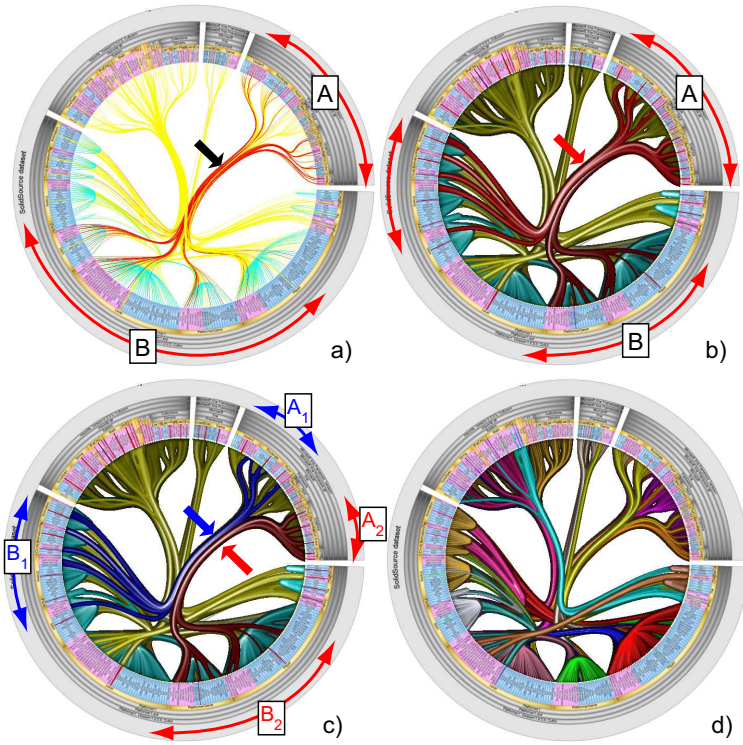

Figure 5.9: Software dependency graph exploration with IBEB (see Sec. 5.3).

to refactor in software. To analyze different coupling types, we first use the HEB with type-colored edges (calls = yellow, class member reads/writes = blue, type usage = red) (Fig. 5.9 a). We see a thick red bundle that links subsystems A and B. However, without iterative node selection, we cannot see which *parts* of A connect to which *parts* of B. Also, edge color blending makes it hard to see edge types at overlaps (arrow in the figure).

Next, we use the IBEB with convex shading and bi-level rendering (Fig. 5.9 b). Clusters contain only same-type edges (Sec. 5.2.2) and are colored on this type. We see now that member read/write relations form localized bundles not extending across classes (small light blue bumps, see *e.g.* the light blue arrow in (b)). This is a good sign for information hiding. Also, two red bundles appear. With two clicks, we bring these to front (b). We now see two separate subsystems in A connected to two separate subsystems in B. For illustration, we click on one of the two bundles (A_1B_1) and change its color to blue (Fig. 5.9 c).

We have now split the original red bundle into two relation sets: A_1B_1 and A_2B_2. Fig. 5.9 d shows further insight in the clustering: all bundles colored with different hues and overlaid with the actual pixel-thin edges. Albeit brief for space limitations, this example illustrates one main point: Classical edge bundles, like HEB, effectively show coarse-scale subsystem connections, but do not expose the finer-scale coupling structure within bundles. IBEB further reveals this structure, by showing where actual edges that 'enter' the bundle will 'exit'.

IBEB can be used with other layouts than the HEB. Figure 5.10 shows its usage with the FDEB on the *US migrations* graph from [80] (9780 edges). As a small addition to the edge splatting (Sec. 5.2.3), we now splat two extra radial profiles on both endpoints of an edge. This yields nicely rounded (capped) bundle shapes.

Compared to the original FDEB (Fig 5.10 a), IBEB exposes several bundles, *e.g.* the green one (West Coast migrations), yellow one (coast-to-coast migrations), a high-density small purple one (East Coast NY area), and an interesting high-density, high-coherence blue one (NY area to midwest migration). Figure 5.10 c uses the alternative thin shapes technique (cf. Sec. 5.2.4 Fig. 5.4 f) to further emphasize coarse graph structure. Here, we brought the coast-to-coast bundle (purple) in front. Finally, Fig. 5.10 d uses the *cores* style to emphasize structure and also reduce occlusion. All in all, we argue that the IBEB helps exposing coarse-scale bundle patterns, and seeing which nodes these bundles connect, while the original FEB is better at exposing fine-scale details in regions with little or no overlaps.

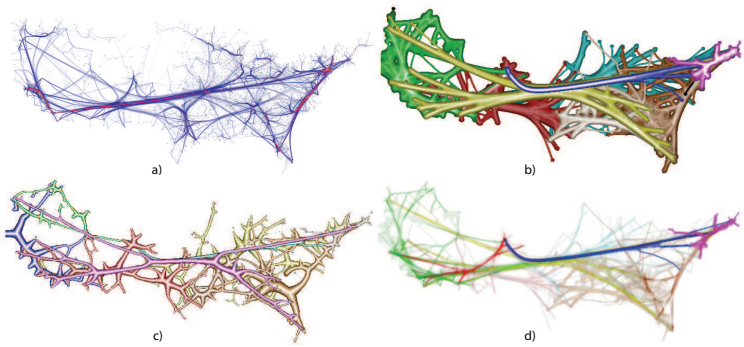

Figure 5.10: Image-based visualization of force-directed edge bundling (FDEB) layouts

To further understand the IBEB strong and weak points, we performed a formative user study. Twenty 3rd year CS students at the Univ. of Groningen, the Netherlands, were given the IBEB implemented atop of the SolidSX software analysis tool using the HEB [159], described in more detail in Chapter 4. The tool imports dependency graphs (inheritance, class field usage, function calls, and containment hierarchy) from Visual C++, .NET/C#, and Java. The C# software discussed earlier was provided by the tool developers as an interesting use-case. Participants were asked to find dependencies between several indicated modules; list the four most important call and field usage paths in the system; and comment on the overall system modularity. Search, filter, and node selection (available in the original tool) were disabled, so the tasks had to be completed mainly focusing on edges. One week was given to familiarize with the tool (which has a detailed manual) and execute the tasks. Effective usage time was 5 to 8 hours. The images in Fig. 5.9 come from this study.

Besides the actual answers, the following points were mentioned by all users (except two who did not complete the study):

- Classical HEB is very effective when (a) there are few bundle overlaps, or (b) one does not need to visually determine which *parts* of a large bundle go to which specific node groups;
- Although overlap exists, IBEB reveals several end-to-end (node-to-node) coarse-scale bundles which are not visible with classical HEB;
- The digging lens is effective in locally unraveling occluded bundles at a location of interest;
- The IBEB has an 'organic' look which is pleasing and invites exploration.

Overall, the IBEB combines the advantages of HEB with an easier understanding of *dense* bundles. In the traditional HEB, a thick, dense, bundle is seen as such but one cannot directly see whether there is finer-level structure, *e.g.* the bundle actually consists of several sub-bundles which connect different node groups, like in Fig. 5.9. This can be done by a trial-and-error selection of individual nodes to see if their edges indeed pass through the bundle of interest. Such selection is easily done in the HEB, but harder in layouts that draw nodes as small points, *e.g.* the FDEB. In contrast, IBEB makes bundles explicitly, and

individually, visible, so users can easier relate bundles to the nodes they connect. The fact that IBEB shows less fine-scale detail than the HEB does not seem to be a major problem, as individual edges are mainly explored once one has decided which few node(s) one wants to inspect. When this is known, both the HEB and IBEB are equally effective - in IBEB, brushing over a node and/or bundle highlights its edges, drawn as individual lines, just like in the HEB. For the several selection and brushing features we support, we refer to [159].

Our users also mentioned several desirable additions. First, even though shading and back-to-front rendering were seen as effective, overlaps still exist. The digging lens helps to analyze overlaps, but only locally. Secondly, edge direction cues are required. We tried several methods for this, *e.g.* luminance or saturation modulation of our bundle shapes (see Section 5.2.6), but this was found to darken images too much. Further work in this area is needed.

5.4 DISCUSSION

We next discuss several technical aspects of our method.

Generality: The only assumption made is that of a graph layout that delivers points along edges, and that edges are spatially bundled in a meaningful way. The layout and/or input graph do not need to obey other constraints, *e.g.* to be hierarchic or acyclic.

Parameters: The user has to set only a few values: level of detail d_{min} (Sec. 5.2.2), splatting radius δ (Sec. 5.2.3), and rendering style (Sec. 5.2.4). Here, only the level of detail does not have, so far, a preset usable for most datasets.

Performance: We ran the IBEB, implemented in C++ and OpenGL 1.1, on several systems running Windows Vista/XP, 1.5 to 3.5 GHz, and 2 GB to 4 GB RAM. The clustering used [40] handles 10..20K edges in under 0.1 seconds. Splatting, shading, rendering, and interaction (OpenGL-based) run in real-time on consumer graphics cards. We obtained real-time response even with Windows Remote Desktop rendering, which uses software-only OpenGL. Skeletonization, whose complexity is $N \log N$ for a binary shape I of N pixels (Sec. 5.2.4), takes 90% of the entire time. For simplicity, we used a software-only implementation [179] which takes 0.1 seconds/shape at 800^2 resolution, *i.e.* 1..2 seconds for a typical full frame. If desired, OpenGL-based skeletonization [167] can be readily used, which would deliver

subsecond/frame speed. Memory needs are around 100 MB for *e.g.* a graph with 10K edges discretized to a total 200K points.

Image-based vs geometric implementation: After edge clustering, IBEB works fully image-based. We also implemented a point cloud-based (geometric) version. We build the shapes I (Sec. 5.2.3) as alpha shapes from points p_{ij} on all edges e_j in a cluster C_i, using the CGAL library [29], similarly to [32]. We compute distances for the shading profiles H (Sec. 5.2.3) with a fast spatial search structure [6]. Speed is similar to the image-based variant. However, the alpha value (of alpha shapes) is hard to control [50]: High values fill in all gaps between bundle branches, low values yield holes inside what would be a compact branch. Resulting alpha shapes are rendered as shaded triangulated meshes. To yield the level-of-detail in I and H achieved by the image-based variant, we need a very high mesh resolution. All in all, we thus prefer the image-based approach.

Visual metaphor: The IBEB *convex* rendering style resembles the shaded edge bundles in illustrative parallel coordinates (IPC) [117], with some differences. Our shapes have a much higher variability, depending on the EBL used. We use hierarchical agglomerative clustering, while IPC uses *k*-means. We use skeletons in shading to emphasize the bundles' branching structure, to reduce overlaps (shrink shapes globally or locally by the digging lens), and for the *cores* rendering style. IPC uses different shapes and a shading style that mainly emphasizes line density.

Open points: The IBEB's main limitation is *visual* scalability. Using 10..30 shapes shows the coarse graph structure. More shapes create too many overlaps. However, we aim to provide a simplified view, not a full-blown replacement for bundled edges.

A useful result implies meaningful bundle shapes. This implies an edge bundling layout (EBL) that spatially groups related edges, and a clustering method (and edge similarity metric d) that yields meaningful edge clusters. The EBL and clustering used are generic, *e.g.* the HEB or FDEB (layout) and hierarchical agglomerative or k-means (clustering). However, d is application and task dependent. So far, we only considered edge types in d. Attributes such as edge weights or node types are open to exploration.

Finally, we stress that we select the visualization level-of-detail globally, and, so far, use only a single 'cut' in the cluster tree, purely based on similarity (Sec. 5.2.2). Locally refining bundles of interest, *e.g.* on user input, thus changing the shape of the cut, is a direction of further study. Here, we can draw inspi-

ration from the interactive navigation techniques from [5] for exploration of graphs structured along multiple hierarchies.

5.5 CONCLUSIONS

We have presented an image-based simplified visualization for edge bundles (IBEB). Given a layout that creates spatially close edge bundles, we visualize bundles using shaded overlapping compact shapes. We reduce the visual complexity of classical bundle visualizations, emphasize the coarse-scale structure, and help navigating from bundles to the connected nodes. We make bundle overlaps explicit, and add interaction to locally disambiguate these. Level-of-detail techniques help to select the visualization granularity and further explore overlaps.

Many extensions are possible. Different shading and edge clustering strategies can be used to address additional use cases, *e.g.* emphasize connections of particular types and/or topologies in a graph. New techniques can be designed to convey additional edge data such as direction or metrics atop of our metaphor. Finally, the IBEB can be extended to other fields, such as flow or tensor visualization. We plan to explore these avenues in future work.

ACKNOWLEDGEMENTS

We are grateful to Dennie Reniers and Lucian Voinea for the code of the SolidSX tool [159], datasets, and use-cases, and to Danny Holten for the force-directed bundling layout data (Sec. 5.3) and many insightful comments.

This chapter is based on:

Alexandru Telea and Ozan Ersoy. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. *Computer Graphics Forum* **29**, 843-852 (2010) *(2nd Best Paper Award, EuroVis'10).*

6

SKELETON-BASED EDGE BUNDLING

ABSTRACT: *In Chapter 5, we have shown that image-based techniques, in particular distance transforms and shape skeletons, can be used to simplify the rendering of large bundled layouts as a post-processing step to existing bundling algorithms. In this chapter, we take the image-based idea a step further, and show how we can use similar image-based techniques to actually bundle large graphs which come with or without hierarchical information. As layout cues for bundles, we use medial axes, or skeletons, of edges which are similar in terms of position information. We combine edge clustering, distance fields, and 2D skeletonization to construct progressively bundled layouts for general graphs by iteratively attracting edges towards the centerlines of level sets of their distance fields. Apart from clustering, our entire pipeline is image-based with an efficient implementation in graphics hardware. Besides speed and implementation simplicity, our method allows explicit control of the emphasis on structure of the bundled layout,* i.e. *the creation of strongly branching (organic-like) or smooth bundles. We demonstrate our method on several large real-world graphs.*

6.1 INTRODUCTION

As we have discussed in Chapter 2, when the number of nodes and edges of a graph increases, node-link graph visualizations become challenged by *clutter*, *i.e.* unorganized groups of nodes and edges onto small screen areas. To reduce clutter, and also address use-cases which focus on simplified depiction of large graphs with an emphasis on graph structure, several methods have emerged. Specifically, edge bundling layouts (EBLs) are an interesting alternative for classical node-link metaphors. Bundling typically starts with a given set of node positions, either present in the input data, or computed using a layout algorithm. Edges found to be close in terms of graph structure, geometric position of their endpoints, data attributes, or combinations thereof, are drawn as tightly bundled curves. This trades clutter for overdraw and produces images which are easier to understand and/or better emphasize the graph structure.

As we have seen in Chapter 5, bundling visualizations can be further enhanced in terms of reducing visual clutter by using image-based techniques. Specifically, we show how distance transforms and two-dimensional shape skeletons can be used to

visually 'cluster' edges which are part of the same bundle into a single shaded shape. Rendering such shapes, instead of the original fine-grained edges, effectively simplifies the bundled visualization and allows one to focus on its coarse-level structure.

In this chapter, we take the image-based approach outlined above a step further. Specifically, we use two-dimensional shape skeletons for actual edge bundling rather than simplifying an existing EBL. In detail, we combine edge clustering, distance fields, and 2D skeletonization to construct bundled layouts by iteratively attracting edges towards the centerlines of level sets of their distance fields. Apart from clustering, our pipeline is image-based, which allows an efficient implementation in graphics hardware. Besides speed, our method allows users to explicitly control the emphasis on bundle structure, *i.e.* create strongly branching (organic-like) or smooth bundles which always have a tree structure. This type of control can be helpful in applications where one is interested to see how several edges 'join' together into, or split from, main structures, for example when exploring the structure of a network. Instances hereof are examining the local hierarchy of traffic connections in a road or airline network, or identifying the number and size of branches (fan in/out patterns) in software structures.

The structure of this chapter is as follows. Section 6.2 presents our bundling algorithm. Section 6.3 details implementation. Section 6.4 presents applications on large real-world graphs. Section 6.5 discusses our method. Section 6.6 concludes the chapter and outlines future work directions.

6.2 ALGORITHM

The inspiration behind our method relates to a well-known fact in shape analysis: given a 2D shape, its skeleton is a curve locally centered with respect to the shape's boundary [37]. Skeleton branches capture well the topology of elongated shapes [101, 156]. Hence, if we could create such shapes from sets of edges in a graph, their skeletons could be suitable locations for bundling. This observation is also supported by the image-based edge bundling (IBEB) method presented in Chapter 5: Skeletons are there used to construct simplified bundle renderings, and distance fields induced by these skeletons are used to construct the shading of such bundles (bright in the middle of the

bundle, close to the skeleton, and dark away from the bundle's middle).

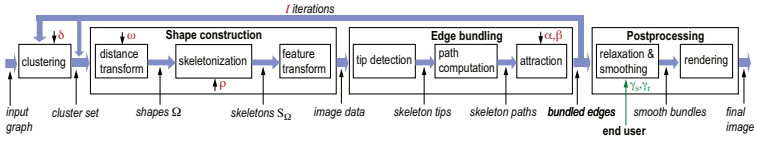

Figure 6.1: Skeleton-based edge bundling pipeline. End user parameters are marked in green. System preset parameters are in red

To this end, we propose a skeleton-based edge bundling method, as follows (see Fig. 6.1):

1. we *cluster* edges into groups, or clusters, C_i which have strong geometrical and optionally attribute-based similarity;
2. for each cluster C, we compute a thin shape Ω surrounding its edges using a distance-based method;
3. for each shape Ω, we compute its skeleton S_Ω and feature transform of the skeleton FT_S;
4. for each cluster C, we attract its edges towards S_Ω using FT_S;
5. we repeat the process from step 1 or step 2 until the desired bundling level is reached;
6. we perform a final smoothing and next render the graph using a cushion-like technique to help understanding bundle overlaps.

We start with an unbundled graph $G = (V, E)$ with nodes V and edges E. We assume that we have node positions $v_i \in \mathbf{R}^2$, either from input data, or from laying out G with any existing method *e.g.* spring embedders [192]. Edges $e_i \in E$ are sampled as a set of points connected by linear interpolation; other schemes such as splines work equally well. The start and end points of an edge, denoted e_i^s and e_i^e respectively, are the positions of the nodes the edge connects. Edge points may come from input data, *e.g.* when we bundle a graph which has explicit edge geometry. If no edge positions are available, we initialize the edge points by uniformly sampling the line segments (e_i^s, e_i^e) with some small step. Our bundling algorithm iteratively updates these edge points. Its output is a bundled layout

of G which keeps node positions intact and adjusts the edge points to represent bundled edges.

The six steps of our method are explained next.

6.2.1 *Clustering*

To obtain elongated 2D shapes, needed for our bundling (described next in Sec. 6.2.3), we first cluster edges using a similarity metric which groups same-direction, spatially close, edges, using the clustering method described in Chapter 5. We have tested several clustering algorithms: hierarchical bottom-up agglomerative (HBA) clustering using full, centroid, single, and average linkage, and k-means clustering, both with Euclidean and statistical correlation (Pearson, Spearman's rank, Kendall's τ) distances. HBA with full linkage and Euclidean distance given by

$$d(e_i, e_j) = \sqrt{\sum_{k=1}^{N} \|e_{ik} - e_{jk}\|^2} \tag{6.1}$$

where $e_{ik,k\in\overline{1,N}}$ are uniformly spaced sample points along the edges, with $N \in [50, 100]$, gives the best results, *i.e.* clusters with geometrically close edges which naturally follow the graph structure. Using the same N for all edges removes edge length bias. HBA delivers a dendrogram $D = \{C_i\}$ with the edge set E as leaves and similarity (linkage) values $d(C)$, equal to the full linkage of cluster C based on the distance metric in Eqn. 6.1, increasing from root to leaves. We select a 'cut' in D, or partition, $P = \{C_i \in D | d(C_i) < \delta\}$ of E based on a similarity value δ, set by our algorithm as explained further in Secs. 6.2.5 and 6.3. If desired, d in Eqn. 6.1 can be easily adapted to incorporate edge data attributes, as outlined in Chapter 5 (see Sec. 5.2.2).

6.2.2 *Shape construction*

Clustering delivers sets of spatially close edges, *i.e.*, the bundling candidates. Given such a cluster $C = \{e_i\}$, we consider its drawing $\Delta(C) \subset \mathbf{R}^2$, *e.g.* the set of polylines corresponding to its edges e_i if we use the default linear edge interpolation. We construct a compact 2D shape $\Omega \subset \mathbf{R}^2$ surrounding $\Delta(C)$, as

follows (see also Fig. 6.2). Given any shape $\Phi \subset \mathbf{R}^2$, we first define its distance transform $DT_\Phi : \mathbf{R}^2 \to \mathbf{R}_+$ as

$$DT_\Phi(\mathbf{x} \in \mathbf{R}^2) = \min_{\mathbf{y} \in \Phi} \|\mathbf{x} - \mathbf{y}\| \tag{6.2}$$

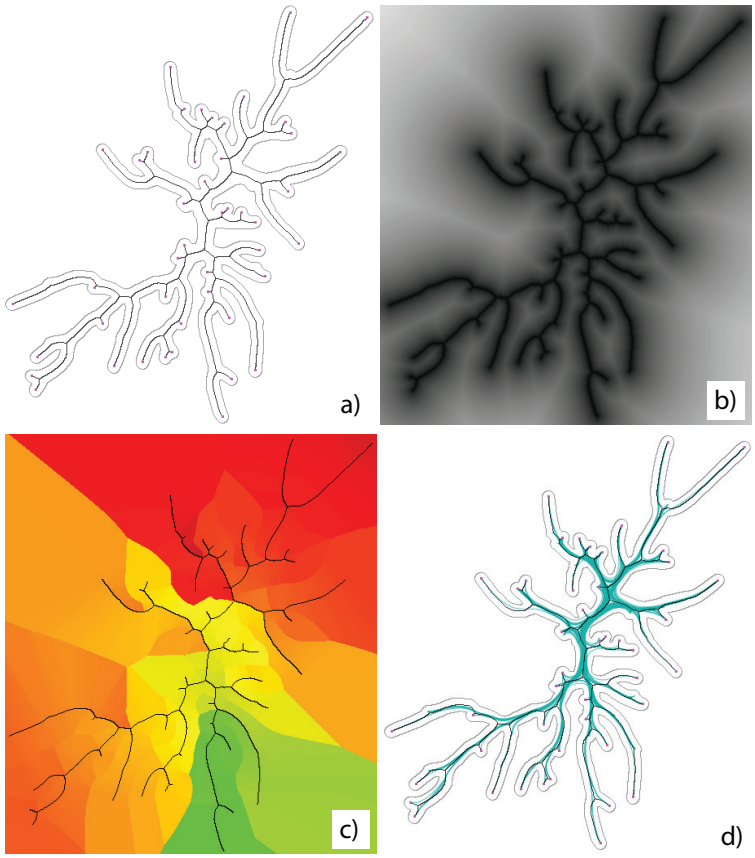

Figure 6.2: Shape construction: a) $\partial\Omega$ and S; b) DT_S; c) FT_S; d) bundling result (see Secs. 6.2.2-6.2.4 for details).

Given a distance value ω, we next define our shape Ω as

$$\Omega = \{\mathbf{x} \in \mathbf{R}^2 | DT_{\Delta(C)}(\mathbf{x}) \leqslant \omega\} \tag{6.3}$$

where $DT_{\Delta(C)}$ is the distance transform of the drawing $\Delta(C)$ of C's edges. The shape's boundary $\partial\Omega$ is the level set of value ω of $DT_{\Delta(C)}$ (see Fig. 6.2 a). This is equivalent to inflating $\Delta(C)$ with a distance ω in all directions. In practice, we set ω to a

small fraction (*e.g.* 0.05) of the bounding box of G. Efficient computation of distance transforms is detailed further in Sec. 6.3.

6.2.3 *Shape creation*

Given a shape Ω computed from an edge cluster drawing as outlined above, we next compute its skeleton S_Ω defined as

$$S_\Omega = \{\mathbf{x} \in \Omega | \exists \mathbf{y}, \mathbf{z} \in \partial\Omega, \mathbf{y} \neq \mathbf{z}, \|\mathbf{x}-\mathbf{y}\| = \|\mathbf{x}-\mathbf{z}\| = DT_{\partial\Omega}(\mathbf{x})\} \tag{6.4}$$

i.e. the set of points in Ω which admit at least two different so-called feature points on $\partial\Omega$, at distance equal to the distance transform of $\partial\Omega$ (Fig. 6.2 a).

Given S, we now compute its so-called one-point feature transform $FT_S : \mathbf{R}^2 \rightarrow \mathbf{R}^2$, defined as

$$FT_S(\mathbf{x}) = \{\mathbf{y} \in S | DT_S(\mathbf{x}) = \|\mathbf{x}-\mathbf{y}\|\} \tag{6.5}$$

i.e. one of the feature points of $\mathbf{x}$. Figure 6.2 b,c show the DT_S and FT_S of a skeleton. Gray values in Fig. 6.2 b indicate the DT_S value (low=black, high=white). Colors in Fig. 6.2 c indicate the identity of different feature points: same-color regions correspond roughly to the Voronoi regions of the skeleton branches [179]. The skeleton is the identity set of FT_S, *i.e.* $\forall \mathbf{x} \in S, FT_S(\mathbf{x}) = \mathbf{x}$. Note that, in Eqn. 6.5, we use the distance transform DT_S of the skeleton S, and not the distance transform $DT_{\partial\Omega}$ of the shape. Also, note that the one-point feature transform is simpler than the so-called full feature transform

$$FT_S^{full}(\mathbf{x}) = \underset{\mathbf{y}\in S}{\operatorname{argmin}} \|\mathbf{x}-\mathbf{y}\| \tag{6.6}$$

which records *all* feature points of $\mathbf{x}$ [37].

In practice, we compute distance transforms, one-point feature transforms, and skeletons in discrete image (screen) space. This allows efficient implementation (see Sec. 6.3) and also further processing of the skeleton for edge bundling, as described next.

6.2.4 *Edge attraction*

Using the skeleton S and its feature transform FT_S, we now bundle the edges $e_i \in C$ by attracting a discrete representation of

each edge towards S. This idea is based on the following observations. First, given the way we combine clustering and edge bundling, a cluster contains only edges having close trajectories; the reasons for this are detailed in Sec. 6.2.5. By construction, the skeleton S of a cluster is locally centered with respect to the (similar) edges in that cluster, *i.e.* a good candidate for the position to bundle towards. Secondly, $FT_S(\mathbf{x}) - \mathbf{x}$ gives, for each point $\mathbf{x} \in \mathbf{R}^2$, the direction vector from $\mathbf{x}$ to the closest skeleton point to $\mathbf{x}$, *i.e.* the direction to bundle towards. We use these observations to bundle e_i as follows.

First, we compute all branch termination points, or *tips*, $T = \{\mathbf{t}_i\}$ of S. Given that S is represented in image space, we use a simple and efficient 3×3 pixel template-based method [96] to locate t_i. Next, we compute all skeleton paths $\Pi = \{\pi_i \subset S\}$ between any two tips $\mathbf{t}_i$ and $\mathbf{t}_j$. The paths are represented as pixel chains and are found using depth-first search from each $\mathbf{t}_i$ on the skeleton pixel-adjacency-graph. We next use these paths to robustly attract the edges towards the skeleton.

For each $e_i \in C$ with start and end points e_i^s and e_i^e respectively, we select a path passing through the feature points of both edge end points $\pi(e_i) \in \Pi$ so that $\{FT_S(e_i^s), FT_S(e_i^e)\} \subset \pi(e_i)$, *i.e.* a skeleton path. If there are several such paths in Π, we pick any one of them, the particular choice having no influence on the algorithm.

We now use $\pi(e_i)$ to bundle e_i along the skeleton, as follows. Consider a point $\mathbf{x} \in e_i$ located at arc-length distance $\lambda(\mathbf{x})$ from $\mathbf{e}_i^s$. We move $\mathbf{x}$ towards $FT_S(\mathbf{x})$ with a distance which is large if $\mathbf{x}$ is far away from $FT_S(\mathbf{x})$ and/or close to the middle of the edge:

$$\mathbf{x}^{new} = \left[1 - \alpha\phi\left(\frac{\lambda(\mathbf{x})}{\lambda(\mathbf{e}_i^e)}\right)\right]\mathbf{x} + \alpha\phi\left(\frac{\lambda(\mathbf{x})}{\lambda(\mathbf{e}_i^e)}\right) FT_S(\mathbf{x}) \quad (6.7)$$

Here, $\alpha \in [0,1]$ controls the tightness of bundling: Large values bring the edge closer to the skeleton, whereas small values bundle less. The function $\phi : [0,1] \to [0,1]$ defined as

$$\phi(t) = [2\ \min(t, 1-t)]^K \quad (6.8)$$

modulates the motion amount so that the edge's end points $\mathbf{e}_i^s$ and $\mathbf{e}_i^e$ do not move at all, points close to these end points move less, and points around the middle of the edge move most. This produces the curved edge profile we require for bundling, and also keeps edge end points fixed to their node locations. The parameter K controls how smoothly edges twist, or curve, from

their nodes to reach their bundled location. Higher K values produce more twists, and low K values produce smoother twists. Values of $K \in [3, 6]$ give very similar results to known bundling methods *e.g.* [78, 80, 103]. Also, for any $\mathbf{x} \in S$, $FT_S(\mathbf{x}) = \mathbf{x}$ (Sec. 6.2.3), so for such points we have $\mathbf{x}^{new} = \mathbf{x}$ (Eqn. 6.7), *i.e.* points which have reached the skeleton, the extreme bundling location, do not move any longer.

Equation 6.7 is equivalent to advecting edge points $\mathbf{x}$ in the gradient field $-\nabla DT_S$. Distance transforms of any shape except a straight line have div $\nabla DT_S \neq 0$ [157]. Hence, our attraction typically shortens and/or lengthens edges, since these get immediately curved after one application of Eqn. 6.7. We compute the edge points $\mathbf{x}$ used in Eqn. 6.7 by uniformly sampling edges in arc-length space with a distance equal to a small fixed fraction (0.05) of the layout's bounding box. This removes points where the edge contracts (div $\nabla DT_S < 0$) and inserts points where the edge dilates (div $\nabla DT_S > 0$) as needed, thus ensuring a uniform edge sampling density.

6.2.4.1 *Attraction singularities*

As explained, Eqn. 6.7 is equivalent to advecting $\mathbf{x}$ in the field $-\nabla DT_S$. This field is smooth everywhere in $\mathbf{R}^2$ except on points $\mathbf{x}$ where $\|FT_S^{full}(\mathbf{x})\| > 1$, *i.e.* points located on the skeleton of the skeleton's complement, or Voronoi diagram of S, $\overline{S} = S_{\mathbf{R}^2 \setminus S}$. Intuitively, $\overline{S}$ corresponds in Fig. 6.2 c to color discontinuities. Although this singularity set is small, *i.e.* a set of curves in 2D, we need special treatment for such situations. If we were to directly advect a curve using Eqn. 6.7 with no further precaution, singularities would appear where the curve crosses $\overline{S}$, since ∇DT_S has a high absolute divergence, *i.e.* changes direction rapidly, in such areas [157]. Such singularities appear as sharp kinks in the curve, which defeats our purpose of creating smooth bundles. For example, attracting the blue edge e in Fig. 6.3 a towards the Y-shaped skeleton yields the red line which shows two kinks, where e crosses $\overline{S}$ (dotted line) at points $\mathbf{a}$ and $\mathbf{b}$. The problem is made only more complex by the fact that we use a sampled edge representation, so $\mathbf{x}$ may be close, but not on, $\overline{S}$.

