PROVENANCE OF EXPLORATORY TASKS IN SCIENTIFIC VISUALIZATION: MANAGEMENT AND APPLICATIONS

by

Carlos Eduardo Scheidegger

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Carlos Eduardo Scheidegger**

has been approved by the following supervisory committee members:

Cláudio T. Silva	, Chair	**2009/10/23** Date Approved
Juliana Freire	, Member	**2009/10/23** Date Approved
Christopher Johnson	, Member	**2009/10/23** Date Approved
Robert M. Kirby II	, Member	**2009/10/23** Date Approved
Pat Hanrahan	, Member	**2009/10/23** Date Approved

and by **Martin Berzins** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Visualization has emerged as an effective means to quickly obtain insight from raw data. While simple computer programs can generate simple visualizations, and while there has been constant progress in sophisticated algorithms and techniques for generating insightful pictorial descriptions of complex data, the process of building visualizations remains a major bottleneck in data exploration. In this thesis, we present the main design and implementation aspects of VisTrails, a system designed around the idea of transparently capturing the *exploration process* that leads to a particular visualization. In particular, VisTrails explores the idea of *provenance* management in visualization systems: keeping extensive metadata about how the visualizations were created and how they relate to one another.

This thesis presents the provenance data model in VisTrails, which can be easily adopted by existing visualization systems and libraries. This lightweight model entirely captures the exploration process of the user, and it can be seen as an electronic analogue of the scientific notebook. The provenance metadata collected during the creation of pipelines can be reused to suggest similar content in related visualizations and guide semi-automated changes. This thesis presents the idea of building visualizations *by analogy* in a system that allows users to change many visualizations at once, without requiring them to interact with the visualization specifications.

It then proposes techniques to help users construct pipelines *by consensus*, automatically suggesting completions based on a database of previously created pipelines. By presenting these predictions in a carefully designed interface, users can create visualizations and other data products more efficiently because they can augment their normal work patterns with the suggested completions.

VisTrails leverages the workflow specifications to identify and avoid redundant operations. This optimization is especially useful while exploring multiple visualizations. When variations of the same pipeline need to be executed, substantial speedups can be obtained by caching the results of overlapping subsequences of the pipelines. We present the design decisions behind the execution engine, and how it easily supports the execution of arbitrary third-party modules. These specifications also facilitate the reproduction of previous results. We will present a description of an infrastructure that makes the workflows a complete description of the computational processes, including information necessary to identify and install necessary system libraries. In an environment

where effective visualization and data analysis tasks combine many different software packages, this infrastructure can mean the difference between being able to replicate published results and getting lost in a sea of software dependencies and missing libraries.

The thesis concludes with a discussion of the system architecture, design decisions and learned lessons in VisTrails. This discussion is meant to clarify the issues present in creating a system based around a provenance tracking engine, and should help implementors decide how to best incorporate these notions into their own systems.

To Adriana, who makes it all worthwhile

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I want to thank many people that have made my years at the University of Utah as enjoyable as they have been.

First of all, I want to thank Prof. Cláudio Silva for giving me the opportunity to be here in the first place, and in particular for believing in me during the many times I did not. I started graduate school with very little academic experience, and working in his group was a great way to learn to be a researcher. I can only hope to have acquired some of his ability to pick good problems to work on. I also want to thank Prof. João Comba, my undergraduate advisor, without whose advice and recommendations I would also not have been here in Salt Lake City. I hope to have met their expectations.

Cláudio was kind enough to encourage me to think about research in many different areas. I had a lot of fun because of it, but, as a result, I probably annoyed more than my fair share of other faculty in the School of Computing and SCI Institute. With that said, I want to thank all of them for the very enlightening discussions I have had, from which I have learned so much. These include Prof. Sarang Joshi, Prof. Luiz Gustavo Nonato, Prof. Valerio Pascucci, Prof. Mike Kirby, Prof. Suresh Venkatasubramanian, Prof. Hal Daumé and Prof. Pete Shirley. Recently, Prof. Adam Bargteil's advice on the mechanics of academic job interviews and talks was invaluable. These discussions constantly reminded me how fortunate I am to spend time in a place that rewards learning and teaching, and that this is a precious and rare environment.

When I was quite stuck in writing, Prof. Juliana Freire told me, "draw a picture!" That worked wonderfully, and I like to think that being given visual advice from a data management expert means that visualization is, after all, quite useful. I am very grateful for that advice. I would also like to thank the rest of my committee for making me view the work in a broader perspective, and keeping me engaged and interested, be it through phone calls, hallway conversations or written exam questions.

I want to also thank many office mates, friends and colleagues. In particular, the early years of sharing an office with Steve Callahan and John Schreiner were among the most fun I can remember, even when we were pulling all-nighters before submission deadlines. Erik Anderson, David Koop, Huy Vo, Emanuele Santos and Lauro Lins were all part of the research group while I was here, and

we exchanged ideas and thoughts so often that I consider them all more than friends and colleagues. Mike Steffen lent the house a piano for a long time, and although the neighbors claim otherwise, I am very thankful for that. David Nellans let us take care of his house, giving us a year and a half of living like kings. Roni Choudhury and I have had more discussions about things neither of us understand than we should have, but I am glad he has as much fun as I do trying to figure them out. Gordon Kindlmann has always reminded us that scientific visualization is supposed to scientific — since there is hardly science if there is no reproducibility, publishing scientific code as open source is a matter of scientific ethics. I could not agree more with that sentiment, and can only hope to instill it in other people as well as he has in me.

I want to thank my mom, Norma, and dad, Jorge, for the understanding and support they have shown throughout these years, even though it meant that I would be sixty-two hundred miles away. One of the earliest memories I have of my childhood is of dad hacking our refrigerator so he could track its energy usage with an old 8-bit personal computer. Another memory is of my uncle Paulo talking about how his toast would come out perfectly every time because he would take a stopwatch to the kitchen to get the timing just right, after having thoroughly experimented and documented the alternatives. That science and engineering not only works well in the real world, but that it is also a lot of fun, is something I love to carry with me. Uncle Paulo and aunt Liane have also been extremely supportive of me and my brother through some tough times, and I want to thank them for it.

It is well known that I loathe paperwork almost as much as I am bad with it. Thankfully, I could always count on our extremely competent and helpful admin staff in the SCI Institute and the School of Computing for help with that. In particular, I want to thank Raelynn Potts, Deb Zemek, Karen Feinauer and Prof. Suresh Venkatasubramanian for putting up with my terrible mix of unwillingness and inability to navigate bureaucracy over these years.

I also want to acknowledge the funding sources for my research during these years, including the NSF and an IBM Student Fellowship.

Some visualizations created using VisTrails and presented in these thesis figures used a number of existing open-source libraries and data repositories, including Teem [61], VTK [110], ITK [52], trimesh2 [107], and matplotlib [4]. The dataset used for the chapter on pipeline completions was created from vistrails generously donated by the students of the Scientific Visualization courses taught at the University of Utah in the fall of 2007 and fall of 2008.

CHAPTER 1

INTRODUCTION

In recent years, with the explosion in the volume of scientific and measurement data, we have observed a paradigm shift in how scientists and engineers use visualization. Projects such as CORIE, an environmental observation and forecasting system for the Columbia River, generate and publish on the Web thousands of new images daily which depict river circulation simulation forecasts and hindcasts, as well as real-time sensor data. Figure 1.1 shows examples of the kinds of visualizations used in the project. Visualization techniques have become key to understanding complex phenomena, and the field has grown into a mature area with an established research agenda [92].

This demand for visualization has driven the development of software systems that provide flexible frameworks for creating complex visualizations. Visualization systems can be broadly classified as turnkey applications (e.g., ParaView, VisIt, Amira) [22, 62, 85] and dataflow-based systems (e.g., VTK, SCIRun, AVS, OpenDX) [53, 99, 110, 122]. In this thesis, I will focus on dataflow systems. These are more general and often serve as the foundation of turnkey applications: both ParaView and VisIt are based on VTK [22, 62], and turnkey application development systems have been proposed using both SCIRun [80] and VisTrails [108].

More specifically, I will focus on visual programming systems that are (loosely) based on a dataflow architecture. These systems represent computations by directed graphs, and allow users to edit the representations visually, by adding and removing vertices and edges. Such graphs are typically called *pipelines*. The graph vertices represent different atomic computational tasks, and are usually called *modules*. The graph edges specify the flow of inputs and outputs in the computation, and are called *connections*. More recently, visual programming systems have been proposed for more general computational tasks [76], or based around web services [96]. One of the stated goals of visual programming systems is to facilitate the creation of programs by nonprogrammers [53]. However successful these systems may be, creating effective scientific visualizations is very much about exploration of the design space. While most dataflow-based systems have sophisticated user interfaces that ease the creation of visualizations, the path from "data to insight" requires a laborious,

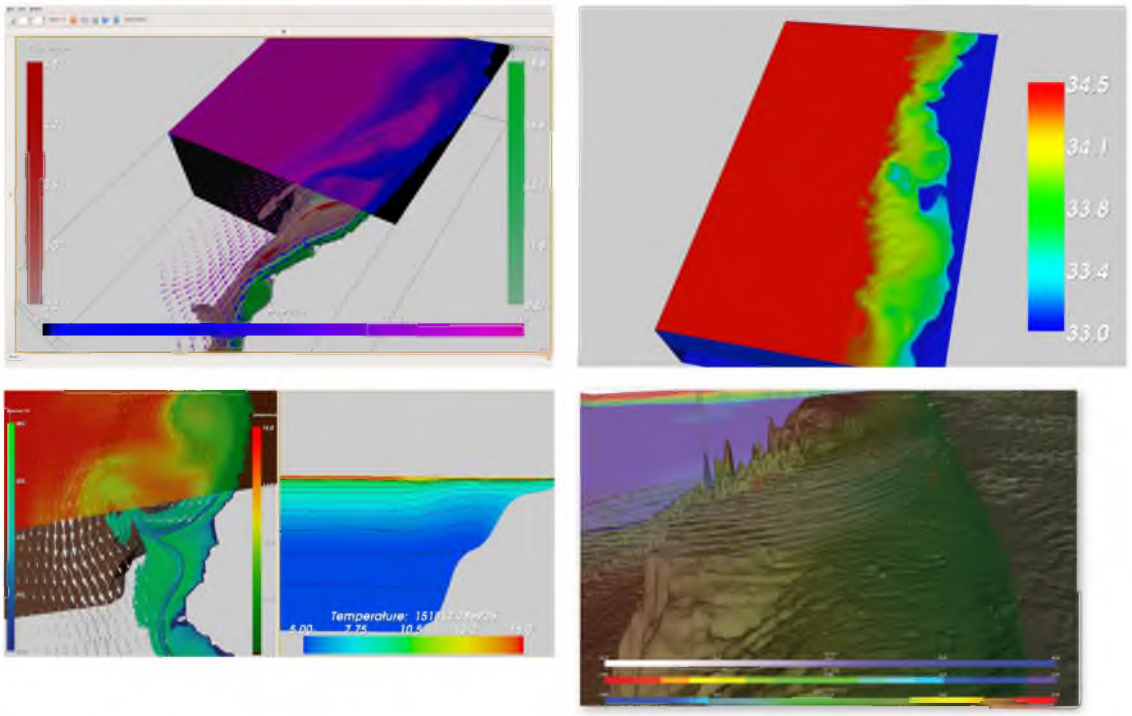


Figure 1.1: The CORIE project produces a large number of visualizations on a daily basis, posing a large data management problem. http://www.stccmop.org/CORIE

trial-and-error process, where users successively assemble, modify, and execute pipelines [123].

In general, visual programming systems lack the infrastructure to help explore the space of possible visualizations. In the course of exploratory studies, users often build large collections of visualizations, each of which helps in the understanding of a different aspect of their data. A scientist working on a new computational fluid dynamics application might need a collection of visualizations such as 3D isosurface plots, 2D plots with relevant quantitative information, and some direct volume rendering images. Although in general, each of these visualizations is implemented in a separate dataflow, they have a certain amount of overlap (e.g., they may manipulate the same input data sets). Furthermore, for a particular class of visualizations, the scientists might generate several different versions of each individual dataflow while fine tuning visualization parameters or experimenting with different data sets. Often, insight comes from comparing the results of a variety of visualizations [104]. For example, Figure 1.1 shows a set of CORIE images used to study the salinity variation at the mouth of the Columbia river. A large portion of the motivation behind the work in this thesis is the realization that visualization fundamentally requires a variety of approaches and visualizations examined in combination.

Even in the simplest exploration scenarios, visualization practitioners will typically try several different isosurface values, transfer functions and colormaps before deciding which is more effec-

tive. Manually managing these different parameter sets and dataflow structures in typical systems requires an ad-hoc naming scheme for files saved on disk, which is not scalable beyond a handful of visualization pipelines. Because these systems are designed to interact with a single visualization and pipeline at a time, any particular pipeline might be lost in a sea of files. While Apple's Spotlight technology provides an API that allows programmers to expose file metadata to a centralized file system searching tool [55], integrating data management in the system itself has several advantages, including the ability to use the metadata for other applications besides search, as we will present in the following chapters.

1.1 Contributions and Impact

This document comprises several contributions to the field of scientific visualization. Specifically, my thesis is that a comprehensive, transparent mechanism for capturing the steps involved in the creation of a scientific visualization (in our case, in the form of a visualization pipeline) can make the exploratory process of creating visualizations simpler and more effective, and, just as importantly, also help make the generated visualizations more reproducible. This mechanism also serves as infrastructure in which novel applications can be built. The specification of the provenance model itself is described in Chapter 3, while the applications, such as building visualizations by analogy and data-driven suggestion of visualization pipeline fragments are described in Chapters 4 and 5. Chapter 8 contains a preliminary evaluation of the design principles of VisTrails and its implementation in the system, and a discussion of the issues involved with a proper evaluation. I believe these contributions substantially improve the state-of-the-art in scientific visualization systems, and help bridge the gap between visualization designers and consumers.

The execution model of VisTrails workflows is mostly characterized by the transparent caching of previously computed results. It is well-suited for exploratory visualization, where users are trying to understand the relation between different but related visualizations and the role of specific parameters in the final results. This execution model has been shown useful for other similar systems, and the caching strategy has now been implemented in Kepler [9], a popular scientific workflow system.

While this thesis presents the major design decisions and some major features of VisTrails, important parts and extensions of the system are described elsewhere. We note that many of these extensions were made easier by the uniform change-based provenance model of VisTrails (to be described in Chapter 3). Ellkvist et al. use the provenance model to synchronize the interaction of multiple users, enabling groupware-like collaboration in VisTrails [32] as described by Ellis and Gibbs [31] in a straightforward fashion.

Santos et al. designed an automatic user interface generation tool for workflows in VisTrails,

where the provenance model is used to simultaneously track the changes in the workflow specification, and parameter sets executed by a user in the generated interface [108]. By changing the domain object being tracked from VisTrails workflows to other objects, it is possible to provide process provenance for third-party applications such as ParaView [17].

The provenance model used in VisTrails is, to the best of our knowledge, the first model used in visual programming systems for scientific visualization that uniformly captures both parameter and structural changes for visualizations. This model is successful in capturing the exploration process of more complicated objects than previously done, and so it allows process provenance to be used in practical real-life scenarios.

The ideas of creating visualizations by analogy and of presenting data-driven completions in scientific visualization systems use the data generated by the provenance model to facilitate the creation of visualizations. In particular, they should be seen as attempts to interpret the provenance model as "externalized knowledge": the collected lessons of potentially many users as they generate visualizations. This notion of "wisdom of the crowds" is particularly important in a world where multiple-user collaborations are the norm rather than the exception, and these chapters present contributions to scientific visualization in that context. I believe this notion of specifying visualizations using exploration data collected from users — in a sense, *metaprogramming visualizations* — will become more practical as we provide better tools to easily store and process these collected data. This thesis is a step in that direction.

As previously mentioned, the provenance model and techniques proposed in this thesis have been implemented in VisTrails, which is publicly available and open-source. VisTrails has been downloaded more than 10000 times over the course of two years. In addition, VisTrails has been used in classrooms to teach Scientific Visualization at the University of Utah and UNC-Chapel Hill. The provenance model presented in Chapter 3 has also been implemented in other scientific visualization systems, including ParaView and VisIt, and is currently part of a commercial plugin for history management in Maya [56].

1.2 Document Structure

This thesis is roughly structured in three parts. The first part, composed of Chapters 1 and 2, cover preliminaries and generally related work. The next part provides a detailed discussion of core concepts in VisTrails, and comprises Chapters 6, 3 and 9. The final part of the thesis presents techniques that use the provenance model to facilitate the creation of visualizations, in Chapters 4 and 5.

Chapter 3 describes the provenance model in VisTrails. The data captured by this model represent the user's exploration process during the creation of a visualization, and allow users to

easily navigate and compare the different visualizations created. This model also prevents the user from accidentally erasing a potentially useful visualization. Scientists have long recognized in the laboratory notebook the need for comprehensive annotation of every experiment performed, and every question asked, for later analysis. The VisTrails provenance model provides an electronic analogue of the laboratory notebook. It is not intended to replace best practices in note-taking: it complements and facilitates them, leaving the visualization practitioner to do what they want to do most: create novel and insightful visualizations that helps them better understand the matter being studied. One defining theme of this thesis is that the data in the provenance model can actually be used to enable novel applications. In particular, Chapter 4 uses these data to let the user create novel visualizations *by analogy*, without referring to the visualization pipeline itself. This is possible by combining the history information in the provenance model with a novel inexact graph matching technique that transfers the user's exploration from one context to another.

Still, without knowledge of the underlying computational components, it is hard to understand what series of modules and connections should be added to obtain a desired result. In essence, there is no "roadmap"; visual programming systems provide little feedback to help the user deduce which modules can or should be added to the pipeline. Even with a mechanism such as the one described in Chapter 4, a novice user, or even an advanced user performing a new task, must often resort to manually searching for existing pipelines to use as examples. These examples are then adapted and iteratively refined until a solution is found. This manual, time-consuming process is the current standard for creating visualizations rather than the exception. Chapter 5 describes VisComplete, a recommendation system for visualization pipelines based on data collected during the exploration process. The system learns common paths used in existing pipelines and predicts a set of likely module sequences that can be presented to the user as suggestions during the design process.

The goal of exploratory visualization also influenced the choice of execution strategy for VisTrails pipelines. Because scientific visualization techniques are often computationally- and memory-intensive, visualization pipelines may comprise long sequences of expensive steps with large temporary results, which might limit a user's ability to interactively explore their data. Chapter 6 describes the execution model of VisTrails pipelines, with particular emphasis on a caching mechanism that allows different pipelines to reuse each other's partial results. This mechanism makes exploratory visualization more practical while imposing relatively few barriers to the way execution modules are designed.

1.3 Terminology

In this thesis, I will use the noun "visualization" to mean a visual depiction meant to increase the understanding of some set of data.

VisTrails was originally designed for scientific visualization scenarios. However, its execution model is general enough to support more varied execution tasks, more akin to scientific workflow systems such as Taverna and Kepler. Because of that, I will sometimes use the terms *visualization pipeline* and *scientific workflow* (or simply *workflow*) interchangeably. In the original VisTrails paper [10], we used the term "vistrail" to mean one such pipeline. We have since decided to use the term to mean the version tree corresponding to an exploration.

Although the term applies in a more general sense, in this thesis, we will take *scientific visualization systems* to mean systems that allow users to create visualizations via a visual programming interface, most often using a dataflow execution model.

CHAPTER 2

BACKGROUND AND RELATED WORK

Visualization pipelines are a particularly simple form of computer programs and, in a sense, facilitating the creation of programs is as old a problem as computer science itself. We cannot hope to cover the entire range of proposed solutions for this general formulation of the problem. We instead focus on works most closely related to scientific visualization, workflow systems, and, in particular, to solutions related to the process provenance model we propose, and the different ways of storing, representing and manipulating a user's history. Work related to specific techniques used in subsequent chapters will be presented in the chapters themselves.