We solve such situations by an implicit *regularization* of the advection field determined by FT_S. First, we enforce the constraint that points $\mathbf{x} \in e$ can only be advected to points on the edge's path $\pi(e)$. This ensures that, during advection, parts of e cannot be attracted towards other skeleton branches than the

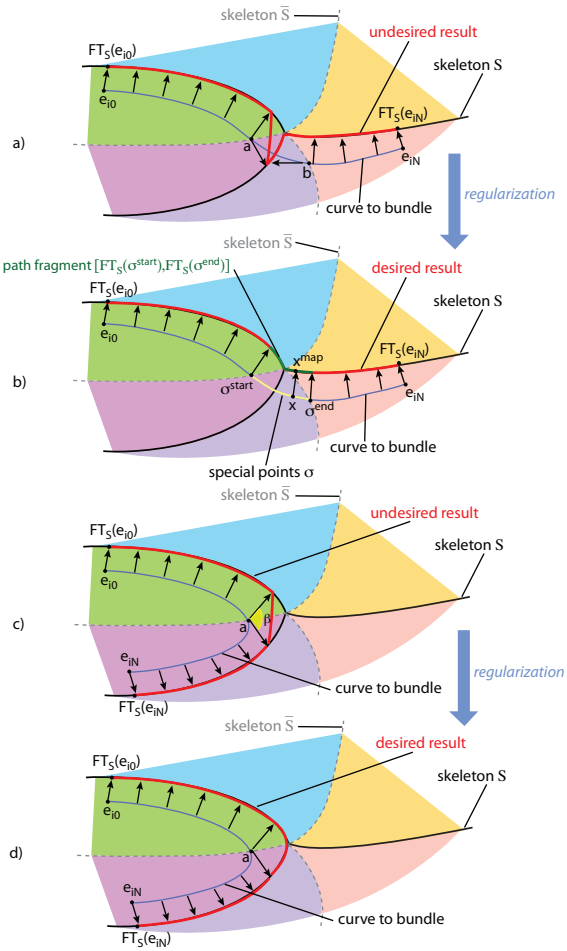

Figure 6.3: Attraction singularities. Naive solution (a,c) and corresponding solutions with regularization (b,d). Final bundled curve is shown in red. Voronoi regions of the branches of $\bar{S}$ are shown in different hues

set of *contiguous* branches which form π. Intuitively, Eqn. 6.7 should not pull e towards non-connected skeleton branches. We achieve this constraint as follows (see Fig. 6.3 b). For each $\mathbf{x} \in e$, we evaluate its $FT_S(\mathbf{x})$. If $FT_S(\mathbf{x}) \in \pi(e)$, we attract the 'regular' point $\mathbf{x}$ using Eqn. 6.7, else we mark $\mathbf{x}$ as special case. Special points along e (yellow in Fig. 6.3 b) form compact sets σ_i, which are preceded and followed on e by regular points σ_i^{start} and σ_i^{end} respectively, whose feature points belong to $\pi(e)$ by construction. We next map each special point $\mathbf{x}$ to a

corresponding point $\mathbf{x}^{map}$ on $\pi(e)$ using arc-length interpolation along both σ_i and their corresponding path fragments $[FT_S(\sigma_i^{start}), FT_S(\sigma_i^{end})] \subset S$ (dark green in Fig. 6.3 b), and use $\mathbf{x}^{map}$ in Eqn. 6.7 instead of $FT_S(\mathbf{x})$. This ensures that both special and regular points are attracted to the same path $\pi(e)$, and thus, since $\pi(e)$ is a compact curve, that the motion of e is smooth.

However, the above regularization does not eliminate *all* the sharp kinks in the advection of an edge: Consecutive points of the edge can 'see' points on the same skeleton path π, and still be separated by a singularity (see point **a** in Fig. 6.3 c). As explained, advecting such points **a** using Eqn. 6.7 would produce undesirable bends. Since the feature-point of **a** is located on the same path $\pi(e)$ as those of **a**'s neighbors on the edge, we cannot find **a** using the path-based detection criterion outlined above. We solve this problem by using an angle-based criterion: Given our discrete edge representation $e = \{\mathbf{x}_i\}$, we test if the feature vectors $FT_S(\mathbf{x}_i) - \mathbf{x}_i$ and $FT_S(\mathbf{x}_{i+1}) - \mathbf{x}_{i+1}$ of consecutive edge sample points $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ form a large angle β. If β exceeds a user-defined value β_{max}, we mark $\mathbf{x}_i$ as a special point and treat it as explained earlier for the path-based detection criterion. In practice, $\beta_{max} = \pi/4$ has given good results for all graphs we tested. The overall effect is that sharp edge angles are eliminated and edges are advected smoothly towards the skeleton (Fig. 6.3 d). As a more complex example of our regularization, Fig. 6.2 d shows the bundling of a set of edges (green) close to the skeleton in Fig. 6.2 a.

Our angle criterion is a one-dimensional version of the divergence-based Hamilton-Jacobi skeleton detector of [157]. It subsumes the path-based criterion. In theory, it would be sufficient to use the angle criterion to achieve smooth motion. However, the path-based criterion is more numerically robust as it involves no angle estimation or thresholding. Since its application is equally fast (we need paths anyway to regularize the attraction in both cases), we use it when applicable to reduce any chance for numerical instabilities.

6.2.5 *Iterative algorithm*

For a given graph layout, one application of the clustering, shape construction, and edge attraction steps outlined above yields a new layout whose edges are closer to their respective cluster skeletons. To achieve full bundling, we repeat this process itera-

tively until a user-specified number of iterations I is reached. More iterations yield tighter bundled edges. This process is strictly monotonic, *i.e.* edges can only get closer to their clusters' skeletons (hence to each other) by construction, as explained below (see also Fig. 6.4).

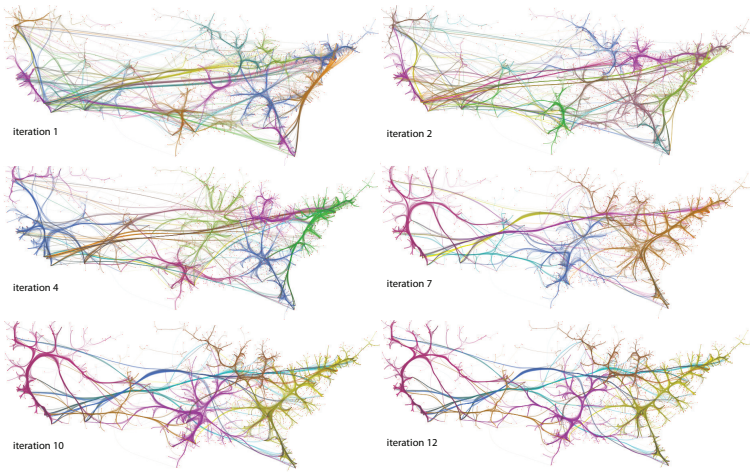

Figure 6.4: Iterative bundling of the US migrations graph. Colors indicate edge clusters (see Sec. 6.2.5)

First, let us explain why clustering needs to be repeated during the iterative process. For the first clustering, we use a high similarity threshold δ in order to guarantee elongated, thin, clusters regardless of the edge spatial distribution in the input graph (Sec. 6.2.1). This is essential for getting the initial bundling under way. Indeed, if we had weakly coherent clusters, these would contain edges that intersect each other at large angles, hence the shapes surrounding them, and their skeletons, would be meaningless as bundling cues. For subsequent iterations, we decrease δ and recluster the graph each few $(3\text{to}5)$ iterations. This produces fewer, increasingly larger, clusters, which allows fine-scale bundles to group into coarse-scale ones. However, these large clusters are *locally* elongated, since they contain already partially bundled edges. Hence, coarsening the clustering will not group unrelated edges. The overall effect is bottom-up bundling: First, the closest edges get bundled, yielding fine-scale local bundles, followed by increasingly coarser-scale bundle merging.

Similarly, we decrease α during the iterative process. Initial large α values yield strongly coherent initial bundles, needed

for clustering stability as explained above. Subsequent relaxed α values allow edges in more complex, larger, bundles to adjust themselves. Concrete values for δ and α are given in Sec. 6.3.2.

6.2.6 *Postprocessing*

6.2.6.1 *Relaxation and smoothing*

The output of our bundling algorithm has a strong branch-like structure (see *e.g.* Fig. 6.6 b). This is the inherent effect of using skeletons as bundling cues. Indeed, skeleton branches asymptotically meet at large angles [134]. This visual signature of our bundles may be desirable for use-cases where one is interested to see the branching structure of a graph. However, often the fact that two bundles join at some point in a thicker bundle is irrelevant, and should not be over-emphasized. We offer this possibility by performing a final postprocessing on the bundled layout. Here, two variations are proposed. First, we apply a simple Laplacian smoothing filter along the edges γ_s times, much like [80].

This removes sharp bundle turns, which by construction appear precisely, and only, where skeleton branches meet. Indeed, as known from medial axis theory, a skeleton branch is always a smooth curve; the only curvature discontinuities along a skeleton appear at branch junctions [134]. A second postprocessing we found useful is to interpolate linearly with a value $\gamma_r \in [0, 1]$ between the bundled graph and its initial layout. This relaxes the bundling, which is desirable when users want to see the individual edges within a bundle and/or where these come from in the initial layout. The effect is similar to the spline tightness parameter in [78].

Figure 6.5 a,b show the effect of smoothing on a graph whose nodes use a radial layout. Smoothing (b) removes the strong branching effect visible in (a) at the locations indicated by arrows. The result is very similar to the HEB layout [78]. However, it is important to stress that we obtain our bundling with no graph *hierarchy* information. Figures 6.6 a,b show the effect of smoothing and relaxation on the well-known US airlines graph, whose bundled layout is shown in Fig. 6.9 f. Smoothing removes the 'skeleton effect' from the bundles, while relaxation makes these thicker with less effect on their curvature. As such, the two effects serve complementary goals.

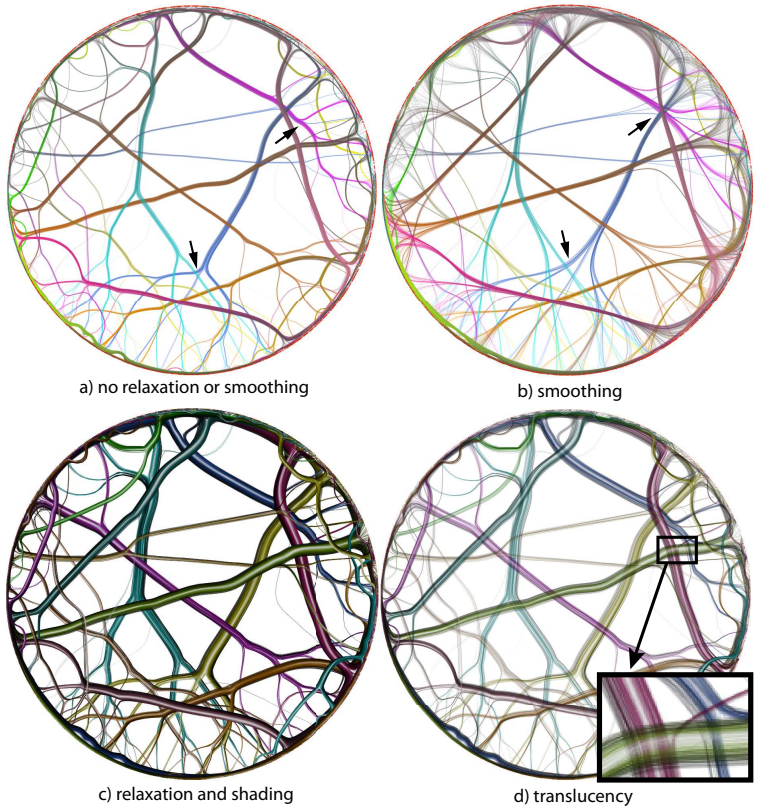

Figure 6.5: Layout postprocessing on a graph with radial layout. Edge smoothing (a vs b). Cushion shading (c), half-transparent detail (d).

6.2.6.2 *Rendering*

Finally, we propose a simple but effective rendering technique for easier visual following of the rendered bundles (Fig. 6.5 c,d). The principle follows [178]: We render each bundle in back-to-front order, decreasingly sorted by skeleton pixel count $|S|$, as if they were covered by a 3D cushion profile bright at the bundle's center and dark at its periphery. This helps following a given bundle, especially in regions where several bundles cross. In contrast to [178], we use a much simpler technique (see Fig. 6.7). Edges are rendered as alpha-blended polylines. We modulate the saturation S and brightness B of each polyline point $\mathbf{x}$ based

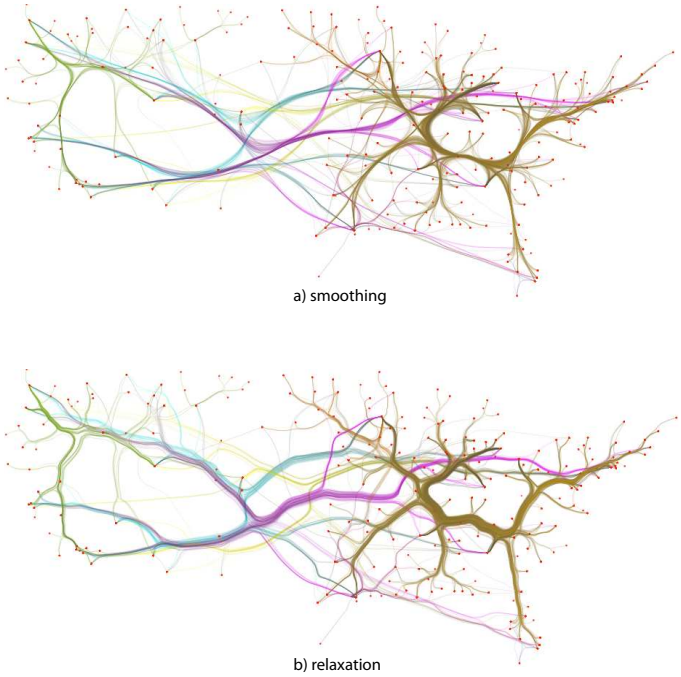

Figure 6.6: Layout postprocessing on US airlines graph. Edge smoothing (Fig. 6.9 f vs a). Edge relaxation (Fig. 6.9 f vs b).

on its distance to the skeleton $d(\mathbf{x}) = DT_S(\mathbf{x})$, which is already computed for the attraction phase (Sec. 6.2.3). For this, we use

$$S(d) = \sqrt{1 - d/\delta_S} \tag{6.9}$$

$$B(d) = 1 - \sqrt{d}/\delta_B \tag{6.10}$$

This yields thin, specular-like, white highlights in the middle of the bundles (where the skeleton is located) and darkens the edges as they get further from the skeleton. The parameter δ_B is the local thickness of the bundle. For an edge point $\mathbf{x} \in \Omega$, $\delta_B(\mathbf{x}) = DT_S(FT_{\partial\Omega}(\mathbf{x}))$, *i.e.* the distance of the closest point on the shape boundary $\partial\Omega$ to the shape's skeleton. This does not require any extra computations, since we anyway compute $FT_{\partial\Omega}$ and DT_S as part of the shape construction (Sec. 6.2.2, see also Sec. 6.3 for implementation details). The parameter $\delta_S < \delta_B$ controls the highlight thickness and is set to a small fraction (*e.g.* 0.2) of δ_B. This technique has several differences as compared to splatting-based shading techniques for bundles in [103, 178]. First, our rendering does not change the screen-space thickness

of a bundle, which is determined by the bundling layout – thin bundles stay thin. In contrast, splatting techniques tend to make thin bundles relatively thicker, which consumes screen space and increases occlusion chances. Secondly, if we relax the bundling as described earlier, individual edges become visible but still show up as a coherent whole due to the cushion shading. Figure 6.5 d shows this. To better illustrate the effect, we decreased here the overall opacity of the edges. The inset shows how bundles appear as shaded profiles even though they are not, technically speaking, compact surfaces. Thirdly, although we could use a physically correct shading model (like [103]), we found our pseudo-illumination adequate in terms of our goal of understanding overlapping bundles.

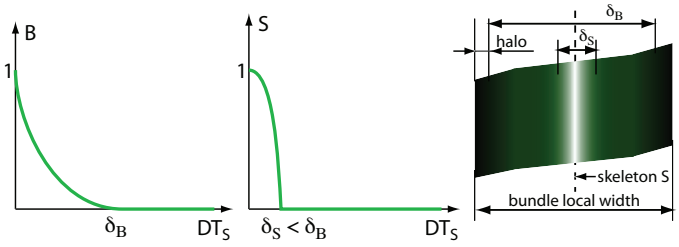

Figure 6.7: Cushion shading for bundles (Sec. 6.2.6.2)

6.2.6.3 *Interaction*

We have experimented with several types of interactive exploration atop of our method. In particular, our image-based pipeline and explicit representation of edge clusters allows us to easily brush or select groups of edges showing up as bundles or branches thereof. Three types of selection were found useful, as follows (see also Fig. 6.11 a-c and example discussed in Sec. 6.4). Given the mouse position $\mathbf{x}$, we first select all bundled edges within a disc of small radius r centered at $\mathbf{x}$ by computing the feature transform of the *bundled* edges and then selecting all edges which contain feature points in the disc. This query is useful for basic edge brushing and for building the next two queries. Secondly, we want to select all edges in the most prominent bundle, or bundle branch, passing through the disc. We repeat the basic selection, count the number of selected edges having the same cluster id, and retain the ones having the cluster id for which the most edges were found. This selects the thickest bundle branch close to the mouse, since edges within any bun-

dle branch always have the same cluster ids, by construction. Finally, to select an entire cluster, we do the basic selection and return all edges in the cluster whose id is the one for which the most edges were found.

6.3 IMPLEMENTATION

Several implementation details are crucial to the efficiency and robustness of our method, as follows.

6.3.1 *Image-based operations*

We compute shapes, skeletons, skeleton tips, and distance and feature transforms in an image-based setting. First, we render all edges using standard OpenGL polylines. Next, we use a Nvidia CUDA 1.1 based implementation of exact Euclidean distance-and-feature transforms [28]. We extended this technique to compute robust skeletons based on the augmented fast marching method (AFMM) in [179]. In brief, we arc-length parameterize the shape boundary $\partial\Omega$ and detect S_Ω as pixels whose neighbors' feature points subtend an arc on $\partial\Omega$ larger than a given value ρ. The value ρ indicates the minimal detail size on $\partial\Omega$ which creates a skeleton point. Since $\partial\Omega$ is a level-set of a distance transform at value ω of a set of smooth curves (edges), it only contains 'sharp' details at the curve end points. Hence, setting $\rho = \pi\omega$, *i.e.* half the perimeter of a circle of radius ω, guarantees that skeleton tips correspond to edge end points. The skeletonization method choice is essential: the AFMM guarantees that no spurious branches appear due to boundary perturbations, which in turn guarantees stable bundling cues. However, even if all skeleton *tips* correspond to edge end points, this does not mean that all edge *end points* correspond to skeleton tips. Short edges within a large cluster do not produce skeleton tips. This is another reason for using the displacement function ϕ (Eqn. 6.8) to guarantee that no edge end points move during bundling.

The original CPU-based AFMM [179] is too slow for our task. Table 6.2 show the inflation (Eqn. 6.2) and skeletonization times (Eqn. 6.4), the latter also including the skeleton feature transform, on a 2.8 GHz quad-core Windows PC (Sec. 6.4) for several graphs at an image size of 1024^2. Table 6.1 gives statistics on these graphs, including the (decreasing) number of clusters at several iterations. On the average, the time needed by

Graph	Nodes	Edges	Clusters/iteration			Total (GPU)
			$I = 1$	$I = 5$	$I = 10$	(sec.)
US airlines	235	2099	90	15	9	6.3
US migrations	1715	9780	57	14	7	4.1
Radial	1024	4021	94	30	24	7.4
France air	34550	17275	207	40	26	29.2
Poker	859	2127	86	28	23	5.2

Table 6.1: Graph statistics for datasets used in this chapter

Graph ($I = 5$)	Tips	Points	Inflation (ms)	Holes (ms)	Skel. (ms)	Paths (ms.)	Attraction (ms)
US airlines	22	8388	77	120	314	98	20
US migrations	28	9780	78	134	339	170	77
Radial	14	21580	80	96	357	45	17
France air	34	23759	81	148	374	222	88
Poker	28	2385	64	117	238	146	13
CUDA implem.			2	8	2	< 12	3

Table 6.2: SBEB performance. Figures are averages for all clusters at iteration $I = 5$ for different graphs. First rows show CPU timings. Last row shows CUDA-based timings (which are uniform for the tested graphs).

the AFMM to process a cluster sums up to 0.4 seconds (in line with [179]). For a graph with 200 clusters (Fig. 6.8 a,b), this yields 80 seconds/iteration. The AFMM is $O(\delta|C| \log(\delta|C|))$ where $|C|$ is the number of pixels on all edges in a cluster C, since the AFMM computes within a band of thickness δ around its input shape, *i.e.* $|\Omega| = O(\delta|C|)$. In contrast, our CUDA implementation takes 4 milliseconds per distance, feature transform, and skeletonization for the same image on a Nvidia GT 330M GT card, in line with performance reported in [28], *i.e.* 0.8 seconds per iteration for the graph in Fig. 6.8 a,b. Graphs with fewer clusters require proportionally less time, since the speed of the CUDA method is $O(N)$ for an image of N pixels, thus image-size-bounded. Overall, the CUDA solution is roughly 100 times faster than the CPU-based AFMM.

The complexity of the skeleton path computations (Sec. 6.2.4) is discussed next. Following earlier comments on the distance-

level-set nature of $\partial\Omega$, the number of skeleton tips $|T|$ for a shape is $O(|\partial\Omega|/(\pi\omega))$. Since we set ω to a fixed fraction of the image size (0.05, see Sec. 6.2.2), we get on the average a few tens of tips per skeleton, regardless of the number of edges in a cluster (Tab. 6.2 (Tips)). AFMM guarantees 1-pixel-thin skeletons [179], so all nodes in the skeleton pixel-adjacency-graph are of degree 2, except skeleton junctions which are $O(|T|)$ in number. The length of the skeleton of a shape $\partial\Omega$ is $O(|\partial\Omega|)$. Hence, the depth-first-search finding of skeleton paths between tips (Sec. 6.2.4) is $O(|T|^2|\partial\Omega|)$ using a brute-force method. Table 6.2 (Paths) shows the costs for the graphs in this chapter using quad-core multithreading with one depth-first-search per thread. The same implementation on CUDA reduces the costs to 12 milliseconds (or less for skeletons with fewer tips) as more cores are available. This cost could be reduced further, if desired, by using the same depth-first search on the much simpler graph whose nodes are skeleton tips and skeleton branch junctions and edge weights given by skeleton branch lengths, or faster all-pairs shortest path algorithms at the expense of a more complex implementation [91].

The attraction step is linear in the number of edge discretization points, *i.e.* tens of thousands for large graphs (Tab. 6.2 (Points)). Edges are attracted independently to their cluster skeleton, so CUDA parallelization of this step is immediate.

Inflating edges can produce shapes of genus > 0, *i.e.* with holes. Technically, this is not a problem, as skeletonization, path computation, and attraction can handle this. However, we noticed that such holes are rarely meaningful. Holes create loops in the skeleton and thus loops in a *single* bundle, which is supposed to be a tight object. To remove this, we fill all holes in our shapes prior to skeletonization using an efficient CUDA-based scan fill method, as follows: Given a background seed pixel outside the image Ω, *e.g.* the pixel $(0, 0)$, we mark it with a special value ν. Next, we fill horizontal scan line segments of background value from each ν-valued pixel in parallel, one scan line per thread. We repeat alternating horizontal with vertical scan line passes until no pixel is filled any more. Checking the stop condition requires only non-synchronized writing to a global boolean variable, set to false before each pass. This parallelizes more efficiently than classical scan line or flood fill. Marking all non-ν pixels as foreground fills all holes in Ω. The entire fill takes under 20 scan iterations for all images we examined. CUDA filling adds around 8 milliseconds/image of 1024^2 pixels in comparison with around 0.15 seconds/image for clas-

sical CPU flood fill (Tab. 6.2 (Holes)) up to a total of roughly 25 milliseconds per cluster per iteration. Note that, due to filling, all skeletons, and thus the created bundles, become trees rather than graphs. Although we do not use this property now, it may enable future interaction work such as user manipulation of the layout by means of bundle handles.

Clustering using HBA is fast. The CPU implementation in [40] constructs the complete dendrogram of a graph of 10K edges in 0.1 seconds on our considered machine. We next added the GPU-based clustering in [30], which is roughly 10 to 15 times faster. Note that only a few clustering passes are needed for a complete layout (Sec. 6.2.5). Also, we do not need to construct the entire dendrogram, but only the bottom-most part thereof, until we reach the cut value δ (Sec. 6.2.1) at which we extract the clusters to bundle further.

Finally, postprocessing (Sec. 6.2.6) poses no performance problems, so we implement it in real-time using standard OpenGL polyline rendering and CPU-based smoothing and relaxation. All in all, the CUDA-based bundling takes 5 to 30 seconds for producing a final layout for the graphs we tested (Tab. 6.1, right column), *i.e.* 25 milliseconds per cluster times the total number of clusters processed during the $I = 10$ iterations plus the clustering time. In terms of memory, our method is scalable: we only need a few 1024^2 images (distance and feature transforms and skeletons) and discard these once a cluster is processed; all paths between skeleton tips for the current cluster; and the graph edge polylines. For all graphs presented here, this amounts to under 100 MB total application memory requirements per graph.

6.3.2 *Parameter setting*

Our entire method has a few parameters: the clustering similarity threshold δ, edge advection factor α, total number of iterations I, and smoothing and relaxation amounts γ_s and γ_r. These parameters allow covering a number of different scenarios, as follows.

Clustering similarity threshold δ: This parameter specifies the granularity level at which we cut the cluster dendrogram to obtain sets of edges to bundle at the current iteration (Sec. 6.2.1). We set δ as a linearly decreasing function on the iteration number $t \in [1, I]$ from $\delta(1) = 0.95$ to $\delta(I) = 0.7$. This yields strongly

coherent clusters in the first iteration, regardless of the initial edge position distribution, and also *locally* strongly coherent clusters in the subsequent iterations (Sec. 6.2.5).

Edge advection factor α: The advection value $\alpha \in (0, 1)$ controls how much edges approach the skeleton at one iteration. This implicitly controls the bundling convergence speed. Too high values yield tight bundles and convergence after the first few iterations, which is fine for graphs which already have relatively grouped edges, but limits the freedom in decluttering complex graphs. Too low values allow the iterative process to adapt itself better to newly discovered clusters as the edges approach each other, but convergence requires more iterations. In practice, we set α as a linearly decreasing function of the iteration number from $\alpha(0) = 0.9$ to $\alpha(I) = 0.2$.

Number of iterations: In practice, after $I \in [10, 15]$ iterations, we obtain tight bundles of a few pixels in width for all graphs we worked with. This is expectable, given that $(1 - \alpha)^I$ becomes very small for $\alpha < 1, I > 10$. In practice, we always set $I = 10$ and then use smoothing and relaxation to interactively adjust the result as desired.

Smoothing: The smoothing amount $\gamma_s \in \mathbf{N}$ describes the number of Laplacian smoothing steps executed on the bundled layout (Sec. 6.2.6). Values $\gamma_s \in [3, 10]$ give an optimal amount of smoothing which keeps the structured aspect of the layout but eliminates the skeleton-like look. Larger values make our layout look similar to the force-directed method of [80]. In practice, we noticed that the smoothing amount strongly depends on the task at hand: In some cases, users attach semantics to the branching structure, *i.e.* want to clearly see which groups of edges get merged together, so no smoothing is needed. In the general case, however, the exact bundle merging events are not relevant, so we use by default $\gamma_s = 5$.

Relaxation: The relaxation amount $\gamma_r \in [0, 1]$ controls the interpolation between the fully bundled layout and original one (Sec. 6.2.6). Relaxation is most conveniently applied interactively, after a bundled layout has been computed. Values $\gamma_r \in [0, 0.2]$ give a good trade-off between bundling and overdraw.

Overall, the entire method is not sensitive to precise parameter settings. For the graphs in this chapter and other ones we in-

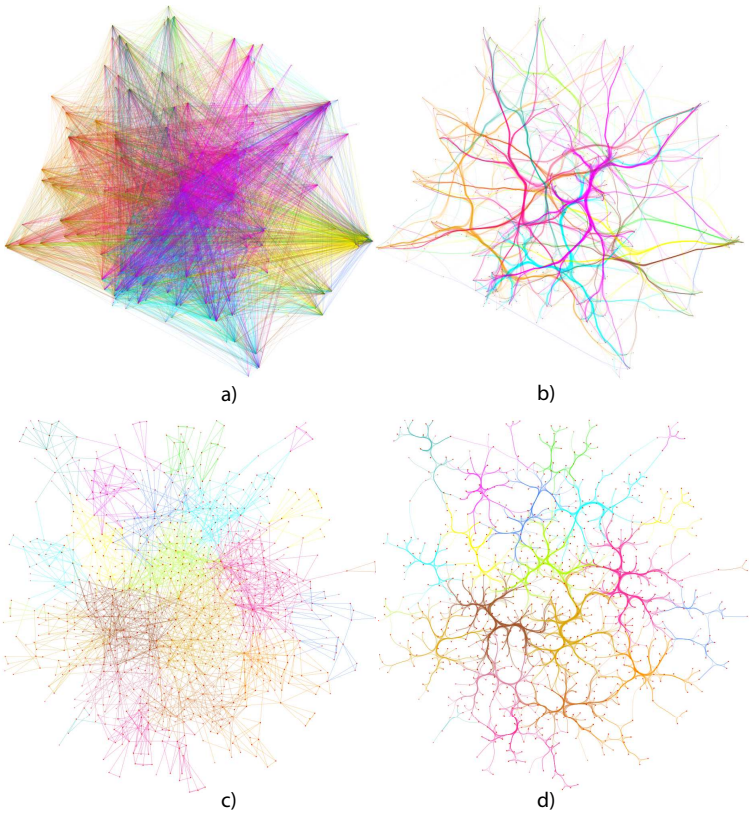

Figure 6.8: Air traffic graph (a: original, b: bundled). Poker graph (c: original, d: bundled). Colors in (a-d) indicate clusters (displayed for method illustration only).

vestigated, we have obtained largely identical bundled layouts with different parameter settings in the ranges indicated above. We explain this by the stability of the inflated shape skeletons to small local variations of the positions of edges, and the smoothing effect of the entire iterative process on the layout. As such, the only two parameters we expose to users are γ_s and γ_r, the others being set to predefined values as explained above.

6.4 APPLICATIONS

We now demonstrate our skeleton-based edge bundling (SBEB) method for several large, real-world, graphs. Statistics on these graphs are shown in Tab. 6.1.

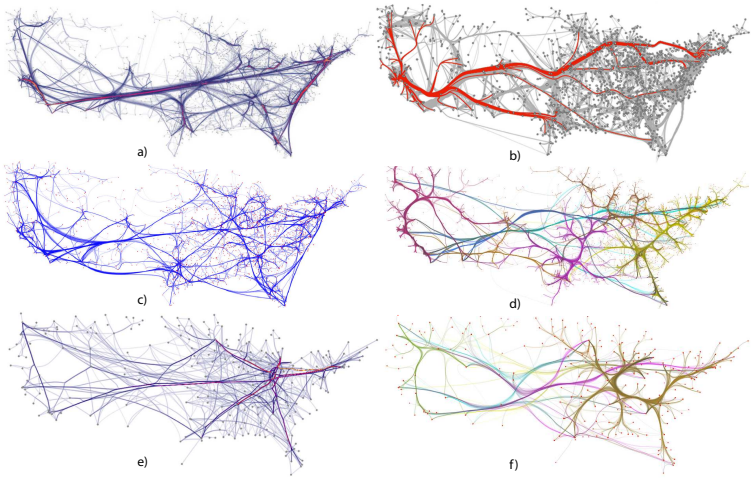

Figure 6.9: US migrations graph (a: FDEB, b: GBEB, c: WR, d: SBEB). US airlines graph (e: FDEB, f: SBEB). Colors in (d,f) indicate clusters (displayed for method illustration only)

Figure 6.8 and Figure 6.9 illustrate the SBEB and compare it with several existing bundling methods. Note that in all images here generated with our method, we used simple additive edge blending only, as our focus here is the layout, not the rendering. Fig. 6.8 a,b show an air traffic graph (nodes are city locations, edges are interconnecting flights). Fig. 6.8 c,d show a graph of poker players from a social network. Edges indicate pairs of players that played against each other. The node layout is done with the spring embedder provided by the Tulip framework [9, 10]. Given the average node degree and node layout algorithm used, related nodes tend to form relatively equal-size cliques. Bundling further simplifies this structure; here, bundles can be used to find sets of players which played against each other.

Fig. 6.9 a-d show the US migrations graph bundled with the WR, GBEB, FDEB, and our method (SBEB) respectively. Overall, SBEB produces stronger bundling, due to the many iterations $I = 10$ being used), and emphasizes the structure of connections between groups of close cities (due to the skeleton layout cues). If less bundling is desired, fewer iterations can be used (Fig. 6.4). Adjusting the postprocessing smoothing and relaxation parameters, SBEB can create bundling styles similar to either GBEB (higher bundle curvatures, more emphasis on the graph structure) or FDEB (smoother bundles). Finally, Fig. 6.9 e,f show the US airlines graph bundled with the FDEB and SBEB respec-

tively. SBEB generates stronger bundling (more overdraw) but arguably less clutter. Note also that SBEB generates tree-like bundle structures which is useful when the exploration task at hand has an inherent (local) hierarchical nature, *e.g.* see how traffic connections merge into and/or split from main traffic routes.

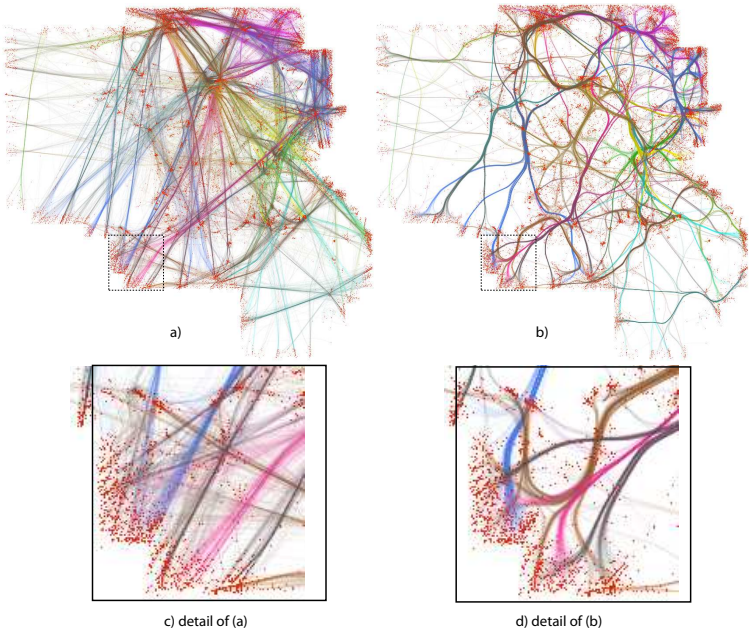

Figure 6.10: Bundling of airline trails (a,b) and details (c,d).

Figure 6.10 and Figure 6.11 show further examples. Fig. 6.10 a,b show flight paths within France, as recorded by the air traffic authorities [83]. Edge endpoints indicate start and end locations of flight records. The original edges are not straight lines, but actual flight paths (polylines). Note that this dataset is not a graph in the strict sense, since only very few edge endpoints are exactly identical within the dataset. This has to do with the fact that flight monitoring systems record flights (trails). However, edge endpoints are spatially grouped since flights typically start and end in geographically concentrated locations such as airports. Given this, our method is able to create a bundled layout of this dataset with the same ease as for actual graphs. Bundling puts close flight paths naturally into the same cluster. The bundled version emphasizes the connection pattern between concentrated take-off and landing locations, which are

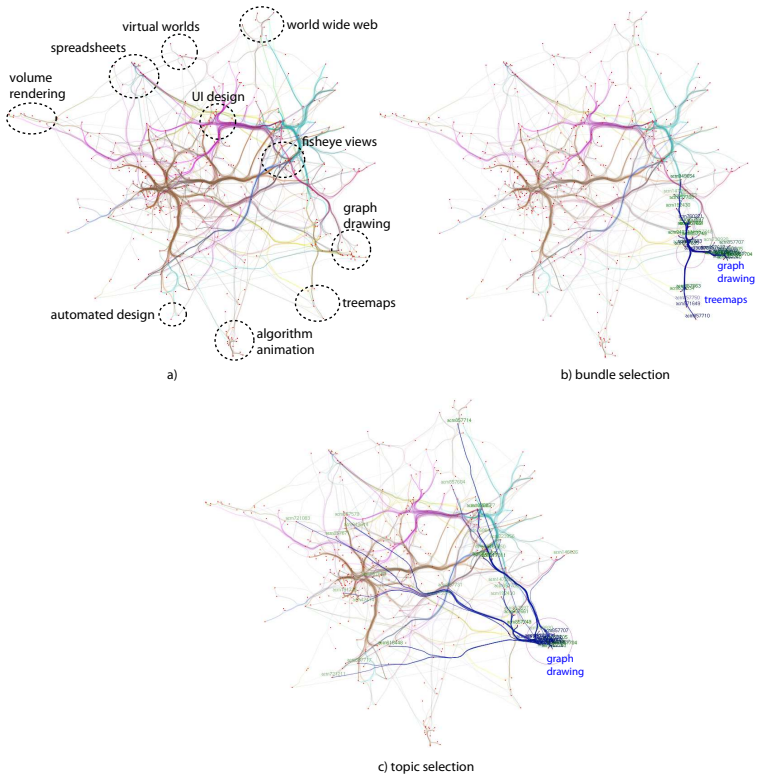

Figure 6.11: Bundling of citations graph (a). Selected bundle (in dark blue) shows citations involving two topics (b). Citations to a selected topic (c). In (b,c), node labels indicate edge direction (citing papers=green,cited papers=blue)

naturally the airports. The zoom-in details (Fig. 6.10 c,d) show the organic effect achieved by bundling.

Fig. 6.11 a-c show a citations graph (433 nodes, 1446 edges). Nodes are InfoVis papers, laid out according to content similarity: close nodes indicate papers within the same, or strongly related, topics. The layout algorithm used for the nodes is multidimensional scaling with least-square projection [132]. Paper similarity is measured using cosine-based distance between term feature vectors [147]. Topics were added as annotations to the image to help explanation. Bundling exposes a structure of the citations between topics. We use the bundle-based selection (Sec. 6.2.6.3) to highlight one of the bundles, which becomes now dark blue (Fig. 6.11 b). It appears that this bundle connects papers related to the Graph drawing and Treemap topics. The direction of edges is indicated by node label colors: citing pa-

pers are green, cited papers are blue. Green and blue labels are mixed within this bundle, which is expected, since papers in these two topics typically cross-reference each other. Fig. 6.11 c shows a selection of all edges which end at nodes within the ball centered at the mouse cursor. Concretely, we highlighted here all papers citing papers in the Graph drawing topic. Note that this selection is a purely node-based one, *i.e.* it does not use bundles for choosing the edges. However, bundles have now another use: they allow *highlighting* specific edges in the graph without increasing clutter, since these edges follow the already computed bundles. Also, note that for this type of node layout, our clustering-based bundling makes sense: edges will be grouped in the same bundle if they have similar positions, meaning start/end from similar topics; if the node layout effectively groups nodes into related topics, then bundles have a good chance to show inter-topic relations in a simplified manner.

6.5 DISCUSSION

In comparison to existing bundling techniques, our method has the following advantages and limitations:

Generality: Our method can treat directed or undirected graphs. By default, we assume the graph is directed, so edges running between the same sets of nodes in opposite directions will belong to different clusters, hence create different bundles. For undirected graphs, we only need to symmetrize the edge similarity function (Eqn. 6.1).

Structured look control: Users can control the 'structured look' of a bundled layout, ranging between smoothly merging bundles and bundles meeting at sharp angles, by manipulating a single parameter (smoothing γ_s, Sec. 6.2.6). This implicitly allows removing sharp ramifications when these are meaningless. Other methods, with the exception of HEB, do not allow explicit control of this aspect, since there is no explicit hierarchy aspect in the bundles. In our case, hierarchy is modeled by the cluster skeletons (at fine level) and by the progressively simplified cluster structures (at coarse level).

Robustness: Our method operates robustly on all graphs we experimented on, *i.e.* yields a set of stable skeletons and bundles

progressively converging towards an equilibrium state. This is explained by the regularization of the feature transform (Sec. 6.2.4) and the inherent robustness of the skeletonization method used (Sec. 6.2.3). Briefly put, adding or removing a small number of nodes or edges will not change the bundling since the distance-based shapes are robust to small changes in the input graph and so are their skeletons too.

Speed and simplicity: Due to the CUDA implementation of its core image-based operations, our method is considerably faster than [80] and slightly faster than [103]. However, we should note that it is not clear if the timings reported in [103] include also the cost of computing the Voronoi diagram underlying the grid graph. The only faster bundled method we are aware of is the MINGLE method [70], which takes 1 second for the US migrations graph and 0.1 seconds for the US airlines graph, in contrast to our 4.1 seconds and 6.3 seconds respectively. MINGLE and SBEB share some resemblance in bottom-up aggregation of edges, but also have some differences. MINGLE compares edges essentially based on end point positions, whereas we use the entire edge trajectory (which may allow us to bundle graphs with curved edges better). The complexity of MINGLE is $O(|E|\log|E|)$ for a graph with E edges, whereas SBEB is essentially $O(|C|)$ where C is the average cluster size. By using a better cluster selection than our current iso-linkage cut in the cluster tree (Sec. 6.2.1), it is possible to reduce $|C|$ and thus make SBEB faster.

Apart from this, our method works entirely image-based, rather than manipulating a combination of hierarchical image-based and mesh-based data structures. The CUDA-based image processing code used by our method is available at [176].

Apart from the above, there are several other differences between our method and recent edge bundling techniques. In contrast to force-directed bundling [80] which bundles pairs of edges iteratively, in a point-by-point manner, we bundle increasingly larger groups of edges (our clusters) along their common center in one single step, using skeletons. In the limit, our method can behave like the force-directed bundling, *i.e.* if we were to treat, at each iteration, only the most cohesive leaf cluster. However, this is practically not interesting, as it would artificially increase the computational cost without any foreseeable benefits. Further, while Lambert *et al.* [103] use shortest paths in a node-based grid graph to route edges, in our method edges bundle themselves using only edge information. As such, there

is no relation between the Voronoi diagrams used in [103] and our skeletons (which, formally, can be seen as a Voronoi diagram in which inflated edges are the sites). Distance fields and skeletons are also used in [178], but in different ways; first, an edge distance field is computed using a considerably less accurate quad-splat-based method, whereas the distance transform we use here is pixel-accurate. Secondly, skeletons are used as *shading* cues and not for layout, whereas here we use skeletons to actually compute edge layouts. In comparison to [133], where bundles split in exactly two sub-bundles, our bundle splits can have in general any degree, as implied by the underlying skeletons. Also, our method can handle general graphs.

Limitations: There is no fundamental reason why a skeleton-based layout should be preferable to other bundling heuristics, apart from the intuition that a skeleton represents the local center of a shape. Hence, the quality of our layouts (or any other bundled layout) is still to be judged subjectively. Moreover, any bundling inherently destroys information: edges are overdrawn, so cannot be identified separately; and edge directions are distorted. Hence, bundling should be used for those applications where one is interested in coarse-scale connectivity patterns *and* when one cannot apply explicit graph simplification *e.g.* due to the lack of suitable node clustering guidelines and metrics. If desired, SBEB can be modified to incorporate additional bundling constraints *e.g.* maximal deformation of certain edges - the skeletons provide only bundling *cues* but the attraction phase can decide whether, and how much, to bundle any given edge. In the longer run, it is interesting to use shape perception results from computer vision [37, 101] to quantitatively reason about the quality of a bundled layout. Here, our image-based approach may prove more amenable to quantitative analysis than other bundling heuristics which are harder to describe in terms of operators having well-known perceptual properties. However, this is a challenging task and requires further in-depth study.

6.6 CONCLUSION

We have presented a new method for creating bundled layouts of general graphs. We exploit the known property of 2D skeletons of being locally centered within a shape to create elongated shapes from a graph with given node positions, and use skeletons as guidelines to bundle similar edges. To guarantee the

stability and smoothness of the bundled layout, we regularize the feature transforms of 2D skeletons to eliminate singularities. Using an iterative process, our layout amounts to a sequence of edge clustering and image processing operations. We present a CUDA-based implementation which achieves comparable or higher performance than existing edge bundling methods, but keeps implementation simple. Finally, we present a simple and efficient scheme to emphasize edge bundles using shaded cushion techniques computed directly on the bundled edges.

In future work, the geometric properties of 2D skeletons could be further exploited to generate bundled layout variations. Modifying the Euclidean distance metric, we could create constrained-angle skeletons leading to layouts similar to cartographic diagrams [167]. Apart from that, bundle-to-bundle and bundle-to-node distance fields could be used to globally optimize the layout of different edge bundles in order to maximize readability and allow for the introduction of spatial constraints such as labels, bundle crossing minimization, and node-edge overlap reduction. Such an application is presented next in Chapter 7. In the longer run, a promising work direction would be to study the optimality criteria of bundled layouts by using existing results from shape perception in computer vision which are directly applicable to our skeleton-based layout method.

This chapter is based on:

Ozan Ersoy, Christophe Hurter, Fernando V. Paulovich, Gabriel Cantareira, and Alexandru Telea. Skeleton-Based Edge Bundling for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, **17(12)**, 2364-2373 (2011).

7

GRAPH BUNDLING BY KERNEL DENSITY ESTIMATION

ABSTRACT: *In Chapter 6, we have shown that graph bundling can be formulated as an image processing problem by the use of distance transforms and 2D skeletons. Bundling is defined as the process of iteratively moving edges to the local maxima of their spatial density, which is computed using shape skeletons. In this chapter, we further refine this observation and reformulate the graph bundling problem as an iterative edge-density sharpening process. To this end, we present a fast and simple method to compute bundled layouts of general graphs. For this, we first transform a given (straight-line) graph drawing into a density map using kernel density estimation. Next, we apply an image sharpening technique which progressively merges close height maxima by moving the convolved graph edges into the density gradient. Our technique can be easily and efficiently implemented using standard graphics acceleration techniques and produces graph bundlings of similar appearance and quality to state-of-the-art methods at a fraction of the cost. Additionally, we show how to create bundled layouts constrained by obstacles and use shading to convey information on the bundling quality. We demonstrate our method on several large graphs and networks.*

7.1 INTRODUCTION

In Chapter 5, we have presented a first application of image-based operations such as distance functions and shape skeletons for the simplification of edge bundling layouts (EBLs). In Chapter 6, we have taken this process a step further, by showing how similar operations can be efficiently and effectively used for the creation of EBLs from a given straight-line graph.

In this chapter, we take our image-based bundling approach a step further. Specifically, we exploit the observation on which the work in Chapters 5 and 6 is based, *i.e.* that bundles are centered structures within the drawing of a straight-line graph. As such, we now propose to construct EBLs directly from such a drawing, without the need for an additional image segmentation and skeletonization step, as we did describe in Chapter 6. Also, in contrast to the techniques presented in Chapters 5 and 6, we now work entirely image-based: Given a graph drawing, we first convolve the edges with a special kernel to construct a density map. Next, we advect edges in the gradient of this map and iterate the process for a few steps with decreasing

kernels. This delivers a layout with well separated and smooth bundle structures. Separately, we modify our density map to obtain bundles which avoid user-specified obstacles of arbitrary sizes and shapes. Finally, we propose a new shading technique which conveys the bundling quality in an easy to interpret way. Our contributions are as follows:

- a bundling technique for general graphs which is robust, simple to implement, and up to one order of magnitude faster than state-of-the-art techniques;
- a technique to generate bundled layouts that smoothly avoid obstacles of arbitrary shape and position;
- a way to visually convey bundling quality via shading.

The structure of this chapter is as follows. Section 7.2 presents our new bundling method. Section 7.3 details implementation and shows results on real-world graphs. Section 7.4 presents our obstacle-driven bundling and bundling quality visualization. Section 7.5 discusses our method. Section 7.6 concludes the chapter.

7.2 ALGORITHM

Most general-graph bundling methods (Chapter 2) use edge-to-edge neighborhood information: Given a graph drawing $G \subset \mathbf{R}^2$ and a point $\mathbf{x} \in G$, we can think of bundling as an operator $B : \mathbf{R}^2 \to \mathbf{R}^2$ which displaces $\mathbf{x}$ based on the spatial information in $G \cap \nu_\epsilon(\mathbf{x})$ where $\nu_\epsilon(\mathbf{x})$ is a small neighborhood centered at $\mathbf{x}$. The result $B(G)$ is a new layout whose edges are gathered in dense groups (bundles) separated by low edge-density areas (white space) to minimize drawing ink. Intuitively, we can see B as an image processing function which *sharpens* the local spatial density ρ of edge points.

We model ρ using kernel density estimation (KDE) methods [158]: Given a graph drawing $G = \{e_i\}_{1<i<N}$ consisting of edges $e_i \subset \mathbf{R}^2$, we can estimate $\rho : \mathbf{R}^2 \to \mathbf{R}^+$ as

$$\rho(\mathbf{x}) = \sum_{i=1}^{N} \int_{\mathbf{y} \in e_i} K\left(\frac{\mathbf{x}-\mathbf{y}}{h}\right) \tag{7.1}$$

where $K : \mathbf{R}^2 \to \mathbf{R}^+$ is a density kernel of bandwidth $h > 0$. Typical kernel choices are Gaussian and Epanechnikov (quadratic)

functions. The density map ρ can be computed by convolving G with K, or building an accumulation map of K over G.

The density map ρ reflects the local edge density. A graph drawing with uniformly distributed edges yields a flat map. Large ρ values are zones of high edge density. More interestingly, local maxima of ρ are located roughly in the *middle* of local edge agglomerations. In Chapter 6, we have shown that these are good positions for placing edge bundles, and compute these points as the medial axes, or skeletons, of the Euclidean distance transform of G thresholded at a small value $\tau > 0$. In contrast, here we define bundling centers as the local maxima of a continuous density map computed with nonlinear kernels. As we shall see later, this implies several differences and advantages for our method.

Given the density map ρ, we next define our kernel density estimation edge-bundling (KDEEB) operator B as the solution of the following ordinary differential equation

$$\frac{d\mathbf{x}}{dt} = \frac{h(t)\nabla\rho(t)}{\max(\|\nabla\rho(t)\|, \epsilon)} \tag{7.2}$$

for all points $\mathbf{x}$ in the graph drawing, with initial conditions given by the input graph. The density gradient $\nabla\rho$ is normalized in a regularized manner – the $\epsilon = 10^{-5}$ denominator value takes care of zero gradients. Normalizing $\nabla\rho$ constrains the movements $\|d\mathbf{x}\|$ to the kernel bandwidth $h(t)$. Since $h(t)$ decreases in time (as explained next), this stabilizes the advection process. Eqn. 7.2 is solved by Euler integration, *i.e.* we construct $B(G)$ by iteratively computing the density map ρ and advecting the points $\mathbf{x} \in G$ in the direction of $\nabla\rho$. The effect of Eqn. 7.2 is to sharpen the density ρ starting with the (typically straight-line, unbundled) input graph G and ending with a tightly bundled graph whose density map asymptotically reaches bundle-aligned Dirac impulses.

The choice of the kernel K and bandwidth h are discussed next. We use an Epanechnikov kernel $K(\mathbf{x}) = 1 - \|\mathbf{x}\|^2$, which optimally approximates the ρ in a minimal variance sense [54, 86]. At each step i of the numerical integration, we decrease h by a geometric series $h_i = \lambda^i h_{max}$, where h_{max} is the initial kernel bandwidth, set to the average inter-edge distance in the input graph G, and λ is a kernel bandwidth reduction factor. Setting $\lambda \in [0.5, 0.9]$ yields a kernel size which follows the average edge density. The initial value h_{max} creates a smooth density ρ where any edge point is influenced by at least one

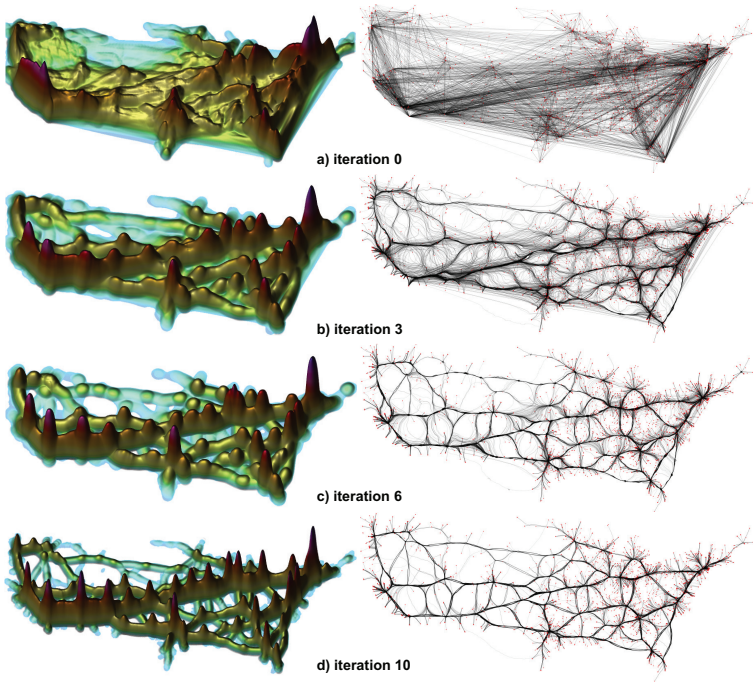

Figure 7.1: Evolution of density map and corresponding bundling for the US migrations graph.

other edge and also avoids density overestimations. During integration, edges get closer, so we decrease the kernel h_i to avoid density overestimation. Decreasing h_i also decreases the advection speed, which stabilizes the process as the signal ρ is increasingly 'sharpened'. In other words, edges converge towards the local density maxima instead of jumping from one side to the other of such maxima. More advanced methods for estimating the kernel bandwidth, such as data-based adaptive selectors can be used, if desired [154, 86]. However, we do not need an *exact* density estimation for graph bundling since we only use the density's *gradient* and recompute the density iteratively, so our simple heuristic suffices.

Figure 7.1 shows several iterations of the density map, drawn as a height plot (normalized in height for display) and corresponding bundled layouts for the US migrations graph [80, 55]. The density map gets sharper during the iterative solving of Eqn. 7.2. This bundles edges along the density local maxima. As the density map gets sharper, the average distance between

local maxima increases, so bundles get tighter and separated by more white space.

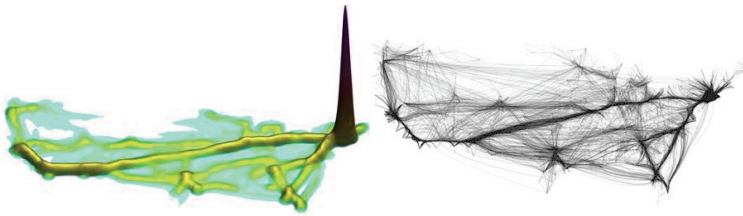

Figure 7.2: Density map (left) and corresponding bundling for non-normalized advection (compare to Fig. 7.1)

Figure 7.2 shows iteration 10 of bundling the same graph, this time without gradient normalization (Eqn. 7.2). Compared to Fig. 7.1, the local maxima vary more, *i.e.* edge density non-uniformities in the input graph get amplified during the bundling. Edges close to the high peak top-right in Fig. 7.2 get bundled strongly, while other edges converge very slowly.

7.3 IMPLEMENTATION

An efficient implementation of our method uses a GPU image-based approach, as follows (see also Fig. 7.3).

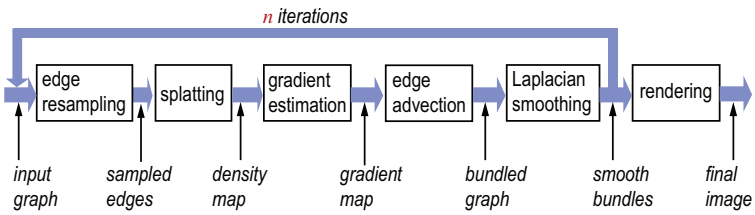

Figure 7.3: KDE edge bundling pipeline.

7.3.1 *Graph representation*

First, we discretize all edges e_i of the input graph into sets of points $\mathbf{x}_{ij}$, by using a small sampling step δ equal to roughly 1% of the size of the graph's bounding box, similarly to other methods [80, 78, 55, 103]. This typically yields several tens of sample points per edge on average.

7.3.2 *Density computation and gradient estimation*

To compute the density map ρ (Eqn. 7.1) and gradient $\nabla\rho$, we can splat the kernel K, precomputed into an OpenGL 2D luminance texture, at all edge sample points $\mathbf{x}_{ij}$, and accumulate results into a floating-point buffer by additive blending. Maximal efficiency is achieved by drawing OpenGL point sprites scaled by the bandwidth h_i (Sec. 7.2). The accumulation buffer size matches the screen size. From this accumulation map, we compute $\nabla\rho$ by finite differences. A more accurate way is to precompute $\partial K/\partial x$ and $\partial K/\partial y$ as two separate luminance textures and accumulate the two components of $\nabla\rho$ by splatting the two textures separately. The two approaches are identical speed-wise: The former uses two passes (accumulate, compute gradient); the latter uses a single pass but creates two separate accumulation maps.

A better approximation of the kernel density estimation (Eqn. 7.1) is obtained if we use edge-aligned kernels. For this, we use elliptical kernels aligned with the edge segments $(\mathbf{x}_{ij}, \mathbf{x}_{ij+1})$, *i.e.* draw rectangles textured by the radial kernel K centered at the edge sample points, aligned with the edge segments, and of size h (across the edge) and equal to the average of $\|\mathbf{x}_{ij} - \mathbf{x}_{ij+1}\|$ (along the edge). Another option is to use one-dimensional half-kernels stored as 1D textures and drawn as rectangles tangent to the edge segments. In Chapter 5, we used the latter method, with a different (distance) kernel, to create distance profiles. Edge-aligned kernels allow a lower edge sampling rate, since kernels are scaled separately along and across edges, thus increase splatting speed without decreasing the KDE quality.

7.3.3 *Advection*

After obtaining the gradient of our edge density map, we advect each edge by Euler integration of Eqn. 7.2 on the edge sample points $\mathbf{x}_{ij}$. Edge endpoints are kept fixed. Since we first compute the gradient map and then advect all edge points, integration is explicit, which parallelizes easily. After each advection step, we resample the edges (Sec. 7.3.1). This is needed since $\operatorname{div} \nabla\rho \neq 0$ *and* edge endpoints are fixed, so advection stretches and/or shrinks edges, which can lead to edge self-intersections or subsampled edge fragments.

7.3.4 *Smoothing*

After each iteration, we do 5..10 Laplacian smoothing iterations of the advected edges with a kernel of fixed size, roughly 8δ, similar to [80]. This removes the small-scale advection artifacts caused by the imprecise estimation of the density map ρ which is due to errors in the kernel bandwidth estimation (Sec. 7.2), on the one hand, and to discretization errors in the finite edge sampling and finite kernel splat texture resolution (Sec. 7.3.2), on the other hand. Artifacts show up as small-scale undulations in the density map, which cause extra divergence points, *i.e.* slight rotations, of $\nabla\rho$. In turn, gradient imprecisions cause edges to become jagged during advection, thus yield slight zig-zags in the final bundles. Laplacian smoothing completely removes this problem and generates smooth bundles. Our smoothing is equivalent to anisotropically filtering the density map, prior to gradient estimation, with a kernel aligned with the map's curvature minor eigenvector, *i.e.* along its ridges [211]. However, this type of image filtering is considerably more expensive, and more complicated, than our Laplacian edge smoothing.

7.3.5 *Iterative bundling*

For all tested graphs, 8..10 iterations of gradient computation, advection, and smoothing yields a stable layout. The process is monotonic: edges move in a single direction rather than back-and-forth. This is due to the structure of the density map gradient: If two edge points $\mathbf{x}, \mathbf{y} \in G$ are within each other's bandwidths at an iteration, both are equally advected towards the midpoint $(\mathbf{x}+\mathbf{y})/2$, since we use the same kernel size and shape at all points.

7.3.6 *Examples*

Figure 7.4 and Figure 7.5 compare our KDEEB with recent bundling methods: FDEB [80], GBEB [39], SBEB [55], and WR [103]. Overall, we produce tighter bundles than FDEB and GBEB, and smoother bundles than SBEB. While SBEB requires an edge pre-clustering on similar directions and positions (Fig. 7.4 a,c,f and Fig. 7.5 a), we obtain similar or better results, *i.e.* tight, smooth, well-separated bundles, with no clustering at all. If edge clusters are provided, we can use these by bundling each cluster separately. For example, in Fig. 7.4 (a,b), which shows a software

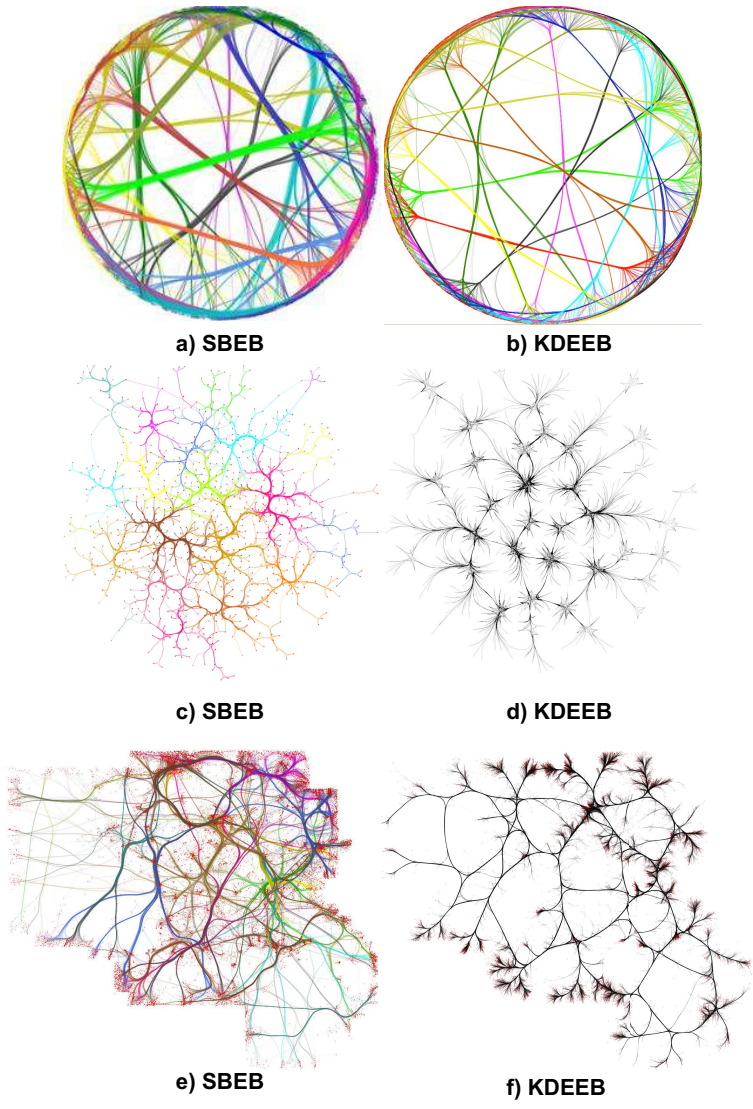

Figure 7.4: Bundling examples. Radial graph (a,b); Poker graph (c,d); France airlines (e,f). Colors mark different edge clusters.

dependency graph with edges grouped by structural similarity, KDEEB delivers better separated bundles, than SBEB. Also, compare Fig. 7.5 a (US migrations graph, pre-clustered on edge similarity, bundled with SBEB) with KDEEB where we bundle each cluster separately (Fig. 7.5 b). Our result is more similar to bundlings which do not use clustering (*e.g.* our method, Fig. 7.5 d or WR, Fig. 7.5 f) than to SBEB. This indicates that our

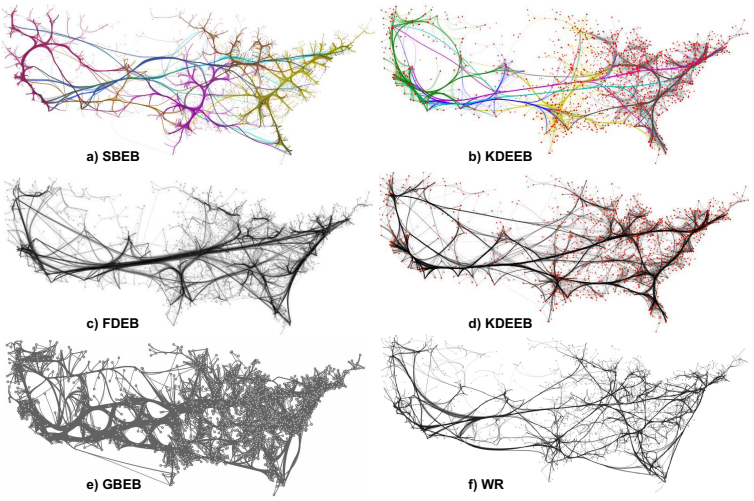

Figure 7.5: Bundling examples. US migrations, clustered (a,b); US migrations, unclustered (c,d,e,f); Colors mark different edge clusters. More examples at www.cs.rug.nl/svcg/Shapes/KDEEB

method could be used in cases where we want to bundle parts of a graph separately, *e.g.* interactive exploration or online graph bundling. Per-cluster bundling does not decrease the speed of our method, since its complexity is $O(EI/\delta)$ for a graph with E edges, I bundling iterations, and an edge sampling step δ. Figure 7.6 shows the US airlines graph bundled by FDEB, SBEB, MINGLE, and our method. Again, our results are tighter and arguably less cluttered than other methods.

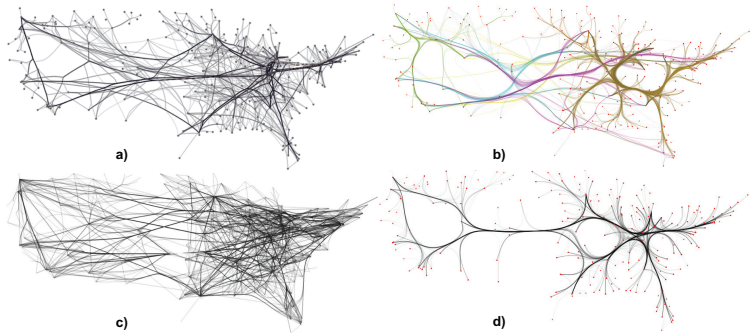

Figure 7.6: Bundling examples. US airlines (FDEB (a), SBEB (b), MINGLE (c), KDEEB (d)).

7.4 ADDITIONS

We describe next two visual additions for bundled graphs that are easily added atop of our bundling method: obstacle-constrained bundles and visualizing bundling quality.

7.4.1 *Obstacle-constrained bundles*

Often, a layout needs to avoid some areas in the embedding space, *e.g.* labels, icons, or other zones of interest. Although many methods for laying out graphs with spatial constraints exist, this use case has not been studied, to our knowledge, for bundled layouts. We next present such an approach.