2.1 Visual Programming Systems for Visualization

Visualization systems have been quite successful at bringing visualization to a greater audience. Seminal systems such as AVS Explorer and Data Explorer [53, 122] enabled domain scientists to create visualizations with much less training and effort than previously possible. The early success of these systems led to the development of several alternative approaches. SCIRun [99] focuses on computational steering: the intentional placement of human intervention and visualization in the process of generating simulations. The Visualization Toolkit [110] is a library that directly exposes a powerful dataflow API for several programming languages. Development in workflow systems for visualization is ongoing, as evidenced by projects such as MeVisLab [86] for medical visualization, and VisTrails itself [121], which helps users to easily combine existing visualization libraries with user-defined modules in a provenance capturing framework.

As scientific visualization became more widely used, several scalability issues have arisen, ranging from ensuring good performance, handling large amounts of data, capturing provenance, and providing interfaces to interact with a large number of visualizations. Distributed, parallel systems [22] have been developed to address performance and dataset size concerns. Such systems provide a scalable architecture for creating and running visualization pipelines with large data.

Visualization has also reached business and financial users. In these scenarios, data is typically stored in a relational database. These data stores have much richer structure associated with them

and allow one to design an associated *algebra of visualizations*. This approach was pioneered by Mackinlay [79] and served as inspiration for Stolte, Tang and Hanrahan in the Polaris system and formalism [114]. This was then extended to facilitate the visualization of data cubes in Rivet [113]. The Polaris effort culminated in Tableau, a commercial system for database visualization, which is the basis of an excellent study by Heer et al. of the tradeoffs in designing and presenting graphical histories of the user's exploration process [49].

Some systems focus on distributed execution of visualization workflows in the Grid. sgViz, in particular, is a system that extends IRIS Explorer for execution on Grid resources [15]. Its layered architecture allows different bindings for a given pipeline specification in each layer. This allows different visualization systems and strategies to be integrated. Heterogenous execution is supported in VisTrails via a fairly complete web services package. One drawback of the execution model in VisTrails is that its caching mechanism requires centralized control, and the execution of a visualization pipeline is currently orchestrated by a single process. Still, the execution model itself is well-suited for parallel execution. The requirement that modules behave functionally, as described in Chapter 6, implies that the order in which the modules are executed does not change the result, which simplifies parallelization. We have, however, chosen to focus on other implementation issues, and automatic parallelism remains as future work.

2.2 Exploration History Management

An important requirement that has come to the attention of developers and users of visualization systems is the ability to record provenance so that computational experiments can be reproduced and validated. Provenance-aware scientific workflow systems have been developed that record provenance both for data products (i.e., how and from which files a given result was generated [34]) and for the exploratory process, the sequence of steps followed to design and refine the workflows used to process the data [16, 59, 100, 109].

User interface designers have long recognized the inadequacy of imposing a linear structure on the actions performed by a user in a system. Vitter described US&R, one of the earliest published studies of different Undo/Redo models, in the context of text editors [125]. Vitter describes a set of formal rules to implement Undo, Skip and Redo (US&R) operations. Flagging operations as Skip allows users to "undo" nonterminal actions. Vitter's model, however, is inadequate for large branching histories: any navigation in branches of the history requires a manual context resolution at every step to determine if the branching actions should be merged. Subsequently, Berlage described a mechanism similar to Vitter's Skip for selective undo [12]. Berlage's model for selectively undoing some actions amounts to taking all downstream actions that are *not* to be undone and applying

them afresh in a new branch of the history. Many other issues with correctly modeling the undo functionality have been researched by the HCI community, including compositionality [93], effect locality [30] and multi-user operation [7]. While the most popular strategy for implementing branching histories is based on the Command design pattern [41], other approaches include specialized logics based on branching time models [27] and explicit programming language support [70].

Specialized mechanisms have also been proposed for visualization-centric systems. Kreuseler et al. [65] proposed a history mechanism for exploratory data mining, but their ideas are also applicable to exploratory visualization. They use a tree structure to represent the change history, and describe how undo and redo operations can be calculated in this tree structure. This mechanism provides a detailed provenance of the exploratory process and is very similar to our process provenance proposal discussed in Chapter 3. One difference in the approach of Kreuseler et al. is that their model is opaque with respect to the underlying object: the changes are essentially black boxes. This hinders the use of the exploration record for anything besides archival and reproducibility. By specifying actions as workflow changes (and having a description of the types of possible changes), our model captures features of these changes which allow most applications described in Chapter 3 and, especially, Chapter 4.

A formal model for capturing the visualization exploration process was proposed by Jankun-Kelly et al. [59], who argued that a richer representation was necessary for a more effective analysis of the user's exploration. The authors' *P-Set* model captures an exploration session of the parameters in a single visualization. In contrast, the provenance model described in this thesis uniformly captures both parameter and pipeline changes. In exploratory visualization, it is likely that users will try different pipelines, and we argue that a model that treats the exploration more uniformly is more likely to be applicable in other scenarios. One advantage of the process provenance framework we propose in Chapter 3 is that it is somewhat agnostic to the execution strategy in use. This means it is an attractive candidate for adoption in specialized systems. For example, Dolgert et al. use the VisTrails provenance model to keep track of workflows in High-Energy Physics [28]. In the area of user interfaces, Kurlander et al. [66,67] have presented approaches to streamline the repetitive tasks users often face. They suggested directly editing the user's history to generalize it and turn it into macros that could be used in other contexts. The creation of visualizations by analogy, described in Chapter 4, can be seen as an attempt to draw upon the structure imposed by the graph structure of workflows to automatically infer these generalizations. In addition, because the storage model of VisTrails is actually directly based on the actions, it is possible to build such higher-level operations *retroactively*, not only while the undo stack is present.

The algorithm we describe for matching two pipelines is similar to a technique developed to

match database schemas [84]. It is also reminiscent of well-known variations of PageRank, which is the basis for Google's successful ranking algorithm [14, 69]. As we will show in Chapter 4, the strength of the correspondence between two modules in our algorithm can be seen as given by the PageRank of the edge in the product graph, given carefully constructed initialization probabilities. Our visualization-by-analogy mechanism shares some of the objectives of programming-by-example techniques [72].

The management of different versions is the driving problem behind source code version control systems such as CVS [33]. While CVS was considered the standard for publicly-available open source version control systems in the 90s, there has been a recent flurry of systems that focus on *distributed* operation, where every copy of a source tree under control is considered to be its own repository. Notably, most of these systems, including DARCS [105], git [3] and Mercurial [97], use *changeset-based* representations, similar to the change-based mechanism we present in Chapter 3. DARCS is particularly notable because it uses a "theory of patches" based on computing whether two patches commute: whether the effects of applying the patches will change if applied in different orders. This patch algebra is similar to the operators that some authors have proposed to manipulate the user's history, including the Rotate command of Yang [128]. We note that this notion of commutation is related to the computation of the domain and range contexts described in Section 3.3. The management of scientific data and processes has attracted a lot of attention in the recent literature (see e.g., [76, 115]). Although our effort was motivated by visualization, the VisTrails framework is extensible and can also be used for general scientific workflows.

2.3 Recommendation Systems

Recommendation systems have been used in widely different application areas. Like the system we present in Chapter 5, these are based on techniques that predict users' actions based solely on the history of their previous interactions [51]. Examples include Unix command-line prediction [64], prediction of Web requests [40, 94], and autocompletion systems such as IntelliSense [87]. Senay and Ignatius have proposed incorporating expert knowledge into a set of rules that allow automated suggestions for visualization construction [111], while Gilson et al. incorporate RDF-based ontologies into an information visualization tool [43]. However, these approaches necessarily require an expert that can encode the necessary knowledge into a rule set or an ontology.

Fu et al. [40] applied association rule mining [8] to analyze Web navigation logs and discover pages that co-occur with high frequency in navigation paths followed by different users. This information is then used to suggest potentially interesting pages to users. VisComplete also derives predictions based on user-derived data and does so in an automated fashion, without the need for

explicit user feedback. However, the data it considers is fundamentally different from Web logs: VisComplete bases its predictions on a collection of graphs and it leverages the graph structure to make these predictions. Because association rule mining computes rules over *sets* of elements, it does not capture relationships (other than co-occurrence) amongst these elements.

In graphics and visualization, recommendation systems have been proposed to simplify the creation of images and visualizations. Design Galleries [81] were introduced to allow users to explore the space of parameters by suggesting a set of automatically generated thumbnails. Igarashi and Hughes [54] proposed a system for creating 3D line drawings that uses rules to suggest possible completions of 3D objects. Suggestions have also been used for view point selection in volume rendering. Bordoloi and Shen [127] and Takahashi et al. [117] present methods that analyze the volume from various view points to suggest the view that best shows the features within the volume. Like these systems, we provide the user with prioritized suggestions that the user may choose to utilize. However, our suggestions are data-driven and based on examples of previous interactions.

An emerging trend in image processing is to enhance images based on a database of existing images. Hays and Efros [48] recently presented a system for filling in missing regions of an image by searching a database for similar images. Along similar lines, Lalonde et al. [68] recently introduced Photo Clip Art, a method for intelligently inserting clip art objects from a database to an existing image. Properties of the objects are learned from the database so that they may be sized and oriented automatically, depending on where they are inserted into the image. The use of databases for completion has also been used for 3D modeling. Tsang et al. [120] proposed a modeling technique that utilizes previously created geometry stored in a database of shapes to suggest completions of objects. Like these methods, our completions are computed by learning from a database to find similarities. But instead of images, our technique relies on workflow specifications to derive predictions.

Another important trend is that of social visualization. Web-based systems such as VisPortal [13, 57] provide the means for collaborative visualization from disjoint locations. Web sites such as Sens.us [50], Swivel [116], and ManyEyes [124] allow many users to create, share, and discuss visualizations. One key feature of these systems is that they leverage the knowledge of a large group of people to effectively understand disparate data. Similarly, VisComplete uses a collection of pipelines possibly created by many users to derive suggestions.

2.4 Multiple-view Visualizations

VisTrails provides an infrastructure that enables the effective use of several visualization techniques which aid users to explore and understand the underlying information. In particular, it supports the three primary techniques described by Roberts [103]: *Multiform visualization*, the

ability to display the same information in different ways, can be achieved by appropriately setting parameters or modifying a vistrail (e.g., the top row of Figure 2.1 shows the Visible-Human dataset rendered using isosurfaces whereas the bottom row uses volume rendering); *abstract views*, which apply the same visualization algorithm using different simplification criteria, can be constructed by refining and modifying a trail specification; and *direct manipulation*, which allows objects to be directly interrogated, scaled and positioned, can be achieved both through the Vistrail Builder and through the VisTrails Spreadsheet, and in the latter, different views can be coordinated (see Figure 2.1).

The use of spreadsheets for displaying multiple images was proposed in previous works. Levoy's Spreadsheet for Images (SI) [71] is an alternative to the flow-chart-style layout employed by many earlier systems using the dataflow model. SI devotes its screen real estate to viewing data by using a tabular layout and hiding the specification of operations in interactively programmable cell formulas. The 2D nature of the spreadsheet encourages the application of multiple operations to multiple data sets through row or column-based operations. Chi [20] applies the spreadsheet paradigm to information visualization in his Spreadsheet for Information Visualization (SIV). Linking between cells is done at multiple levels, ranging from object interactions at the geometric level to arithmetic operations at the pixel level. The difficulty with both SI and SIV is that they fail to capture the history of the exploration process, since the spreadsheet only represents the latest state in the system.

The VisTrails Spreadsheet supports concurrent exploration of multiple visualizations. The interface is similar to the one proposed by Jankun-Kelly and Ma [58] and it provides a natural way to explore a multidimensional parameter space. The separation between parameters and the actual network makes the vistrail model especially suitable to be used in such an interface. Users can change any of the parameters present in a workflow, creating new versions in the vistrail; they can also synchronize different views over a set of vistrail parameters – changes to this parameter set are reflected in related workflows in the same vistrail, shown in different cells of the spreadsheet.

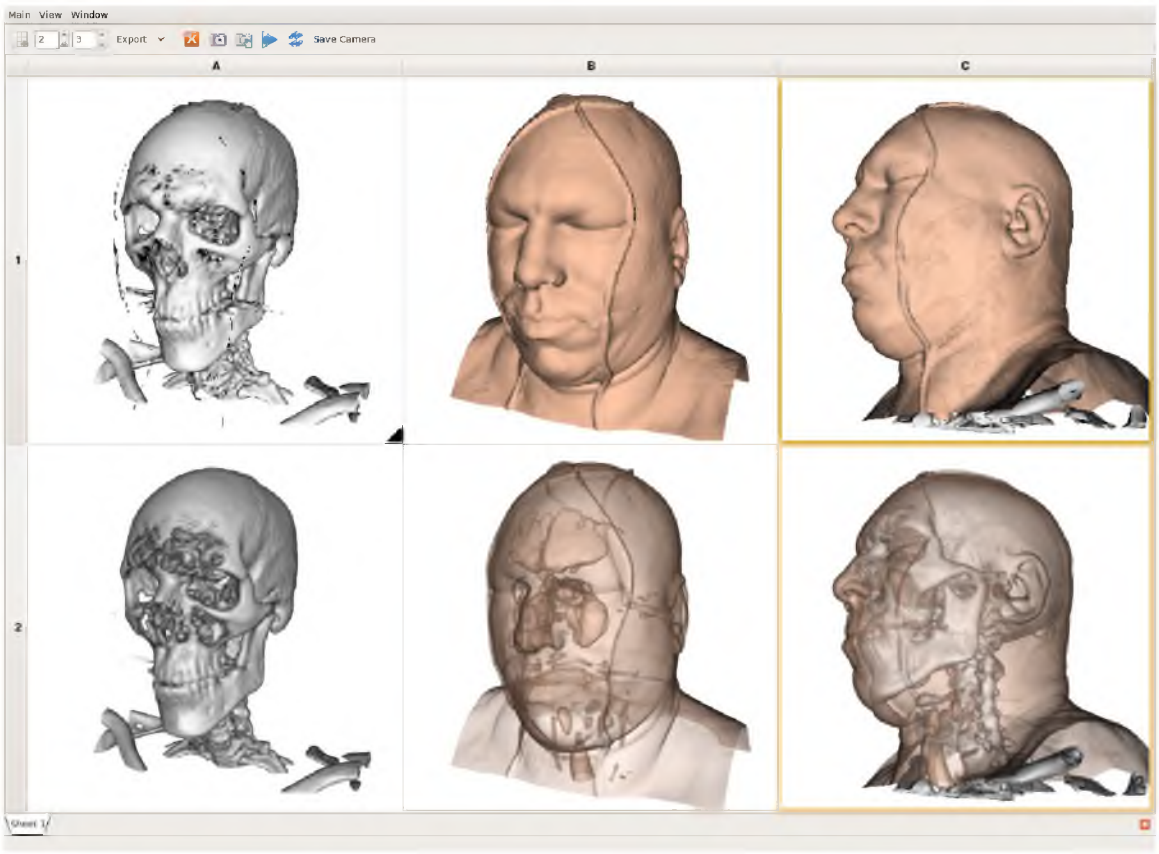


Figure 2.1: VisTrails provides support for multiple-view visualization via the VisTrails Spreadsheet. Cameras as well as other vistrail parameters for different cells can be synchronized. This spreadsheet shows visualizations of the Visible Human dataset using different algorithms. The top row shows the rendering of two isovalues, while in the second row, images are created using volume rendering. The cameras for the cells in the rightmost column in the spreadsheet above are synchronized.

CHAPTER 3

CHANGE-BASED PROVENANCE

The most visible distinction of VisTrails when compared to other similar systems is its *version tree*, shown schematically in Figure 3.1, and in a screenshot of the system GUI in Figure 3.2. This version tree is a user-accessible record of the entire exploration process at any point in time. Because users try different alternatives during the exploration process, and this process can be laborious [123], VisTrails eschews the notion of there being one single important visualization in a process, and considers instead the problem of managing this potentially large set of alternatives. We first present some necessary definitions. Then, we describe our model in detail, and relate it to the history management models presented in Chapter 2. This provenance model is the basis for the techniques presented in Chapters 4 and 5.

3.1 Definitions

A *visualization system* is a software system that provides functionality for graphically displaying data according to a specific set of rules. In our case, we restrict ourselves to *visual programming systems*, where the programmatic rules can be seen as a representing a program by a graph, and they are called *pipelines*. Executing the pipeline in the visualization system produces a *visualization*. The pipeline is composed of *modules* which define specific operations and *connections* which determine the flow of data between modules. The modules are the pipeline graph's vertices, and the connections are the edges. The modules and connections have additional information to specify the behavior of the resulting visualization. Modules have *input ports* that specify the set of possible inputs, and *output ports* that specify the outputs. Each connection links an output port of one module (the *source*) with an input port of another module (the *destination*). In addition, modules might require additional state settings, represented by a set of *module parameters*. Each module parameter is specified by an ordered pair of parameter name (one of a set specified by the module's designer, as described in Chapter 6) and value.

We denote the set of all visualization pipelines as $\mathbb{V}$. The VisTrails provenance model is intended to help the user manage a potentially large subset $v \subset \mathbb{V}$ of these pipelines.

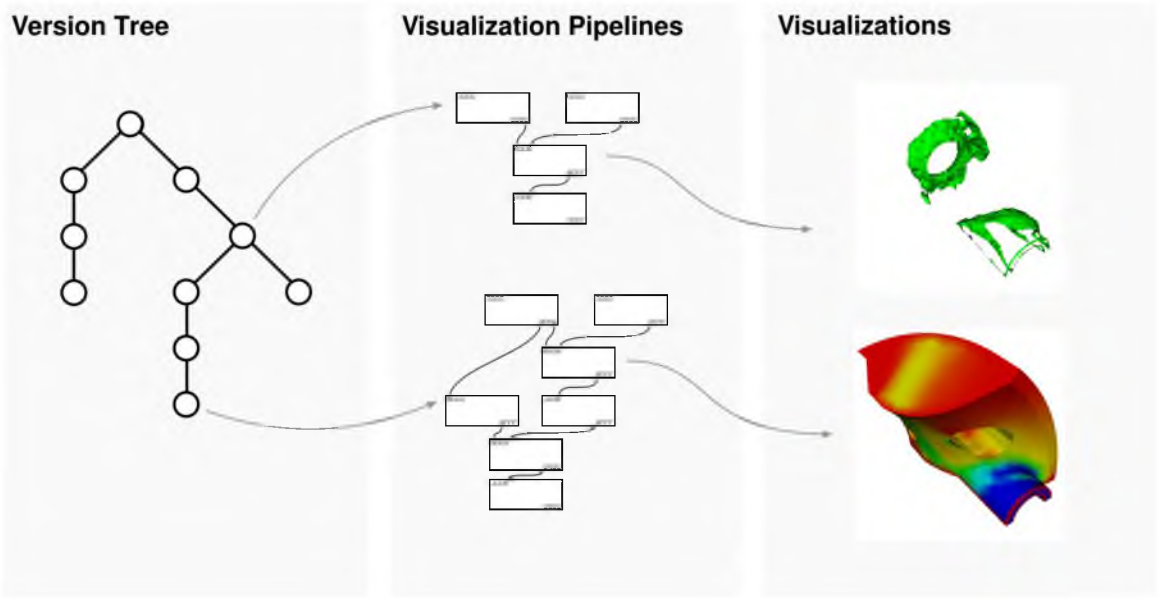


Figure 3.1: A review of terminology from VisTrails. The *version tree* (shown on the left) captures the history of the exploration process, where each node corresponds to a *visualization pipeline*. Edges in this tree are functions from $f : \mathbb{V} \rightarrow \mathbb{V}$. Each pipeline comprises *modules*, *connections*, and *parameter values*, and generates a *visualization* (shown on the right).