Given a set of 2D *obstacles* $\Omega_{1 \leqslant i \leqslant B} \subset \mathbf{R}^2$, we want to create a bundled layout which (a) follows the general paradigm of bundling close edges into smooth and tight bundles, and (b) routes bundles around obstacles without creating sharp bends or lengthening the bundles needlessly. Obstacles are shapes of arbitrary geometry and topology, *e.g.* can have dents, protrusions, or holes, and can be placed freely. We model such shapes as binary images, with foreground pixels (Ω) inside the shape and background pixels ($\overline{\Omega}$) outside.

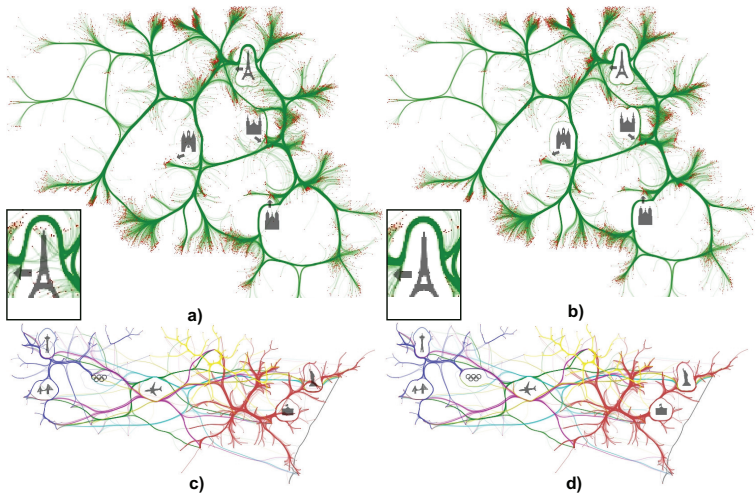

Figure 7.7: Obstacle-constrained bundling without endpoint displacement (a,c) and with endpoint displacement (b,d).

To constrain bundles, we modify the density ρ used by our method. Instead of the density ρ in Eqn. 7.1, we use now

$$\rho_{obs} = \rho - DT\left(T\left(DT\left(\cup_{i=1}^{B}\overline{\Omega}_i\right), \tau\right)\right) \tag{7.3}$$

where $DT(\Omega) : \Omega \rightarrow \mathbf{R}^+$ is the *distance transform* (DT) of the shape's boundary $\partial\Omega$ [37], computed on the foreground, and $T(\cdot, \tau)$ is the lower thresholding of a DT with a value τ. Hence, we subtract from ρ the DT of an inflated version $\Omega_{infl} = T(DT(\overline{\Omega}), \tau)$ of our obstacle Ω with a distance τ. Since the gradient $\nabla DT(\Omega)$ is a vector that points from each point $\mathbf{x} \in \Omega$ to the closest point on $\partial\Omega$ to $\mathbf{x}$, by using ρ_{obs} instead of ρ in Eqn. 7.2, we force edges that cross obstacles to move in the shortest direction towards the obstacles' boundaries, *i.e.* route edges outside obstacles with minimal stretching. Once edges exit an obstacle, this repelling effect ceases, since $DT(\Omega) = 0$ outside obstacles. For shapes with sharp convex corners, $\nabla DT(\Omega)$ is not a smooth field: $\nabla DT(\Omega)$ has discontinuities along the skeleton of $\partial\Omega$, which in turn has one separate branch for each such corner [37, 179]. However, such discontinuities create no kinks or sharp bends in the advected edges, for several reasons. First, outside obstacles, edges are only influenced by the smooth $\mathcal{C}^\infty$ component ρ. Secondly, since we use *inflated* obstacles Ω_{infl}, any corners are rounded out, so edges never get sharp bends when following the obstacles' contours. This matches our goal of smooth obstacle avoidance. The parameter τ (10..20 pixels) controls how much corners are smoothed, and also creates a thin halo-like band between the routed edges and the obstacles, which helps better separating the former from the latter.

This method has one singular case. Consider a rectangle Ω crossed by an edge which is parallel to, and far from, its short sides. The edge is parallel with $\nabla DT(\Omega)$, so it only gets shifted tangentially by ρ_{obs}. Laplacian smoothing (Sec. 7.3) eliminates tangential shifts, so the edge never exits Ω.

We solve this problem as follows (see Fig. 7.8). For each edge e that crosses an inflated obstacle Ω_{infl}, we compute the intersection points $\{\mathbf{p}_i\} = e \cap \partial\Omega_{infl}$. For simplicity, we next consider that there are only two such points $\mathbf{p}_1$ and $\mathbf{p}_2$; the method works the same for more intersection points. We compute the shortest pixel path $\gamma \subset \partial\Omega_{infl}$ between $\mathbf{p}_1$ and $\mathbf{p}_2$. If there are two such paths, we take any of them. Next, we replace the edge segment $e \cap \Omega$ inside the obstacle with γ. This pushes e outside Ω_{infl} with a minimal deformation. Finally, we apply

Laplacian smoothing on e (Sec. 7.3.4), but forbid the smoothed points to re-enter $\Omega_{\mathtt{infl}}$. This effectively rounds *concave* corners made by e as it follows $\partial\Omega_{\mathtt{infl}}$. Since *convex* corners are already rounded off by using the inflated version of Ω, we obtain edges that smoothly avoid obstacles.

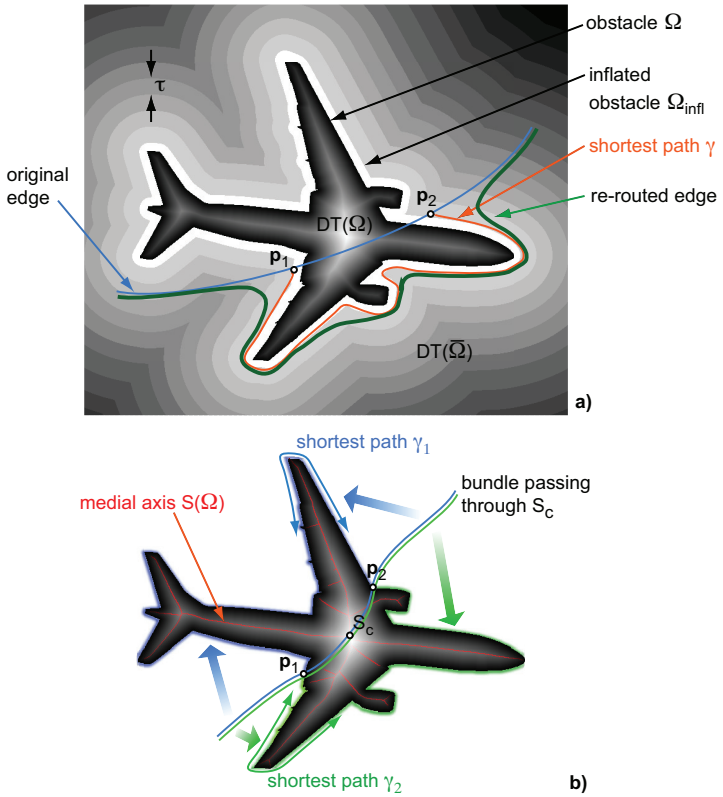

Figure 7.8: a) Obstacle-constrained bundling refinement; b) Bundle splitting singularity. The background shows the shape's distance transform for illustration (Sec. 7.4.1).

Figure 7.7 shows several obstacle-constrained bundles. Images (a,b) show our method on the France airlines graph (Sec. 7.5.2), with and without endpoint displacement. Icons show cities close to large flight endpoint agglomerations. Images (c,d) show obstacle avoidance on the US airlines graph (Sec. 7.5.2). Edges starting or ending inside an obstacle are routed straight to the obstacle boundary, after which they follow the bundle they are part of. If we allow node displacement, endpoints inside obstacles are moved too. The technique works both with our new bundling (Sec. 7.2, images (a,b)), but also on graphs

bundled by other methods, *e.g.* Fig. 7.7 c,d whose bundling was generated by SBEB presented in Chapter 6.

Finally, we present a different type of obstacle avoidance: global whole-area avoidance, or *outward bundling*. In this use-case, we want to create a bundled layout where bundles are routed, if possible, outside the entire area where nodes are placed. This frees up space close to and/or between nodes which can be used to show other information *e.g.* maps, annotations, or different types of (unbundled) edges. In contrast to obstacle avoidance, this is a global process, as we now want to avoid an entire, large, area rather than isolated obstacles. We achieve this by shifting the splat kernels (Sec. 7.3.2) slightly along the vector between the barycenter of the graph node positions and the position of the current splatting point. This effectively offsets the kernels outside the edges, and thus pulls the edges globally away from the graph center. Edges which connect nodes *radially*, *i.e.* in directions roughly leading to the barycenter, will stay unchanged. Bundles which connect nodes at relatively similar distances from the graph center will, however, be repelled further from this center. Figure 7.10 b shows this technique on the France airlines graph. We see that, even though the bundle constraints are large, bundles stay coherent but get routed outside the nodes' agglomerations, if possible. The inner space thus freed can be used for additional visualizations. Implementation-wise, this technique is trivial, as it requires only shifting the splatting locations in a given direction when evaluating Eqn. 7.1.

Obstacle avoidance is simple to implement: We compute the obstacles' distance transforms, inflations, and shortest boundary paths using the AFMM method [179] on images up to 1000^2 pixels in subsecond time. If higher speed is needed, a CUDA version hereof can be used, which takes under 10 milliseconds on modern graphics cards [55].

Obstacle avoidance can be done during, or after, bundling. In the former case, obstacles affect bunding: different edges may get bundled than when no obstacles are used. In the latter case, same-bundle edges get re-routed together on the same side of an obstacle, which keeps bundled edges together *except* in the rare case when a bundle intersects the center of the obstacle's medial axis $S(\Omega)$ (Fig. 7.8 b). In this case, edges which intersect $S(\Omega)$ on different sides of S_c are re-routed to the two different shortest paths along the obstacle's boundary (blue and green curves γ_1 and γ_2 in Fig. 7.8 b). This creates a natural bundle 'flow' around the obstacle.

7.4.2 *Visualizing bundling quality*

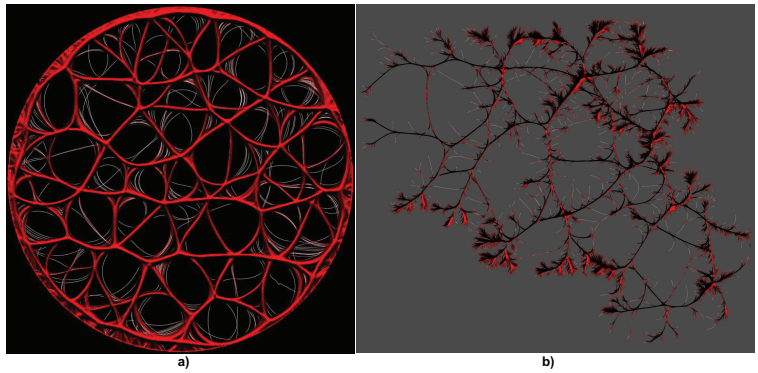

Figure 7.9: Bundling quality visualized by shading. Shaded colorful structures indicate dense bundles. Outlier edges are white. Radial graph (a), France airlines (b)

Given any bundling method, how to measure its quality? One can measure the results' fitness for a given task *e.g.* by user studies. Secondly, one can measure the quality of the produced images by some given image metrics. For the latter approach, little work exists so far. We use here the second approach: We model a graph's bundling *strength* by measuring how densely packed its edges are. Areas with high edge density, separated by areas with zero density, indicate strong, clearly delimited, bundles, and minimize ink [70]. Low edge density areas indicate spurious edges which could not be bundled. These are either limitations of the bundling method or actual data *outliers*, *i.e.* edges with no other similar-direction edges in their proximity.

We address the above as follows. We compute our density map ρ, we compute its normal $\mathbf{n}$, and next its Phong shading, with diffuse color set to a user-chosen 'graph material' color and specular strength inversely proportional to the density ρ. This creates two effects. First, strong bundles appear as shaded cushions in the graph's color, similar to [178, 55]. Secondly, outlier edges appear as strongly specular (*e.g.* white). Edges are rendered as lines with classical alpha blending and shading applied at the edge sample points $\mathbf{x}_{ij}$. Technically, this method is simpler than the image-based shading in [178]: We only need to apply Phong lighting to the edge sample points, whereas in [178] we construct 2D shaded bundle images by means of splatting, thresholding, and skeletonization. Thin (outlier) edges appear clearly in our shading here, whereas in Chapter 5 we

only shade bundles having a minimal thickness of several pixels.

Figure 7.9 shows two examples. The first graph (a) encodes software dependencies *i.e.* nodes are functions and edges are function calls. Shaded red structures show strong bundles indicating groups of functions *i.e.* software subsystems calling each other. These are clearly separated from outlier, unbundled, edges (white). We see that many edges are not bundled. In Fig. 7.9 b (France airlines graph), most edges are well bundled, as there are very few white outliers. Note that the above visualization is just an aid to reflect on the bundling strength and not a self-contained bundled graph visualization technique in itself: To be effective, it should be combined with suitable shading showing edge types, directions, and nodes.

7.5 DISCUSSION

7.5.1 *Comparison*

Several differences are visible between our method *vs* existing methods (Fig. 7.4 and Fig. 7.5): We produce smoother, less twisting, bundles than GBEB and SBEB, and tighter bundles than FDEB and MINGLE. Figure 7.10 a shows the effect of edge-aligned kernels (Sec. 7.3.1): The obtained bundling (US migrations graph) resembles now more the style of GBEB (Fig. 7.5 e) than the smooth style of FDEB or WR (Fig. 7.5 c,f).

Figure 7.10 c shows bundling of a synthetic graph of 100K edges with nodes randomly placed in a square. The result is a set of well structured, smooth, bundles, with little clutter. There is no semantic associated to such bundles, since our graph was random. However, this shows that KDEEB can effectively declutter and bundle very dense graphs.

Our bundling (Eqns. 7.1 and 7.2) shares some aspects with FDEB [80] and SBEB [55]. As FDEB, we move edge points close to each other, but we do not need any additional edge compatibility metrics ([80], Sec. 3.2). As SBEB, we move edges close to their local center. While SBEB computes this center *explicitly* as medial axes of thresholded distance functions of similar-direction edges, we move edges towards their *implicit* local center via the density map gradient. Eqn. 7.2 resembles solving the Eikonal equation [179], as we move edges with equal speed along a radial kernel gradient, which resembles the gradient of an Euclidean distance map. However, we recompute this gradi-

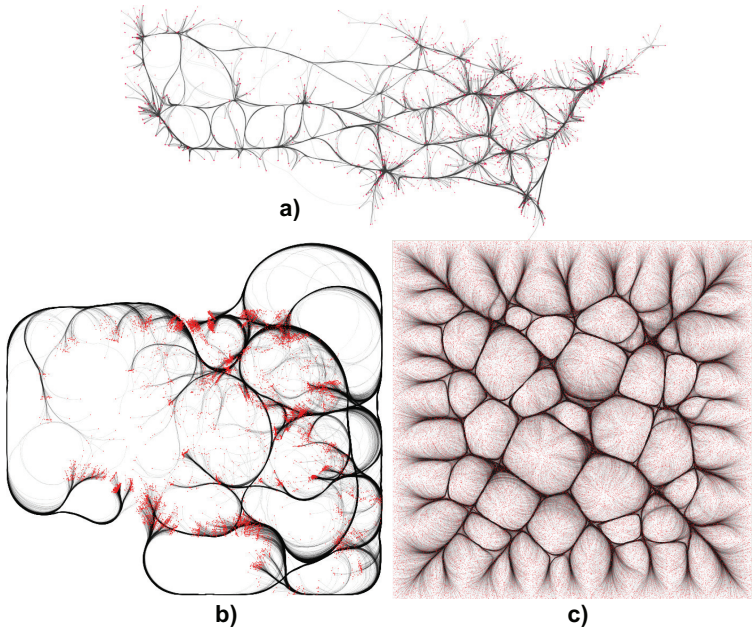

Figure 7.10: Additional examples. GBEB-style layout (a); Outward bundling (b); Random 100K edge graph bundling (c).

ent at each step, while [179] uses a fixed motion direction given by an explicit initial boundary.

GPU image-based techniques based on a density map computed from a graph drawing are also used by [64]. However, the aim is different: We 'concentrate' the density signal, and keep nodes fixed, to bundle edges, while [64] works in the opposite direction, spreading nodes towards less dense areas in order to declutter a given layout.

7.5.2 *Performance and simplicity*

Our entire bundling code is under 1000 lines of C#, and consists of four simple steps: density computation (Sec. 7.3.2), edge advection (Sec. 7.3.3), and edge smoothing (Sec. 7.3.4). Compared to other bundling methods whose implementations we could study [80, 103, 55], our pipeline is simpler, *e.g.* we do not require graph clustering, skeletons, Voronoi diagrams, or spatial search structures. We only use OpenGL 1.1 as compared to the more complex CUDA or pixel shader code in [103, 55].

Graph	Nodes	Edges	Edge samples	Bundling time (sec.)	
				8800 GTX	GeForce 580
US airlines	235	2099	86K	1.4	0.5
US migrations	1715	9780	220K	3.6	1.5
Radial	1024	4021	290K	4.5	1.5
France air	34550	17275	330K	3.8	1.8
Poker	859	2127	50K	0.8	0.4
Random	200K	100K	4.8M	43	18

Table 7.1: Graph statistics for datasets used in this chapter.

Table 7.1 shows running times on two Nvidia cards, both on a 3.3 GHz Core i5 PC, for 10 iterations. The *Edge samples* column shows the number of sample points on all graph edges. Advection, resampling, and smoothing are done in C# on 4 threads, which takes about 40% of the entire time, the remainder being OpenGL-based splatting. These steps can be easily accelerated further with *e.g.* vertex shaders or CUDA. However, even without this extra boost, KDEEB is much faster than similar approaches - on average for the tested graphs, 16 times *vs* FDEB [80], 6 times *vs* GBEB [39], 5 *vs* than SBEB [55], and 4 *vs* WR [103]. The only faster bundling method we know is MINGLE [70]: 2..3 times faster than KDEEB for graphs up to 2000 edges, and about the same speed for larger graphs. The lower performance of KDEEB for small graphs is due to the relatively large amount of work done in C# on the CPU for these graphs, which gets dominated by GPU computations for larger graphs. Also, MINGLE arguably produces more cluttered, less bundled, layouts (Fig. 7.6 c vs Fig. 7.6 d), as it uses only the start and endpoints of edges to bundle these, whereas we use the entire edge paths.

Memory-wise, we only need to store three frame buffers equal to the screen size (density map and its two gradient components). This means practically zero data overhead atop of the edge samples which describe the bundled layout.

7.6 CONCLUSION

We have presented a new method for creating bundled layouts of general graphs. Our approach offers a simple, (GPU) parallelizable method which is several times faster, and arguably

simpler to implement, than comparable methods. Our method produces bundled graph layouts with tight and smooth structures, robustly handles graphs of widely variable complexity and size, and requires no complex user parameter settings. We show how to constrain bundling to avoid arbitrary-shaped obstacles placed in the embedding space at user-selected positions, and also a way to globally route bundles outside the nodes' position area. Our approach, which follows an image sharpening technique, opens new ways for analyzing and refining graph bundling based on well understood image processing techniques.

Compared to the skeleton-based edge bundling (SBEB) technique presented in Chapter 6, we outline several similarities and differences. At a technical level, we also use the density map of the edges of the input graph to compute the bundling locations. At a conceptual level, we also construct bundles at the centers of shapes inferred from this density map. The key difference is that, while SBEB constructs these centers *epxplicitly*, by first clustering the graph edges (to construct separate groups) and next computes the shapes corresponding to these groups by density thesholding, and their respective centers by skeletonization, we now determine the shapes and their centers *implicitly*, as the local maxima of the density map. This makes our current method considerably simpler and faster, as we do not need to cluster edges, threshold the density graph, and compute shape skeletons.

Several future work directions exist. Speed-wise, our method can directly use a fully-parallel (*e.g.* CUDA) optimization. Secondly, by modifying the splat kernels, different bundling styles could be obtained *e.g.* orthogonal layouts. Last but not least, our image sharpening technique may have direct applications in image processing and simplification, beyond the confines of information visualization.

This chapter is based on:

Christhophe Hurter, Ozan Ersoy, and Alexandru Telea. Graph Bundling by Kernel Density Estimation. *Computer Graphics Forum* **31**, 865-874 (2012) *(cover image on Proc. EuroVis 2012).*

8

SMOOTH BUNDLING OF LARGE STREAMING AND SEQUENCE GRAPHS

ABSTRACT: *In the previous chapters, we have shown that we can efficiently and effectively bundle large graphs using several image-based techniques. In this chapter, we extend the scope of our work one dimension further by considering time-dependent, or dynamic, graphs. Dynamic graphs are increasingly pervasive in modern information systems. We present here two techniques for simplified visualization of dynamic graphs using edge bundles. The first technique applies to streaming graphs, and naturally extends the kernel-density edge bundling in Chapter 7 by letting the iterative bundling run in parallel with the actual graph time information. The second technique applies to discrete graph sequences, and incorporates additional graph-to-graph correspondence data to emphasize structural changes between such graphs. We illustrate our methods with examples from real-world large dynamic graph datasets from flight traffic control and software evolution.*

8.1 INTRODUCTION

In the previous chapters, we have presented several applications of edge bundling layout (EBL) methods, and also introduced two new methods for constructing EBLs using image-based techniques. However effective in reducing visual clutter and efficiently computing EBLs, the above mentioned methods were only used for static graphs. In this chapter, we explore the usage of image-based EBLs for dynamic (time-dependent) graphs. Dynamic graphs pose their own understanding challenges. The data volumes are far larger than for static graphs. Users are interested in spotting *changes* in the overall graph structure, while maintaining limited clutter. Bundling methods, a promising option to compactly depict dynamic graph changes, have however been mainly used for static graphs.

In this chapter, we present two types of techniques for visualizing dynamic graphs using edge bundles. The first technique considers *streaming graphs*, *i.e.* temporally ordered, unstructured, edge-sequences with start and end lifetime moments (for the definition of streaming raphs, we refer to Sec. 2.3). For this use-case, we extend a recent fast and clutter-free static-graph bundling method. The second technique considers *graph sequences*, *i.e.* a discrete set of graphs between which higher-

level correspondences can be inferred (for their definition, we refer again to Sec. 2.3). For this use-case, we exploit additional edge-correspondence information to further highlight events of interest such as the appearance, change, and disappearance of edge groups, and show results based on different underlying static bundling algorithms. We present efficient GPU implementations of both our techniques which scale to large dynamic graphs, ensure spatial and temporal continuity, and are simple to implement. We demonstrate our techniques on real-world dynamic graphs from the air-traffic and software engineering application domains.

The structure of this chapter is as follows. Section 8.2 details a first visualization method for streaming graphs. Section 8.3 presents a second visualization method for graph sequences. Section 8.4 discusses the two methods in terms of desirable features. Section 8.5 concludes the chapter.

8.2 VISUALIZING STREAMING GRAPHS

Given a graph G, which includes (2D) node positions, we can think of (2D) bundling as an operator $B : G \rightarrow \mathbb{R}^2$ which creates a drawing $B(G)$ which maps edges that are close in G to close spatial positions (bundles) as explained in Chapter 7. Different bundling algorithms propose different ways to model edge closeness in G: tree-distance of edge end-nodes in a hierarchy [78], closeness of edges in a straight-line drawing of G as seen in [80, 55, 84], or the more general combination of graph-theoretic and image-space distances [126].

Consider now a streaming graph (Eqn. 2.1), the "instantaneous" graph $G(t) = \{e \in G | t \in [t_{start}(e), t_{end}(e)]\}$ and its bundling $B(t) = B(G(t))$ by some bundling operator B. Ideally, we want that $B(t)$ (a) varies continuously, or smoothly, in time with respect to the input $G(t)$ and also (b) keeps the spatial properties of the underlying bundling operator B, *i.e.*, puts close edges in tight bundles.

Property (b) is readily satisfied by using a "good" bundling algorithm B that guarantees that for any input graph, the result will be (strongly) bundled, such as *e.g.* [70, 103, 39, 55, 84], or to a lesser extent [78, 80], as we shall see. Property (a) means that, when $G(t)$ changes only slightly, then $B(G(t))$ should also change only slightly, so graph structures which are stable in time are also stable in the final visualization. Conversely, if there is an abrupt change in the graph, then there should be a visi-

ble change in the animation. However, even in the presence of such large changes in the input, discontinuous bundle *jumps* in the animation should be avoided, since visually tracking such jumps is hard.

A partial answer to (a) can be achieved by reducing the dynamics of $G(t)$, *e.g.* by applying a low-pass filter to $G(t)$. In other words, the bundling result shown at moment t is $B(\tilde{G}(t))$ where $\tilde{G}$ is the filtered graph. This is the solution proposed by StreamEB, who pioneered bundled layouts for streaming graphs [126]. They use a sliding window technique (finite-support box filter) to compute $\tilde{G}$ as all edges alive in $[t, t+\Delta t]$.

However, this approach has two limitations. First, the smoothness of the final animation depends strongly on the variation rate of $\tilde{G}$. If graphs for two consecutive time moments $\tilde{G}(t)$ and $\tilde{G}(t+\Delta t)$ differ too much, *e.g.* there are too many edges added or deleted per time unit, or the filtering time-window is too small, then there is no guarantee that the corresponding bundlings $B(\tilde{G}(t))$ and $B(\tilde{G}(t+\Delta t))$ are spatially close. If this is not the case, users notice a disruptive visual jump from t to $t+\Delta t$. Secondly, the computational efficiency of the approach in [126] strongly depends on the scalability of the underlying static bundling operator B. Algorithms which ensure good spatial stability, *e.g.* [80, 39] are also quite expensive, roughly $O(|\tilde{E}|^2)$ for $|\tilde{E}|$ edges in $\tilde{G}(t)$. Faster bundling algorithms [70, 55, 103] cannot ensure continuity, *i.e.*, a small change in the input graph may generate a large change in the bundled image, so are less suitable for stream bundling.

8.2.1 *Algorithm*

We address the above challenges by exploiting the properties of a recent bundling method for large graphs: kernel-density estimation edge bundling (KDEEB) [84], which was presented in detail in Chapter 7. Let us recall that, given a graph drawing $G = \{e_i\}_{1<i<N}$, KDEEB estimates the spatial edge density $\rho : \mathbb{R}^2 \to \mathbb{R}^+$

$$\rho(\mathbf{x}) = \sum_{i=1}^{N} \int_{\mathbf{y}\in e_i} K\left(\frac{\mathbf{x}-\mathbf{y}}{h}\right) \tag{8.1}$$

where $K : \mathbb{R}^2 \to \mathbb{R}^+$ is an Epanechnikov kernel of bandwidth h. KDEEB iteratively moves each point $\mathbf{x}$ of each edge upstream along $\nabla\rho$ following

$$\frac{d\mathbf{x}}{dt} = \frac{h(t)\nabla\rho(t)}{\max(\|\nabla\rho(t)\|, \epsilon)} \tag{8.2}$$

where ϵ is a small normalization constant. After a few Euler iterations for solving Eqn. 8.2, during which one decreases h and recomputes ρ, edges converge into bundles. A final 1D Laplacian smoothing pass is done on edges to remove small wiggles. Full details are given in [84]. Upon a closer analysis, one can see that this process is nothing else but performing the well-known mean-shift algorithm [34] on the drawn edges. In other words, the bundled graph is a *clustering* of the graph drawing based on edge similarity. This observation is important, since smoothness, noise robustness, and stability results proven for mean shift [34] can be readily extrapolated to KDEEB.

Core to KDEEB is the fast computation of the density map ρ. This is done by splatting the kernel K, encoded as an OpenGL texture, into an accumulation map. This allows bundling graphs of tens of thousands of edges in a few seconds on a modern GPU.

Our main idea for bundling streaming graphs is now to let the KDEEB iterations vary in sync with the stream time t. The principle is simple (see also Algorithm 1): We move a sliding window $[t, t + \Delta t]$ over the entire time range of the input streaming graph, compute $\rho(t)$ from the graph $\tilde{G}(t)$, and advect edges following Eqn. 8.2. There are two key advantages to this approach. First, $\rho(t)$ can be very efficiently computed by the underlying KDEEB algorithm, which is $O(|\tilde{E}|)$, *i.e.* proportional to the edge count in the current graph $\tilde{G}$. Secondly, and most importantly, the original KDEEB required $I = 5..10$ iterations for a single static graph to be bundled. We remove this iterative process by letting $\tilde{G}$ bundle *while* sliding the time-window. This makes sense since (a) if $\tilde{G}$ changes very slowly, advancing the stream time t is nearly equivalent to performing iterations for a fixed t, so we obtain a strongly bundled $\tilde{G}$, which is what we want to see. If (b) $\tilde{G}$ changes rapidly, then our process has less time to bundle, and thus we see looser bundles, which conveys us precisely the dynamics of $\tilde{G}$. More details on performance and parameter settings are given in Sec. 8.4.2.

Intuitively, our dynamic bundling method can be thought of as a process where edges continuously track the local density

```
1  t ← 0
2  while stream not ready do
3  |  ρ ← 0
4  |  E_live ← {e ∈ E|[t_start(e), t_end(e) ∩ [t, t + Δt] ≠ ∅}
5  |  foreach e ∈ E_live do
6  |  |  splat e into ρ ;                    //Splat live edges (Eqn. 8.1)
7  |  end
8  |  foreach e ∈ E|t_end(e) ∈ [t − δt, t] do
9  |  |  relax e towards its original position ;    //Vanishing
   |  |  edges
10 |  end
11 |  foreach e ∈ E_live do
12 |  |  advect e one step ;                  //See Eqn. 8.2
13 |  |  apply 1D Laplacian smoothing on e ;
14 |  |  draw e in the visualization ;
15 |  end
16 |  t ← t + δt ;                           //Advance sliding window
17 end
```

Algorithm 1: Bundling streaming graphs with KDEEB

maxima of a dynamically-changing graph. Since at each advection step edges move with a bounded amount h (line 12, Alg. 1), and since advection is done while advancing the stream time t, the maximal amount an edge-point can move at any time is h (Eqn. 8.2). Hence, the bundles move smoothly on the screen.

For disappearing edges, we can perform an additional step: We interpolate these edges from their current (bundled) position towards their original (unbundled) position they had in the input stream (line 9, Alg. 1). This makes the animation symmetric: New edges progressively bundle as times goes by, while disappearing edges relax, or unbundle, towards their original positions after exiting the sliding time-window. To further emphasize this effect, we modulate the edges' transparencies in a similar fashion. We note that this effect is optional. If left out, disappearing edges will exit silently, without relaxation. The choice of using relaxation or not depends on whether users want to edge vanishing events or not.

An important goal of animated visualizations is to help users detect deviations from regular patterns [194, 207]. We support this by shading bundles to convey their speed of change, using a simple and fast image-based method: We compute the density moving-average $\tilde{\rho}(t)$ over $[t, t + \Delta t]$, and color bundles by the

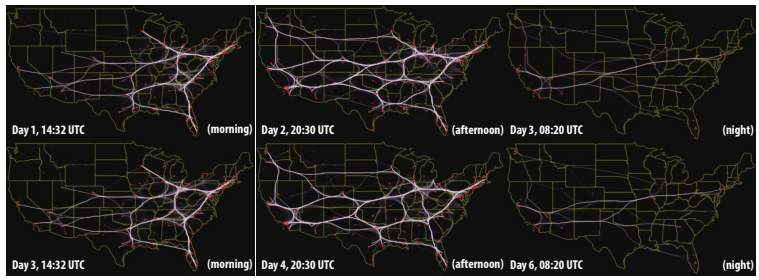

Figure 8.1: Streaming visualization for 6-days US airline flight dataset (Sec. 8.2.2)

normalized difference $|\rho(t) - \tilde{\rho}(t)|/\tilde{\rho}(t)$ using a white-to-purple colormap. Results hereof are shown next.

8.2.2 *Applications*

Figure 8.1 shows several frames from a streaming visualization of US flights [165] (6 days, 41K flights). The streaming graph contains flights with start and end date-and-time and geographical locations. The resulting flight-trail bundles are smooth, clutter-free, and exhibit a continuous variation in time. From the stills, we see that same time-of-day flight patterns are quite similar for several days. However, they vary strongly over a day: During the evening, the East coast has the most intense traffic. During the afternoon, the entire US is quite uniformly covered with flight routes. During the night, flights linking the two coasts dominate.

Figure 8.2 shows a similar visualization for flights over France (7 days, 54K flight trails). Similar to the US dataset, bundles are smooth and clutter-free in both space and time. Colors indicate the bundles' speed of change (white=stable, purple=rapid changes, see Sec. 8.2.1). Red dots show the first and last positions when a plane was monitored. Dots inside France are actual airports. Dots outside the French territory indicate international flights which enter/exit the French airspace. We see that, overday, the main "backbone" flight pattern is quite stable over different days, and contains mainly north-south routes, with Paris as a key hub (Fig. 8.2 top row). A different pattern, also quite stable, appears at night (Fig. 8.2 bottom row): A salient vertical bundle shows Southern flights bound to Paris. We also see more purple, which shows that night-time flight paths are much less stable than overday flight paths.