An important observation leveraged throughout this thesis is that every user operation performed on a pipeline, such as adding and deleting modules, connections and parameters, can be directly expressed as a function $f : \mathbb{V} \rightarrow \mathbb{V}$ (this function can be *partial*, that is, not specified for some of the elements in its domain). If the original user operation on a pipeline A resulted in a pipeline B, the only requirement for the representation of f is that $f(A) = B$. Many of our techniques depend on making these functions first-class elements in the visualization system.

We are now ready to present the actual model used in VisTrails to represent the user's exploration process in $\mathbb{V}$.

3.2 Modeling the Evolution of Pipelines

VisTrails uses an action-based model to capture provenance. As the user makes modifications to a particular workflow, the provenance mechanism records those changes. Instead of storing the set of all workflows, we store the operations or actions that are applied to them, such as the addition of a module, and the modification of a parameter. This representation is both simple and compact—it uses less space than the alternative of storing multiple *versions* of a workflow. In addition, it enables the construction of an intuitive interface that allows users to both understand and interact with the history of the workflow through these changes. A tree-based view allows a

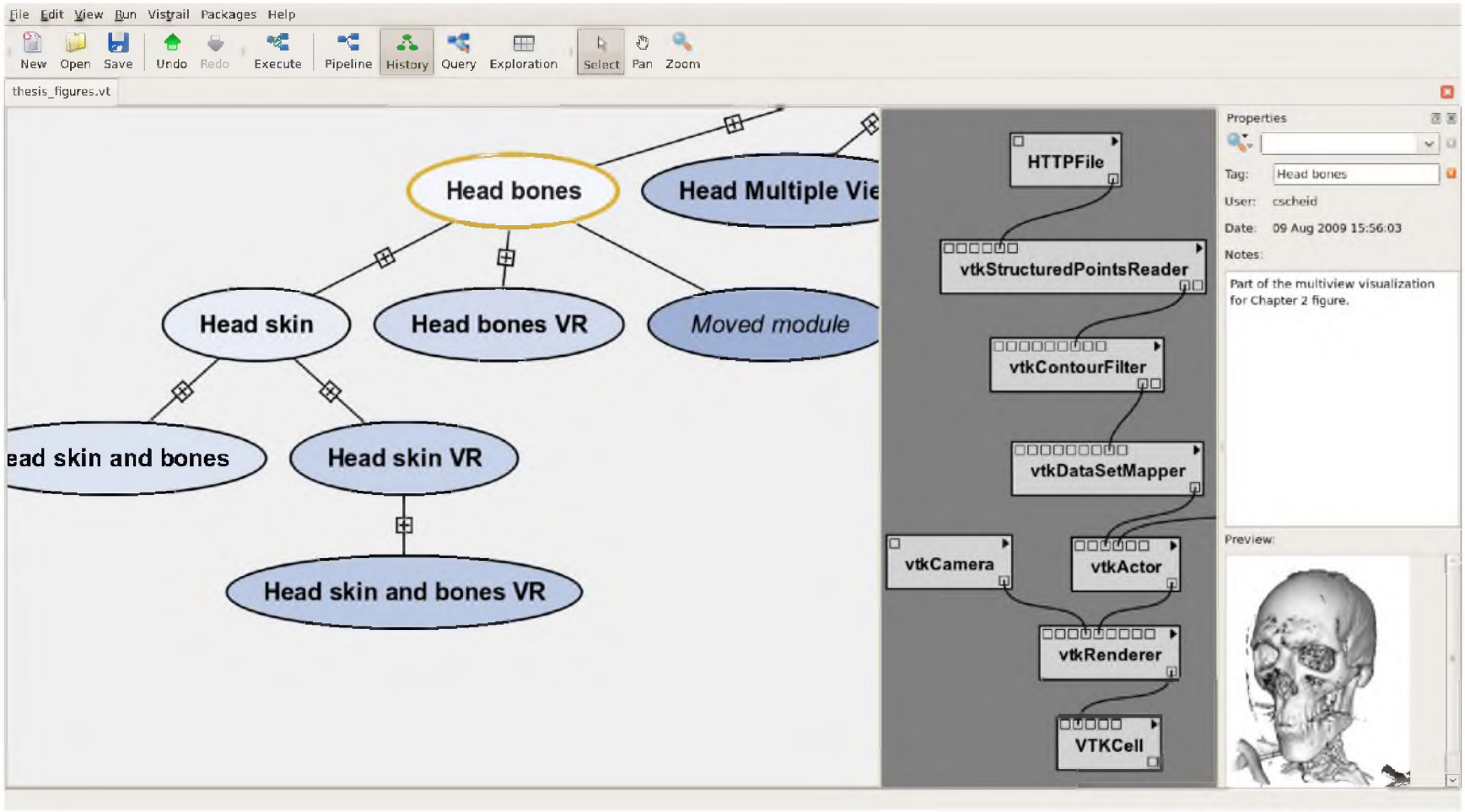


Figure 3.2: An overview of the main VisTrails GUI. Shown on the left side is the version tree view, where users can navigate the captured history of the exploration process. In the middle, we show the highlighted workflow, and associated notes and thumbnail on the right side.

user to return to a previous version, to undo bad changes, to compare different workflows, and to be reminded of the actions that led to a particular result. This, combined with a caching strategy that reduces redundant computations described in Chapter 6, facilitates the exploration of a large number of related visualizations.

VisTrails provides a mechanism that uniformly captures provenance information for both the data and the processes that derive the data. We will first describe the mechanism to capture the user's exploration history, and then show how it can be used to capture data provenance as well. Along with the benefits of previously proposed provenance mechanisms, such as reproducibility and documentation, the action-based model has additional benefits. As we discuss in Section 3.8, it enables operations that streamline the data exploration process, and is the basis for Chapters 4 and 5.

A vistrail VT is a tree in which each node corresponds to a *version* of a pipeline, and each edge between nodes p_p and p_c, where p_p is the parent of p_c, corresponds to the function applied to p_p which generated p_c. As we described before, let $\mathbb{V}$ be the domain of all possible visualization pipelines. In addition, we will assume the existence of one special visualization pipeline, $\emptyset \in \mathbb{V}$, that corresponds to an empty pipeline, with no modules or connections, which generates no visualizations. This special version has no action associated with it, and hence has no parent. We will treat it as the *root* of the tree. Also, let $\delta : \mathbb{V} \to \mathbb{V}$ be a function that transforms a workflow instance into another, and $\mathscr{D}$ be the set of all such functions. when a function represents an atomic change performed by a user, we will denote it by f. A vistrail node corresponding to a pipeline p is constructed by composing a sequence of actions, where each $\delta_i \in \mathscr{D}$:

$$p = (f_n \circ (f_{n-1} \circ \cdots \circ (f_1 \circ (\emptyset))...)) \tag{3.1}$$

In what follows, we use the following notation: we represent pipelines as p_p, and if a workflow p_c is created by applying a sequence of actions on p_p, we say that $p_p < p_c$ (i.e., the vistrail node p_c is a descendant of p_p). Note that, necessarily, for all $p \in \mathbb{V}$, if p is different than $\emptyset$, then $\emptyset < p$.

An excerpt of the vistrail XML schema is shown in Figure 3.3. For simplicity of the presentation, we only show a subset of the schema and use a notation less verbose than XML Schema. A vistrail has a unique ID, a name, an optional annotation, and a set of actions. Each action is uniquely identified by an ID (`@id`), the ID of its parent (`@parentId`), its creator (`@user`), and the time of creation (`@date`). The different actions we have currently implemented include adding, deleting and replacing workflow modules, adding and deleting connections, and setting parameter values. To simplify the navigation and querying of the history, a version may have a name (the optional attribute `tag` in the schema) as well as additional notes from the creator (`annotations`).

```
type Vistrail = vistrail [ @id, @name, Action*,
                           annotation? ]
type Action =
 action [ @id, @parentId, @user, @date,
          tag?, annotation?,
          (addModule|deleteModule|replaceModule|
   addConnection|deleteConnection|
   |setParameter|changeParameter)]
```

Figure 3.3: Excerpt of the schema for a vistrail. Notably, the model does not store any workflows directly; it instead encodes them by actions that change the workflow.

3.3 Determining Pipeline Differences

Workflow-based systems allow users to create a variety of pipelines, rather than being restricted to a predefined set of visualizations. In the process of deriving insightful visualizations, a series of pipelines is often created by iterative refinement. To understand this process as well as the derived visualizations, it is useful to compare the different pipelines. The standard representation of a pipeline is a directed graph, with labeled vertices representing operations. Given a pair of such pipelines, we want to determine the difference between the visualizations they generate. In the following, we show how to describe and manipulate differences between pipelines.

We previously defined $\delta : \mathbb{V} \to \mathbb{V}$ as a function on the space of visualizations. Now, let $\Delta : \mathbb{V} \times \mathbb{V} \to \delta$ be a function that takes two pipelines p_a and p_b and produces another function that will transform p_a to p_b. For brevity, let $\delta_{ab} = \Delta(p_a, p_b)$. From now on, we will use δ to refer to an arbitrary $\Delta(a,b)$. It is clear that δ is not unique: even though $\delta_{ab}(p_a) = p_b$ is a necessary constraint, there are no further restrictions. Loosely speaking, we would like to pick the δ_{ab} that minimally changes all other pipelines. If we define the distance between p_a and p_b as the number of changes necessary to perform the transformation, then a natural way to determine δ_{ab} is as one such function that satisfies the distance between p_a to p_b. This is easily seen to be as hard as finding a best matching between two pipelines p_a and p_b, which is the problem of subgraph isomorphism. Furthermore, the subgraph isomorphism problem is trivially reducible to the MAX-CLIQUE problem, a well-known NP-hard problem [42]. Additionally, MAX-CLIQUE is a particularly hard problem: there is no approximation algorithm for it with approximation factor better than $n^{1-\varepsilon}$, for any constant positive value of ε. [46].[1] This is about as bad a negative result as one can get, since outputting a single vertex gives an approximation factor of n. Since we cannot possibly get a good approximation, heuristics for both problems are well justified.

[1] This result assumes that NP $\neq$ ZPP. Quoting Hastad [46], "The faith in the hypothesis NP $\neq$ ZPP is almost as strong as in NP $\neq$ P."

We restrict our initial analysis to the simple case where p_b is *derived* from p_a — the user created p_b by applying a finite set of changes to p_a. We denote this relationship as $p_a < p_b$. Then, a system with some knowledge of how the pipelines were constructed should be able to determine the differences between related pipelines using this history.

When $p_a < p_b$, we can then say δ_{ab} is the sequence of operations that was used to derive p_b from p_a. However, few pairs of versions satisfy the $<$ relation, and we would like Δ to be less restrictive. We start with a simple extension: if δ_{ab} exists, so should δ_{ba}. Specifically, we would like the following functional equation to hold:

$$\delta_{ab}\delta_{ba} = e \tag{3.2}$$

where e is the identity function. We can achieve this if our sequence of changes consists of invertible atomic operations. Specifically, suppose $\delta_{ab} = f_n \circ \cdots \circ f_1$ where each f_i has a well-defined inverse. For example, if f_i is the operation of adding a module to the pipeline, f_i^{-1} is the operation of deleting that module from the pipeline. Then,

$$\delta_{ba} = \delta_{ab}^{-1} = f_1^{-1} \circ \cdots \circ f_n^{-1} \tag{3.3}$$

From now on, we assume that any function that operates on $\mathbb{V}$ has an inverse on a, that is, that for every f such that $f(a) = b$, we can construct an f^{-1} such that $f^{-1}(b) = a$. Note that both functions might still be partial.

3.4 Performance

In terms of storage, the action-based model we use for history management is optimal in the sense that it only requires $O(1)$ storage per new version. In fact, storage cost for a vistrail file stored in disk is dominated by the thumbnails generated for executed workflows. To give an idea of the actual storage necessary, we computed the average action size by inspecting every vistrail collected from students performing a set of scientific visualization tasks in a course taught at the University of Utah (the dataset is described in more detail by Lins et al. [73]). Every action, stored in disk takes an average of 137.6 bytes. This is computed by taking the entire XML description of a vistrail (including annotations and tags), compressing it with the traditional ZIP tool (which is actually how a vistrail is stored in disk), and dividing the total amount of disk space taken by the total number of actions in all vistrails. Figure 3.4 shows a scatterplot with the distribution of actions versus file sizes.

The obvious problem the model presents, however, is its time complexity per access of any particular version. Without any other modification, the scheme described above requires $O(d)$ time to materialize any version by replaying all the actions from the root, where d is the distance from the

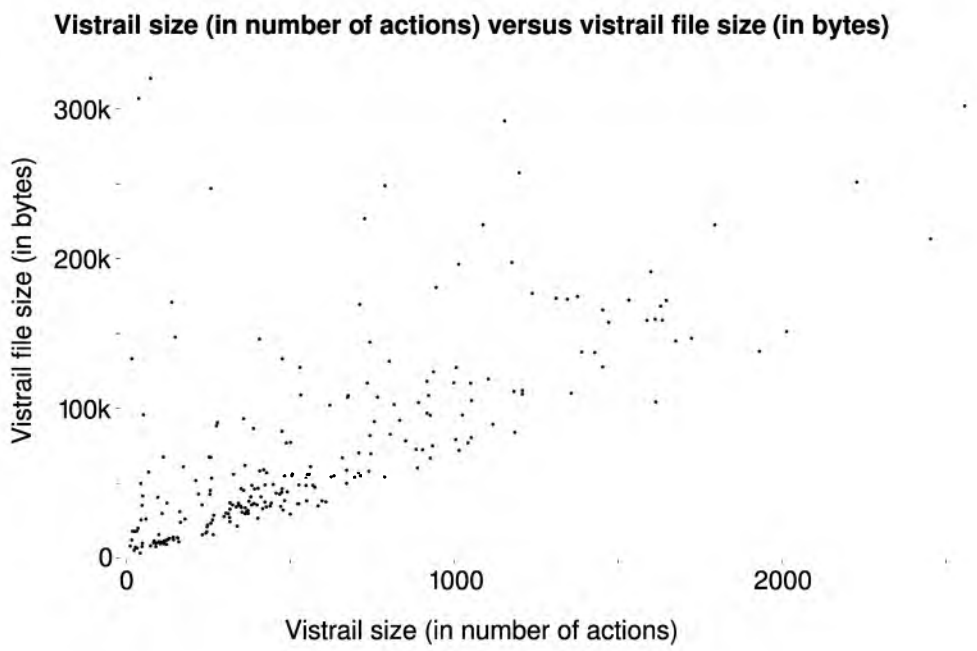


Figure 3.4: Scatterplot of file size versus action count using 251 vistrails provided by students in two scientific visualization courses. The average action size is 137.6 bytes. One outlier was cropped to increase plot legibility.

desired version to the root node, assuming each action requires $O(1)$ time to execute. In addition, d might be much larger than the size of the visualization pipeline under consideration (call that s), so replaying through these versions is likely to take much more work than would be necessary to produce a copy of said pipeline. For example, again using the same dataset as before, the average visualization pipeline size (combined number of modules, connections and parameter settings) was 40.44, while the average depth of a tagged version was 118.20 (we show a scatterplot of this in Figure 3.5). If space complexity was not an issue, it is obvious that we could simply store every intermediate version on a vistrail of n nodes directly for a total storage cost of $O(ns)$ and access time $O(1)$. Our goal is to find a trade-off between storage cost and version access time.

More generally, this problem occurs any time a data structure *changes* over time and we are required to have access to its past states. These are known as *permanent* data structures [60]. More specifically, if any version can only be changed once (that is, if past versions can be accessed but not mutated), they are known as *partially* persistent. If any version can be changed, these structures are known as *fully* persistent. One way to see the version tree proposed in this chapter is as a data structure for fully-persistent visualization pipelines. Finally, persistent data structures can be *confluent*, which means that there exists an operation to combine two versions. Since we do not

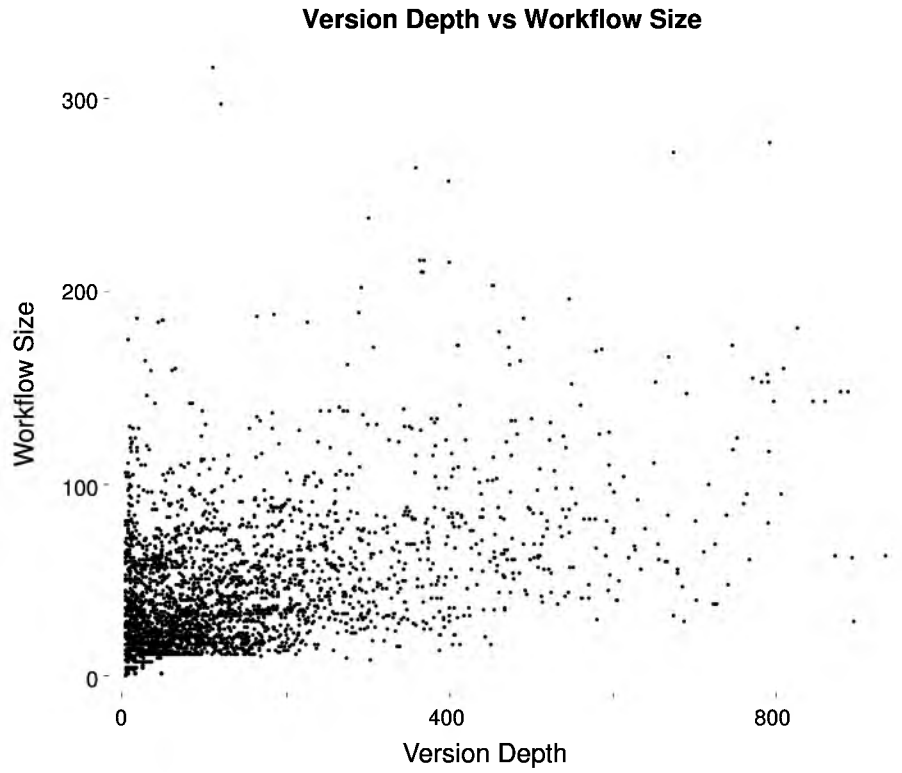


Figure 3.5: A scatterplot that compares the depth of workflows in a vistrail and their size, measured by the sum of module count, connection count and parameter setting. The data used here are vistrails provided by students during two scientific visualization courses. The average workflow size is 40.44, and the average depth is 118.20. This scatterplot is slightly cropped (with about 20 outlier workflows removed) to increase legibility.

support such operations, we will not discuss them any further.

Driscoll et al. showed how by paying close attention to the actual data structures being stored, it is possible to achieve an optimal $O(m)$ total space usage (where m denotes the total number of versions) and $O(1)$ amortized time to access any given version [29]. Their analysis and techniques for fully persistent data structures, however, are quite involved. More importantly, they require the data structure under investigation to be comprised of nodes of bounded in-degree, which is not the case in our situation. We restrain ourselves to an analysis of checkpointing, which is fully general and applies to black-box cases where we do not have any access to the data structures aside from storing copies.

Let us first look at checkpointing for partially-persistent data structures, those whose history is linear. It is possible to show that to minimize the worst case of space blowup or time spent per access, we must store $O(\sqrt{n})$ copies of the data structures, spaced $O(\sqrt{n})$ versions apart from each other. We now show that picking $O(\sqrt{n})$ versions uniformly at random also allows us to achieve

average-time $O(\sqrt{n})$ time access to any version. This random technique will be combined with an online replacement of old versions to allow $O(\sqrt{n})$ when n is not known in advance, such as when users interact with the version tree and simultaneously create new versions, which is the typical mode of operation.

Consider the selection of a version at depth d in a partially-persistent data structure with n versions, k of which are checkpointed. Assuming $O(1)$ cost per action, the cost of accessing this version is equal to the run of uncheckpointed versions needed to find the checkpoint. We proceed via a straightforward counting argument, illustrated in Figure 3.6.