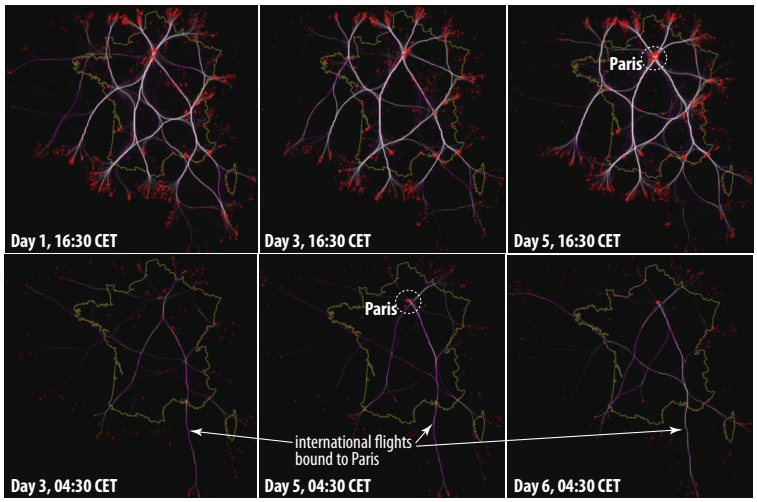

Figure 8.2: Streaming visualization for 7-days France airline flight dataset (Sec. 8.2.2)

For the US dataset, a qualitative comparison of our results with those produced by StreamEB framework [127] shows that using KDEEB produces stronger bundles and an overall smoother animation. This can be explained by the different properties of the two underlying bundling algorithms (KDEEB can easily produce bundles with many inflexion points, while FDEB has a smoothing factor built in its edge compatibility metric that discourages such shapes); by the finer-grained time sampling that we use; but most importantly by the built-in smoothness of our algorithm which bundles edges as they arrive in the input stream.

8.3 VISUALIZING GRAPH SEQUENCES

Graph sequences G^i (Sec. 2.3) exhibit different properties from streaming graphs, as follows. First, streams allow defining an infinity of "instantaneous" graphs $G(t) = (V, \{e \in E | t_{start} < t < t_{end}\}), \forall t \in \mathbb{R}$ (see Sec. 8.2.1). Some of these graphs may not have a direct meaning or usefulness. In contrast, graph sequences contain a finite set of graphs which have been explicitly computed in specific ways, *e.g.* for particular time moments, *e.g.* (major) revisions of a software system. Secondly, keyframe correspondences add higher-level, edge-centric, information, *e.g.* the fact that two files f_1, f_2 share a common piece of text in ver-

sion 1, and next f_1 shares the same text with a file f_3 in version 2. In contrast, streaming graphs (Eqn. 2.1) only specify how edges appear and disappear in time, but do not necessarily encode logical connections between edges at different time moments. Thirdly, graph sequences do not necessarily come with birth and death moments for individual edges. Finally, keyframes in graph sequences must be wholly available before processing, whereas edges in a streaming graph can be, in most cases, analyzed "online" as they appear. All in all, the above make a case for treating graph sequences differently from graph streams.

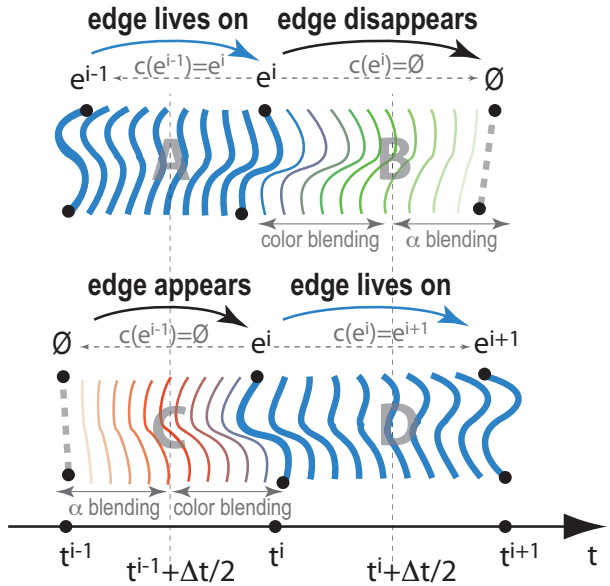

Figure 8.3: Interpolation for graph sequence visualization

8.3.1 *Algorithm*

For graph sequences, we propose the following bundling method: For each keyframe G^i, we compute its bundled layout $B^i = B(G^i)$, using a given bundling algorithm B. Next, we interpolate these layouts between a keyframe i and the previous and next keyframes $i-1$ and $i+1$ respectively using the correspondence data (see Fig. 8.3). Consider a time axis t along which we place keyframes *e.g.* at moments $t_i = i\Delta t$ (any other definition of t_i can be easily used, if available). For each edge $e \in G^i$, if $c(e) = e^{i+1} \in E^{i+1}$, we linearly interpolate $B^i(e)$ to $B^{i+1}(e^{i+1})$

over the interval $[t_i, t_{i+1}]$ (Fig. 8.3D). If $c(e)$ is the empty set, *i.e.* e has no correspondence in E^{i+1}, we interpolate $B^i(e)$ to the line segment $L(e) = (n_{start}(e), n_{end}(e))$ over the same time interval (Fig. 8.3B). Symmetrically, if $c^{-1}(e) = e^{i-1}$, we interpolate from $B^{i-1}(e^{i-1})$ to $B^i(e)$ over $[t_{i-1}, t_i]$ (Fig. 8.3A), else we interpolate from $L(e)$ to $B^i(e)$ over the same time interval (Fig. 8.3C).

We emphasize appearing and disappearing edges by shading: the edges that have correspondences between two keyframes i and $i+1$ are blue and thick. Edges that disappear from i to $i+1$ get a color linearly interpolated between blue (at t_i) and green (at $t_{mid} = \frac{t_i+t_{i+1}}{2}$), followed by a transparency (alpha) value decreasing from opaque (at t_{mid}) to fully transparent (at t_{i+1}). Edges that appear from $i-1$ to i get an alpha increasing from fully transparent (at t_{i-1}) to opaque (at t_{mid}), followed by a color interpolated from red (at t_{mid}) to blue (at t_i). Hence, edges appear by fading in to red (highlights their incoming), then smoothly merge in a blue bundle, and disappear by unbundling, becoming green (highlights their vanishing), and fading out. Examples next explain the color choice.

8.3.2 *Applications*

We illustrate our sequence-graph visualization with two datasets from software engineering. The first dataset contains 22 releases of Mozilla Firefox [123]. For each revision, we extracted the code hierarchy (folders and files), and also the so-called *clones*, or code duplicates, using the freely available clone detector SolidSDD [145, 161]. Hence, for each revision, we obtain a compound hierarchy-and-associations graph where two files are linked by an edge if they share a code duplicate. If a code fragment is cloned in several files, then all these files are pair-wise linked by associations.

Figure 8.4 shows snapshots from SolidSDD's HEB visualization for such graphs. Node colors show duplication amount (red=high, green=low). Seeing how subsystems share clones is useful in perfective maintenance, where one needs to plan code clone removal with minimal impact on system architecture. It is also important to assess how much, and where, did adaptive maintenance (*i.e.* adding new features) introduce new clones, and how much, and where, did perfective maintenance succeed to remove clones in the past [131]. For this, we need to easily

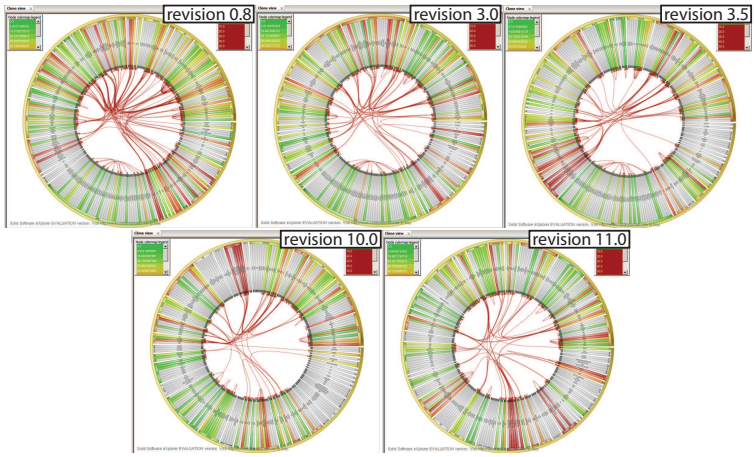

Figure 8.4: Small multiples visualization for clones in Mozilla Firefox for five selected revisions (Sec. 8.3.2)

compare the clone evolution patterns. This is hard to do using such small-multiple visualizations.

To support such a task, we proceed as follows. First, we create a so-called union hierarchy containing all graph nodes in the analyzed releases [15]. This contains 13856 file and folder nodes. Next, we build correspondences between clones in consecutive releases: Two clone relations e^i and e^{i+1} correspond if they link the same files, *i.e.* files having the same fully qualified names, in G^i and G^{i+1}. Other ways to find correspondences, *e.g.* using the actual text content of the clones [131], can be readily used too, if desired. The above steps deliver a graph sequence G^i in the sense described in Sec. 2.3.1. This sequence contains 5687 unique edges (that is, when counting corresponding edges as one) and 48591 edges in total.

We now use our graph-sequence visualization method to analyze this sequence. Fig. 8.5 shows several frames from this animation. The second and fourth rows show results produced using KDEEB as underlying bundling method. Disappearing edges are green (removing clones is good); appearing edges are red (introducing new clones is bad). Additionally, we color hierarchy nodes as follows: Nodes which contain a changing clone count are colored by the clone count change, using red for positive values and green for negative values. Nodes where the clone count stays constant are colored blue. In all cases, we use saturation to indicate absolute values (saturated=high, desaturated=low values).

We note several events of interest. First, there is a relatively stable "core" clone-structure that lives for a long time (blue bundles). These can be hard to remove clones, or clones that maintainers were not aware of, *e.g.* if no clone detector was actively used on this system during perfective maintenance. We also spot several moments when major clone-pattern changes occur, *e.g.* from revision 2.0.0.10 to 3.0, many green edges appear, so many clones are removed (Fig. 8.5 d). Node coloring helps spotting high-clone-density subsystems. For instance, from revisions 3.6.10 on, we see two such dark-blue groups (dotted circles in Fig. 8.5 fourth row, e-h). Since these groups stay visible in several revision, they indicate "stubborn" clones which, for several reasons, could not be removed for a long time. Although this information is encoded in the bundles too, finding such patterns on nodes is easier than visually following bundles. In other words, node colors help finding *aggregated* patterns, *e.g.* high-clone-density systems during the evolution, while bundle changes help seeing which particular *subsystems* share such clones. We also see a red spot in revision 2.0.0.10 (Fig. 8.5 c): This is a subsystem where many *intra-system* clones have been added. Seeing such clones without node coloring would be hard, since their (bundled) edges are very short.

The transition between revisions 7.0 and 8.0 shows an interesting event: First, several "stubborn" clones are removed (green edges shown after passing revision 7.0) Next, clones between the *same* files are added back again (red edges shown when approaching revision 8.0). This typically happens when one modifies related code in two subsystems *e.g.* by rewriting it by independently applying twice the same given design pattern. However, developers were likely not aware of the clones, otherwise we would expect the clone to be removed during such a perfective refactoring. Finally, comparing the first and last frame shows that the core clone pattern did not change significantly. Also, the bundle pattern shows that clones connect *unrelated* subsystems, *i.e.* nodes in the radial icicle plot that are not close to each other, hence not in the same parent system. This is a negative sign for code quality, since removing such clones requires system-wide understanding and refactoring.

As a second example, we extracted a compound digraph with folders, files, and functions (forming the hierarchy) and function calls (forming the associations) from 14 revisions of the Wicket open-source software [214]. Next, we build the same union hierarchy as in the first example (8799 nodes), and compute correspondences based on the fully qualified signatures of

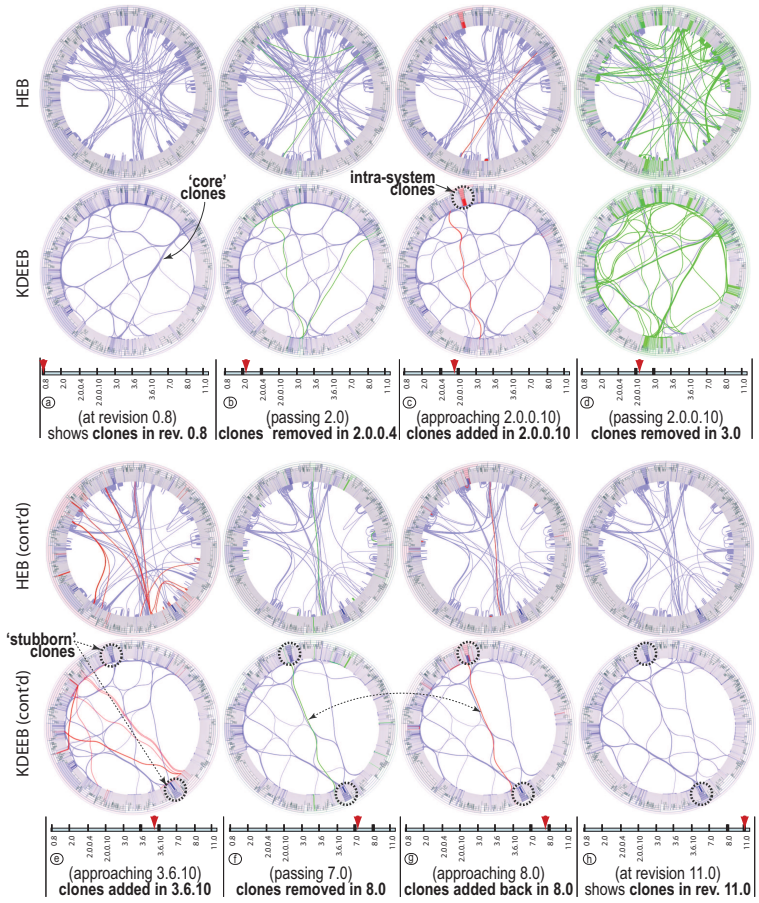

Figure 8.5: Sequence-based visualization for clones in Firefox (8 frames). First and third row: HEB bundling. Second and fourth row: KDEEB bundling. (Sec. 8.3.2)

(caller, callee) pairs. We obtain 11953 unique edges and 92810 total edges. Figure 8.6 shows several frames from the graph-sequence visualization. To better illustrate the animated transitions, we focus here on a short period (three revisions). This visualization helps reasoning about the system's (change of) modularity, a challenging task in program comprehension [15]. The interpretation is as follows: The stable pattern (blue bundles) shows the stable control-flow logic of the system, *i.e.* calls that do not change much across versions. We see that this pattern is quite complex, *i.e.* connects many subsystems in different hierarchy parts, so the overall modularity of this software is *and* stays relatively low. In more detail, we see that in version

1.4.18, a significant coupling is added between systems A and B (large red bundle A-B, Fig. 8.6 c). Interestingly, at the *same* moment (1.4.18), many calls are removed between the same systems (large green bundle A-B, Fig. 8.6 f). This indicates a refactoring of the A-B system interaction – note the similarity with the clone insertion-and-deletion pattern, and interpretation thereof, discussed above for the Firefox dataset.

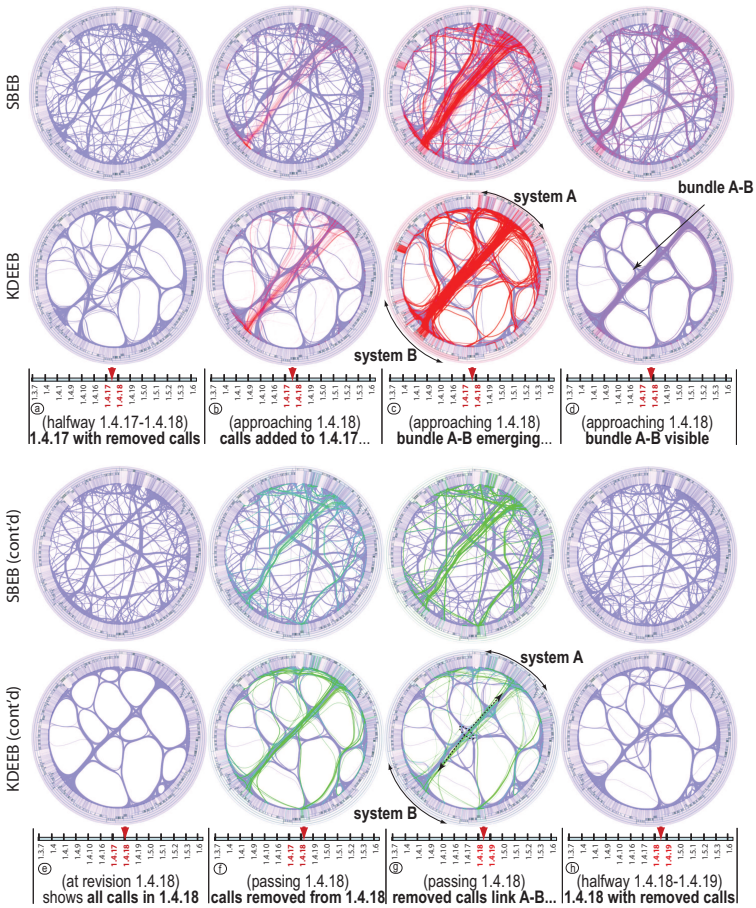

Figure 8.6: Sequence animation – Wicket call graphs (8 frames around release 1.4.18). First and third row: SBEB bundling. Second and fourth row: KDEEB bundling. (Sec. 8.3.2)

8.4 DISCUSSION

8.4.1 *Streaming* vs *sequence graphs*

In Sections 8.2 and 8.3 we have presented two techniques for visualizing streaming and sequence graphs. One question is: Can we use the streaming algorithm for a sequence graph, and/or conversely? Why do we need two techniques? Below we analyze this aspect.

8.4.1.1 *Streams with sequence-based visualization*

For the first experiment, we convert our France air-traffic streaming graph (Sec. 8.2.2) to a sequence graph of 7 keyframes. For this, we divide the 7-days stream into 7 one-day periods. Edges are assigned to keyframes based on start time. Next, we add correspondences between edges in consecutive keyframes (days) whose geographic start and end locations are very similar and flight IDs are identical. We obtain a 7-keyframe sequence, with 8811 unique edges (when counting corresponding edges as one), and 54K edges in total.

In sequence-based visualization of this dataset, we observe that bundled patterns are much less structured, and their change is harder to follow. This is not too surprising, since the stream-to-sequence conversion quantized the fine-grained time information. Hence, while the streaming-based visualization uses this information to *continuously* bundle edges as they appear, the sequence-based visualization only bundles at keyframes, and uses edge interpolation in between. Additionally, visualizing streams as graph sequences involves delicate data modifications, *e.g.* cutting the stream at possibly irrelevant moments into disjunct chunks, and adding edge-correspondences that may not be meaningful. When such a transformation is not evident, and when fine-grained time data is important for comprehension, one should not visualize graph streams as graph sequences.

8.4.1.2 *Sequences with stream-based visualization*

For the second experiment, we convert our Wicket graph sequence (Sec. 8.3.2) to a streaming graph, by inserting 100 uniformly-spaced time moments between each two consecutive keyframes. Figure 8.7 shows three frames from the resulting animation, taken between revisions 1.5.0 and 1.5.1. The sequence method (top row) clearly shows a stable core indicating un-

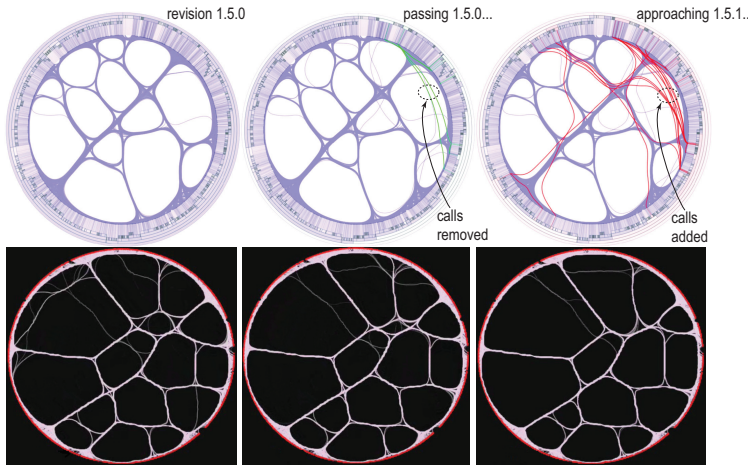

Figure 8.7: Streaming visualization of graph sequence (3 frames around revision 1.5.0, Wicket software dataset)

changing call patterns (blue bundles), and also outlines the removed calls (green) and added calls (red). The streaming method (bottom row), although doing a good job in creating a smooth and stable bundling, cannot emphasize such additions and removals, since it has no correspondence data to separate the treatment of stable and (dis)appearing edges.

8.4.2 *Scalability*

The streaming graph visualization 8.2.1 has a complexity of $O(|\tilde{E}|)$ per animation frame, where $|\tilde{E}|$ is the average number of edges in any time-window of size Δt at any moment t in the stream. This is so since we run the bundling process in sync with the stream time, as explained in Sec. 8.2.1. In other words, there is a single density-splat and advection step for each edge present in a frame. In comparison, the method in [126] is $O(|\tilde{E}|^2)$ per frame. We implemented our graph streaming and sequence visualizations in C# using the KDEEB algorithm which is itself written in C# using OpenGL 1.1, and run them on a 2.3 GHz PC with 8 GB RAM and an NVidia GT 480 card. On this platform, producing one streaming-animation frame took 0.05 seconds for the US dataset ($|\tilde{E}| = 2K$ edges on average) and 0.17 seconds/frame for the France dataset ($|\tilde{E}| = 15K$ edges on average). Per frame, we are roughly 10 times faster than the original KDEEB ([84], Tab. 1), which is expected, as we do only one it-

eration per frame (see Sec. 8.2.1). In contrast, if we were to use FDEB, we would need, for the US dataset, 19 seconds/frame on comparable hardware ([80], Sec. 4.2), or 6 seconds/frame for a graph of $|\tilde{E}| = 900$ edges on a 1.7 GHz PC ([126], Fig. 12). Of course, the total time needed for a stream depends on the stream's length.

The sequence graph visualization 8.3.1 has a complexity of $O(BN)$ for a sequence of N graphs, and an underlying bundling algorithm of complexity B. This is basically the same cost as in [126], modulo the fact that our algorithm B is faster, as already explained. However, note that our visualization is different, since we (a) emphasize appearing and disappearing edges and (b) smoothly interpolate consecutive bundled layouts by using edge correspondences.

8.4.3 *Bundling algorithm choice*

For streaming visualizations, KDEEB is arguably a very good solution: KDEEB works for general graphs, produces bundles with little clutter even for very complex graphs, and is robust and simple to use. However, the most important point is that KDEEB's design allows to *incrementally* update the graph *during* the bundling. In contrast, most other bundling layouts require a full recomputation of the bundling when the input graph changes. This is due to various technical factors, *e.g.* use of spatial search data structures and compatibility metrics that need reinitialization upon graph changes [55, 39, 70], or encoding the bundle polylines separately from the input graph's straight-line edges [70, 103, 150]. FDEB comes closest to KDEEB in flexibility, as it represents (partially) bundled edges as a set of unstructured polyline curves, so it can be used for incremental smooth bundling upon input graph changes. However, KDEEB's linear complexity in the input graph size makes it more suitable than FDEEB which is quadratic in the same input size.

For sequence visualizations, any bundling algorithm can be technically used. However, here KDEEB also proved better than alternatives. Figure 8.5 shows the differences between using HEB (first and third rows) *vs* KDEEB (second and fourth rows). As visible, HEB produces less structured and compact bundles. A similar effect can be seen in StreamEB [126]. Figure 8.6 shows the differences between using SBEB (first and third rows) *vs* KDEEB (second and fourth rows). Here, SBEB produces actually too much structure – the bundles have too many branches.

KDEEB produces less clutter than SBEB, but more structure than HEB, thereby offering a good visual balance.

8.4.4 *Parameters*

Our streaming-based visualization uses the same edge sampling, kernel size, smoothing, and density-map resolution parameters as KDEEB [84]. The parameters added by our streaming method are the size of the time-window Δt and time-step δt for sliding this window (see Alg. 1). Δt controls how much one sees in one animation frame: Larger Δt values show more (bundled) edges, but inherently smooth out the dynamics of the animation. Smaller values show more of the instantaneous graph $G(t)$, but make short-lived edges (dis)appear faster. In our examples, we used a Δt corresponding to a 5% change in the number of edges in $\tilde{G}$, so that animation goes faster over uninteresting time periods, similarly to [126]. δt controls the ratio between the animation speed and the stream's own speed *and* also the bundling tightness. Large δt values subsample the stream, *i.e.* make the animation go faster and show less tight bundles, since, as outlined in Sec. 8.2.1, bundling occurs in sync with the stream time. Smaller δt values supersample the stream, *i.e.* make the animation go slower and also create tighter bundles. In practice, getting tight bundles with KDEEB requires roughly $I = 5..10$ iterations [84]. Hence, we set δt to $1/I$ of the average edge lifetime in the stream. A good side-effect of this setting is that bundling reflects the edge lifetime: Short-lived edges, likely outliers, do not strongly bundle. Long-lived edges, which contribute to the coarse-scale structure of the graph, get strongly bundled. Apart from Δt and δt, our algorithm has no other parameters.

8.4.5 *Limitations*

Currently, we showed that we can bundle graph streams and sequences in a fast, smooth, and clutter-free manner, and that such animations help assessing connection stability and spot fast-changing bundles (Secs. 8.2.2 and 8.3.2). However, the animation and visual mapping metaphors, *i.e.* speed, shape, tightness, and shading of bundles, would need to be adapted to support seeing finer-grained events of interest such as bundle splitting, or merging; similar bundles in far-apart time frames; and separating bundles based on additional edge attributes. Also, a

quantitative and qualitative measurement of the effectiveness of animated bundles is needed.

8.5 CONCLUSION

In this chapter, we have presented two algorithms for the animated visualization of graph streams and sequences. By exploiting the smoothness, stability, speed, and incremental nature of our KDEEB image-based bundling algorithm (Chapter 7), we succeed in creating streaming graph animations which exhibit the same desirable properties. Next, we use the same algorithm to generate sequence-based graph visualizations where edge appearance and disappearance events are emphasized. We apply our techniques on four large datasets, and present evidence that supports our choice for KDEEB as underlying layout.

Future work can address animation, visualization, and interaction refinements to find and emphasize finer-grained events of interest, such as bundle merging and splitting, and support tasks such as detecting graph patterns that match problem-specific patterns of interest. Furthermore, multilevel refinements can be proposed to emphasize such patterns and events on several scales in a single animation. At a more conceptual level, the parallel drawn between graph bundling and the mean shift prcess used in image processing [34] could be further analyzed in order to gain a better theoretical understanding of graph bundling.

This chapter is based on:

Christhophe Hurter, Ozan Ersoy, and Alexandru Telea. Smooth Bundling of Large Streaming and Sequence Graphs. *Proc. PacificVis*, 2013 *(Honorable Mention Paper Award)*

9

AN ATTRIBUTE-AND-STRUCTURE SEMANTIC LENS FOR LARGE ELEMENT PLOTS

ABSTRACT: *In previous chapters, we have presented a range of edge bundling methods for static and dynamic graphs. Bundled images, by definition, trade clutter for overdraw, therefore making it hard to reason about individual attributes of edges in a given bundle. In this chapter, we present a set of interaction techniques that help exploring the edge information which is usually aggregated during the bundling process. For this purpose, we introduce MoleView, a novel technique for interactive exploration of multivariate relational data. Given a spatial embedding of the data, in terms of a scatter plot or graph layout, we propose a semantic lens which selects a specific spatial and attribute-related data range. The lens keeps the selected data in focus unchanged and continuously deforms the data out of the selection range in order to maintain the context around the focus. Specific deformations include distance-based repulsion of scatter plot points, deforming straight-line node-link graph drawings, and as varying the simplification degree of bundled edge graph layouts. Using a brushing-based technique, we further show the applicability of our semantic lens for scenarios requiring a complex selection of the zones of interest. Our technique is simple to implement and provides real-time performance on large datasets. We next demonstrate how our technique can be applied also to other datasets than graphs, such as 2D images and 3D volumetric datasets.*

9.1 INTRODUCTION

In recent years, the amount of data which information visualization techniques are confronted with has increased massively, whereas display sizes have remained largely identical. Infovis techniques address this challenge in two main ways. First, datasets are simplified by clustering or subsampling, so they deliver manageable data amounts with respect to available screen size. Secondly, mapping techniques maximize the amount of information displayed per screen space area, or information density.

However effective, techniques of the second type create additional challenges to explorative user interaction. Consider the case of dense node-link layouts or multivariate datasets displayed as scatterplots or parallel coordinates, such as the edge bundling layouts (EBLs) presented in Chapters 3-7. Such techniques create significant amounts of *overlap* between the drawn

elements (points or edges). This simplifies the resulting visualization by reducing the number of perceived elements. However, overlaps make it harder to explore the dataset: In typical 2D visualizations, it is hard or even impossible to see what is hidden 'under' the front-most elements, even when using transparency. Hidden elements cannot be easily selected and/or brushed over without additional interaction effort. We have the situation of a compact visualization (desirable from the viewpoint of scalability and, optionally, clutter reduction) which is suboptimal for interactive exploration. Finally, there are use-cases when a given dataset may be best understood by using several layouts, one for each aspect being examined. Displaying one layout in separate linked views of all layouts can be suboptimal as it increases the effort required from the user to correlate between the different views.

In this chapter, we present MoleView, a framework for interactive exploration of large *element-based plots*, which are sets of discrete data elements, each with several data and/or position (layout) attributes, which are visualized using a single, rather than multiple, views. Examples thereof are node-link layouts, (multidimensional) scatter plots, and images. Our contributions are as follows. First, we extend the well-know semantic lens with a range-based attribute filter to select a 'data layer' at a user-defined point, *i.e.* a set of data elements falling within the lens' position and attribute filter values. Instead of hiding the elements in the lens which fail passing the attribute filter, we use a dynamic re-layouting technique to smoothly push these away from the lens, or pull them back, hence the name of our technique. Second, we extend our data-driven deformation idea to explore bundled graphs. Given a bundled and unbundled version of the same graph, we use the MoleView to control the bundling strength *and* which edges get bundled at a certain location. In this way, users can explore bundled graphs (*e.g.* dig into a bundle to extract edges of interest based on attribute value) or, conversely, interactively simplify a given layout by bundling uninteresting edges. Finally, we extend the semantic lens concept for the task of exploring a dataset by the smooth animated interpolation between two completely different layouts of the same data, using as example the exploration of two-dimensional scalar images. Our technique has just a few parameters which are simple to control by end users, can be efficiently implemented to provide real-time interaction on large datasets, and can be easily incorporated in existing Infovis data exploration applications.

For related work on interaction techniques with a focus to element-based plots in general and node-link graph drawings in particular, we refer to Sec. 2.4. Following this work, in Section 9.2, we describe the principle of the MoleView technique and its three different modes on utilization (elements, bundles, and dual-layout), and illustrate our technique in practice on a range of real-world datasets. Section 9.3 discusses the presented technique. Finally, Section 9.4 concludes this chapter with future work directions.

The 'digging lens' presented in Chapter 5 partially addresses this problem by thinning overlapping bundles at the focus location to allow one to see and/or select bundles obscured due to the inherent overdraw.

9.2 MOLEVIEW PRINCIPLE

The principle of MoleView is as follows (see also Fig. 9.1). As input, we consider a dataset $D = \{s_i\}$ consisting of a set of data elements s_i which all have 2D layout positions $L = \{p_i = (x_i, y_i) \in \mathbf{R}^2\}$. Examples thereof are scatter plots, where s_i are data points; images, where s_i are pixels with color information; and node-link graph drawings, where s_i are nodes, edge control points, entire edges, or entire edge bundles. Any other dataset can be considered as long as it provides 2D position information. Within the given layout, the positions of different elements can overlap, *e.g.* in the case of (bundled) graph drawings (in which case clutter and overdraw are an issue), or not *e.g.* in the case of images. Each s_i can have an application-specific attribute vector $v_i = \{v_{ij}\}$. For simplicity, we next consider only numerical attributes $v_{ij} \in \mathbf{R}$. However, the MoleView principle applies equally well for other attribute value types.

When exploring a 2D rendering of D, users first define a so-called *focus* zone $Z \subset \mathbf{R}^2$. Our central goal is to support tasks which involve exploration of the spatial structure and data attribute distribution of elements $s_i \in D$ within the focus zone (*i.e.* $p_i \in Z$). The provided support is offered in terms of a semantic lens applied on the zone of interest. Our lens combines a flexible, easy to use, animation-based mechanism for specifying the focus zone, containment of data elements in the focus zone, and attribute values to explore, and also the type of spatial deformation applied to the data elements in and/or outside the lens.