If we think of a possible configuration of the checkpoints as an n-digit binary string, then by defining a random variable C to represent the number of left-looking runs of zeros starting at digit d, it is easy to see that its expectation is given by

$$E_{\text{conf}}[C(d,n,k,\text{conf})] = \binom{n}{k}^{-1}\left(d\binom{n-1-d}{k} + \sum_{0\le i\le d} i\binom{n-i}{k-1}\right) \tag{3.4}$$

with the left binomial term being the probability of not seeing any checkpoint all the way up the root, and the binomial term inside the sum being the probability of seeing a run of length i (we use "conf" to denote the possible configurations with k checkpoints). By assumption, the checkpointed versions are independent of the current version picked, so the joint probability $P(d,n,k,\text{conf})$ factors as $P(d)P(n,k,\text{conf})$. In this simple case, versions and depths stand in one-to-one correspondence. The expectation over all possible versions is, then,

$$E_{d,\text{conf}}[C(d,n,k,\text{conf})] = n^{-1}\sum_{1\le d\le n-1} E_{\text{conf}}[C(d,n,k,\text{conf})]. \tag{3.5}$$

Surprisingly, the expression has a simple closed form (found using Mathematica [101]):

$$E_{d,\text{conf}}[C(d,n,k,\text{conf})] = \frac{-2+(-3+n)+k(2+k+n^2)}{(1+k)(2+k)}. \tag{3.6}$$

Now, we simply set $k = n^{1/2}$. With this, we get $(-2+2n^{1/2}-n+n^2+n^{3/2})/(2n+3n^{3/2}+n^2) = O(n^{1/2})$, our desired result.

A full analysis of the optimal number of checkpoints for arbitrary trees is much more complicated. It is clear that special classes of trees allow better worst-case situations (consider a balanced binary tree: since the height is $O(\log n)$, we need *no* copies to achieve the $O(\sqrt{n})$ bound). We would like to know if it is possible to achieve the $O(\sqrt{n})$ bound for general trees. Based on our empirical data collected (and shown in Figure 3.7), the trees in history trees produced by users of VisTrails tend to have relatively large depth. Since we expect these trees to have close to $O(n)$ depth, we settle by convincing ourselves that $O(\sqrt{n})$ checkpoints will always be enough to attain $O(\sqrt{n})$ access times: that is, branching will never hurt the asymptotic expected access time.

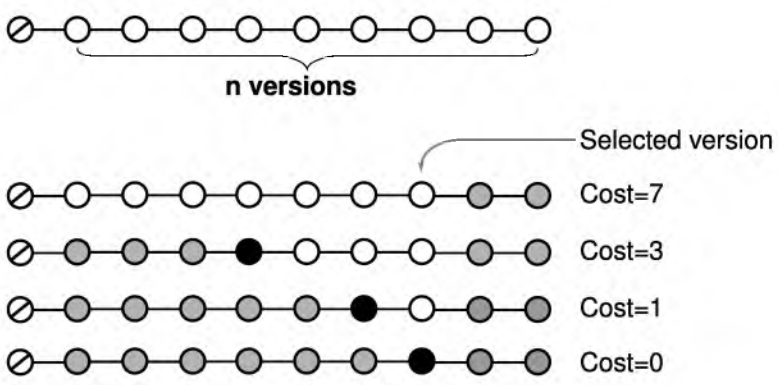


Figure 3.6: The cost of accessing a version at depth d out of n versions with a certain configuration of k checkpoints is equal to the run of uncheckpointed versions until either a checkpoint or $\emptyset$ is found. White nodes denote uncheckpointed versions, black nodes denote checkpoints, and gray nodes denote nodes whose status is irrelevant for the cost of the configuration. The tree root is shown as the leftmost node.x

When computing the expected cost per node, we have to account for the checkpoint versions that might be inaccessible in separate branches of the tree. The expected cost, for any given node, is the same to a node at the same depth in a long list with as many nodes as the tree, except that all the branches are reglued to the back of the branch in consideration. Notice, then, that the transplanted branches always can never have a smaller cost in the list configuration than they did in the tree configuration, and so we are done: $O(\sqrt{n})$ checkpoints are enough. This is illustrated in Figure 3.8.

We have decided to use a randomized version of checkpointing because it lends itself to a simple online extension of the algorithm. This is desirable, since during the course of interacting with the version tree, n will increase (say, from n to n'), and so the original checkpoints will be, on average, $O(\sqrt{n})$ apart, but likely not $O(\sqrt{n'})$. To remedy this situation, every time a new version if added to the version tree, a random checkpoint is created with probability $1/\sqrt{n}$. This ensures that there will be, on average, $O(\sqrt{n})$ checkpointed versions available. To ensure that they will be correctly distributed, we also *replace* the oldest checkpointed version with the currently selected version when the user navigates to a different version in the tree (which, under our assumption, will be any one of the versions with equal probability). This way, the checkpointed versions gradually "spread out" and become uniformly distributed on the version tree, regardless of the growing.

Notice that these checkpointed workflows contain information that is dependent on the set of VisTrails modules currently enabled on the system. If a vistrail with checkpointed workflows is shared, and opened in a version of the system with upgraded VisTrails packages, or missing libraries, a linear-time pass over each checkpointed workflow can determine if the checkpoint can be used. If it cannot, this checkpoint is simply removed from the vistrail. Section 6.6 deals with the problem of

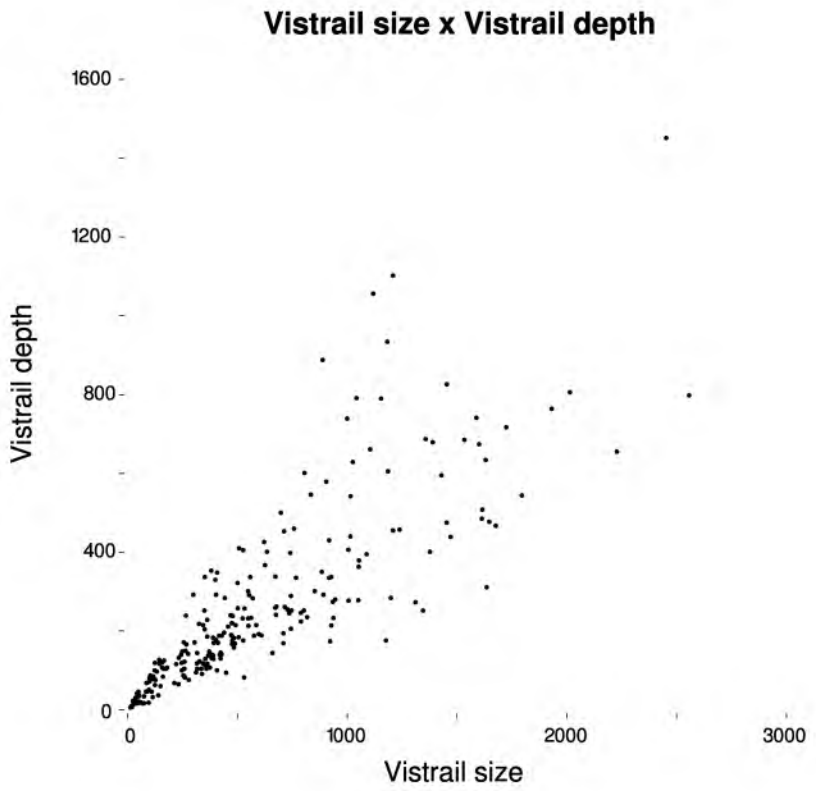


Figure 3.7: Scatterplot of maximum depth versus version count in 251 user vistrails containing tasks collected from two scientific visualization courses. These trees tend to have relatively few branches and deep nodes: note the linear trend.

evolution of VisTrails packages in more detail.

3.5 Distributed Operation

In this section, we describe how we use the action-based provenance mechanism to allow several users to collaboratively, and in a distributed and disconnected fashion, modify a vistrail—collaborators can exchange patches and/or synchronize their vistrails.

An important feature of the VisTrails provenance model is *monotonicity*: nodes in the vistrail history tree are never deleted or modified—once pipeline versions are created, they never change. Having monotonicity makes it possible to adopt a collaboration infrastructure similar to modern version control systems (e.g., GNU Arch, BitKeeper, git and DARCS), where every user's local copy can act as a repository for other users. This enables them to work offline, and only commit changes they perceive as relevant.

As we discuss below, the main challenge in providing this functionality lies in keeping consistent action timestamps—the global identifiers of actions in a vistrail. Intuitively, in a distributed opera-

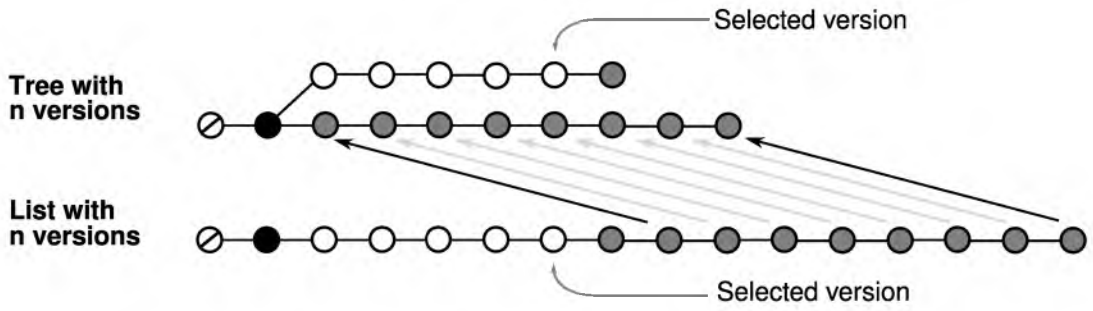


Figure 3.8: The average cost of accessing nodes in a tree with n nodes and k checkpoints is never worse than the cost of a list with the same number of nodes and checkpoints. The average cost at a depth d includes a sum to d, and there is always a mapping of nodes (by the pigeonhole principle) from one such list to a tree that will not increase the depth of any node.

tion, two or more users might try to commit workflows with the same timestamp, and a consistent relabeling must be ensured.

3.5.1 Synchronizing Vistrails

We call *vistrail synchronization* the process of ensuring that two repositories share a set of actions (or, equivalently, visualizations). Figure 3.9 gives a high-level overview of the synchronization process. Because of monotonicity, to merge to history trees, it suffices to add all nodes created in the independent versions of a vistrail. In what follows, we describe an example scenario that illustrates the issues that must be addressed in vistrail synchronization.

Suppose user A has a vistrail which is checked out by both users B and C. User B creates a new workflow, represented by a sequence of actions with timestamps {10, 11, 12}. Unbeknownst to user B, user C also creates a new workflow, which happens to have overlapping timestamps: {10, 11, 12, 13}. User C happens to commit its workflow before user B, so when B decides to commit his changes, there will already be actions with his timestamps. The only solution is for A to provide B with a new set of action timestamps, which A knows to be conflict free *in his vistrail* (say, {14, 15, 16}). The problem appears simple, expect B might himself have served as a repository for user D, who checked out {10, 11, 12} before B decided to commit. If B ever exposed his changed timestamps, a cascade of relabelings might be necessary. Worse than that, D might be offline, or depending on the operation mode, B might even be unaware of D's use of the vistrail.

Our solution is based on a simple observation. Action timestamps need to be unique and persistent, *but only locally so*. In other words, even if user A exposes his actions to B as a set of timestamp values, there is no reason for B to use the same timestamps. The problem lies exactly when actions created by user B have timestamps that may be changed in the future, when committed

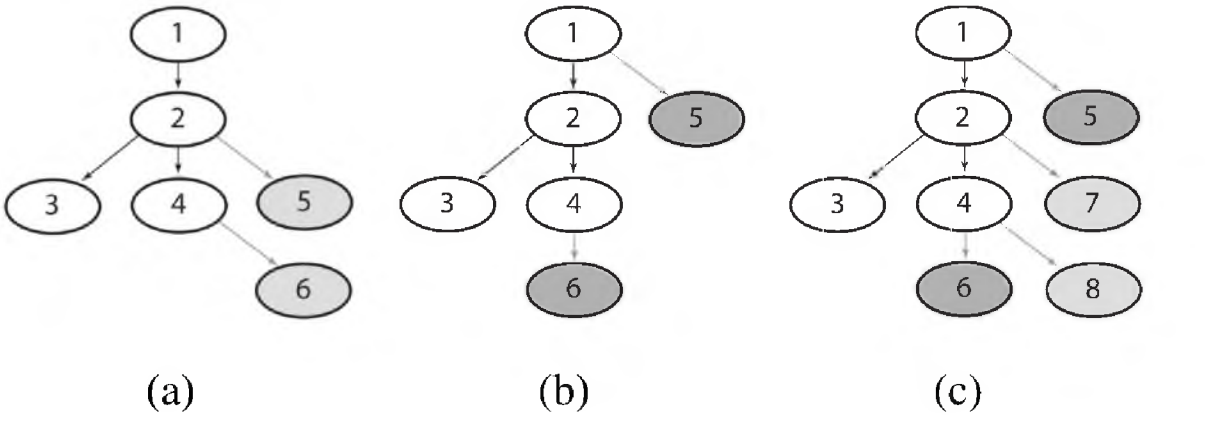


Figure 3.9: Synchronizing vistrails. When users collaborate in a distributed fashion (subfigures (a) and (b)), they might create actions with the same timestamp. When these are committed to the parent repository, some timestamps have to be changed (subfigure (c)).

to A. To avoid that, we introduce what we call *relabeling maps*, a set of bijective function $f_i : \mathbf{N} \to \mathbf{N}$. Each user keeps a relabeling map whose preimage is the set of timestamps given by the parent vistrail i, and whose image is a local set of timestamps which will be exposed in case its vistrail is used as a repository. When the user commits a set of actions, the parent vistrail might provide a new set of timestamps (more specifically, the parent creates new entries on its own relabeling map, and exposes new timestamps). The child vistrail's relabeling map then *only changes the preimage*. In the previous paragraph's example, part of B's relabeling map preimage goes from $\{10, 11, 12\}$ to $\{14, 15, 16\}$, but the image stays the same. If we call f_B the old relabeling map, and f'_B the new one, then $f_B(10) = f'_B(14)$, $f_B(11) = f'_B(15)$ and so on. Notice that in this way, it does not matter what B's relabeling matter is, aside from its image not changing when B commits back to A. Since D's repository only depends on the image of f_B, D will never be affected by any actions of B, a property essential for distributed operation. Figure 3.10 illustrates the idea.

This distribution model of vistrails allows for operation under peer failure. Using the above example, assume user B's hard drive fails, losing his vistrail repository. Even though the local changes are lost, some of the data might be available in user D's vistrail. In failure mode, we allow D to commit changes directly to A (or any other repository). Even though this makes it possible to prevent data loss, some redundancy becomes inevitable. Since user B's relabeling map has been lost, it is impossible to know the mapping between user D and user A's timestamps. We simply assume, then, that all actions user D wants to commit are new. The most important feature of this operation mode is that it does not violate *monotonicity*. User A's vistrail is still valid, user C might still use user A's vistrail, and user D will simply receive a completely new preimage for its relabeling

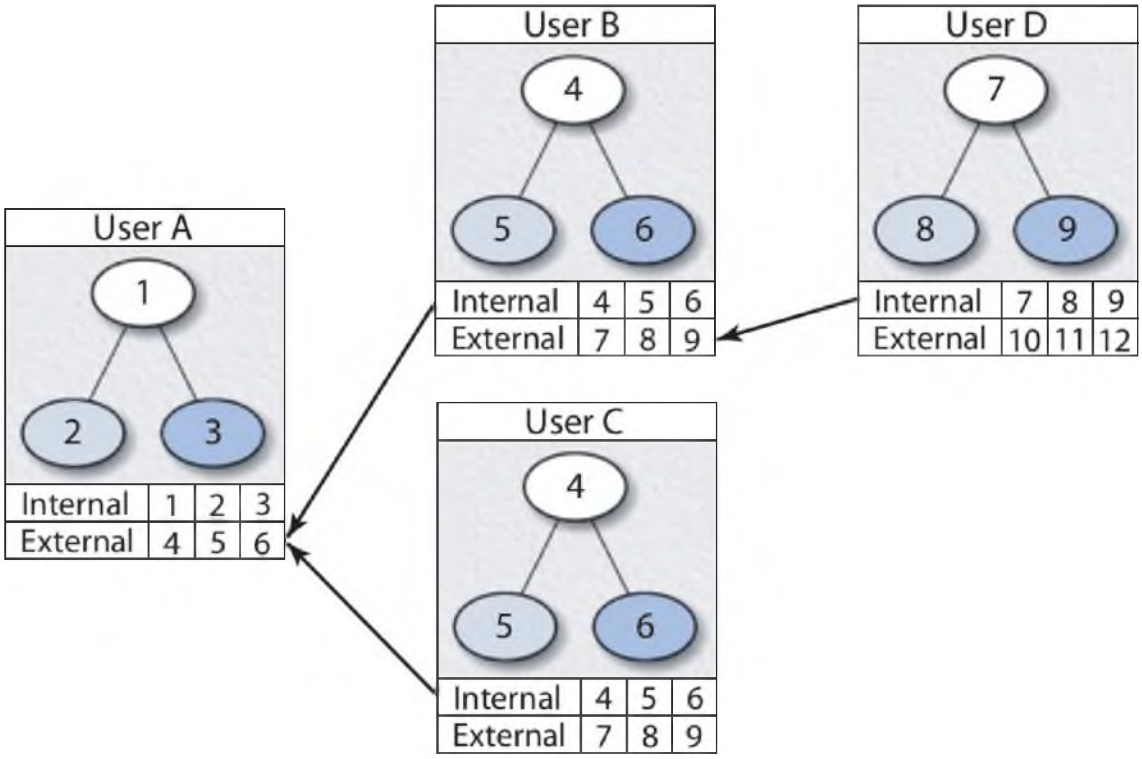


Figure 3.10: Synchronizing vistrails through *relabeling maps*. Even though its local timestamps might change on commits, each vistrail exposes locally consistent, unchanging timestamps to the world, ensuring correct distributed behavior.

map. The most important feature of the scheme is that users that have checked out user D's vistrail need not be aware of user B's failure. This is illustrated in Figure 3.9.

3.6 Data Provenance via Process Provenance

One of the fundamental problems users face when trying to determine the provenance of a certain file is following the chain of results generated by an exploration task. Critically, the answer to the question "where did this file come from?" might be composed of multiple workflows that generate and consume intermediate results. Most systems that tackle the problem of capturing provenance focus directly on the relationship between the *data products* [89].

In contrast, the perspective we present in VisTrails is that it is more valuable to capture data provenance indirectly via the processes that generated the data. Most importantly, it allows us to very naturally answer the related question "what was the change in the underlying process that caused these two results to be different?" Unless a complete record of the changes in the process is transparently and permanently stored, the burden of management is put on the user. In addition, this related question is commonly the important one during provenance analysis [37]. The VisTrails

execution log contains the records of workflow executions. By linking the record of a file being written to disk (or an SQL "insert" statement, for example) with the related read, it is possible to trace these data dependencies — in fact, Koop et al. describe a translation mechanism from the VisTrails provenance model to the Open Provenance Model (OPM), which has recently been suggested as a common model for data provenance [90]. One potential problem with this simple approach is that the raw data files can be moved, deleted or modified.

More sophisticated mechanisms can be used to maintain the raw data [38,115]. For example, the RHESSI Experimental Data Center [115] uses a mapping scheme to locate and retrieve data items, and it ensures data consistency by only allowing raw data to be accessed through the meta-data available in the database. Since VisTrails workflows are extensible by design, these systems could be easily integrated with VisTrails via a specialized VisTrails module package.

3.7 Computing Pipeline Differences

This problem becomes much easier using the action-based provenance model. Let d_1 and d_2 be two vistrail nodes corresponding to the pipelines P_1 and P_2, respectively:

$$d_1 = x_k \circ x_{k-1} \circ \cdots \circ x_1 \circ \emptyset \tag{3.7}$$

$$d_2 = y_i \circ y_{i-1} \circ \cdots \circ y_1 \circ \emptyset. \tag{3.8}$$

Since all the vistrails nodes are represented by paths in a rooted tree, they all share at least one common ancestor. Let $c = x_m = y_j = lca(x_k, y_i)$ be the least common ancestor of x_k and y_i. The common part of d_1 and d_2 is the path from the root of the tree down to node c, i.e., $\{c \circ \cdots \circ \emptyset\}$. The difference computation is simple and efficient—it can be done in linear time: after obtaining the least common ancestor of the nodes, the difference $d_2 - d_1$ is represented by two lists of actions: $\{x_k \circ \ldots \circ x_{m+1}\}$ and $\{y_i \circ \ldots \circ y_{j+1}\}$. Using these lists, we identify the modules that are unique in the two pipelines, and for modules that are present in both pipelines, we identify their differences by examining *setParameter* actions. For display, we divide the nodes into four categories: shared nodes, shared nodes with parameter changes, nodes that belong only to P_1, and nodes that belong only to P_2. Edges are divided into shared edges, and edges that belong to P_1 or P_2, respectively. A remaining issue is how to handle nodes that have been deleted and re-inserted to a given workflow. We handle this case by inspecting module names and function signatures (although we do not do it, another possibility is to use the matching algorithm described in Chapter 4). Obviously, the actual path taken from P_1 to P_2 might not actually encode a minimal set of such changes. However, it does encode a path actually taken by users, and in our experience, the encoded difference seems reasonably short and without many obviously unnecessary changes.

Figure 3.11 shows the visual difference interface provided by VisTrails. A visual difference is enacted by dragging one node in the history tree onto another, which opens a new window with a difference pipeline. The difference pipeline displays modules unique to the first node in orange, modules unique to the second node in blue, modules that are the same in dark grey, and modules that have different parameter values in light grey. The parameter changes are displayed in a subwindow when a light grey module is selected. Using this interface, users can correlate differences between two visualizations with differences between their corresponding pipelines.

3.8 Data Exploration via Pipeline Manipulations

Capturing provenance by recording changes applied to workflows has benefits in both uniformity and compactness of representation. In addition, it allows powerful data exploration operations through direct manipulation of the history tree. First, we describe our interface for interacting with provenance to gain a better understanding of the exploration process than a single visualization specification provides. Second, we show that stored actions lend themselves very naturally to reuse through a macro mechanism. Third, we describe a bulk-update mechanism that allows the creation of a large number of visualizations of an n-dimensional slice of the parameter space of a workflow. Finally, we describe how the differences between workflows can be efficiently computed in the action-based model.

3.8.1 Interacting with the Process Provenance

Figure 3.12 shows an example of a history tree. Each node in the tree corresponds to one visualization pipeline. An edge between two nodes represents one or more actions that changed the parent pipeline. To avoid visual clutter, in addition to a full tree view, we provide a default condensed view of the tree. The condensed view shows only the nodes that the user assigns a tag, or nodes that form a branch in the history. Since a tag is only metadata, it can easily be changed or even removed, thus changing the visual representation of the tree. A series of actions between tagged nodes is represented by a "+" icon that can be expanded and collapsed with a click.

As described above, additional metadata (e.g., annotations, creator, and date of creation) is recorded with each action and is displayed with the history tree. Annotations can be provided by the user to facilitate searching, clarify the exploration process, and provide the additional information that may be needed for a complete audit trail. When a node is selected, the metadata is displayed in the right panel of the history tree viewer, as shown in Figure 3.12 (left). In addition, the creator, date information and thumbnail are incorporated as visual cues in the history tree. The color of the nodes denotes the creator (blue nodes were created by the user and orange nodes were created by someone else) and the saturation of the node denotes chronology (more recent nodes are more saturated).

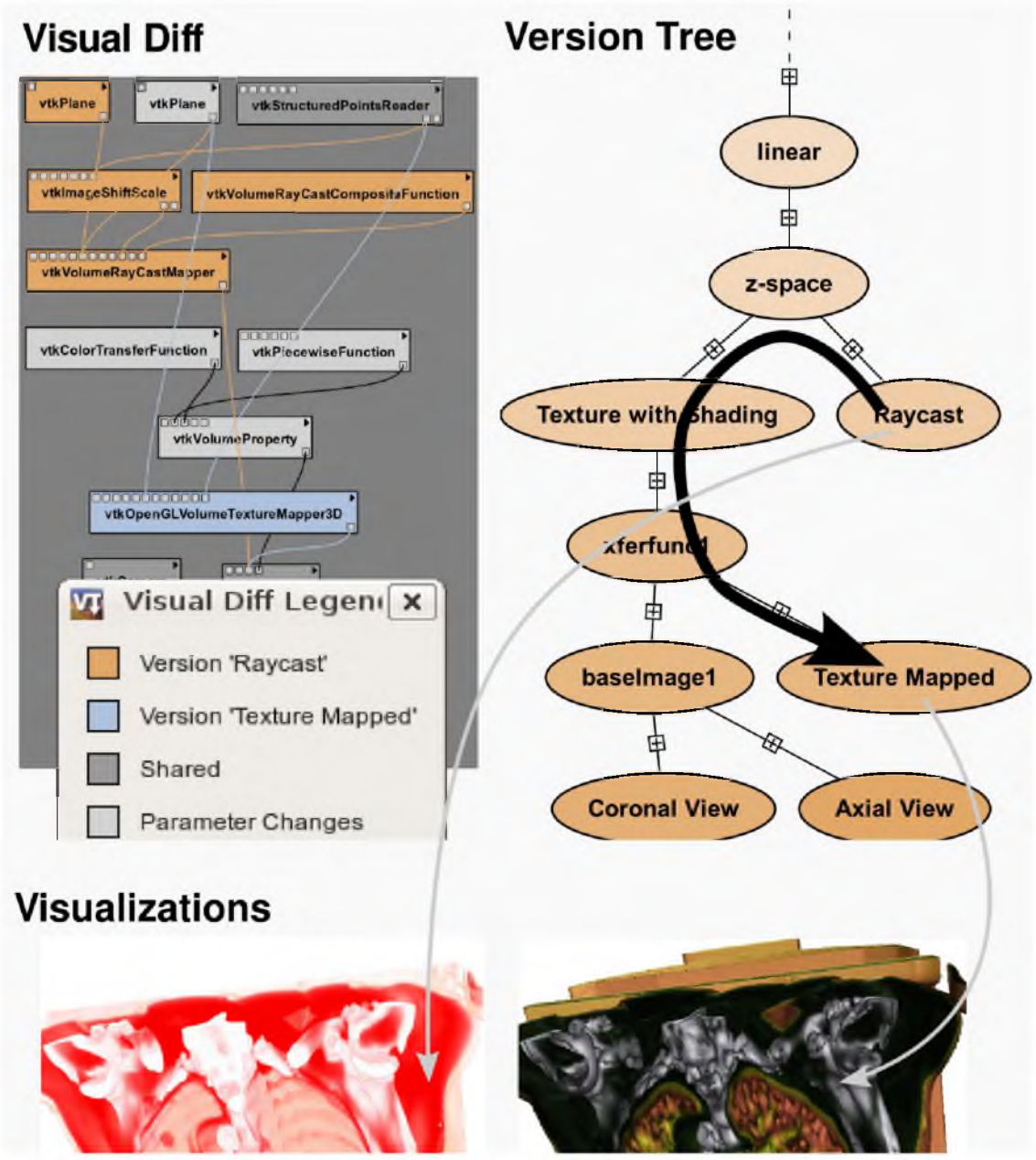


Figure 3.11: Computing the differences between two workflows that derive visualizations of CT data of a lung containing pathological tissue. In VisTrails, users can select nodes (workflows) in the vistrail tree to be compared. Here, the user selected the workflows labeled "RayCast" and "Texture Mapped." Their corresponding visualizations are shown on the right and the differences between the two workflows are shown in the bottom. Modules shown in blue are present only in "Texture Mapped." orange modules are present only in "RayCast," dark grey modules are present in both workflows, and light-grey modules have different parameter values in the two versions as shown on the bottom left for the "vtkColorTransferFunction" module.

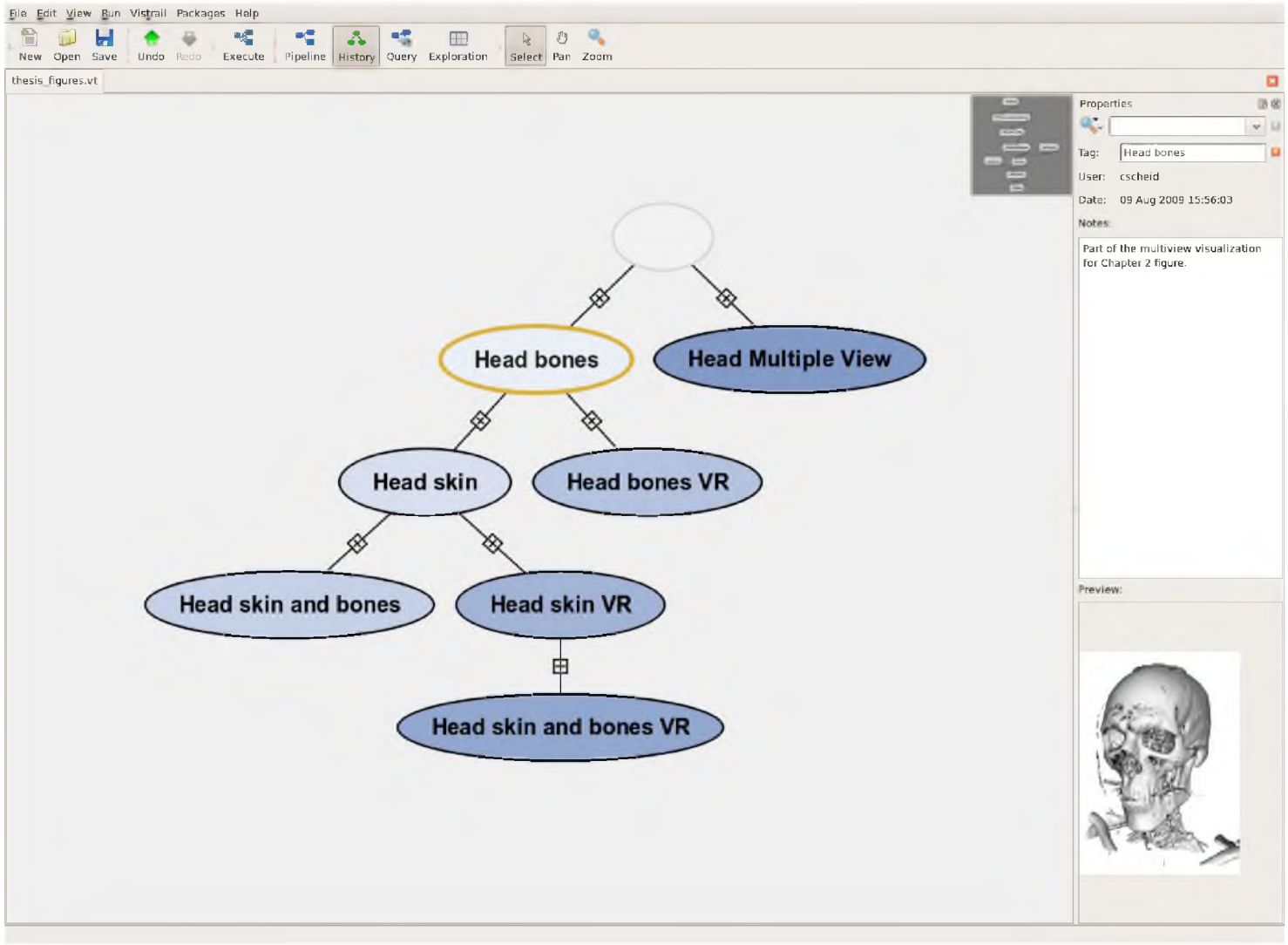


Figure 3.12: An overview of the version tree interface in VisTrails.

These cues, along with the tags and annotations, allow a user to open a previously unseen vistrail and quickly get an overview of the exploration process.

Inevitably, as a user is creating and exploring visualizations, mistakes will be made or the tree will grow too large. Thus, we provide pruning operations that hide the extraneous nodes (no provenance information is actually deleted). Pruning the tree can currently be performed in two ways. The first, and simplest, pruning operation is to manually select the nodes to be removed and press the delete key. The second pruning operation is performed through a search and refine interface. A search will highlight only the nodes that match a query, whereas a refine will collapse the history tree to contain only the matching nodes. Both search and refine features use a keyword-based text input that is provided in the interface. Queries over modules, parameters, tags, annotations, users and dates can be performed by directly typing words or by using a simple query interface. For example, a version created by "stevec" in March that uses a "vtkRenderer" could be queried as *user:stevec date:Mar vtkRenderer*. These features are especially useful when different users share a vistrail in a database, and are similar to the use of queries as *views* into a relational database [112].

3.8.2 Scalable Derivation of Visualizations

The action-oriented model also leads to a very natural means to script workflow changes. The *parameter space* of a workflow d, denoted by $P(d)$, is the set of workflows that can be generated by changing the value of parameters of d. From $x_k, \cdots, x_i$, we can derive $P(d)$ by tracking *addModule* and *deleteModule* actions, and knowing $P(m)$, the parameter space of module m, for each module in the workflow. Each parameter can then be thought of as a *basis vector* for the parameter space of d.

Using the action-based model, instances for the basis vectors can be produced by a sequence of *setParameter* actions. Workflows spanning an n-dimensional subspace of $P(d)$ are generated as follows:

$$setParameter(id_n, value_n) \circ \cdots \circ setParameter(id_1, value_1) \circ d$$

The ability to apply bulk updates over a set of distinct workflows greatly simplifies the exploration of the parameter space for a given task and provides an effective means to create a large number of visualizations. Our bulk-update interface is accessed through a tab in the VisTrails GUI (see Figure 3.13). The parameters that have been set throughout the pipeline are displayed in the right column. All the parameters the user desires to explore are dragged into the main window. Our interface allows the user to select one of four dimensions for a parameter to be mapped to in the

spreadsheet: horizontal rows, vertical columns, multiple sheets, and animations through time in a single cell. Additionally, there are several options for defining the values to explore. The first is a simple interpolation that occurs between a start and end parameter and is most commonly used for simple integer or floating point values. The second is a user-specified list that can be used for any data type. This option is intended for exploring strings such as file names or labels. Finally, the most flexible option provides an interface for creating a Python function that defines the values. This option is intended for providing nonlinear sequences of values for exploration.

Mapping bulk updates to a spreadsheet is relatively straightforward when the pipeline produces a single visualization. However, this is not always the case, as shown by the different slice axes used in Figure 3.13. We provide an interface that allows the user to specify the spreadsheet layout of multiple visualizations during bulk updates. An annotated snapshot of the pipeline is displayed that shows a number on each module that appears multiple times in the pipeline. For example, in the figure, several modules are repeated twice and are thus assigned a "1" or a "2" to distinguish them in the explorations. The layout of the outputs are performed using a virtual cell interface that allows annotated cells to be stacked horizontally and vertically (shown on the bottom right of the figure).

Note that since parameter value modifications and workflow modifications are captured uniformly by the action-based provenance, a similar mechanism can be used to explore spaces of different workflow definitions. For example, if a user needs to compare visualizations generated by different algorithms, she can create a series of visualizations using the first algorithm, and then apply a bulk update that creates new workflows by replacing the occurrences of the first algorithm with the second and keeping all the other modules and parameter settings intact.

The resulting functionality is similar to Jankun-Kelly and Ma's visualization spreadsheet [58], with the added generality of executing arbitrary VisTrails workflows, and full provenance tracking: a version of the visualization found via one such parameter study can be stored back in the version tree, allowing all other provenance techniques to work seamlessly. In addition, since VisTrails identifies and avoids redundant operations (as described in detail in Chapter 6), workflows generated from parameter explorations can be executed efficiently—the operations that are common to the set of affected workflows need only be executed once.

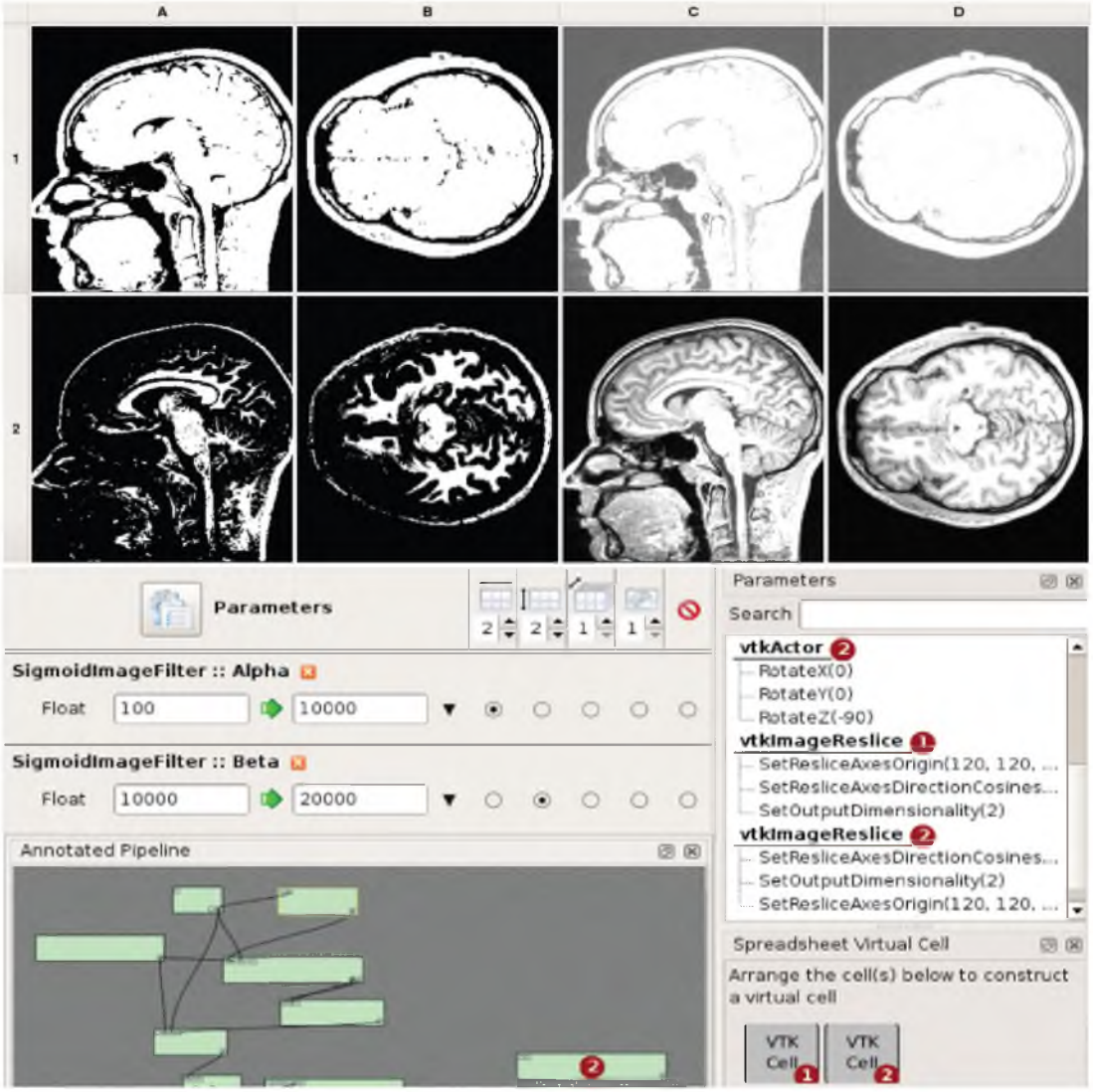


Figure 3.13: Visualization Spreadsheet. Parameters for a vistrail loaded in a spreadsheet cell can be interactively modified by clicking on the cell. Cameras as well as other vistrail parameters for different cells can be synchronized. This spreadsheet contains visualizations of MRI data of a head and was generated procedurally with the parameter exploration interface shown on the right. The horizontal axis varies one parameter of an image filter and the vertical axis explores another. The interface allows the user to manage the layout of cells when multiple visualizations are produced by one pipeline, as shown above with the two views (labeled 1 and 2).