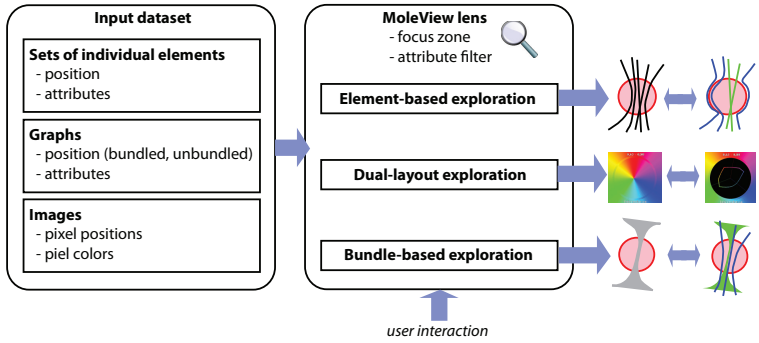

Figure 9.1: MoleView interactive exploration pipeline

We guide the design of our lens by the following:

- exploration should use a *single* view rather than linked views;
- the zone and attribute range of interest should be easily specifiable by simple mouse-driven operations;
- the lens should address the overdraw problem in dense visualizations by allowing users to 'see' behind the front-most elements;
- the lens should provide a focus-and-context metaphor on the dataset D. Changes to the layout L of D should be *smooth* so that users maintain their mental map when using the lens;

The general mechanism proposed is as follows. First, we select the set of data elements $D^Z \subset D$ which are spatially within Z. Spatial containment is determined by the desired effect and type of data elements, *e.g.* points or curves. Secondly, we filter D^Z to a subset of data elements D^{sel} which are within the attribute range of interest A. Like for spatial containment, attribute selection can involve different types of filters for different tasks. We call the set $D^{filt} = D^Z \setminus D^{sel}$ the set of filtered elements, *i.e.* elements that fall in the lens spatially but not data-wise. The most important step is the third one: We apply a smooth, time-animated, spatial deformation $\Delta : \mathbf{R}^2 \times \mathbf{R}^+ \rightarrow \mathbf{R}^2$ from the original layout $L^{filt} = \{p_i \in \mathbf{R}^2 | s_i \in D^{filt}\}$ of the elements in D^{filt} to yield a new layout $L^{filt}_{new} = \Delta(L, t)$. The time parameter $t > 0$ controls the animation of the deformation, *i.e.* morphs in both directions between L^{filt} and L^{filt}_{new} as

the lens is activated, respectively deactivated. Suitable choices of the deformation function Δ allow us to perform decluttering, selective fisheye-like exploration on specific data elements, bundled graph exploration, and also correlation of data elements between different layouts.

We next detail three different instances of the MoleView lens principle outlined above: element-based exploration (Sec. 9.2.1), bundle exploration (Sec. 9.2.2), and dual-layout exploration (Sec. 9.2.3).

9.2.1 *Element-based exploration*

In this mode, we consider the exploration of a dataset D whose elements s_i have the minimal amount of information: position p_i and an attribute value v_i. We first define the zone of interest Z as a distance field $D_Z(P) : \mathbf{R}^2 \to \mathbf{R}^2$. The distance field D_Z is defined using a so-called *control set* $P \subset \mathbf{R}^2$, as follows. First, we compute the distance transform $DT_P : \mathbf{R}^2 \to \mathbf{R}_+$ [37]

$$DT_P(x \in \mathbf{R}^2) = \min_{y \in P} \|x - y\| \tag{9.1}$$

Given DT_P, Z is simply the level set of DT_P at a user-specified distance $\delta > 0$. Hence, we select all data elements *spatially* falling within Z as

$$D^Z = \{s_i \in D | DT_P(p_i) \leqslant \delta\} \tag{9.2}$$

If P is a compact set, then Z is also be compact. However, this is not a constraint – the set P can be any collection of points, lines, or surfaces in 2D. Computing D^Z is simple, no matter how complex the the data element shapes are: We render a shape and apply the point-in-region test (Eqn. 9.2) when visiting each rendered pixel, an operation efficiently supported by graphics hardware.

Given D^Z, we next select the elements $D^{sel} \subset D^Z$ which are within the zone of interest *and* also have attribute values of interest. In this chapter, we use attribute-range selection

$$D^{sel} = \{s_i \in D^Z | v_i \in [v_{min}, v_{max}]\} \tag{9.3}$$

Other attribute tests can be substituted easily without affecting the implementation or ease of use of our method. The spatial and attribute tests (Eqns. 9.2 and 9.3) can be done in a single rendering pass.

The element-based exploration works now as follows. The user specifies the control set P by direct interaction, *i.e.* brushing in the visualization using the mouse. In the simplest case, one selects one or more screen points which will form P, similar to [219]. In this case, DT_P is a superposition of point radial distance functions. The size of the zone of interest δ is via the mouse wheel with a modifier key (Control). More complex interactive specifications of P, yielding more complex distance transforms DT_P, are described in Sec. 9.2.4. Apart from P, the user also specifies an attribute filter to select elements based on their data values. For the filter in Eqn. 9.3, we specify the range $[v_{min}, v_{max}]$ by moving the mouse wheel.

The MoleView comes now into action: We keep the points D^{sel} which fall spatially and data-wise in the lens at their original locations p_i and define the layout L^{filt} for the filtered points D^{filt} so as to push them away from the exploration focus (see Fig. 9.2). For this, we move the points $p_i \in D^{filt}$ in the gradient field $-\nabla DT_P$ with a speed $|\mathbf{v}|$ which decreases as points get close to the lens border and further from the control set P. In detail, the motion field $\mathbf{v} : \mathbf{R}^2 \to \mathbf{R}^2$ is defined by

$$\mathbf{v}(x) = -\nabla DT_P(x)\, \lambda\left(\frac{DT_P(x)}{\delta}\right) \tag{9.4}$$

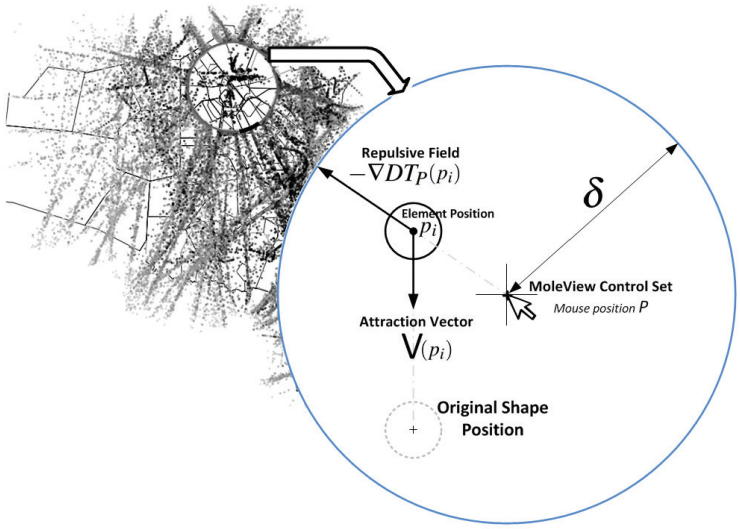

Figure 9.2: MoleView element-based exploration mode

The function $\lambda : [0,1] \rightarrow [0,1], \lambda(0) = 1, \lambda(1) = 0$ lowers the speed, *i.e.* decelerates points, as they get close to the lens border. In practice, exponential decaying profiles give smooth animation results. The advection implicitly yields a deformation $\Delta(t)$ which gradually pushes points away from P and slows them down at the lens border. Different speed profiles as function of the distance to P can be easily substituted.

The advection in Eqn. 9.4 is applied when the lens is activated by mouse clicking and is done as long as the mouse button is kept pressed. During this period, we continuously update the position of the points in D^{filt} and redraw them, thereby creating a smooth animation. As the user changes the control set by moving the mouse, points keep moving as they enter into, or exit from, the zone of interest. When the lens is deactivated by mouse button release, we change $\mathbf{v}$ to an attraction field $\mathbf{V}$, defined at the current location of the displaced points p_i^{disp} as

$$\mathbf{V}(p_i^{disp}) = p_i - p_i^{disp} \tag{9.5}$$

where p_i are the point positions before displacement. The effect is that the displaced points smoothly go back to their original positions with decelerating speed, thus reversing the lens effect.

For additional cues, we change the rendering of the displaced elements p_i using $DT_P(p_i)$ by linearly interpolating their transparency between a low value α_{min} at $DT_P = 0$ and a maximal value $\alpha_{max} = 1$ at $DT_P = \delta$, *i.e.* on the border of Z.

Point dataset example: We first consider a 2D point plot of a multivariate dataset using multidimensional scaling (MDS) [132]. The points are text documents placed on the 2D plane so as to reflect the similarity of topics they contain. Document similarity is computed using a cosine-based distance between term vectors extracted from the documents' text [147]. Document topics, found by the classification algorithm underlying the MDS layout, are saved as point data attributes. Due to overdraw, it is hard to see which are the point topics within a given spatial region. This insight is important for MDS plot users, *e.g.* to detect data points which are close to a topic classification border, and for MDS algorithm designers, to assess the algorithm ability to separate different topics.

Figure 9.3 shows the element-based lens applied to this dataset. The selected attribute range-of-interest matches the purple-colored topic. When the lens is activated, points outside this topic are smoothly pushed towards the lens periphery, while

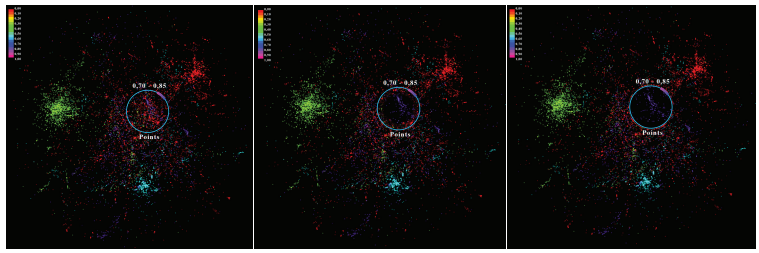

Figure 9.3: Element-based exploration of an MDS plot for text documents. Colors are document topics. Points outside the range of interest are gradually pushed to the lens border

points within the topic stay unchanged. By changing the attribute range with the mouse wheel, we can browse through the topics overlapping at a given location. Points are pushed or attracted with respect to the lens center as they exit, respectively enter, the range of interest, yielding a sequence of smooth transitions, which helps understanding the image.

Trail dataset example: Our second example dataset is a set of trajectories (trails) whose end points indicate airport locations in France. Trails are flight routes between airports, recorded by air traffic authorities (17275 flight routes) [83]. Each trail is a sequence of points with geographical and altitude data at the respective location. Altitude is visualized by color mapping. Note that this dataset is not, strictly speaking, a graph since trails do not always share start and end points.

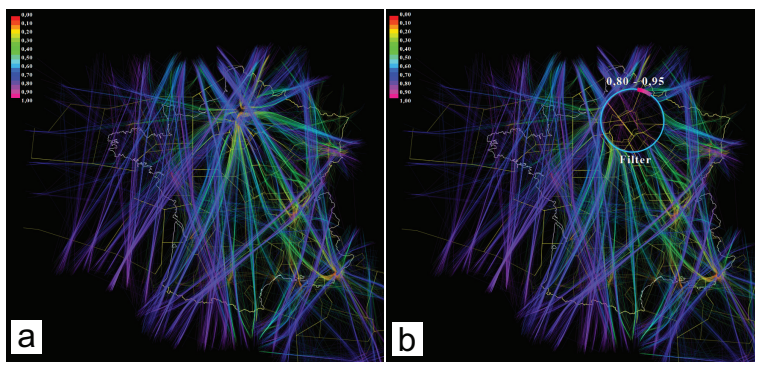

Figure 9.4: Flight trails dataset (a) and element-based MoleView lens (b)

Rendering the entire trail set with altitude-colored edges produces an image of very limited usefulness, given the high data

occlusion and clutter (Fig. 9.4 a). An important task here is to find flights with a certain altitude, or altitude variation, over a given spatial region, *e.g.* high-altitude flights, or take-off and/or touch-down flight segments [83]. We could use the technique of Niels *et al.* [215] to reduce clutter, but this would not address the specific task of emphasizing specific flight segments. Also, the method in [215] uses blending to eliminate overdraw, which makes it hard to see individual flight routes.

Figure 9.4 b shows the element-based MoleView on the flight dataset. We select a circular zone of interest by moving the lens to the desired location. Next, we tune the radius and altitude range for the zone of interest using the mouse wheel. The selected altitude range $[v_{min}, v_{max}]$ is shown by the colored bar on the lens's periphery, which moves around the center as the mouse wheel is turned. As we change these two parameters, flight routes are dynamically pushed to the lens periphery or brought back to their original position. The edge control points are moved smoothly the gradient field of DT_P, which yields a smooth visualization, allowing to follow how edges are filtered in or out from the lens. Overall, edges continuously move in or out of the lens as parameters are changed. Edges which are selected in the lens stay unmoved, which makes them easy to spot. The obtained effect reminds of a mole pushing earth (data elements) around as it digs at several locations, hence the name for our technique.

Bundled graph example: We next show element-based exploration for bundled graphs. Data elements s_i are individual control points of the bundled edges. Figure 9.5 a shows the graph in Fig. 9.4 bundled by the skeleton-based edge bundling (SBEB) method presented in Chapter 6. Any other bundling methods can be used equally well, *e.g.* [80, 39, 103]. Compared to the unbundled view (Fig. 9.5 a), bundling reduces clutter and allows us to spot groups of close flight routes. However, we now cannot see the *altitudes* of these flights, *e.g.* if flight connection patterns captured by the bundles are similar or different for different altitudes, given the inherent overdraw. With the MoleView, we select a zone of interest around an agglomeration and push control points for edges in that area matching our altitude filter outside of the bundle. The effect is similar to locally bundling edges within the desired attribute range. Figure 9.5 b-d show this for three altitude ranges (low, medium, and high) at the same location. We additionally emphasize the selection effect by rendering selected elements D^{sel} with their colors as set by

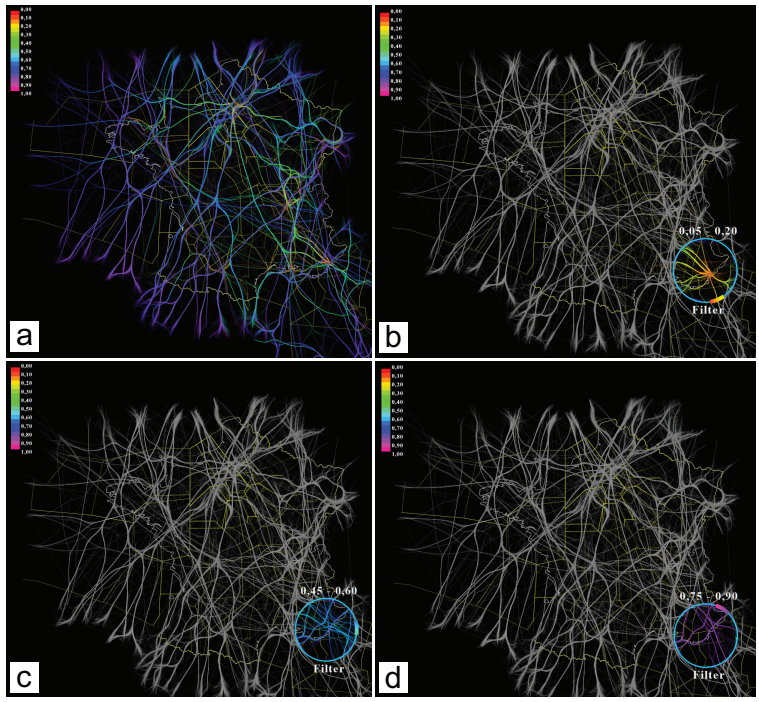

Figure 9.5: Bundled flight trails (a). Attribute-based MoleView lens for three altitude levels (b-d)

the original visualization and desaturate the elements in D^{filt}. We now see that the bundling patterns of these flights are different, *i.e.* plane routes group differently on altitude. The exploration above is useful in answering questions such as whether a certain group of flights (bundle) contains flights of a specific altitude range. If the graph would encode a software system structure, like the one in Fig. 9.10 (discussed further), the question addressed would be whether a given system-to-system connection contains dependencies of a given type.

Image data example: Figure 9.6 shows the element-based lens applied to image data. The elements of our dataset D are pixels in an image. A pixel with image (x, y) coordinates is attributed by its grayscale or color value. Images (a-c) show the lens applied to an ultrahigh-resolution angiography image of the human eye [107]. The attribute filter was selected to retain the bright pixels corresponding to important blood vessels and push the darker pixels away from the focus of interest. The three images show how filtered pixels are pushed away, revealing the

blood vessels in context. Images (d-f) show the lens applied to a color-coded image of the traffic in Lisbon at night [38]. Green hues show relatively slow moving vehicles. This time, the attribute filter was set to work on hues, retaining the green range. The three frames reveal the slow motion traffic close to the focus of interest. However, the spatial map context is preserved, as filtered pixels are gradually pushed away from the focus (or brought back in, when releasing the mouse). In contrast, traditional value-based filtering would not preserve the context but abruptly eliminate elements out of the attribute range of interest from the visualization.

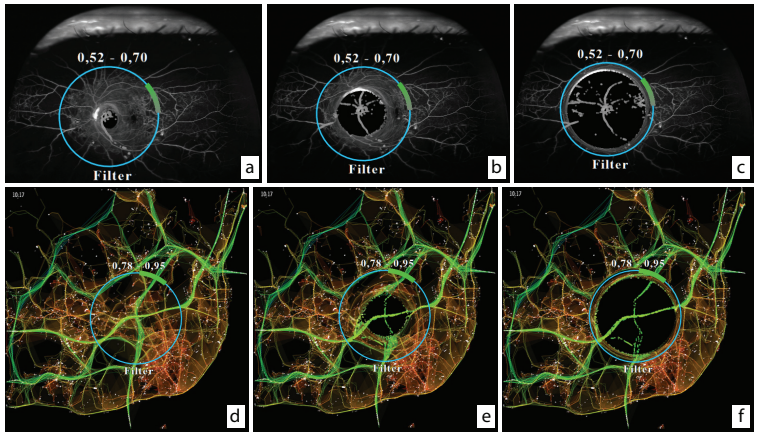

Figure 9.6: Element-based MoleView applied to grayscale angiography image (a-c) and color-mapped traffic speed image (d-f)

9.2.2 *Bundle-based exploration*

Our second scenario, called *bundle-based exploration*, considers the more specific case of a dataset D representing a bundled graph. Data elements s_i are now either individual edge control points, entire edges, or entire bundles. Such datasets can be obtained using one of the many available bundling methods [78, 39, 80, 103]. As explained in Chapter 2, bundled layouts provide simplified visualizations of large graphs but also increase overdraw. This makes it difficult to understand which edges exactly are part of a given bundle, unless the bundling is data-driven, which is not the case in all examples we are aware of. For instance, hierarchical edge bundles (HEBs) used in software visualization have proved of limited success beyond as-

sessing the overall modularity of a system [78, 79]. Such edges are annotated with data attributes *e.g.* type of dependency (call, uses, inherits, includes, reads, writes, owns), or number of times and moment when a function gets called. Real-world software comprehension tasks such as reverse-engineering, architecture quality assessment, and performance assessment need to understand how such attributes are distributed over the edges in a bundle.

Given a control set and zone of interest defined by the user (Sec. 9.2.1), we consider a bundled layout L^b and an unbundled layout L^u of the explored graph . We now apply our semantic lens pipeline (Sec. 9.2) by setting the original and deformed layouts L and L^{filt} to the bundled and unbundled layouts L^b and L^u respectively. The deformation Δ smoothly interpolates between the two layouts rather than moving points away from the zone of interest as for the element-based exploration (see Fig. 9.7):

$$\Delta(t, p_i) = \lambda(t)L^b + (1 - \lambda(t))L^u \tag{9.6}$$

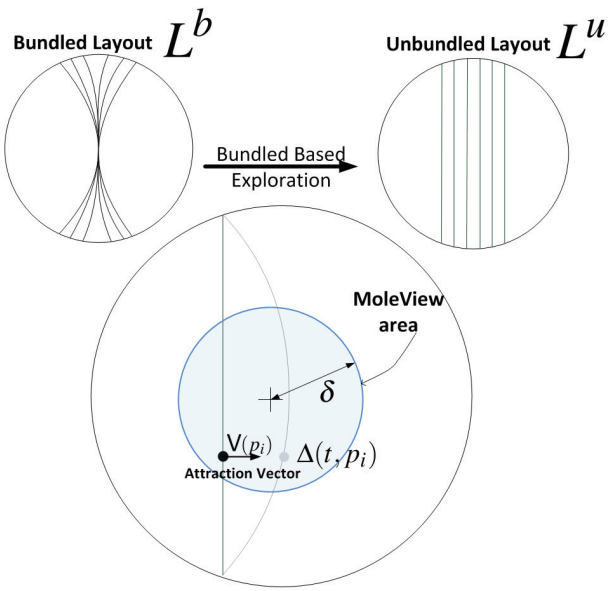

Figure 9.7: MoleView bundle-based exploration mode

Just as for the distance-field-based deformation (Eqn. 9.4), different speed profiles λ can be used to control the animation. The

attraction term **V** (Eqn. 9.5) is identical to the element-based exploration. When the lens is deactivated, displaced elements snap back smoothly from the positions in one layout to the positions in the second layout.

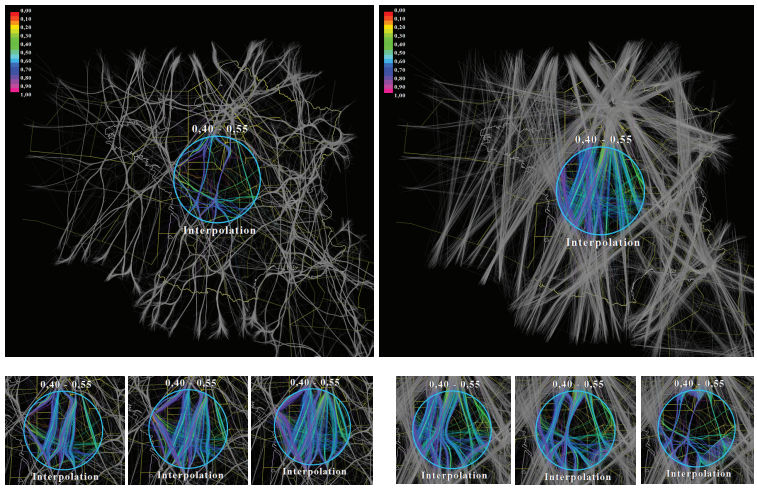

Figure 9.8: Bundle-based exploration (Sec. 9.2.2). Local unbundling (left). Local bundling (right)

Point-level exploration: Figure 9.8 left shows the bundle-based lens for the flights graph. Compared to Fig. 9.5, filtered elements are now moved towards their unbundled locations rather than being pushed towards the lens periphery, yielding a smooth local transition between the bundled and unbundled layouts for the selected edge portions. Since the lens uses both position and attribute values, this is different than simply unbundling the *entire* bundle in the zone of interest. The reverse scenario where selected elements are moved towards their unbundled layout is obtained by applying the deformation (Eqn. 9.6) on the set D^{sel} rather than on D^{filt} (see Fig. 9.8 right). In this case, the lens supports the task of locally showing selected elements in their original spatial context, and filtered elements using the simplified bundled view.

By swapping the layouts L^b and L^u in Eqn. 9.6 and applying the lens on an unbundled graph, we obtain two complementary effects, *i.e.* we can locally bundle selected elements while leaving all filtered elements at their original locations, or locally bundle filtered elements leaving all selected ones at the original locations. These scenarios are useful when the user wants to

keep the original *context* (unbundled graph) and wants to apply the structural simplification (bundling) on the *focus* zone. Figures 9.8 illustrate the above scenarios.

Edge-level exploration: We can also apply our lens on entire edges. Elements s_i of our dataset D are now whole edges rather than edge control points. The method stays the same, but we now apply the deformation (Eqn. 9.6) to *all* control points of edges in the lens rather than to points in the lens. The lens has now bundles (or unbundles) an entire set of selected edges (Fig. 9.9). Here, flights through the Paris area are smoothly bundled, while other flights are drawn at their original locations. This is useful when one wants to explore a set of trals in detail, *i.e.* see them in their entirety in their original positions, rather than applying unbundling to a spatially confined region.

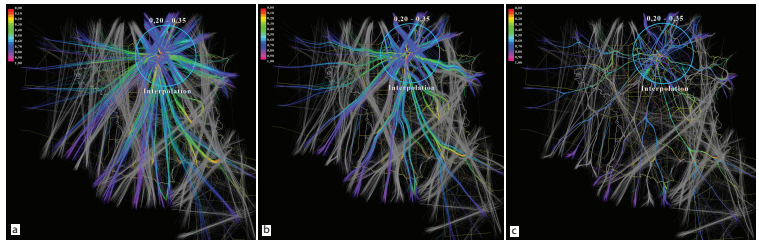

Figure 9.9: Smooth bundling of entire flight paths within a zone of interest. The original unbundled dataset (a) is gradually bundled within the zone of interest (b), finally yielding the bundled dataset (c)

Bundle-level exploration: At the coarsest level, we can apply our lens on entire *bundles*. For this, we need explicit bundle identities as groups of edges. Given a bundled layout, we compute such edge groups, or clusters, using the bottom-up hierarchical agglomerative clustering scheme based on Euclidean distance between edge control points presented in Chapter 5. This gives a partition $C = \{c_i\}$ of the edge set in the input graph into clusters which contain edges which we visually perceive as a bundle. Other edge clustering schemes can be used, if desired.

Given such a partition C of the edge set, we can now directly use our lens on entire bundles by considering a whole bundle as a data element s_i in any of the exploration modes described above. The advantage is now that users can brush a single branch of a bundle and directly explore the entire bundle. Figure 9.10 shows this for a radial layout depicting the structure

of a software system (nodes are software entities, while edges are dependencies). Bundles are explicitly identified using edge clustering and assigned different colors (a). Alternatively, this can be done by clustering edges relating specific coarse-scale subsystems, if a software containment hierarchy is available. Local unbundling reveals the structure of a specific zone of interest (b). This can be useful *e.g.* if edges are colored on another attribute than the one used for bundling, *e.g.* edge type, as it allows one to explore the different types within a bundle, without modifying the overall bundled layout. If we consider entire edges as elements, the lens can be used to unbundle one or more entire bundles under the lens (b). Finally, we can combine the local and whole-edge unbundling effects to achieve a two-stage unbundling effect (c). When animated, this gives additional cues as to the identities of the bundles brushed by the lens, but keeps clutter minimal within the lens area. This is useful *e.g.* when we do not use colors to show bundle identities and users are interested in seeing all edges within a certain bundle passing through a spatial region.

9.2.3 *Dual-layout exploration*

Our third scenario, called *dual-layout exploration*, considers a dataset D which is explored via two completely different spatial layouts. An example thereof are images, seen either as pixels arranged according to a Cartesian layout or histogram layout. The two layouts serve different purposes: the Cartesian one allows finding specific shapes; the histogram shows data value distributions. Typical visualizations interested in above aspects use two views linked via brushing and/or selection. However, as outlined earlier, a two-view mode is suboptimal as it requires users to explicitly correlate two images. This applies even more so if correlation is needed only at certain zones of interest.

We can use our semantic lens (Sec. 9.2) to address the correlated exploration of datasets which use different layouts for different views on the data. To illustrate this, we consider two layouts of an image: the inherent Cartesian layout L^C of pixels in the image, and a polar coordinate plot L^P with hue mapped to the angular axis and saturation mapped to the radius. Value (luminance) plays the role of the attribute values v_i of our data elements which are affected by the attribute filter.

To apply the semantic lens, we define a time-dependent deformation $\Delta(t)$ which links the positions of corresponding data

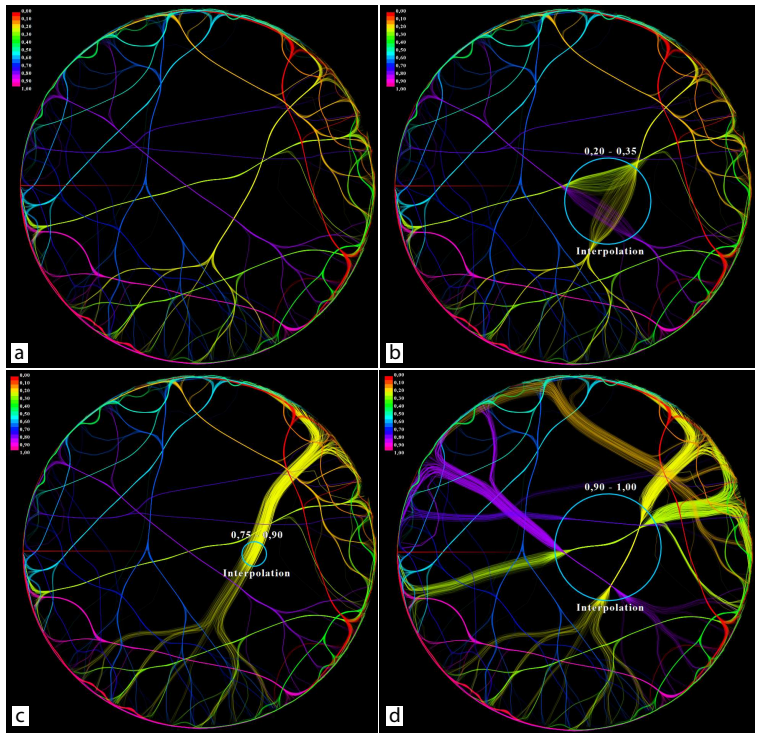

Figure 9.10: MoleView applied at bundle level on a software dependency graph using a radial layout. Original bundled graph, with bundles colored by bundle id (a). Local unbundling effect (b). Whole-edge unbundling (c). Combined local and whole-edge unbundling (d).

elements (pixels) p_i^C and p_i^P in the two layouts L^C and L^P respectively

$$\Delta(t, p_i) = \lambda(t)p_i^C + (1 - \lambda(t))p_i^P \tag{9.7}$$

Compared to the element-based exploration mode, our goal is now different: We wish to correlate the spatial distribution of data elements in two layouts rather than filter away elements having a certain attribute range. For this, we apply our semantic lens on all elements falling within the zone of interest, *i.e.* $D^{sel} = D^Z$.

Figure 9.11 and Figure 9.12 illustrate the dual-layout on a simple image containing the full color spectrum. Fig. 9.11 uses a HSV polar layout L_P in which the zero hue value, red, is at the top (as shown by the arrow). When applying the dual-layout

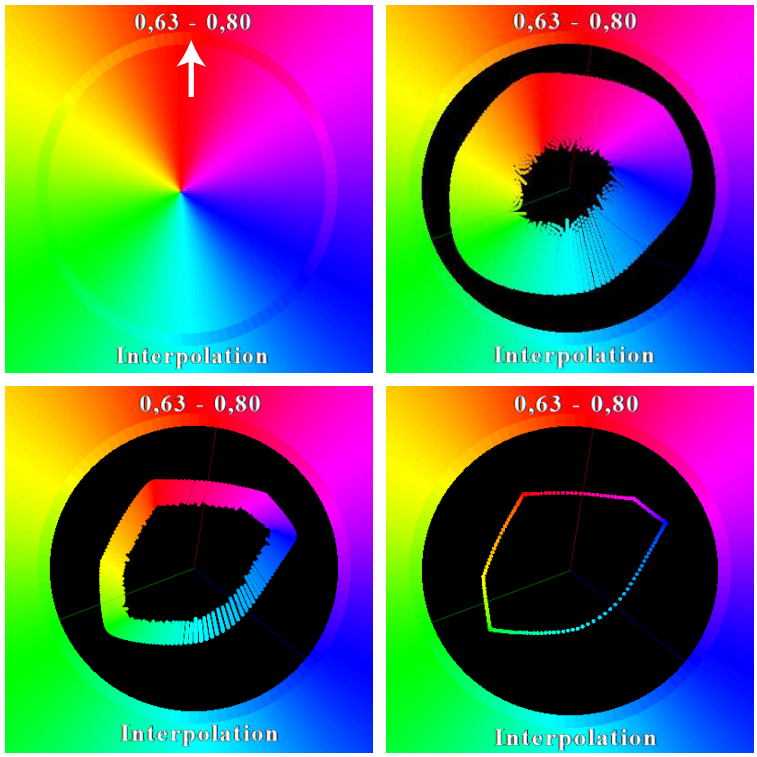

Figure 9.11: Dual-layout lens applied to a simple image. Origin on the angle axis is indicated by the arrow.

lens, pixels are smoothly advected in the deformation field $\Delta(t)$ (Eqn. 9.7) from their location in the Cartesian layout L_C, *i.e.* original image to their location in the HSV polar coordinate layout L_P. This allows the user to locally query an image and see the hue and saturation distribution over that zone of interest. If we draw the points in L_P using alpha blending, we effectively obtain a histogram of the hues and saturations of the pixels in the zone of interest.