CHAPTER 4

CREATING VISUALIZATIONS BY ANALOGY

The *visualization by analogy* component provides a mechanism for reusing these pipelines to construct new visualizations in a semi-automated manner—without requiring users to directly manipulate or edit the dataflow specifications. As Figure 4.1 illustrates, this technique works by determining the difference between a source pair of analogous visualizations, and transferring this difference to a third visualization. This forms the basis for scalable updates: the user does not need to have knowledge of the exact details of the three visualization dataflows to perform the operation.

In Chapter 3, we defined a provenance model that uniformly captures changes to pipeline and parameter values during the course of data exploration. This detailed history information, combined with a multi-view visualization interface, simplifies the exploration process. It allows users to navigate through a large number of visualizations, giving them the ability to return to previous versions of a visualization, compare different pipelines and their results, and resume explorations where they left off. Of particular importance to this chapter, we showed how the process provenance can be promptly used to compute differences between two visualization pipelines in the version tree.

Here, we show how this provenance information can also be used to simplify and partially automate the construction of new visualizations. Constructing insightful visualizations is a process that requires expertise in both visualization techniques and the domain of the data being explored. We propose a framework that enables the effective reuse of this knowledge to aid both expert and nonexpert users in performing data exploration through visualization. When creating visualizations, users often have to integrate new features into existing pipelines. For example, a user may wish to improve a given visualization by adjusting parameter so they match a published result. The user might also simply want to switch to a different visualization algorithm. In either case, there usually exists an example that demonstrates the given technique. A user can infer the necessary changes, and then apply them to a particular visualization. This analogical reasoning is very powerful, and we show that *visualization by analogy* can be (partly) automated.

Two ordered pairs are *analogous* if the relationship between the first pair mirrors the relationship between the second pair. Therefore, if we know what the relationship is between the first pair, and

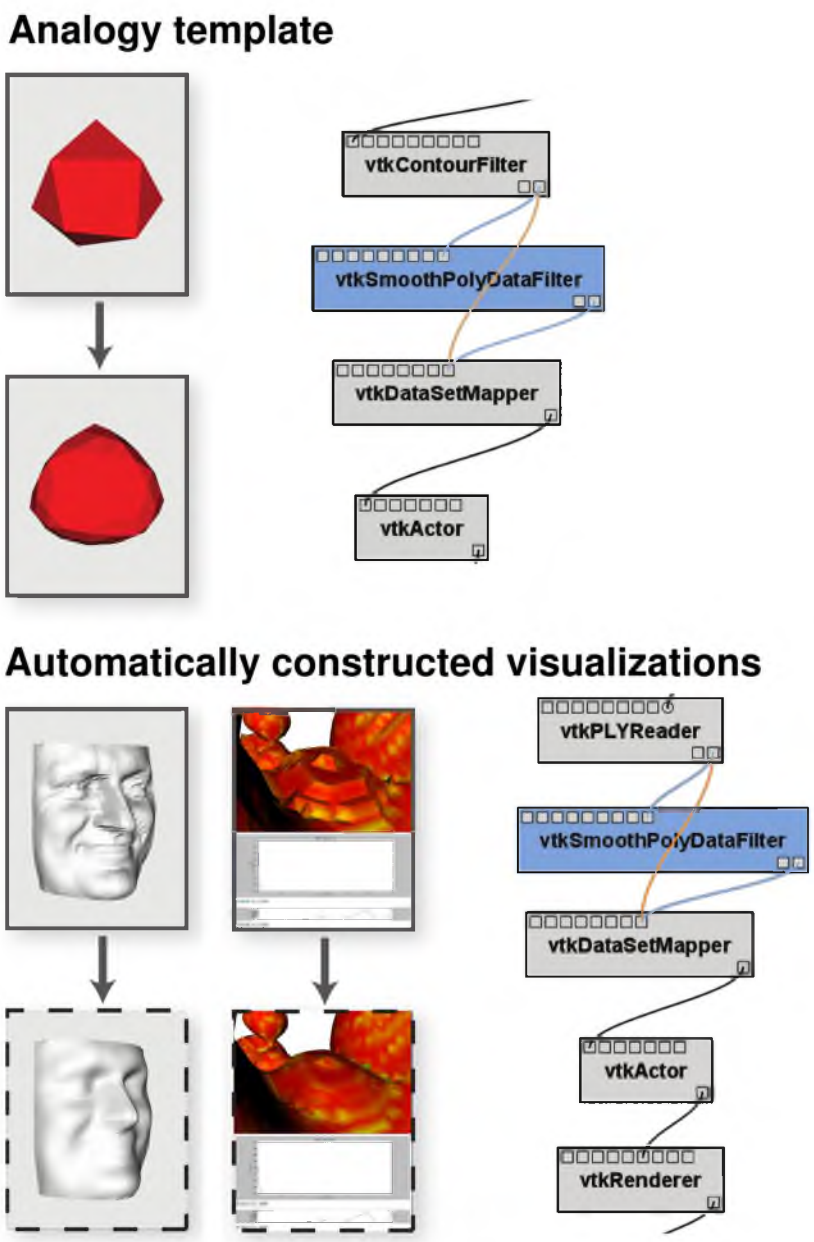


Figure 4.1: Visualization by analogy. The user chooses a pair of visualizations to serve as an analogy template. In this case, the pair represents a change where a file downloaded from the WWW is smoothed. Then, the user chooses a set of other visualizations that will be used to derive new visualizations, with the same change. These new visualizations are derived automatically. The pipeline on the left reflects the original changes, and the one on the right reflects the changes when translated to the last visualization on the right. The pipeline pieces to be removed are portrayed in orange, and the ones to be added, in blue. Note that the surrounding modules do not match exactly: the system figures out the most likely match.

are given the first entity of the second pair, we should be able to determine the other entity of that pair. More concretely, given a difference between δ two pipelines, we should be able to modify an arbitrary pipeline so that the resulting changes mirror δ.

4.1 Updating Pipelines

Finding differences is not only a useful technique for analyzing pipelines, but it can also be used to create new visualizations. As a reminder, in Chapter 3, we defined δ to be a function that changes one pipeline into another, typically created as the encoding of the difference between two pipelines.

Our ultimate goal is to apply the pipeline difference result δ to pipelines other than those used to create it. To analyze where δ is applicable, we introduce the domain and range context of δ. Formally, the *domain context* of δ, $D(\delta)$, is the set of all pipeline primitives required to exist for δ to be applicable. We represent these contexts as sets of identifiers. For example, if δ is a function that changes the file name parameter of a module with id 32, $D(\delta)$ is the set containing the module with id 32. Similarly, the *range context* of δ, $R(\delta)$, is the set of all pipeline primitives that were added or modified by δ. Note that $D(\delta^{-1}) = R(\delta)$, which provides an easy way to compute range contexts. In other words, these contexts encapsulate the partiality information of the functions that comprise δ.

We will use the δ objects similarly to applying patches in source control systems: the difference results can be applied to modify an existing pipeline. Given a δ, it is straightforward to apply it to a pipeline. Recall that δ is a sequence of actions that transform a pipeline. Thus, updating a pipeline p_a is as simple as computing $\delta(p_a)$. Note, however, that the operation can *fail* if an element of $D(\delta)$ does not exist in p_a. If we denote the set of identifiers in p_a as $\mathrm{Id}(p_a)$, then $\delta(p_a)$ will fail if $D(\delta) - \mathrm{Id}(p_a) \neq \emptyset$. We will now present a technique that changes δ to try and ensure that the operation succeeds.

It is important to notice here that previously published techniques for manipulating the user's history can be seen as particular ways of changing δ. For example, Berlage's selective undo mechanism [12] allows users to manually remove a certain set of actions from the history before applying it in a different context, which makes it possible for users to remove the violating elements of $D(\delta)$. Kurlander and Feiner's macro-by-example mechanism also allows users to explicit change the domain of the actions in consideration [67]. In our case, the objects under consideration, visualization pipelines, are conceptually much simpler than a full-fledged vector drawing system. As we will discuss in Section 4.5, it is this simplicity that allows us to effectively use an automatic method for pipeline matching. This matching operation is what we turn to now.

4.2 Matching Pipelines

While computing pipeline differences is an integral part in reasoning about multiple visualizations, another important operation is to match similar pipelines, i.e., we wish to find correspondences between pipelines. The result of pipeline matching can either be a binary decision (whether the pipelines match) or a mapping between the two inputs. Note that different metrics and thresholds can be used to determine the similarity of two pipelines. In the remainder of this section, we discuss an approach for finding a good mapping between two pipelines.

Let D represent the set of all domain contexts and define $\mathsf{map} : \mathbb{V} \times \mathbb{V} \to (D \to D)$ as a function which takes two pipelines, p_a and p_b, as input and produces a (partial) map from the domain context of p_a to the domain context of p_c. The map may be partial in cases where elements of p_a do not have a match in p_b or vice versa. Notice that if $p_a < p_b$, $\mathsf{map}(p_a, p_b) = \mathsf{map}_{ab}$ is the identity on all elements that were not added or deleted in the process of deriving p_b.

To construct such a mapping, we formulate the problem as a weighted graph matching problem. Let $G_a = (V_a, E_a)$ be the graph corresponding to the pipeline p_a. In a straightforward definition, V_a would be the modules in p_a and E_a the connections in p_a. However, one could consider other definitions such as the dual of this representation. For V_a, we define a scoring function $s : V_a \times V_b \to [0.0, 1.0]$ that defines the compatibility between vertices. For example, the similarity score of two modules that are exactly the same can be set to 1.0 and the score of modules M_1 and M_2 such that M_1 is a subclass of M_2 may be set to 0.6.

We define a matching between G_a and G_b as a set of pairs of vertices $M = \{(v_a, v_b)\}$ where $v_a \in V_a$ and $v_b \in V_b$. A matching is *good* when

$$\sum_{(v_a, v_b) \in M} s(v_a, v_b) \tag{4.1}$$

is maximized. A good matching on pipelines is one that corresponds to a good matching on their representative graphs. Given a good matching M, we can define a mapping from p_a to p_b as $v_a \to v_b$ for all $(v_a, v_b) \in M$.

To automate this operation, we need to compute the difference between two pipelines and apply this difference to another (possibly unrelated) pipeline. Suppose that we have three pipelines p_a, p_b, p_c, and wish to compute p_d so that $p_a : p_b$ as $p_c : p_d$. We discussed the problem of finding the difference in Section 3.3, but recall that updating a pipeline p_c with an arbitrary δ will fail if p_c does not contain the domain context of δ. When this is the case, we need to map the difference so that it can be applied to p_c.

We wish to express δ_{ab} so that $\delta_{ab}(p_c)$ succeeds. This is exactly what map_{ac} does; recall that to construct this operator, we need to find a match between p_a and p_c. More precisely, we first compute $\delta^*_{cb} = \mathsf{map}_{ac}(p_a, p_b)$ and then find $\delta^*_{cb}(p_c)$.

In summary, our algorithm is:

1. Compute the difference: $\delta_{ab} = \Delta(p_a, p_b)$
2. Compute the map: $\mathsf{map}_{ac} = \mathsf{map}(p_a, p_c)$.
3. Compute the mapped difference: $\delta^*_{cb} = \mathsf{map}_{ac}(\delta_{ab})$
4. Compute $p_d = \delta^*_{cb}(p_c)$

4.3 A Soft Graph Matching Algorithm

In our matching algorithm, we use the standard graph representation where vertices correspond to modules and edges to connections. In addition, even though we still discriminate between input and output ports, we do not enforce directionality on the edges so that we can diffuse similarity along them.

Recall that our goal in pipeline matching is to determine a *mapping from the context of one pipeline to another*. To do so, we convert the pipelines to labeled graphs and define a scoring function for nodes based on their labels. With a graph for each pipeline, we compute the mapping by pairing nodes that score well and enforcing connectivity constraints between these pairs.

Let G_a and G_b be the graphs corresponding to p_a and p_b. For our implementation, we define modules as vertices and connections as edges. Denote a connection between two vertices a and b as $a \sim b$ and define the scoring function that measures the pairwise compatibility of vertices by

$$c(v_a, v_b) = \frac{|\text{ports}(v_a) \cap \text{ports}(v_b)|}{|\text{ports}(v_a)| + |\text{ports}(v_b)|}$$

where $\text{ports}(v)$ denotes the ports of the module corresponding to the vertex v. This measure emphasizes port matching: it gives higher scores to modules that can be more easily substituted for each other. Such a substitution depends solely on the *compatibility* of the input and output ports and not on module name or functionality. Figure 4.2 shows an example of such an approximate matching.

Notice that this scoring function is defined only for nodes, and therefore, it does not help us in comparing the topologies of the pipelines. While a simple maximum bipartite matching [24] between nodes may succeed in finding a map between nodes, we would like to enforce some neighborhood relationship constraints on the graphs. Intuitively, we want to define the *similarity* between vertices as a weighted average between how *compatible* the *modules* are and how *similar* their *neighborhoods* are. The similarity score should strike a balance between the locality of pairwise compatibility and the overall similarity of the neighborhood. In a way, we will try to build a matching that is progressively less local, by diffusing "potential isomorphisms" that are easy

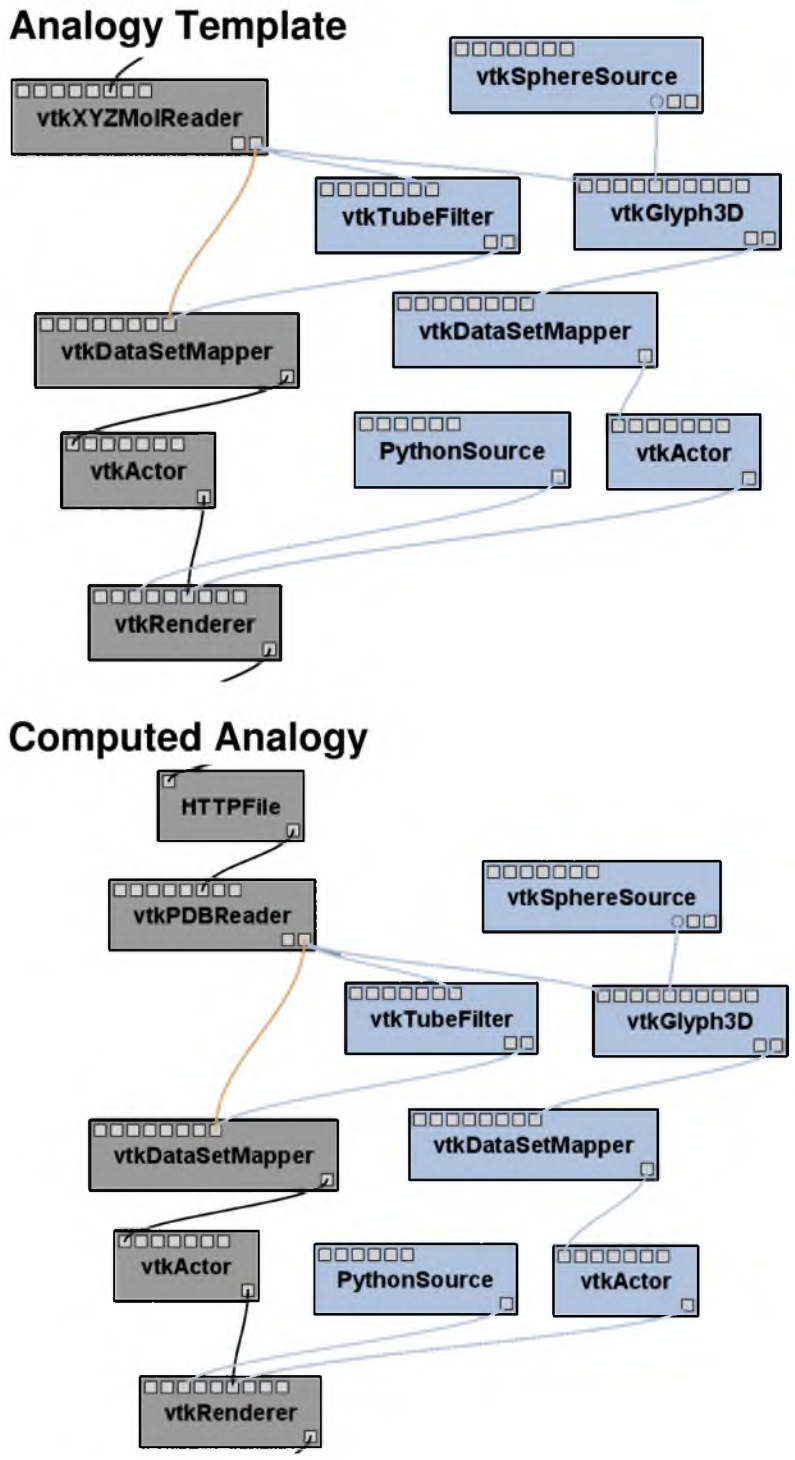


Figure 4.2: Example of an analogy between pipelines where there is no perfect module matching. The difference in the left pipeline pair is transferred to the right pipeline pair. Note, however, that the modules are not the same—the system must find the most likely pairing based on the similarity measure described in the text.

to compute. This definition seems circular, but it leads to a simple matching technique based on the dominant eigenvector of a Markov chain [69].

We create a graph $G = G_a \times G_b$ that combines both G_a and G_b. In this graph, we define a vertex $v_{a,b}$ for each pair of vertices $v_a \in V_a, v_b \in V_b$. Similarly, an edge $v_{i,j} \sim v_{k,\ell}$ exists when $v_i \sim v_k$ in G_a and $v_j \sim v_\ell$ in G_b. (G is the *graph categorical product* of G_a and G_b.) Notice that the connectivity of G encodes the pairwise neighborhoods of the vertices in G_a and G_b. We now want to translate our intuitive algorithm from the previous paragraph into an iterative algorithm. First, we need the following notation:

- $\pi_k(G)$ is the measure of pairwise similarity after k steps
- $A(G)$ is the adjacency matrix of G normalized so that the sum of each row is one (a row with sum zero is modified to be uniformly distributed)
- $c(G)$ is the normalized vector whose elements are the scores for the paired vertices in G: $c(G) = (c(v_a, v_b), v_a \in G_a, v_b \in G_b)$
- α is a user-defined parameter that determines the trade-off between pairwise scoring and connectivity

To iteratively refine our estimate, we *diffuse* the neighborhood similarity according to the following formula:

$$\pi_{k+1} = \alpha A(G)\pi_k + (1-\alpha)c(G) = M_G\pi_k. \tag{4.2}$$

The final pairwise similarity between modules is given by $\pi_\infty = \lim_{k\to\infty}\pi_k$. For our purposes, $c(G)$ gives a good measure of similarity so $A(G)$ is used mainly to break ties between two alternatives. Thus, we choose a small weight for the neighborhood in our implementation ($\alpha = 0.15$).

Though this formulation makes intuitive sense, we want to ensure that repeated iteration always converges and does so quickly. It is clear that M_G in Equation 4.2 is a linear operator; therefore, if π converges, it does so to an eigenvector. The theory of Markov chains tells us that because of the special structure of M_G, it has spectrum $(1, \alpha, \alpha^2, \cdots)$ [69], and so the iteration is exactly the power method [45] for eigenvalue calculation. Hence, the iteration will converge to the single dominant eigenvector, and each iteration will improve the estimate linearly by $1-\alpha$. Since we are using a small α, this ensures quick convergence. From this iteration, we obtain π_∞ which contains the relative probabilities of $v_a \in G_a$ and $v_b \in G_b$ matching for each possible pair. For each vertex in v_a, the vertex in v_b whose pair has the maximum value in π_∞ will be considered the match. Figure 4.3 illustrates how the matchings are refined as the mapping algorithm iterates.

There are two steps involved in applying an analogy to a pipeline. First, the user defines the analogy template by selecting the two pipelines whose difference is to be applied to another pipeline.

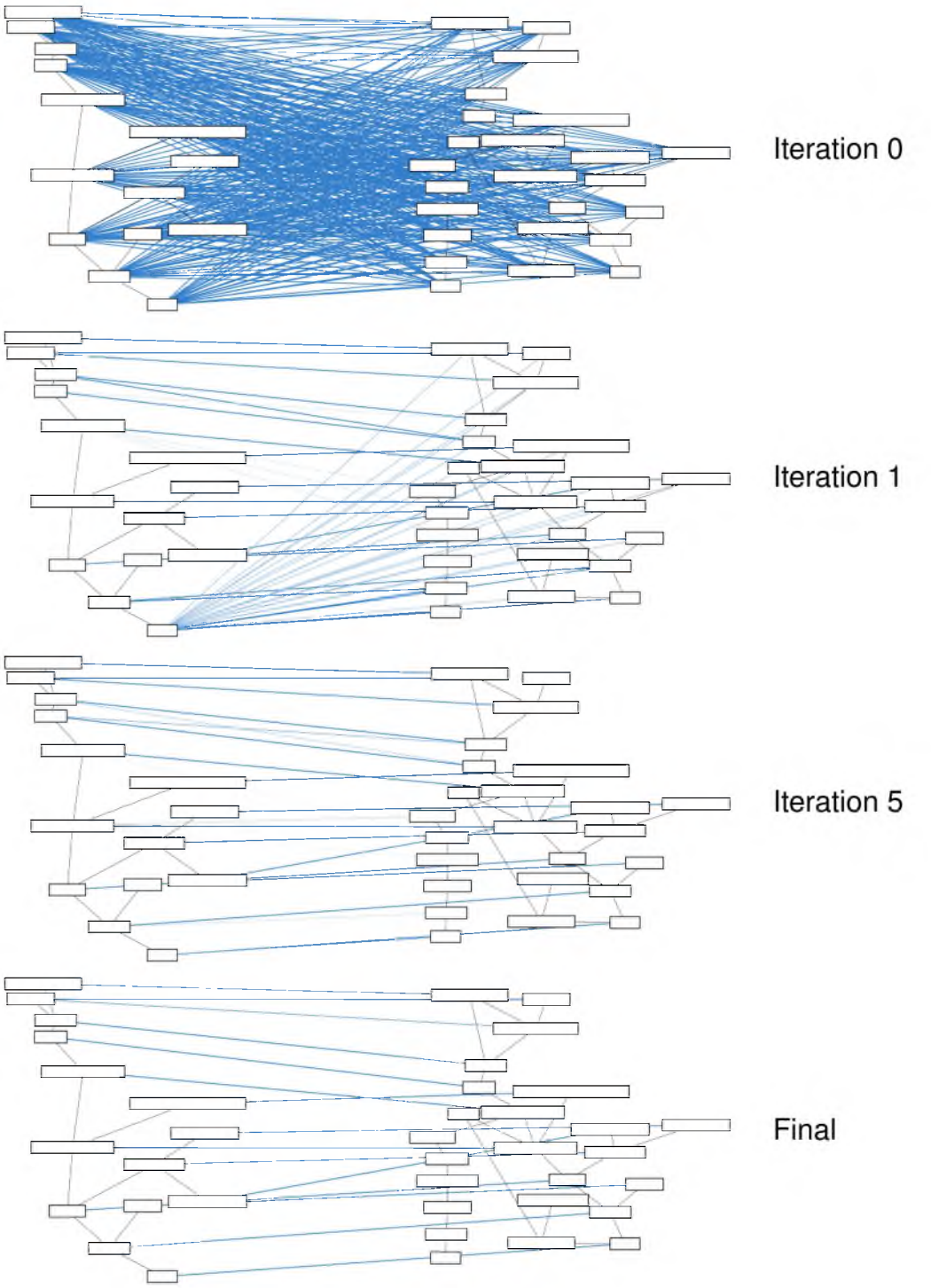


Figure 4.3: Example matching generated by the pipeline matching algorithm. Thicker edges correspond to stronger correspondences. Notice that the correspondences get progressively better as the algorithm iterates. This matching corresponds to Example 2 in Section 4.4.

Second, the user selects another pipeline and applies the analogy to that pipeline, creating a new pipeline. In VisTrails, these operations can be executed in either the version tree pane of the builder window or the visualization spreadsheet. In either case, the application of the analogy creates a new version in the vistrail.

In the version tree, an analogy is defined by dragging the version representing initial pipeline to the version representing the desired result. This operation displays the difference between the pipelines and the user is able to click a button to create an analogy from these pipelines. To apply the analogy, the user right-clicks on the version representing the pipeline and selects the desired analogy.

Creating and applying analogies in the VisTrails Spreadsheet is similar. The spreadsheet supports a viewing mode and a composition mode. In the composition mode, a user can create an analogy by dragging one cell into another cell. To apply the analogy, the user drags the pipeline to be modified to a new cell, at which point the analogy is applied and the new visualization displayed.

The computation of the analogy in this case is the same as the one described previously. More concretely, for pipelines p_a and p_b defining the analogy and the pipeline to be updated p_c, we derive δ_{ab} using the version tree. We then match G_a and G_c using the algorithm described above to obtain map_{ac} and use this function to compute δ^*_{cb} which can then be applied to p_c to produce a new pipeline p_d.

4.4 Case Studies

We present three examples that illustrate the proposed primitives.

4.4.1 Updating Inputs in Multiple Pipelines

In this scenario, we want to compare different isosurface extraction techniques. In particular, we wish to investigate how resilient the techniques are to subsampled or oversampled data. Typically, the techniques are first compared using raw inputs. The task, then, is to update the pipelines with the new test data.

There are several ways to address this problem. The most straightforward one is to develop a preprocessing script that converts the files. Although this is feasible, it is not desirable since it puts the burden to manage the data on the user. At the least, it requires explicit management of intermediate files, and it does not provide an explicit record of the desired experiment. A better alternative is to directly create new dataflows that exercise the test regime. It is clear, however, that this can be time consuming if the specialist must first examine each pipeline to determine whether it needs to be updated and only then perform the required modifications.

It is therefore desirable to automate this process. With query-by-example, we can find all matching pipelines with one operation. With analogies, we can perform the desired update once, capture that change as an analogy and apply it to the matching pipelines. Not only does this save time and effort, but it ensures that all pipelines are updated. In addition, each update is done in a similar manner; the possibility that the updates are inconsistent is reduced.

In this example, we construct a query template by copying the relevant portion of the pipeline onto the Query Canvas. This procedure returns a set of pipelines which we need to update. We first update one of the pipelines by directly adding the resampling step. Then, we define an analogy template using the original pipeline and the updated one. We apply this analogy to automatically update the other pipelines that match the query. As result, several new results are produced without requiring the user to manually update each individual pipeline.

4.4.2 Changing a Rendering Algorithm

In this example, we show a moderately complex change in a pipeline that replaces an entire rendering technique with another. When designing an effective visualization, one algorithm tends to perform better than the alternatives. It is natural, then, that a single visualization will be tried with different algorithms. When the best result is identified, the user must change the other visualizations to reflect this. In this example, we show that it is possible to replace an entire algorithm by analogy.

The visualization portrayed in Figure 4.4 renders an ITK [52] scalar field in VTK [110], using the Teem tools [61] to generate the appropriate data format. While in the original change, there was only one generated view, in the analogy target there are two renderings, so the system must correctly decide the proper one to modify.

4.4.3 Chaining Analogies

We have discussed that one can modify a pipeline by analogy as a single update operation. However, one can also use analogies to quickly combine multiple examples. In this example, illustrated in Figure 4.5, we show how three different techniques can be combined to transform a very simple pipeline into a visualization that is not only more complicated but also more useful.

In many scientific fields, the amount of data and the need for interaction between researchers across the world has led to the creation of online databases that store much of the domain information required. Scientists are concerned not only with using data from these centralized repositories but also publishing their own results for others to view. In this example, we show how analogies can be used to modify a simple pipeline that visualizes protein data stored in a local file to obtain data from an online database, create an enhanced visualization for that protein, and finally publish the results as an HTML report.

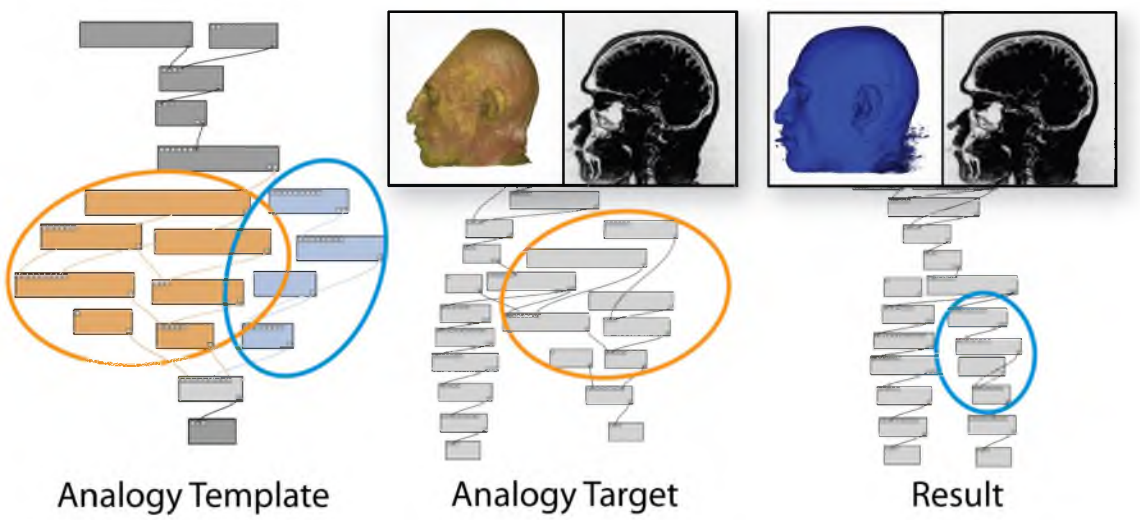


Figure 4.4: Switching the rendering technique by analogy. The analogy template on the left specifies that volume rendering modules should be replaced by isosurfacing ones. The analogy target and the resulting pipeline are shown, together with the resulting visualization.

We begin with a vistrail that contains pipelines that accomplish each of the individual tasks outlined above. Specifically, we have a simple pipeline p_0 that reads a file with protein data and generates a visualization of that data. We also have pipelines p_1 and p_1' where the difference between the two is that p_1 reads a local file and p_1' reads data from an online database, pipelines p_2 and p_2' where p_2 features a simple line-based rendering and p_2' improves the rendering to use a ball-and-stick model. Finally, p_3 displays a visualization while p_3' generates an HTML report that contains the visualized image.

To create the new pipeline, we compute the analogy between p_1 and p_1' and apply it to p_0. Then, we compute the analogy between p_2 and p_2' and apply that the result of the previous step. Finally, we compute the analogy between p_3 and p_3' and apply it. The new pipeline p_0^* prompts the user for a protein name, uses that information to download the data for that protein, creates a ball-and-stick visualization of the data, and embeds that image in an HTML report.

The benefits of using analogies to generate this new pipeline not only include faster results but also a lower level of knowledge needed to modify pipelines. One can imagine a scientist who executes a pipeline to create a visualization downloading a pipeline which publishes data to the web and adding the same capability to their pipeline via analogy. Instead of trying to find the correct modules and manually modifying the pipeline, the scientist can use the analogy from the example pipeline to add the new feature automatically.

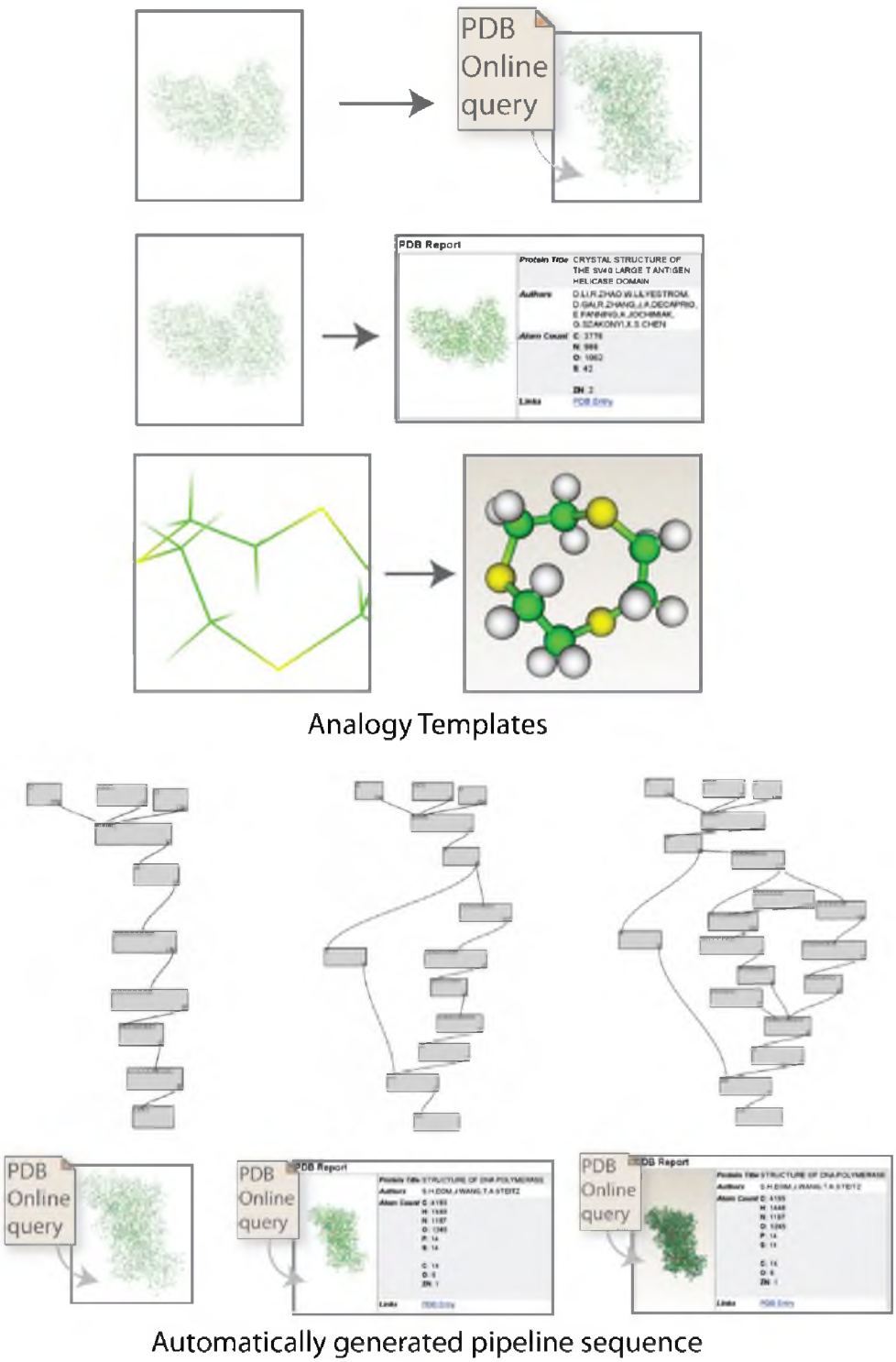


Figure 4.5: Creating complex pipelines by chaining simple analogies. From three simple examples, the user creates a complex visualization that creates a web page with enhanced molecule rendering, whose results are fetched from the Protein Database, an online macromolecular database.

4.5 Discussion

We argue that visualization by analogy is a useful operation that efficiently solves what are otherwise manual, time-consuming tasks. The basic operations introduced in Section 4.1 rely both on the graph structure of pipelines and on pipeline modification history. As discussed, global comparisons of graphs are intractable in general, but the fact that visualization pipelines translate to labeled graphs where the nodes are largely distinct allows us to define effective heuristics. We believe that this framework can be used to develop additional primitives that significantly reduce the amount of work required to maintain and integrate ensembles of visualizations.

The proposed primitives can be easily implemented in dataflow-based visualization systems that provide undo/redo capabilities. As long as undo/redo operations are represented explicitly in the system (for example, using the Command design pattern [41]), a straightforward serialization of these would achieve the wanted capabilities. Module and connection representations may vary across systems, but the framework and techniques apply as long as the elements can be translated to labeled graphs.

As with most heuristics-based approaches, our approach to matching is not foolproof, and there are cases where it may fail to produce the results a user expects. For example, if a user applies an analogy to a pipeline that shares little or no similarity with the starting pipeline, the matching algorithm will return a mapping which is likely to be meaningless. However, when application of an analogy fails or produce poor results, the user can either discard or refine the resulting pipeline: analogies always construct *new* pipelines—they do not modify existing pipelines.

Analogies can be highly subjective. In some cases, applying an analogy can lead to ambiguity and derive bad results. Figure 4.6 shows an example of an analogy that is supposed to resample an input file before continuing with the rest of the pipeline. Instead of removing a connection from the raw file to downstream modules, the application keeps the old connection in addition to adding the new connection to the resampling module. In this case, a user might have to "clean up" the results of the pipeline. Our current pairwise similarity score tries to establish a compromise in the absence of domain-specific knowledge about modules. Formulating and incorporating such knowledge into the matching is certainly possible and desirable. An interesting avenue for future work is to investigate how to acquire this information in an unobtrusive way, for example, by taking user feedback about derived analogies into account. Furthermore, our current implementation finds the best mapping solving a maximum-weight assignment problem. There are alternative ways of using π_∞, and this investigation is part of future work.