Figure 9.13 and Figure 9.14 show the dual-layout lens applied to two color-mapped scalar fields. The first field (Fig. 9.13)) shows the frequency of lightning occurrences on the surface of the Earth with a heat colormap (cold colors = low frequency, hot colors = high frequency) [124]. Using the dual-layout lens, we see that zones in the geographical areas (b) and (c) have a similar distribution of lightning occurrences: the pixel pattern in the HSV space within the lens is nearly identical. This is not evident from the original image, since the pixels in the two indicated

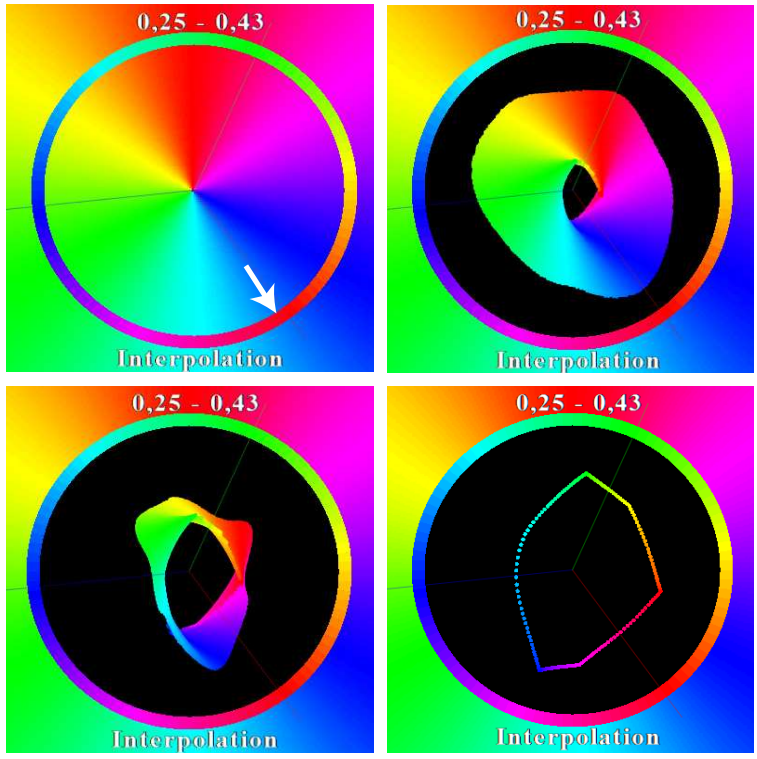

Figure 9.12: Dual-layout lens applied to the same simple image in Fig. 9.11, with a different rotation of the HSV space. Origin on the angle axis is indicated by the arrow.

regions have relatively complex color patterns. In contrast, the zone under the lens in figure (d) shows a different pixel color distribution than the zones (b) and (c) – the green-blue 'tail' of the shape we see in the lens in (b) and (c) is now missing. This indicates that this geographical zone has no lightning frequencies corresponding to these value ranges. Again, the original image (a) does not show this – the pixel color patterns in the three regions are looking relatively similar.

The second scalar field (Fig. 9.14) shows a 3D skeleton, or medial axis, of a cow model. The skeleton is computed using the voxel-based method in [144]. Skeleton voxels are colored with their so-called importance with a blue-to-red rainbow colormap. Less important skeleton points (blue) correspond to small-scale object features, *e.g.* the horns or hoof tips. Most important points (yellow and red) correspond to large-scale object features, like the rump. Skeleton importance can be used to simplify the ob-

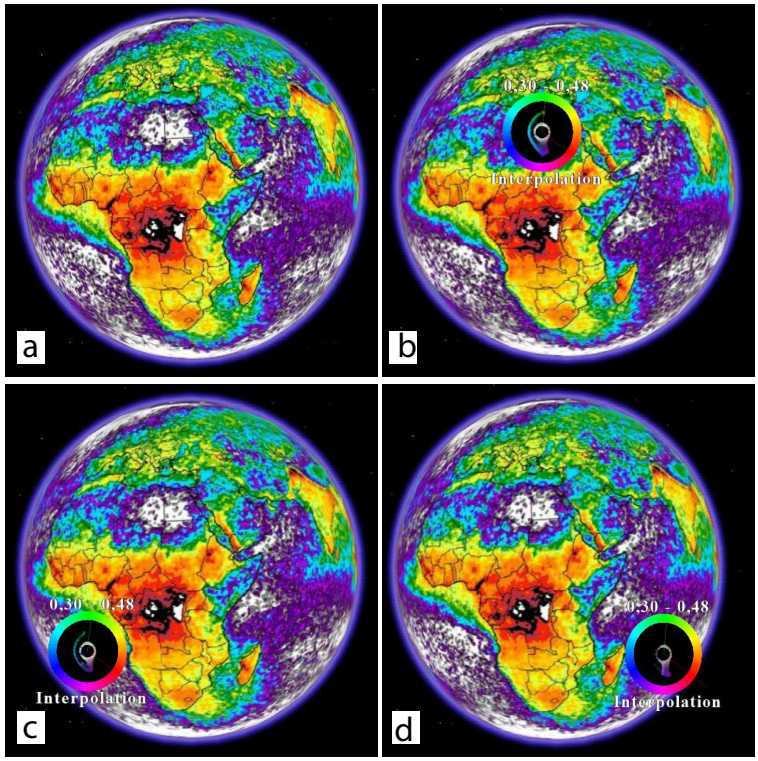

Figure 9.13: Dual-layout lens applied to a color-coded scalar field image: Lightning frequency on the surface of the Earth (heat colormap).

ject by pruning away less important points. The skeletonization method in [144] *conjectures*, but does not rigorously prove or disprove, that the importance of skeleton points varies smoothly over small, connected, areas of the skeleton.

We use our dual-layout lens to investigate this hypothesis. Image (b) shows the lens applied to the head region. We see here a continuous blue-to-green curve, which shows that voxels in this region have, indeed, importances which compactly cover the low-to-medium range. Applying the lens to the back rump region (c) shows, as expected, a broader color spectrum, since points in this area have importances spanning from very low (blue) to highest in the model (red). However, this curve is not continuous, but broken in the yellow range. This indicates that there are no voxels here with medium-high importance values, which raises questions on the validity of the conjecture in [144]. Applying the lens to the front rump region (d) shows a sim-

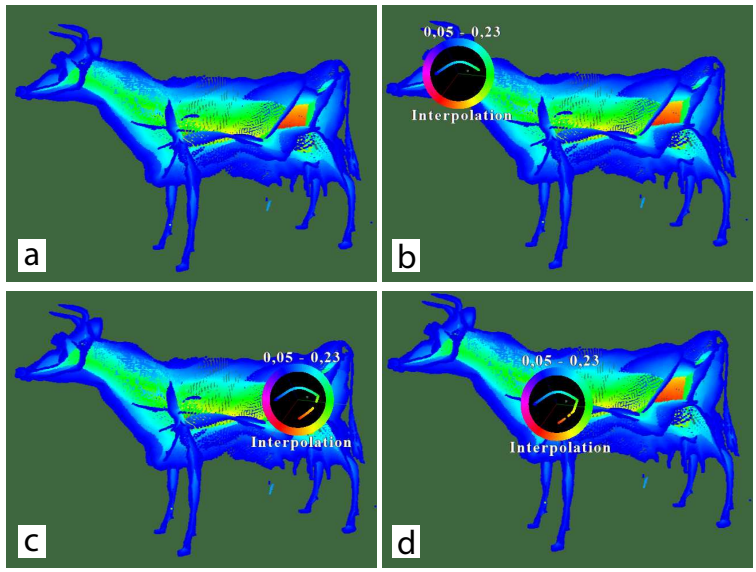

Figure 9.14: Dual-layout lens applied to a color-coded scalar field image: 3D skeleton color-coded by importance (rainbow colormap).

ilar curve as in region (c). Again, we see small interruptions of the curve, which strengthen our supposition that the conjecture may not be valid. Moreover, we see a red portion in the curve, showing that there are high-importance voxels in this area. Manual direct inspection of the model from different viewpoints such as the one shown in (a), however, does not show such voxels, which potentially may lead analysts to the conclusion that the model's highest-importance region is only located in the back rump region. Given that we worked with this 3D skeletonization method and this specific model for about a year in a different project, this was an unexpected result, which we only discovered using the MoleView lens. Close examination revealed the answer: the front rump region does, indeed, contain high-importance voxels, but these are hidden from virtually any viewpoint, as they are located precisely at the intersection of several 3D skeletal manifolds which meet in that region, so they are hardly visible from the outside. Hence, standard examination of the 3D color-coded voxel set did not reveal these outliers, but application of the MoleView lens did.

9.2.4 *Specification of the zone of interest*

The exploration modes described in the previous sections use a simple selection of the zone of interest as one, or several, radial regions determined by user-specified points or foci. Alternatively, whole edges or entire edge bundles that intersect such regions can be selected. However, in more complex scenarios, users are interested to specify zones of interest on a finer-grained, more flexible, level. For example, in the flight visualization, one can be interested to unbundle, or emphasize attribute-based edges, which are part of a given geographical area.

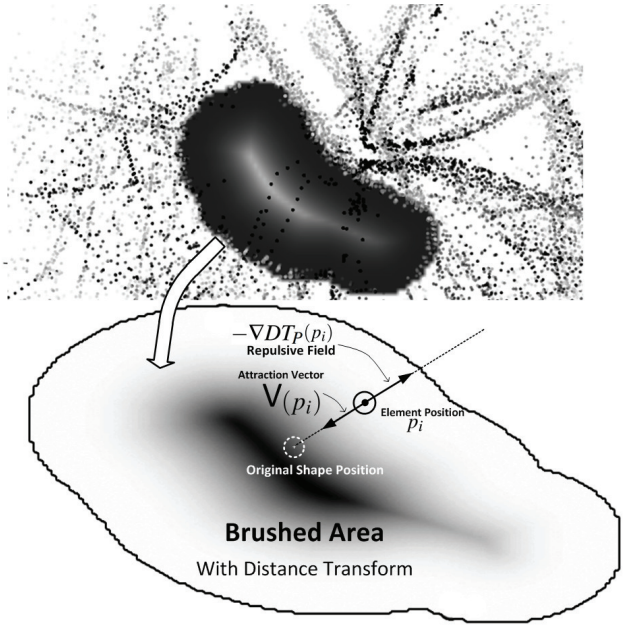

Figure 9.15: Interactively painting zones of interest (see Sec. 9.2.4)

We achieve this by allowing the user to 'paint', or brush, the control set P directly on the screen using the mouse (see Fig. 9.15), by recording the mouse path on the screen, and using this path as control set P. The remainder of our entire method stays identical, as we can directly compute distance transforms of such pixel paths in exactly the same way we do it for individual points (Sec. 9.2.5).

Figure 9.16 illustrates the specification of zones of interest for the flight dataset. Here, the user is interested in seeing low-altitude flights that pass over geographical zones located close

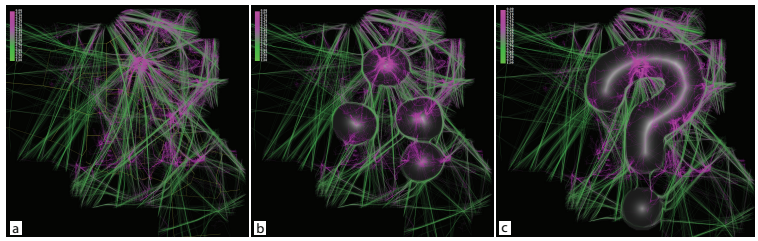

Figure 9.16: Interactive specification of a zone of interest (see Sec. 9.2.4). Flight visualization without lens (a). Focus on low-altitude flights in areas around the main airports (b). Free-form painting of the zone of interest (c). Distance transform profiles are shown in gray

to some main airports in France. Air traffic controllers are particularly interested in such flight patterns for planning purposes as flight routes can get readily crowded in such zones. In Fig. 9.16 b, the user has painted the areas of interest directly on the visualization. Using the element-based exploration lens (Sec. 9.2.1) smoothly pushes away the mid-to-high altitude uninteresting flights (green), revealing the low altitude critical flights (purple). The distance transform profiles for the brushed zones are shown in grayscale (black=high distance, white=low distance to the control set). Using the same mechanism, arbitrarily complex zones of interest can be easily painted, see *e.g.* Figure 9.16 c for a freehand example.

9.2.5 *Implementation*

The MoleView lens can be efficiently and easily implemented atop of any existing visualization metaphor consisting of several discrete, data-attributed, elements with 2D spatial positions, as follows.

First, we compute the distance transform DT_P of the control set (Eqn. 9.1, Sec. 9.2.1) using augmented fast marching method (AFMM) [179]. The shape on which the AFMM is computed is identical to the control set P, which is interactively drawn by the user, as explained previously. Besides the distance transform, the AFMM also delivers the feature transform of its input shape $FT_P : \mathbf{R}^2 \rightarrow \mathbf{R}^2$ defined as

$$FT_P(x \in \mathbf{R}^2) = \mathrm{argmin}_{y \in P} \|x - y\| \tag{9.8}$$

Since $|FT_P| = -\nabla DT_P$ [179], we can obtain in this way the gradient field we need for deformation with no numerically sensitive operations such as differentiation, regardless of the complexity of the input image. The AFMM efficiently computes the distance and feature transform of an image of 800^2 pixels in roughly 0.25 seconds on a typical 2.5 GHz modern PC. The complexity of the AFMM is $O(N \log N)$ for an image of N pixels. If desired, a significantly faster CUDA-based implementation of distance and feature transforms can be used [176], which provides $O(N)$ complexity and treats images of 800^2 pixels on 0.02 seconds per image on a Nvidia GT 330M. Performance is important when specifying user-drawn zones of interest (Sec. 9.2.4), since such zones may have arbitrarily complex shape, as compared to the simple set of points shown in Secs. 9.2.1 and 9.2.2. Using the above, our entire method can be implemented to achieve real-time frame rates on a typical modern PC for datasets having hundreds of thousands of data elements. For large datasets, implementing the displacements (Eqns. 9.4, 9.6 and 9.7) on the GPU using CUDA is straightforward, as these are independent, simple, point operations.

9.3 DISCUSSION

Animation is a key element to the effectiveness of MoleView: by continuously (and smoothly) changing the position of the points affected by the lens, users can brush through a dataset and obtain a continuous, smooth, change of the visualization. The continuous effect is also present when the lens is toggled between activated and deactivated states: points smoothly move as affected by the lens at activation, or move back to their original position as the lens is deactivated. This type of motion allows the creation of a focus-and-context effect. As opposed to other techniques, this is realized by position changes in time, rather than just spatial distortions. Hence, even when the user does not move the lens, the visualization changes smoothly. The same holds for situations when the lens is moved.

The MoleView and the bubble variant of EdgeLens [219] produce similar results in particular cases. Specifically, this happens if the control set is a set of discrete, relatively widely spaced, points, and we do not apply the attribute filter. However, there are several differences, as follows. First, MoleView is not specifically limited to decluttering edges in node-link diagrams, but can be applied essentially to any set of discrete

elements which have data and 2D position. Examples shown here demonstrate this for bundled and unbundled graphs, scatterplots, and images. For this, the usage of a general advection field, rather than controlling edge shapes using Bézier curves as in EdgeLens, is essential. In particular, the field used to morph an image to its pixel color histogram, is computed by using the two layouts of the image and HSV histogram respectively (Sec. 9.2.3). Another important ingredient of MoleView is the ability to select the attribute range to act upon. This allows one to explore based on data *and* spatial position rather than spatial position only as in EdgeLens. As such, MoleView and EdgeLens address overlapping, but not identical, use-cases.

Our control set (Sec. 9.2) is a general subset $P \subset \mathbf{R}^2$, specified *e.g.* by direct painting in the visualization. The lens shape, and its repulsion vector field computed using the feature transform FT_P, yield very different deformation patterns than displacing a set of control points under the influence of a few discrete foci as in EdgeLens. Specifically, FT_P yields a locally smooth field wherever the control set P does not have strong curvature discontinuities, as known from medial axis theory [156]. Practically, if the user draws P a a set of lines, this field will always be smooth if the lines do not intersect. At intersection points, there is only a null set of discontinuities corresponding to the feature points of the branching points of the skeleton S_Z of the zone of interest Z [179, 156]. For example, if the user draws P as$n$ lines which intersect *exactly* in the same point, we will have n such discontinuities. This poses no robustness or quality problems in practice when advecting elements in FT_P, since these are moved *away* from S_Z.

An attractive aspect of the MoleView set of techniques is that they can be added with minimal intrusion to existing visualizations in a postprocessing phase, *e.g.* without having access to the actual engines which compute multidimensional scaling layouts or bundled edge layouts. In particular, for image data the dual-layout exploration presented in Sec. 9.2.3 can be used directly on 2D image data generated by other applications, without access to the actual underlying data points or, for the application in Fig. 9.14-b-d, the 3D voxel data.

Strictly speaking, the bundle-based exploration lens (see Section 9.2.2) can be seen as a particular case of the more general dual-layout exploration lens, where the two layouts L^u and L^b co-exist in the same conceptual space. The difference is that the dual-layout lens propose a more aggressive semantic change – it changes the meaning of the space within the lens from a Carte-

sian (RGB) plot to a polar (HSV) plot. In contrast, the meaning of both the bundled and unbundled layouts is less different. As such, we choose to allow bundled and unbundled data elements to co-exist in the lens area, whereas in the image use-case the lens are shows only one of the RGB or HSV layouts.

9.4 CONCLUSION

In this chapter, we have presented MoleView, a set of interactive lens techniques for the exploration of large datasets rendered as sets of 2D objects. The principle of the MoleView is based on a combination of attribute-based filtering with local displacements of the data points in a force field determined by the zone of interest and dataset layout values. Three exploration modes are presented. The element-based mode repels filtered data points in a distance field, thus unearthing specific data values which may be obscured due to overdraw. The bundle-based mode locally deforms a bundled layout into an unbundled one or conversely, thus helping users to dig into the structure of tight bundles for edges having specific data values. This mode can be applied to any edge bundling layout (EBL), such as the HEB [78], the image-based edge bundling (IBEB) method presented in Chapter 5, the skeleton-based edge bundling method (SBEB) presented in Chapter 6, or other EBLs mentioned in Sec. 2.2.2. Finally, the dual-layout mode smoothly interpolates point positions between two different layouts which highlight different data aspects allowing the user to correlate between the two data views. From this perspective, the interaction techniques presented here alleviate one of the core problems of EBLs mentioned in Chapter 2, *i.e.* the inherent trade-off between clutter and overdraw. Specifically, overdraw is locally, and interactively, eliminated to uncover details of the bundled edges, while the global context is still left bundled, thus rendered with low clutter.

The set of interaction techniques presented in this chapter can be further extended with additional use-cases. For example, the attribute filter can be made to operate on a histogram of the data values in the lens rather than the values themselves, allowing users to select data outliers from a large mass. Secondly, the dual-layout exploration lens principle can be applied to other layouts than Cartesian RGB plots and HSV polar plots, *e.g.* to smoothly interpolate between completely different graph layouts for graph exploration or between different 2D plots which

show pairs of dimensions in a multivariate dataset in a single view.

This chapter is based on:

Christophe Hurter, Ozan Ersoy, and Alexandru Telea. MoleView: An Attribute and Structure-Based Semantic Lens for Large Element-Based Plots. *IEEE Transactions on Visualization and Computer Graphics*, **17(12)**, 2364-2373 (2011).

10 DISCUSSION & CONCLUSIONS

In this final chapter, we revisit our initial research questions, stated in Section 1.2, and compare our obtained results against these questions. We reflect on the completeness of our results, and further outline possible directions for future work.

First let us revisit our research questions:

1. Is edge bundling an effective instrument for the understanding of large graphs as compared to more classical node-link graph visualization techniques?

To answer this question, we use the formative user evaluation performed in a research context presented in Chapter 3, and the analysis of utilization results of the SolidSX toolset in research and the IT industry, presented in Chapter 4. Our comparison in Chapter 3 of straight-line node-link layouts with hierarchical edge bundles (HEBs), performed on several compound software dependency graphs ranging from small ones (hundreds to thousands of edges) up to large ones (hundreds of thousands of edges), emphasize several points. First, node-link layouts are dominated by visual clutter for graphs larger than a few hundred edges. In contrast, we can read coarse-scale dependency groups (bundles) on HEBs relatively easily even for the larger graphs. Secondly, we see that the readability of HEBs decreases with size, due to the inherent edge overlap built in the HEB method. Finally, we see that the effectiveness of HEBs cannot be judged in isolation from the other parts of the embedding software visual analytics (SVA) pipeline.

The studies presented in Chapter 4 refine and strengthen the last point noted above. We present several program comprehension scenarios in the IT industry where HEBs play a key role. We see that the initial observations emerging from our research-context study (Chapter 3) only get strengthened in this context. Most importantly, we see that the *perceived* end-to-end effectiveness of HEBs, in the eyes of their users, crucially depends on the design and implementation of the *entire* SVA pipeline in which they are embedded. Interestingly, most criticism voiced by our tools' users was not on the HEB method itself, but on other aspects of the SVA pipeline, such as completeness and ease-of-automation of fact extraction, ease of configuration, scalability,

applicability to multiple programming languages, and interoperability. Although such observations have been made earlier by several other researchers in software visualization [143, 99, 31], their implications seem, in our eyes, to be minimalized by many current evaluation studies on the effectiveness of software visualization techniques which focus on evaluating a technique independently on its end-to-end usage context.

The studies presented in Chapters 3 and 4 also outline one important limitation of edge bundling layouts (EBLs) for large graphs: The difficulty of visually following bundles to their end-nodes. In Chapter 5, we present image-based edge bundles (IBEB), a technique which simplifies the rendering of a given EBL to emphasize its main, coarse-level, bundles. Although IBEB cannot show the fine-grained structure of an EBL, it succesfully solves the issue of showing the graph's main connectivity pattern on a coarse scale, even in the presence of many bundle overlaps.

2. How can we design edge bundling techniques which computationally scale to handle large graphs?

To answer this question, we follow the path opened in Chapter 5. Namely, we cast the EBL problem as an image processing problem, and construct EBLs by manipulating two-dimensional images rather than working on the discrete graph structure. This is a fundamentally different approach to graph bundling as compared to earlier methods, for several reasons. From a computational aspect, this allows us to *scale* the EBL computation in two directions – namely, by using a higher or lower image resolution, in line with the available computational resources; and by using the massive parallelism offered by modern graphics cards (GPUs). From the more interesting theoretical aspect, we believe that our image-based approach to EBL computation represents an important paradigm shift in graph visualization, for the following reasons. First, we re-cast the graph bundling problem using the formal, well understood, framework provided by distance functions and shape skeletons (Chapter 6). Secondly, we show that graph bundling is equivalent to the well-known mean-shift image processing technique [34] (Chapter 7). These observations allow to efficient EBL algorithms, and also incorporate complex constraints such as avoiding obstacles of any shape (Chapter 7). Furthermore, these observations allow us to reason formally and quantitatively about the robustness, convergence, and complexity of our proposed EBL algorithms, by reusing known results from image and shape processing.

Our image-based analogy to graph bundling further allows us to easily extend EBL to the realm of dynamic graphs (Chapter 8). For streaming graphs, the extension is trivial, amounting simply to identifying the bundling iteration count to the graphs' time stamp. For sequence graphs, we solve the dynamic bundling problem by adding correspondence information between edges in subsequent graphs. In both cases, the inherent high scalability of our bundling methods ensures that we can bundle large dynamic graphs at interactive rates without additional (implementation) effort.

3. How can we complement edge bundling techniques with rendering and interaction techniques to simplify the resulting images, and also add more contextual information, for a better understanding?

We address this question at two different levels. First, we propose a number of rendering techniques which both simplify a computed EBL and also add supplementary information to it. In Chapter 5, we show how we can shade simplified bundles to emphasize the coarse-level graph structure, using the well-known shaded cushions metaphor [200]. In Chapter 6, we show how a similar effect can be achieved by using computationally simpler mechanisms which operate at the level of a single edge. In Chapter 7, we show how outlier edges, which cannot be grouped into bundles, can be additionally emphasized in the graph rendering. Finally, in Chapter 8, we show how bundle stability (or the lack thereof) can be encoded into hue and transparency. Secondly, we recognize that, for large graphs, the inherent overdraw caused by EBLs prevents showing edge-level details for each edge. We address this issue in Chapter 9 by a set of focus-and-context interactive techniques which locally disentangle a given EBL to show more information, while in the same time keeping the global simplified context offered by EBLs. Apart from demonstrating these interactive techniques on EBLs, we show how they can be used also for other types of element-based plots, such as 2D or 3D images.

10.1 FUTURE WORK

The results presented in this thesis can lead to future research in a number of directions. According to our insights, these directions are as follows, in decreasing order of impact for the information visualization community:

1. Bundling theory: Our parallels between graph bundling and image-processing operations such as distance transforms, skeletons, and mean shift segmentation, open an interesting new direction for (quantitatively) reasoning about edge bundling. So far, many EBL algorithms have been proposed in the literature. However, a formal definition of what an EBL is, and which are its measurable properties, lacks. Further refining the analogy between EBLs and the aforementioned image-processing operations could lead to such a formal definition, and also to ways to quantify, and formally reason about, the desirable properties of an EBL. A first (and simple) example hereof is the analogy between graph bundling and image density sharpening presented in Chapter 7.

2. Bundling customization: Recasting the bundling problem in an image-based setting allows us to define global and local bundling constraints in a very flexible manner. An example is the bundle obstacle avoidance described in Chapter 7. Pursuing this path can lead to the design of efficient and effective EBL methods that incorporate more constraints, such as constrained bundle directions for diagram-like drawings.

3. Visualizing attributes: The inherent overdraw caused by bundling makes it fundamentally hard to show individual edge attributes within a bundle. When edges have more than one attribute, or when these attributes change in time, the problem becomes only harder. The image-based techniques presented in Chapters 5 and 6 could be extended to construct colored, textured, and shaded shapes which aggregate such attributes at a pixel level to convey detailed insight on the information present in the bundled edges. A major advantage of image-based techniques here is that such techniques can synthesize the color of each image pixel separately, and thus offer fine-grained levels of control of what, and how, is shown at each such pixel.

4. Bundling applications: Our current applications have been, for practical constraints, been limited to graphs emerging from software engineering and flight data analysis. However, bundling has a high potential to be effective in simplifying other types of relational-and-spatial structures. One salient example we think of is the simplified visualization of neural fiber tracts computed using tractography techniques from 3D diffusion tensor magnetic resonance imaging (DT-MRI) scans [58]. Showing the simplified structure of such datasets using image-based bundling

has a high potential. One additional major challenge, as compared to our information visualization context, is that spatial positions along the edges is highly significant. Hence, new research is needed to study ways to limit the impact of the inherent deformation produced by bundling, while at the same time offering the structural simplification that EBLs offer.

BIBLIOGRAPHY

[1] J. Abello, F. van Ham, and N. Krishnan. AskGraphView: A large graph visualisation system. *IEEE TVCG*, 12(5): 669–676, 2006.

[2] AbsInt Inc. aiSee graph layout software, 2010. `www.aisee.com`.

[3] N. Andrienko and G. Andrienko. *Exploratory analysis of spatial and temporal data: a systematic approach*. Springer, 2006.

[4] D. Archambault, T. Munzner, and D. Auber. Grouse: Feature-based and steerable graph hierarchy exploration. In *Proc. EuroVis*, pages 67–74, 2007.

[5] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE TVCG*, 14(4):900–913, 2008.

[6] S. Arya and D. Mount. Approximate nearest neighbor searching. In *Proc. ACM Symp. on Discrete Algorithms*, pages 271–280, 1993.

[7] AT&T. The graphviz package, 2010. `www.graphviz.org`.

[8] D. Auber. Visualization of large graphs in the Tulip system, 2003. PhD thesis, U. of Bordeaux, France.

[9] D. Auber. Tulip graph visualization framework, 2011. `tulip.labri.fr`.

[10] D. Auber, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois, and G. Melançon. The tulip 3 framework: A scalable software library for information visualization applications based on relational data. Technical report RR-7860, INRIA, 2012.

[11] Zs. Balanyi and R. Ferenc. Mining design patterns from C++ source code. In *Proc. ICSM*, pages 305–314. IEEE, 2003.

[12] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proc. ACM SOFTVIS*, pages 165–172, 2005.

[13] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.

[14] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE*, pages 625–634. IEEE, 2004.

[15] F. Beck and S. Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 2012. DOI: 10.1007/s10664-012-9225-9.

[16] Bell Labs. CScope, 2007. cscope.sourceforge.net.

[17] F. Bertault and M. Miller. An algorithm for drawing compound graphs. In *Proc. Graph Drawing*, pages 197–204, 1999.

[18] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. H. Gros, A. Camsky, S. McPeak, and D. Engler. A few billion of lones of code later: Using static analysis to find bugs in the real world. *Comm. of the ACM*, 53(2):66–75, 2010.

[19] E. Bier, M. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and magic lenses: The see-through interface. In *Proc. ACM SIGGRAPH*, pages 137–145, 1993.

[20] E. Bier, M. Stone, and K. Pier. Enhanced illustration using MagicLens filters. *IEEE CG & A*, 17(6):62–70, 1997.

[21] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Adv. Comput.*, (62):105–178, 2004.

[22] C. Binuccia, U. Brandes, G. Di Battista, W. Didimo, M. Gaertler, P. Palladino, M. Patrignani, A. Symvonis, and K. Zweig. Drawing trees in a streaming model. *Inform. Process. Lett.*, 112(11):418–422, 2012.

[23] F. Boerboom and A. Janssen. Fact extraction, querying and visualization of large C++ code bases. MSc thesis, Faculty of Math. and Computer Science, Eindhoven Univ. of Technology, 2006.

[24] I. Boyandin, E. Bertini, and D. Lalanne. A qualitative study on the exploration of temporal changes in flow maps with animation and small-multiples. *Comp. Graph. Forum*, 31(3):1005–1014, 2012.

[25] D. Bruls, C. Huizing, and J. J. van Wijk. Squarified treemaps. In *Proc. IEEE VisSym*, pages 33–42, 2000.

[26] M. Burch and S. Diehl. TimeRadarTrees: Visualizing dynamic compound digraphs. *Comp. Graph. Forum*, 27(3): 823–830, 2008.

[27] M. Burch, F. Beck, and S. Diehl. Timeline trees: Visualizing sequences of transactions in information hierarchies. In *Proc. AVI*, pages 75–82, 2008.

[28] T. Cao, K. Tang, A. Mohamed, and T. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games*, pages 134–141, 2010.

[29] CGAL. CGAL library, 2009. `http://www.cgal.org`.

[30] D. Chang, M. Kantardzic, and M. Ouyang. Hierarchical clustering with cuda/gpu. In *Proc. ISCA*, pages 130–135, 2009.

[31] S. Charters, N. Thomas, and M. Munro. The end of the line for Software Visualisation? In *Proc. IEEE Vissoft*, pages 27–35, 2003.

[32] W. Chen, S. Zhang, S. Coreia, and D. Ebert. Abstractive representation and exploration of hierarchically clustered diffusion tensor fiber tracts. *Comp. Graph. Forum*, 27(3): 1071–1078, 2008.

[33] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *Proc. IWPC*, pages 134–143. IEEE Press, 2003.

[34] D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE TPAMI*, 24(5):603–619, 2002.

[35] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1999.

[36] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. ICPC*, pages 49–58. IEEE, 2007.

[37] L. Costa and R. Cesar. *Shape analysis and classification: Theory and practice*. CRC Press, 2000.

[38] P. Cruz. Boundaries in information visualization – towards information aesthetics, 2010. MSc Thesis, U. Coimbra, Portugal.

[39] W. Cui, H. Zhou, H. Qu, P. Wong, and X. Li. Geometry-based edge clustering for graph visualization. *IEEE TVCG*, 14(6):1277–1284, 2008.

[40] M. de Hoon, S. Imoto, J. Nolan, and S. Myiano. Open source clustering software. *Bioinformatics*, 20(9):1453–1454, 2004.

[41] Hayco de Jong and Paul Klint. ToolBus: The next generation. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Formal Methods for Components and Objects*, pages 220–241. Springer LNCS, 2003.

[42] M. Dickerson, D. Eppstein, M. Goodrich, and J. Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. In *Proc. Graph Drawing*, pages 1–12, 2003.

[43] S. Diehl. *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[44] D. Dobkin, E. Gansner, E. Koutsofios, and S. North. Implementing a general-purpose edge router. In *Graph Drawing*, pages 262–271. Springer, 1997.

[45] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proc. ICSE*, pages 367–375, 2007.

[46] S. Ducasse and O. Nierstrasz. On the effectiveness of clone detection by string matching. *Intl. J. on Software Maintenance and Evolution*, 18(1):37–58, 2006.

[47] T. Dwyer and L. Nachmanson. Fast edge-routing for large graphs. In *Graph Drawing*, pages 147–158. Springer, 2010.

[48] T. Dwyer, K. Marriott, and M. Wybrow. Integrating edge routing into force-directed layouts. In *Proc. Graph Drawing*, pages 8–19, 2007.

[49] Eclipse project. Eclipse CDT framework for C/C++, 2010. `www.eclipse.org/cdt`.

[50] H. Edelsbrunner and E. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994.

[51] S.G. Eick, J.L. Steffen, and E.E. Sumner. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Soft. Eng.*, 18(11):957–968, 1992.

[52] G. Ellis and A. Dix. An explorative analysis of user evaluation studies in information visualisation. In *Proc. AVI Workshop on Beyond Time and Errors: Novel Evaluation methods for information visualization*, 2006.

[53] G. Ellis and A. Dix. A taxonomy of clutter reduction for information visualisation. *IEEE TVCG*, 13(6):1216–1223, 2007.

[54] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability and its Applications*, 14:153–158, 1969.

[55] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareira, and A. Telea. Skeleton-based edge bundles for graph visualization. *IEEE TVCG*, 17(2):2364 – 2373, 2011.

[56] C. Erten, S. Kobourov, V. Le, and A. Navabi. Simultaneous graph drawing: Layout algorithms and visualization schemes. In *Proc. Graph Drawing*, pages 437–449, 2004.

[57] M. Ettema and E. Vast. Dependency evolution analyzer, 2010. `www.cs.rug.nl/svcg/SoftVis/DepEvol`.

[58] M. Everts, H. Bekker, J. Roerdink, and T. Isenberg. Depth-dependent halos: Illustrative rendering of dense line data. *IEEE TVCG*, 15(6):1299–1306, 2009.

[59] J. D. Fekete, D. Wang, N. Dang, A. Aris, and C. Plaisant. Overlaying graph links on treemaps. In *Proc. InfoVis (poster)*, pages 82–83, 2003.

[60] N. Fenton and S. Pfleeger. *Software Metrics: A Rigorous and Pracical Approach.* Chapman & Hall, 1998.

[61] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proc. ICSM*, pages 172–181. IEEE, 2002.

[62] D. Forrester, S. Kobourov, A. Navabi, K. Wample, and G. Yee. Graphael: A system for generalized force-directed layouts. In *Proc. Graph Drawing*, pages 454–464, 2004.

[63] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proc. DIMACS'94*, pages 388–403. Springer LNCS, 1994.

[64] Y. Frishman and A. Tal. Uncluttering graph layouts using anisotropic diffusion and mass transport. *IEEE TVCG*, 15 (5):777–788, 2009.

[65] Y. Frishman and Ayellet Tal. Online dynamic graph drawing. In *Proc. EuroVis*, pages 75–82, 2007.

[66] Y. Fua, O. Ward, and E. Rundensteiner. Hierarchical parallel coordinates for exploration of large datasets. In *Proc. IEEE Visualization*, pages 43–50, 1999.

[67] G. Furnas. Generalized fisheye views. In *Proc. ACM CHI*, pages 16–23, 1986.

[68] E. Gansner and Y. Koren. Improved circular layouts. In *Proc. Graph Drawing*, pages 386–398, 2006.

[69] E. Gansner, Y. Koren, and S. North. Topological fisheye views for visualizing large graphs. In *Proc. InfoVis*, pages 175–182, 2004.

[70] E. Gansner, Y. Hu, S. North, and C. Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Proc. PacificVis*, pages 187–194, 2011.

[71] Gccxml Team. The Gccxml C++ parser, 2011. `www.gccxml.org`.

[72] M. Ghoniem, J. D. Fekete, and P. Castagnola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Proc. IEEE InfoVis*, pages 17–24, 2004.

[73] S. Grivet, D. Auber, J. P. Domenger, and G. Melancon. Bubble tree drawing algorithm. In *Proc. Intl. Conf. on Comp. Vision and Graphics*, pages 633–641, 2004.

[74] D. Harel and Y. Horen. Graph drawing by multidimensional embedding. In *Proc. Graph Drawing*, pages 388–393, 2002.

[75] N. Henry and J. D. Fekete. NodeTrix: A hybrid visualization of social networks. *IEEE TVCG*, 13(6):1302–1309, 2007.

[76] I. Herman, G. Melancon, and S. Marshall. Graph visualization and navigation in information visualization: a survey. *IEEE TVCG*, 6(1):24–43, 2000.

[77] R. Holt, A. Winter, and A. Schurr. GXL: Towards a standard exchange format. In *Proc. WCRE*, pages 162–171, 2000.

[78] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12 (5):741–748, 2006.

[79] D. Holten and J. J. van Wijk. Visual comparison of hierarchically organized data. *Comp. Graph. Forum*, 21(4): 759–766, 2008.

[80] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. *Comp. Graph. Forum*, 28(3):670–677, 2009.

[81] H. Hoogendorp, O. Ersoy, D. Reniers, and A. Telea. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *Proc. ACM VISSOFT*, pages 137–145, 2009.

[82] M. Huang, P. Eades, and J. Wang. On-line animated visualization of huge graphs using a modified spring algorithm. *JVLC*, 9(6):623–645, 1998.

[83] C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading data across views to support iterative exploration of aircraft trajectories. *IEEE TVCG*, 15(6):1017–1024, 2009.

[84] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. *Comp. Graph. Forum*, 31(3):435–443, 2012.

[85] Z. Jiang, A. Hassan, and R. C. Holt. Visualizing clone cohesion and coupling. In *Proc. APSEC*, pages 130–137, 2006.

[86] M. Jones, J. Marron, and S. Sheather. A brief survey of bandwidth selection for density estimation. *J. American Stat. Assoc.*, 91(433):401–407, 1996.

[87] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *Proc. ICSE*, pages 98–107. IEEE, 2010.

[88] T. Kamiya. CCfinder clone detector home page, 2010. www.ccfinder.net.

[89] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large-scale source code. *IEEE TSE*, 28(7):654–670, 2002.

[90] I. Kaplan. Implementing graph pattern queries on a relational database. In *Technical Report LLNL-TR-400310*. Lawrence Livermore National Laboratory, USA, 2008.

[91] G. Katz and J. Kider. All-pairs shortest-paths for large graphs on the GPU. In *Proc. Graphics Hardware*, pages 208–216, 2008.

[92] Rick Kazman, Steven Woods, and Jeromy Carriere. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proc. WCRE*, pages 154–163, 1998.

[93] H. Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, Univ. of Victoria, Canada, 2006.

[94] H. Kienle and H. A. Müller. Requirements of software visualization tools: A literature survey. In *Proc. IEEE Vissoft*, pages 92–100, 2007.

[95] H. Kienle and H. A. Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247–263, 2010.

[96] R. Klette and A. Rosenfeld. *Digital geometry: Geometric methods for digital picture analysis*. Morgan Kaufmann, 2004.

[97] KOffice Team. KOffice software repository, 2010. www.koffice.org.

[98] E. Korshunova, M. Petkovic, M. van den Brand, and M. Mousavi. Cpp2XMI: Reverse engineering for UML class, sequence and activity diagrams from C++ source code. In *Proc. WCRE*, pages 297–298, 2006.

[99] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. Soft. Maint. and Evol.*, 15(2):87–109, 2003.

[100] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax trees. In *Proc. WCRE*, pages 253–262, 2006.

[101] I. Kovacs, A. Feher, and B. Julesz. Medial-point description of shape: A representation for action coding and its phychophysical correlates. *Vision research*, 38:2323–2333, 1998.

[102] L. Kwakman. Automatically reducing code duplication. MSc thesis, Univ. of Groningen, the Netherlands, Sept. 2010. `www.cs.rug.nl/~alext/PAPERS/MSc/kwakman10.docx`.

[103] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *Comp. Graph. Forum*, 29(3): 432–439, 2010.

[104] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. In *Proc. Information Visualisation*, pages 329–335, 2010.

[105] M. Lanza. CodeCrawler - polymetric views in action. In *Proc. ASE*, pages 394–395, 2004.

[106] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[107] H. Lawrence and Y. Kulkarni. Anterior segment and fundus photography, 2011. `emedicine.medscape.com/article/1228681-overview`.

[108] A. Lienhardt, A. Kuhn, and O. Greevy. Rapid prototyping of visualizations using Mondrian. In *Proc. IEEE Vissoft*, pages 67–70, 2007.

[109] Y. Lin, R. C. Holt, and A. J. Malton. Completeness of a fact extractor. In *Proc. WCRE*, pages 196–204. IEEE, 2003.

[110] T. Littlefair. C and C++ code counter, 2007. `sourceforge.net/projects/cccc`.

[111] LLVM Team. Clang C/C++ analyzer home page, 2010. `clang.llvm.org`.

[112] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The *Visual Code Navigator*: An interactive toolset for source code investigation. In *Proc. InfoVis*, pages 24–31. IEEE, 2005.

[113] J. Looser, R. Grasset, and M. Billinghurst. A 3D flexible and tangible magic lens in augmented reality. In *Proc. ISMAR*, pages 254–262. IEEE, 2007.

[114] Lua Team. The Lua programming language, 2011. `www.lua.org`.

[115] A. Ludwig. Recoder java analyzer, 2010. `recoder.sourceforge.net`.

[116] A. Marcus, L. Fend, and J. I. Maletic. 3d representations for software visualization. In *Proc. ACM SoftVis*, pages 27–36, 2003.

[117] K. McDonnell and K. Mueller. Illustrative parallel coordinates. *Comp. Graph. Forum*, 27(3):1031–1038, 2008.

[118] S. McPeak. Elkhound: A fast, practical GLR parser generator, 2002. Tech. report UCB/CSD-2-1214.

[119] S. McPeak. The Elsa C++ static analyzer, 2010. `scottmcpeak.com/elkhound/sources/elsa`.

[120] T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.

[121] S. Moreta and A. Telea. Multiscale visualization of dynamic software logs. In *Proc. EuroVis*, pages 11–18, 2007.

[122] T. Moscovich, F. Chevalier, N. Henry, E. Pietriga, and J.-D. Fekete. Topology-aware navigation in large networks. In *Proc. ACM CHI*, pages 320–329, 2009.

[123] Mozilla. Firefox repository, 2012. `http://www.mozilla.org/en-US/firefox/fx`.

[124] NASA Team. Earth lightning map, 2011. `thunder.msfc.nasa.gov/data`.

[125] P. Neumann, S. Schlechtweg, and M. S. Carpendale. ArcTrees: Visualizing relations in hierarchical data. In *Proc. EuroVis*, pages 53–60. IEEE, 2005.

[126] Q. Nguyen, P. Edges, and S.-H. Hong. StreamEB: Stream edge bundling. In *Tech. Report TR-689, Univ. of Sydney, July 2012, ISBN 9781742102801*, 2012. Also to appear in Proc. Graph Drawing.

[127] Q. Nguyen, P. Edges, and S.-H. Hong. StreamEB results, 2012. `http://rp-www.cs.usyd.edu.au/~qnguyen/streameb`.

[128] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. In *Proc. ACM ESEC/FSE*, pages 1–10, 2005.

[129] NSIS Team. NSIS installer, 2012. `nsis.sourceforge.net`.

[130] OINK. The oink C++ static analyzer, 2008. `www.cubewano.org`.

[131] J. R. Pate, R. Tairas, and N. A. Kraft1. Clone evolution: A systematic review. *J. Soft. Maint. Evol. Res. Pract.*, 2012. DOI:10.1002/smr.579.

[132] F. Paulovich, L. Nonato, R. Minghim, and H. Levkowitz. Least square projection: A fast high-precision multidimensional projection technique and its application to document mapping. *IEEE TVCG*, 14(3):564–575, 2008.

[133] D. Phan, L. Xiao, R. Yer, P. Hanrahan, and T. Winograd. Flow map layout. In *Proc. IEEE InfoVis*, pages 219–224, 2005.

[134] S. Pizer, K. Siddiqi, G. Szekely, J. Damon, and S. Zucker. Multiscale medial loci and their properties. *IJCV*, 55(2-3): 155–179, 2003.

[135] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2006.

[136] Prefuse. The Prefuse information visualization toolkit, 2010. `prefuse.org`.

[137] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proc. GD*, pages 248–261, 1997.

[138] H. Qu, H. Zhou, and Y. Wu. Controllable and progressive edge clustering for large networks. In *Proc. Graph Drawing*, pages 399–404, 2006.

[139] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. In *Proc. Conf. Parallel Compilers (CPC)*, pages 81–90, 2000. see also `www.rosecompiler.org`.

[140] M. Raitner. Visual navigation of compound graphs. In *Proc. Graph Drawing*, pages 403–413, 2004.

[141] R. Rao and S. Card. The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In *Proc. CHI*, pages 222–230. ACM, 1994.

[142] Redgate Inc. Reflector .NET API, 2010. `www.red-gate.com/products/reflector`.

[143] S. Reiss. The paradox of software visualization. In *Proc. IEEE Vissoft*, pages 59–63, 2005.

[144] D. Reniers, J. J. van Wijk, and A. Telea. Computing multiscale skeletons of genus 0 objects using a global importance measure. *IEEE TVCG*, 14(2):355–368, 2008.

[145] D. Reniers, L. Voinea, O. Ersoy, and A. Telea. The Solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Sci. Comput. Program.*, 2012. doi:10.1016/j.scico.2012.05.002.

[146] M. Rumpf and A. Telea. A continuous skeletonization method based on level sets. In *Proc. IEEE VisSym*, pages 151–160, 2002.

[147] G. Salton. Developments in automatic text retrieval. *Science*, 253:974–980, 1991.

[148] G. Sander. Graph layout through the VCG tool. In *Proc. Graph Drawing*, pages 194–205. Springer, 1994. see also `rw4.cs.uni-sb.de/~sander/`.

[149] SciTools, Inc. Understand for C/C++, 2010. `www.scitools.com`.

[150] D. Selassie, B. Heller, and J. Heer. Divided edge bundling for directional network data. *IEEE TVCG*, 19(12):754–763, 2011.

[151] M. Sensalire, P. Ogao, and A. Telea. Classifying desirable features of software visualization tools for corrective maintenance. In *Proc. ACM SOFTVIS*, pages 87–90, 2008.

[152] M. Sensalire, P. Ogao, and A. Telea. Evaluation of software visualization tools: Lessons learned. In *Proc. IEEE Vissoft*, pages 156–164, 2009.

[153] M. Sensalire, P. Ogao, and A. Telea. Model-based analysis of adoption factors for software visualization tools in corrective maintenance. Tech. report svcg-rug-10-2010, Univ. of Groningen, the Netherlands, 2010. `www.cs.rug.nl/~alext/PAPERS/Sen10.pdf`.

[154] S. Sheather and M. Jones. A reliable data-based bandwidth selection method for kernel density estimation. *J. of the Royal Statistical Society*, B53(3):683–690, 1991.

[155] B. Shneiderman. Treemaps for space-constrained visualization of hierarchies, 2010. `www.cs.umd.edu/hcil/treemap-history`.

[156] K. Siddiqi and S. Pizer. *Medial Representations: Mathematics, Algorithms and Applications*. Springer, 1999.

[157] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. Zucker. Hamilton-Jacobi skeletons. *IJCV*, 48(3):215–231, 2002.

[158] B. Silverman. Density estimation for statistics and data analysis. *Monographs on Statistics and Applied Probability*, 26, 1992.

[159] SolidSource. SolidSX software explorer, 2009. `http://www.solidsourceit.com/products/SolidSX-source-code-dependency-analysis.html`.

[160] SolidSource BV. SolidSX, SolidSDD, SolidSTA, and SolidFX tool distributions, 2010. `www.solidsourceit.com`.

[161] SolidSource IT. SolidSDD Clone Detector, 2012. `http://www.solidsourceit.com`.

[162] M. Spindler and R. Dachselt. Exploring information spaces by using tangible magic lenses in a tabletop environment. In *Proc. ACM CHI (EA)*, pages 243–248, 2010.

[163] SQLite Team. The SQLite database, 2011. `www.sqlite.org`.

[164] T. A. Standish. An essay on software reuse. *IEEE TSE*, 10 (5):494–497, 1984.

[165] Statistical Computing. US flights dataset, 2012. http://stat-computing.org/dataexpo/2009/the-data.html.

[166] M. Storey and H. Müller. Manipulating and documenting software structures using SHriMP views. In *Proc. ICSM*, pages 275–284, 1995.

[167] R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proc. IEEE VisSym*, pages 221–230, 2003.

[168] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(4): 876–892, 1991.

[169] SVCG. Scientific visualization and computer graphics group, Univ. of Groningen, Software Visualization and Analysis, 2010. www.cs.rug.nl/svcg/SoftVis.

[170] SharpSVN Team. SharpSVN c# library, 2010. sharpsvn.open.collab.net.

[171] A. Telea. An image inpainting technique based on the fast marching method. *J. of Graphics Tools*, 9(1):23–34, 2004.

[172] A. Telea. An open architecture for visual reverse engineering. In *Managing Corporate Information Systems Evolution and Maintenance (ch. 9)*, pages 211–227. Idea Group Inc., 2004.

[173] A. Telea. Combining extended table lens and treemap techniques for visualizing tabular data. In *Proc. EuroVis*, pages 51–58, 2006.

[174] A. Telea. Image inpainting tool source code, 2010. www.cs.rug.nl/~alext/SQAT/Software.

[175] A. Telea. Software quality assurance and testing (sqat) course assignment, 2010. Univ. of Groningen, the Netherlands, www.cs.rug.nl/~alext/SQAT/Assignment.

[176] A. Telea. CUDA skeletonization and image processing toolkit, 2011. http://www.cs.rug.nl/~alext/CUDASKEL.

[177] A. Telea and D. Auber. Code Flows: visualizing structural evolution of source code. *Comp. Graph. Forum*, 27(3):831–838, 2008.

[178] A. Telea and O. Ersoy. Image-based edge bundles: Simplified visualization of large graphs. *Comp. Graph. Forum*, 29 (3):543–551, 2010.

[179] A. Telea and J. J. van Wijk. An augmented fast marching method for computing skeletons and centerlines. In *Proc. IEEE VisSym*, pages 251–258, 2002.

[180] A. Telea and L. Voinea. A tool for optimizing the build performance of large software code bases. In *Proc. IEEE CSMR*, pages 153–156, 2008.

[181] A. Telea and L. Voinea. An interactive reverse-engineering environment for large-scale C++ code. In *Proc. ACM SOFTVIS*, pages 67–76, 2008.

[182] A. Telea and L. Voinea. Visual software analytics for the build optimization of large-scale software systems. *Computational Statistics*, 26(4):635–654, 2011.

[183] A. Telea, A. Maccari, and C. Riva. An open toolkit for prototyping reverse engineering visualizations. In *Proc. Data Visualization (IEEE VisSym)*, pages 67–75. IEEE, 2002.

[184] A. Telea, A. Voinea, and H. Sassenburg. Visual tools for software architecture understanding: A stakeholder perspective. *IEEE Software*, 27(6):46–53, 2010.

[185] A. Telea, L. Voinea, and O. Ersoy. Visual analytics in software maintenance: Challenges and opportunities. In *Proc. EuroVAST*, pages 65–70. Eurographics, 2010.

[186] M. Termeer, C. Lange, A. Telea, and M. Chaudron. Visual exploration of combined architectural and metric information. In *Proc. IEEE Vissoft*, pages 21–26, 2005.

[187] A. Teyseyre and M. Campo. An overview of 3D software visualization. *IEEE TVCG*, 15(1):87–105, 2009.

[188] James J. Thomas and Kristin A. Cook. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. National Visualization and Analytics Center, 2005.

[189] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX and XMI. In *Proc. WCRE*, pages 296–300, 2000.

[190] S. Tilley, K. Wong, M.A. Storey, and H. Müller. Programmable reverse engineering. *Intl. J. Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[191] Frank Tip. A survey of program slicing techniques. *J. of Programming Languages*, 3(3):121–189, 1995.

[192] I. Tollis, G. Di Battista, P. Eades, and R. Tamassia. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999.

[193] C. Tominski, J. Abello, F. van Ham, and H. Schumann. Fisheye tree views and lenses for graph visualization. In *Proc. Information Visualisation*, pages 202–210, 2006.

[194] B. Tversky, J. Morrison, and M. Betrancourt. Animation: Can it facilitate? *Intl. J. Human Computer Studies*, 57:247–262, 2002.

[195] F. vam Ham. Using multilevel call matrices in large software projects. In *Proc. InfoVis*, pages 227–232, 2003.

[196] M. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[197] M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[198] M. van den Brand, S. Roubtsov, and A. Serebrenik. SQuAVisiT: A flexible tool for visual software analytics. In *Proc. CSMR*, pages 331–332, 2009.

[199] R. van Liere and W. de Leeuw. Graphsplatting: Visualizing graphs as continuous fields. *IEEE TVCG*, pages 206–212, 2003.

[200] J. J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE InfoVis*, pages 73–80, 1999.

[201] J. J. van Wijk and C. W. A. M. van Overveld. Preset based interaction with high dimensional parameter spaces. In F. Post, G. Nielsen, and G. Bonneau, editors, *Data visualization - State of the art*, pages 391–406. Kluwer, 2003.

[202] J. J. van Wijk, T. Isenberg, J. Roerdink, A. Telea, and M. Westenberg. Visual analytics evaluation. In *Mastering the Information Age: Solving Problems with Visual Analytics*, chapter 8, pages 131–143. Eurographics, 2010.

[203] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proc. ACM SE*, pages 68–80, 2010.

[204] L. Voinea and A. Telea. Visual querying and analysis of large software repositories. *Empirical Software Engineering*, 14(3):316–340, 2009.

[205] L. Voinea and A. Telea. Case study: Visual analytics in software product assessments. In *Proc. IEEE Vissoft*, pages 65–72, 2009.

[206] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Proc. ACM SOFTVIS*, pages 47–56, 2005.

[207] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J.J. van Wijk, J.-D. Fekete, and D.W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comp. Graph. Forum*, 30(6):1719–1749, 2011.

[208] VSG Inc. OpenInventor toolkit, 2011. `vsg3d.com/open-inventor/sdk`.

[209] VTK Team. The visualization toolkit (VTK) home page, 2010. `www.vtk.org`.

[210] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proc. SCAM*, pages 128–135, 2004.

[211] J. Weickert. *Anisotropic diffusion in image processing, Teuber Verlag, Stuttgart, 1998.* Teuber Verlag, 1998.

[212] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proc. ICPC*, pages 231–240. IEEE, 2007.

[213] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proc. WCRE*, pages 219–228. IEEE, 2008.

[214] Wicket. Apache Wicket, 2012. `http://wicket.apache.org`.

[215] N. Willems, H. van de Wetering, and J. J. van Wijk. Visualization of vessel movements. *Comp. Graph. Forum*, 28(3): 959–966, 2009.

[216] N. Wong and S. Carpendale. Using edge plucking for interactive graph exploration. In *Proc. IEEE InfoVis (poster comp.)*, pages 51–52, 2005.

[217] N. Wong and S. Carpendale. Supporting interactive graph exploration with edge plucking. In *Proc. IEEE Visualization (interactive posters)*, 2005.

[218] N. Wong and S. Carpendale. Supporting interactive graph exploration using edge plucking. In *Proc. SPIE*, pages 235–246, 2007.

[219] N. Wong, S. Carpendale, and S. Greenberg. EdgeLens: An interactive method for managing edge congestion in graphs. In *Proc. IEEE InfoVis*, pages 167–175, 2003.

[220] P. C. Wong and James J. Thomas. Visual analytics. *IEEE CG&A*, 24(5):20–21, 2004.

[221] Y. Yang, J. Chen, and M. Beheshti. Nonlinear perspective projections and magic lenses: 3D view deformation. *IEEE CG & A*, 25(1):567–582, 2005.

[222] J. Yi, R. Melton, J. Stasko, and J. Jacko. Dust & magnet: Multivariate information visualization using a magnet metaphor. *J. of Information Visualization*, 4(4):542–551, 2006.

[223] K. Zhang. *Software visualization - From theory to practice*. Kluwer Academic, 2003.

[224] H. Zhou, X. Yuan, W. Cui, H. Qu, and B. Chen. Energy-based hierarchical edge clustering of graphs. In *Proc. PacificVis*, pages 55–62, 2008.

[225] H. Zhou, X. Yuan, H. Qu, W. Cui, and B. Chen. Visual clustering in parallel coordinates. *Comp. Graph. Forum*, 27 (3):1047–1054, 2008.

LIST OF FIGURES

LIST OF ACRONYMS

AFMM Augmented fast marching method 100

ASG Annotated syntax graph 25

AST Abstract syntax tree 25

CR Change request 54

DAG Directed acyclic graph 8

DT Distance transform 121

DT-MRI Diffusion tensor magnetic resonance imaging 172

EBL Edge-bundling layout 2

FDEB Force-directed edge bundling 16

GBEB Geometric-based edge bundling 16

GD Graph drawing 2

GLR Generalized Left-Reduce 10

HBA Hierarchical bottom-up agglomerative 72

HEB Hierarchical edge bundle 3

IBEB Image-based edge bundle 53

InfoVis Information visualization 1

IPC Illustrative parallel coordinates 85

KDE Kernel density estimation 114

KDEEB Kernel density estimation edge-bundling 114

LOC Lines-of-code 43

MDS Multidimensional scaling 150

NLD Node-link diagram 24

SBEB Skeleton-based edge bundling 105

SCM Source control management 8

SME Software Maintenance and Evolution 51

SoftVis Software visualization 1

SQAT Software Quality Assurance and Testing 51

SVA software visual analysis 1

SVN Subversion 51

VA Visual analytics 1

VSTS Visual Studio Team System 11

WPF Windows Presentation Foundation 62

WR Winding roads 16

PUBLICATIONS

Christhophe Hurter, **Ozan Ersoy**, and Alexandru Telea. Smooth Bundling of Large Streaming and Sequence Graphs. *Proc. PacificVis*, 2013 *(Honorable Mention Paper Award)*.

Ozan Ersoy, Christophe Hurter, and Alexandru Telea. Mean shift for graph bundling. *Proc. ASCI/IPA/SIKS 2012*.

Dennie Reniers, Lucian Voinea, **Ozan Ersoy**, and Alexandru Telea. The Solid* Toolset for Software Visual Analytics of Program Structure and Metrics Comprehension: From Research Prototype to Product. *Science of Computer Programming*, Elsevier (2012).

Christhophe Hurter, **Ozan Ersoy**, and Alexandru Telea. Graph Bundling by Kernel Density Estimation. *Computer Graphics Forum 31*, 865-874 (2012).

Christophe Hurter, **Ozan Ersoy**, and Alexandru Telea. MoleView: An Attribute and Structure-Based Semantic Lens for Large Element-Based Plots. *IEEE Transactions on Visualization and Computer Graphics 17(12)*, 2600-2609 (2011).

Christhophe Hurter, **Ozan Ersoy**, and Alexandru Telea. Generalizing Semantic Lenses for Large Element-based Plots *ASCI/IPA/SIKS 2011*.

Ozan Ersoy, Christophe Hurter, and Alexandru Telea. Graph Edge Bundling by Medial Axes. *Proc. ASCI/IPA/SIKS 2011*.

Ozan Ersoy, Christophe Hurter, Fernando V. Paulovich, Gabriel Cantareira, and Alexandru Telea. Skeleton-Based Edge Bundling for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics 17(12)*, 2364-2373 (2011).

Dennie Reniers, Lucian Voinea, **Ozan Ersoy**, and Alexandru Telea. A Visual Analytics Toolset for Program Structure, Metrics, and Evolution Comprehension. *Proc. WASDeTT 10, ed. H. Kienle*, 2010, IEEE.

Alexandru Telea and **Ozan Ersoy**. Bundle-Centric Visualization of Compound Digraphs. *ASCI 2010*.

Alexandru Telea, **Ozan Ersoy**, and Lucian Voinea. Visual Analytics in Software Maintenance: Challenges and Opportunities. *Proc. EuroVAST'10*, 2010.

Alexandru Telea and **Ozan Ersoy**. Image-Based Edge Bundles: Simplified Visualization of Large Graphs. *Computer Graphics Forum 29*, 843-852 (2010) *(2nd Best Paper Award, EuroVis'10)*.

Alexandru Telea, **Ozan Ersoy**, and Hessel Hoogendorp. Comparison of Node-Link and Hierarchical Edge Bundling Layouts: A User Study. *Proc. Dagstuhl Seminar 09211 Visualization and Monitoring of Network Traffic*, 2009.

Alexandru Telea, Hessel Hoogendorp, **Ozan Ersoy**, and Dennie Reniers. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*, 81-88 (2009).

SAMENVATTING

Grote grafen zijn een essentieel element in het begrijpen van dataverzamelingen zoals onstaan in het veld van programmacomprehensie in het gebied van software-engineering. In de laatste jaren, een reeks van technieken zijn onstaat die dit probleem benaderen: *edge bundling layouts* (EBLs). Deze visualisatiemethodes tekenen grote grafen door de nadruk te leggen op het tonen van de versimpelde structuur ervan. Als gevolg, de finjschalige details van de fraaf, zoals individuele verbindingen (*edges*) worden onzichtbaar, en ruimte wordt gemaakt voor het tone van de grofschalige graafstructuur.

In dit proefschrift onderzoeken wij nieuwe visualisatiemethodes voor het afbeelden van grote hiërarchische grafen door middel van EBLs. Wij beginnen ons onderzoek door vast te stellen dat EBLs significante voordelen hebben ten opzichte van klassieke graafafbeeldingstechnieken (*straight-line node-link layouts*) in het kader van programmacomprehensie. Wij verfijnen dit onderzoek door verder te analyseren hoe EBLs belangrijke onderdelen kunnen zijn van complete oplossingen voor programmacomprehensie die wordt toegepast in de IT industrie.

Gebaseerd op deze observaties presenteren wij vervolgens een aantal algoritmes die EBLs efficiënt en effectief kunnen berekenen voor grote grafen. De gemeenschappelijke noemer van deze algoritmes is het gebruik van *image-based* technieken – het omzetten van het *graph bundling* probleem in een reeks beeldverwerkingsoperaties zoals afstandstransformaties, skeletonisatie, en *mean shift* beeldsegmentatie. In dit kader laten wij eerst zien hoe grote EBLs versimpeld kunnen worden zodat de hoofdstructuur van de graaf zichtbaar wordt, zelfs in het geval van dominante occlusies. Vervolgens laten wij zien hoe het berekenen van EBLs voor hiërarchische en ook algemene grafen gedaan kan worden met gebruik van *image-based* technieken. Dit stelt ons in staal om formeel te rederenen over de robuustheid en complexiteit van de voorgestelde algoritmes, en ook deze algoritmes efficiënt te implementeren met behulp van parallelisatie op grafische kaarten (GPUs).

Vervolgens breiden wij ons voorstel voor *image-based* in twee richtingen. De eerste is het berekenen van EBLs voor dynamische (tijdsafhankelike) grafen, waarin de schaal van de datasets en het zichbaar maken van dynamische veranderingspatronen

de twee belangrijkste uitdagingen vormen. De tweede is het tonen van aanvullende details over de gebundelde edges op een lokale schaal, door middel van een reeks van interactieve technieken. Wij laten ook zien hoe deze interactieve technieken toepasbaar zijn voor de exploratie van andere types datasets zoals twee- en driedimensionale beelden.

Wij illustreren onze *image-based* EBL aanpak met verschillende voorbeelden voor programmacomprehensie van grote software-systemen en ook voor datasets afkomstig uit de analyse van luchtverkeerinformatie.

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor, Alexandru Telea, for granting me the opportunity to have this PhD position, and for his great support, guidance and supervision along the whole PhD period. I have enjoyed our meetings and long discussions interchanging ideas, and conversations about various subjects, technical and non-technical. I consider myself very lucky for being your student, and I learned a lot from you.

Second, I want to thank all my friends in Groningen, who made me feel at home. Piray, Can, Ozlem, Leonardo, Volkan, Olha, Onur, Araz, and Hande Özgen. We shared many good and bad, happy and sad moments during this PhD period. Thank you for being my friends, and being part of my life. We will anyway be always in touch, so I don't want this sound like a goodbye message my friends.

Devrim, Orçun, Serra, Turan, Hande Kırbaş, Ercan, Yeliz, and Utku, my dear friends, as well as my Zernike lunch companions, I will miss every moment we spent together in Groningen, lunch breaks, house parties, king (a card game) parties, drinking and talking. Thanks for being by my side.

Now this is going to be the second time I mention your names here, but you well deserve it. Orçun and Devrim, yes I mean you. You have been my very best friends for the past four years, and undoubtedly you will have an important part in my life in the future as well.

Murat abi, Serayi abla, Bayram abi and Eric thank you for being there, for your friendship and for the nice food that helped me to survive the delicious Dutch cuisine. I am very glad that I got to know you.

Maarten and Matthew, my officemates and my friends, I want to thank you for not only making the office an enjoyable place with your good taste of humor, but also for sharing your life experiences and your technical knowledge with me. Maarten, I hope you still remember those few Turkish words you learned.

Next, many thanks to everyone in our research group, namely Jos Roerdink, Tobias Isenberg, Moritz, Yun, Deborah, Bilkis, Jasper, Alessandro, and Andre for their friendship and helpful feed-

backs. I also want to thank to Esmee, Ineke and Desiree for their help and support.

Furthermore, I would like to thank my family for supporting me through all my education life and for generously giving their love to me for all of my life.

Last but not least, my dearest Naima, thank you for every single beautiful thing you brought into my life, your endless support and companion. I am the luckiest guy in the universe, because I have you. I can't wait to spend the rest of my life with you.

Ozan Ersoy *Groningen, <the date>*

COLOPHON

This thesis was typeset with LaTeX 2ε using Robert Slimbach's *Minion Pro* font. The typesetting is based on the *classicthesis* style by André Miede.