One important consideration when introducing new manipulation primitives is the impact on how users interact with them. Also, to extend our analogy tool, users' input could be used to guide

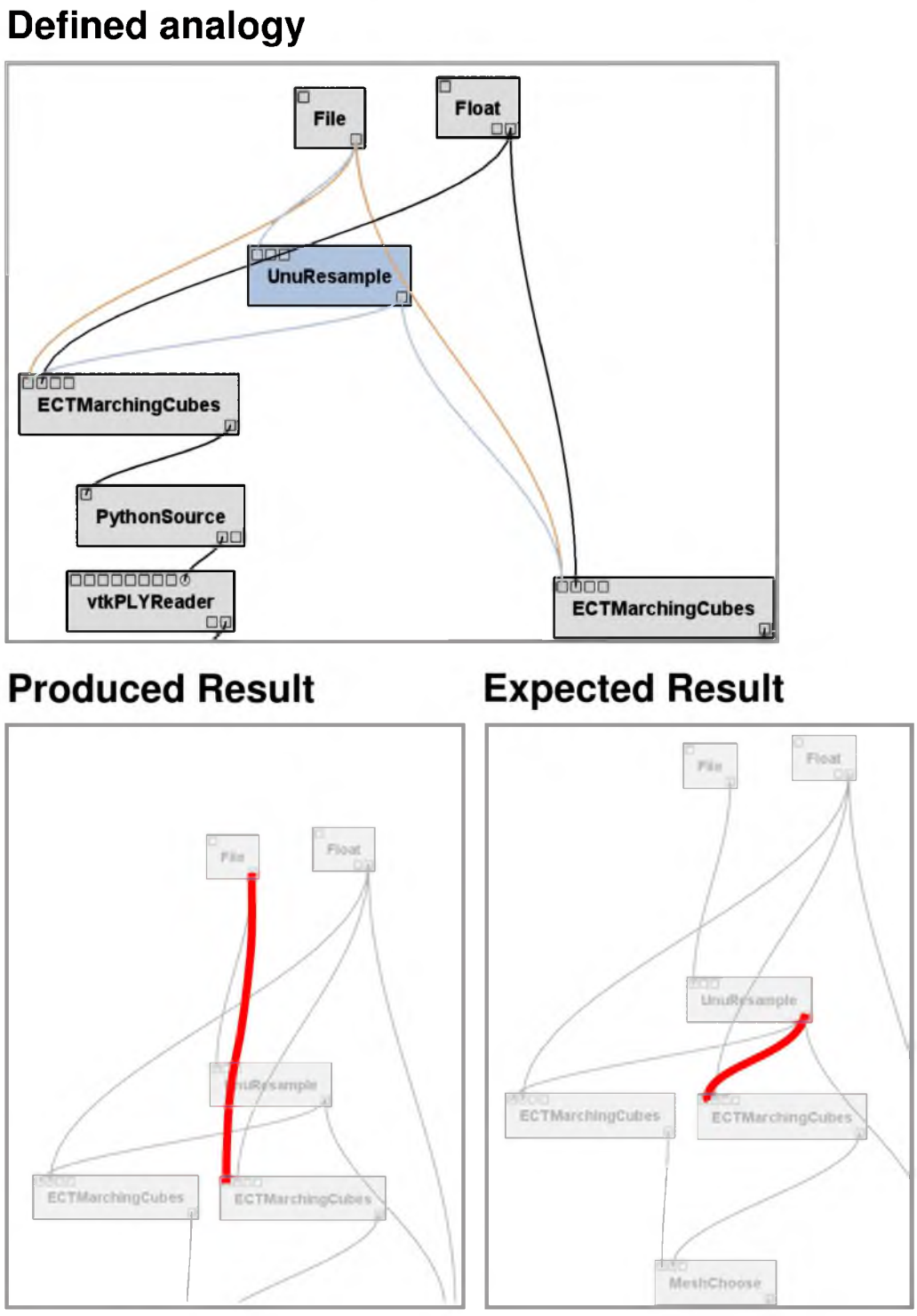


Figure 4.6: A situation where creating pipelines by analogy fails. The intended effect when defining the analogy was to replace the raw file with a preprocessing step. Note, however, that there still is one lingering connection, highlighted in red.

the matching process, especially in cases where the automatic construction fails. Constraint information might be incorporated into the matching, allowing it to generate better results in situations where the information in the pipeline definitions is not sufficient. Along the same lines, it may be useful to allow users to explore the results of many possible matchings.

4.5.1 Matching More Complicated Objects

The technique described above computes a remapping between actions in different contexts. In our case, the context is given by the pipelines. On first inspection, one could think that a generalization would work for other types of data and actions, such as changing pieces of a program in a text editor. Here, we present some of the particularities of the current setup that clarify why such generalizations could fail.

One important aspect of the current algorithm that is not shared by many other editing tools is access to semantic information about the modules being manipulated. For example, it uses the set of available input and output ports (which is the declared interface of the module) to decide the matching. Modern IDEs such as Eclipse also include knowledge of target language semantics, and notably provide *refactoring tools* [35]. A critical difference between automatic refactoring and a generalization of our proposal is that refactoring does not change the semantics of the code, and so requires no external examples.

VisTrails workflow representations are "all semantics," in a sense: the workflow graph encodes the semantics of the execution. In addition, if we assume that the signature generated by the cache manager as described in Chapter 6 encodes the behavior of the module (aside from system-level side effects such as the contents of files), then every different workflow encodes a different behavior. This denseness of representation is not present in other languages, and for good reason: it is undecidable to check whether Turing-complete languages produce the same behavior, by a trivial application of Rice's Theorem [102]. Even in domain-specific languages that are not Turing-complete, doing matching on abstract syntax trees is bound to be problematic. It might be possible to create meaningful analogy changes in limited cases for dataflow-like language such as Sh [83]. Even in that case, if the construction of the pipelines is done programmatically, this is not likely to work.

CHAPTER 5

RECOMMENDING PIPELINE FRAGMENTS

User collaboration and social data reuse has proven to be a powerful mechanism in various domains, such as recommendation systems in commercial settings (e.g., Amazon, e-Bay, Netflix), knowledge sharing on open Web sites (e.g., Wikipedia), image labeling for computer vision (e.g., ESPGame [126]) and visualization creation (e.g., ManyEyes [124]). The underlying theme shared by these systems is that they use information provided by many users to solve problems that would be difficult otherwise. We apply a similar concept to pipeline creation: pipelines created by many users enable the creation of visualizations *by consensus*. For the user, VisComplete acts as an autocomplete mechanism for pipelines, suggesting modules and connections in a manner similar to a Web browser suggesting URLs. The completions are presented graphically in a way that allows the user to easily explore and accept suggestions or disregard them and continue working as they were. Figure 5.1 shows an example of VisComplete incorporated into the VisTrails Builder interface and Figure 5.2 shows some example completions for a single module.

We propose a recommendation system that leverages information in a collection of pipelines to provide advice to users of visualization systems and aid them in the construction of pipelines. We develop a technique that uses a pipeline's representation as a graph for predicting likely completions. In particular, this algorithm searches for common subgraphs in the collection. We also present an interface that displays the recommended completions in an intuitive way. Our preliminary experiments show that VisComplete has the potential to reduce the effort and time required to construct visualizations. We found that the suggestions derived by VisComplete could have reduced the number of operations performed by users to construct pipelines by an average of over 50%. Note that although in this paper we focus on the use of VisComplete for visualization pipelines, the techniques we present can be applied to general workflow systems.

5.1 Generating Data-driven Suggestions

VisComplete suggests partial completions (i.e., a set of structural changes) for pipelines as they are being created by a user. These suggestions are derived using structural information obtained

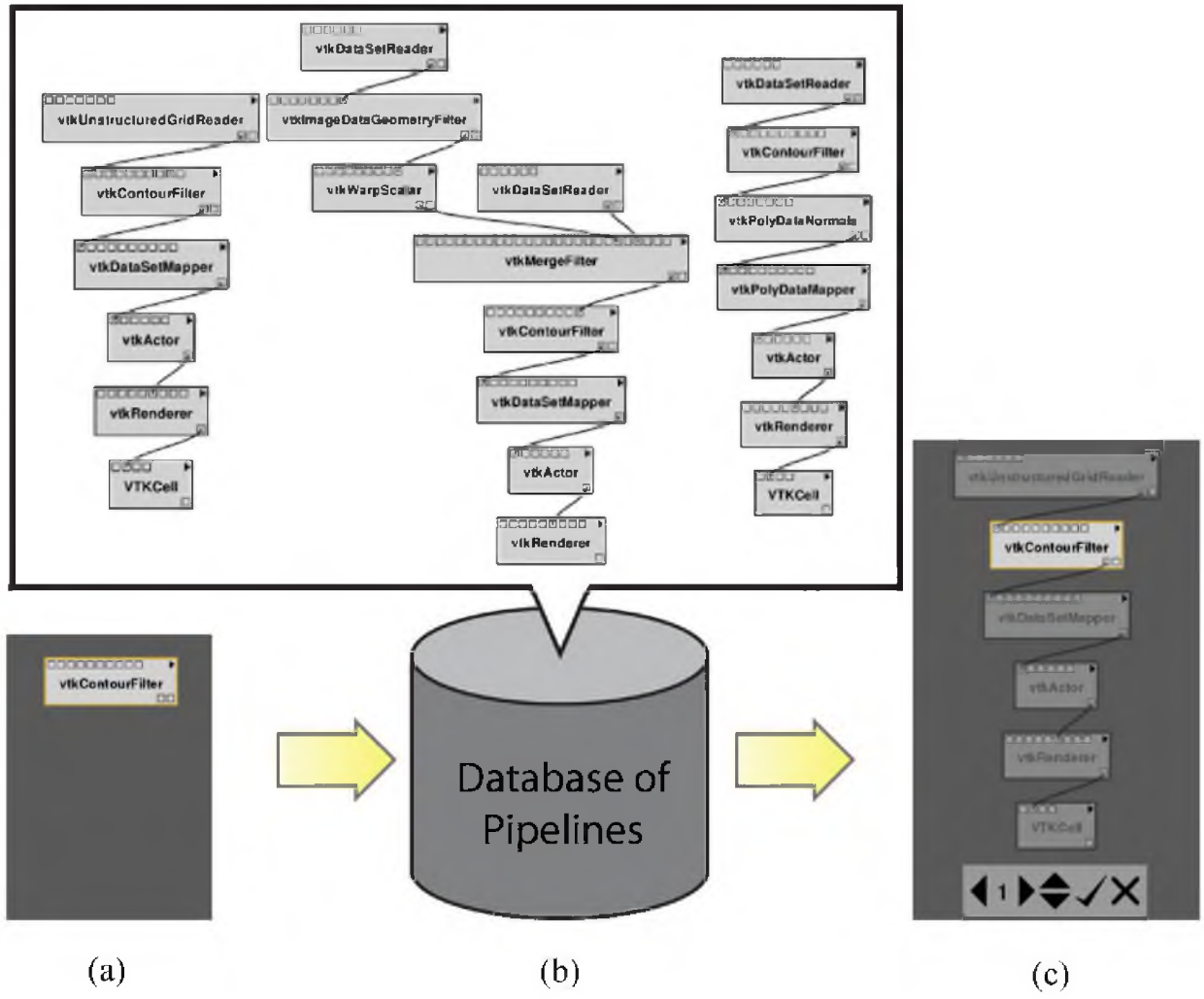


Figure 5.1: The VisComplete suggestion system and interface. (a) A user starts by adding a module to the pipeline. (b) The most likely completions are generated using indexed paths computed from a database of pipelines. (c) A suggested completion is presented to the user as semitransparent modules and connections. The user can browse through suggestions using the interface and choose to accept or reject the completion.

Figure 5.2: Three of the first four suggested completions for a "vtkDataSetReader" are shown along with corresponding visualizations. The visualizations were created using these completions for a time step of the Tokamak Reactor dataset that was not used in the training data.

from a collection $\mathscr{G}$ of already-completed pipelines.

Pipelines are specified as graphs, where nodes represent modules (or processes) and edges determine how data flows through the modules.

As described in Chapter 3, a pipeline is a directed acyclic graph $G(M,C)$, where M consists of a set of modules and C is a set of connections between modules in M. A *module* is an object which contains a set of input and output ports through which data flows in and out of the module. A *connection* between two modules m_a and m_b connects an output port of m_a to an input port of m_b. In this chapter, we will ignore the methods that change a module's state. In addition, we will mostly disconsider that a connection between two modules might be realized by different sets of input and output ports.

The problem of deriving pipeline completions can be defined as follows. Given a partial graph G, we wish to find a set of completions $C(G)$ that reflect the structures that exist in a collection of completed graphs. A *completion* of G, G^c, is a supergraph of G: the vertices and edges of G are subsets of the ones in G^c.

Our solution to this problem consists of two main steps. First, we preprocess the collection of pipelines $\mathscr{G}$ and create $\mathscr{G}_{path}$, a compact representation of $\mathscr{G}$ that summarizes relationships between common structures (i.e., sequences of modules) in the collection (Section 5.1.1). Given a partial pipeline p, completions are generated by querying $\mathscr{G}_{path}$ to identify modules and connections that have been used in conjunction with p in the collection (Section 5.1.2).

5.1.1 Mining Pipelines

To derive completions, we need to identify graph fragments that co-occur in the collection of pipelines $\mathscr{G}$. Intuitively, if a certain fragment always appears connected to a second fragment in our collection, we ought to predict one of those fragments when we see the other.

Because we are dealing with directed acyclic graphs, we can identify potential completions for a vertex v in a pipeline by associating subgraphs downstream from v with those that are upstream. A subgraph S is *downstream* (*upstream*) of a vertex v if for every $v' \in S$, there exists a path from v to v' (v' to v). In many cases where we wish to complete a graph, we will know either the downstream or upstream structure and wish to complete the opposite direction. Note that this problem is symmetric: we can change one problem to the other by simply reversing the direction of the edges.

However, due to the (very) large number of possible subgraphs in $\mathscr{G}$, generating predictions based on subgraphs can be prohibitively expensive. Thus, instead of subgraphs, we use paths, i.e., linear sequences of connected modules. Specifically, we compute the frequencies for each path in $\mathscr{G}$. Completions are then determined by finding which path extensions are likely given the existing paths.

To efficiently derive completions from a collection of pipelines $\mathscr{G}$, we begin by generating a summary of all paths contained in the pipelines. Because completions are derived for a specific vertex v in a partial pipeline (we call this vertex the *completion anchor*), we extract all possible paths that end or begin with v and associate them with the vertices that are directly connected downstream or upstream of v. Note that this leads to many fewer entries than the alternative of extracting all possible subgraph pairs. And as we discuss in Section 5.4, paths are effective and lead to good predictions.

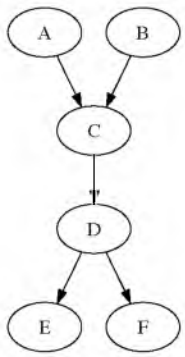
More concretely, we extract all possible paths of length N, and split them into a path of length $N-1$ and a single vertex. Note that we do this in *both* forward and reverse directions with respect to the directed edges. This allows us to offer completions for pipeline pieces when they are built top-down and bottom-up. The path summary $\mathscr{G}_{path}$ is stored as a set of (path, vertex) pairs sorted by the number of occurrences in the database and indexed by the last vertex of the path (the anchor). Since predictions begin at the anchor vertex, indexing the path summary by this vertex leads to faster access to the predictions.

As an example of the path summary generation, consider the graph shown in Figure 5.3. We have the following upstream paths ending with D: $A \rightarrow C \rightarrow D$, $B \rightarrow C \rightarrow D$, $C \rightarrow D$, and D. In addition, we also have the following downstream vertices: E and F. The set of correlations between the upstream paths and downstream vertices is shown in Figure 5.3. As we compute these correlations for all starting vertices over all graphs, some paths will have higher frequencies than others. The frequency (or support) for the paths is used for ranking purposes: predictions derived from paths with higher frequency are ranked higher.

Besides paths, we also extract additional information that aid in the construction of completions. Because we wish to predict full pipeline structures, not just paths, we compute statistics for the in- and out-degrees of each vertex type. This information is important in determining where to extend a completion at each iteration (see Figure 5.4). We also extract the frequency of connection types for each pair of modules. Since two modules can be connected through different pairs of ports, this information allows us to predict the most frequent connection type.

5.1.2 Generating Predictions

Predicting a completion given the path summary and an anchor module v is simple: given the set of paths associated with v, we identify the vertices that are most likely to follow these paths. As shown in the following algorithm, we iteratively develop our list of predictions by adding new vertices using this criteria.



path	vertex
$A \rightarrow C \rightarrow D$	E
$A \rightarrow C \rightarrow D$	F
$B \rightarrow C \rightarrow D$	E
$B \rightarrow C \rightarrow D$	F
$C \rightarrow D$	E
$C \rightarrow D$	F
D	E
D	F

Figure 5.3: Deriving a path summary for the vertex D.

GENERATE-PREDICTIONS(P)

predictions $\leftarrow$ FIRST-PREDICTION(P)
result $\leftarrow$ [ ]
while |*predictions*| > 0
  do *prediction* $\leftarrow$ REMOVE-FIRST(*predictions*)
    new-predictions $\leftarrow$ REFINE(*prediction*)
    if |*new-predictions*| $= 0$
      then *result* $\leftarrow$ *result* + *prediction*
      else *predictions* $\leftarrow$ *predictions* + *new-predictions*

At each step, we refine existing predictions by generating new predictions that add a new vertex based on the path summary information. Note that because there can be more than one possible new vertex, we may add more than one new prediction for each existing prediction. Figure 5.4 illustrates two steps in the prediction process.

To initialize the list of predictions, we use the specified anchor modules (provided as input). At this point, each prediction is simply a base prediction that describes the anchor modules and possibly how they connect to the pipeline. After initialization, we iteratively refine the list of predictions by adding to each suggestion. Because there are a large number of predictions, we need some criteria to order them so that users can easily locate useful results. We introduce *confidence* to measure the goodness of the predictions.

Given the set of upstream (or downstream depending on which direction we are currently predicting) paths, the confidence of a single vertex $c(v)$ is the measure of how likely that vertex is, given the upstream paths. To compute the confidence of a single vertex, we need to take into account the information given by all upstream paths. For this reason, the values in $\mathscr{G}_{path}$ are not normalized; we use the exact counts. Then, as illustrated by Figure 5.5, we combine the counts from each path. This means we do not need any weighting based on the frequency of paths; the

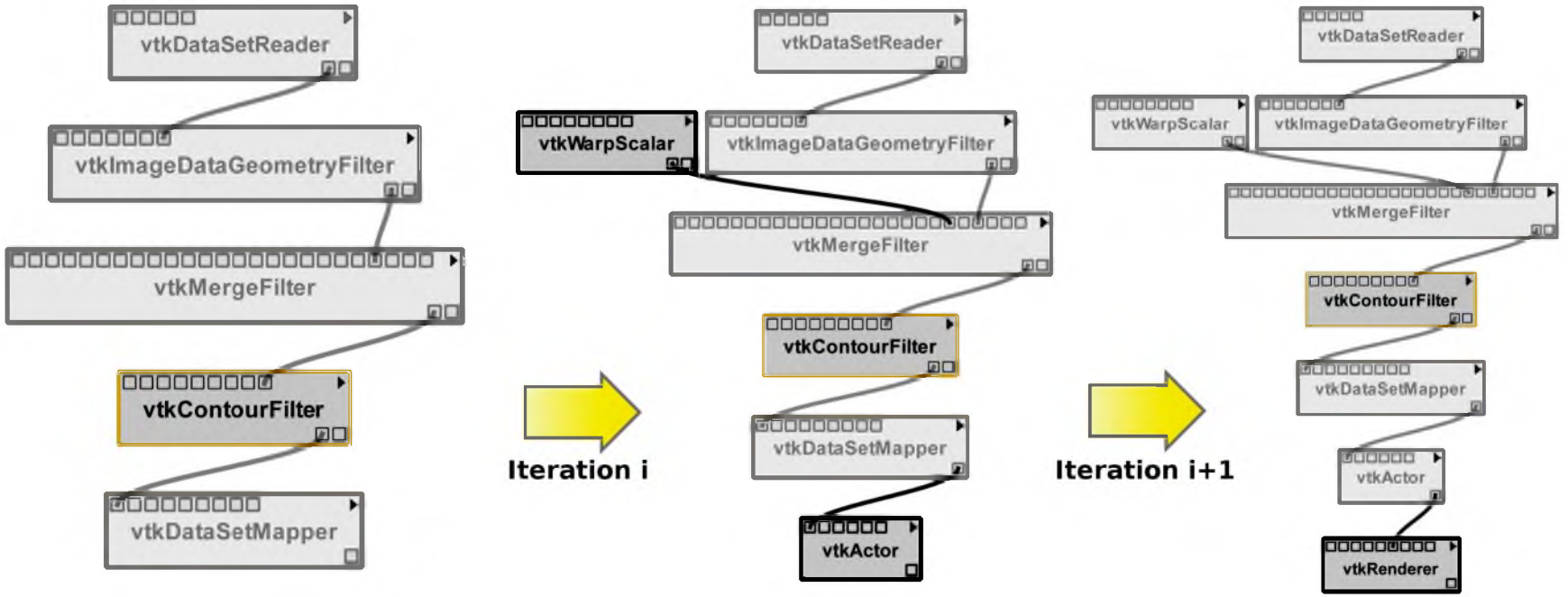


Figure 5.4: Predictions are iteratively refined. At each step, a prediction can be extended upstream and downstream; in the second step, the algorithm only suggests a downstream addition. Also, predictions in either direction may include branches in the pipeline, as shown in the center.

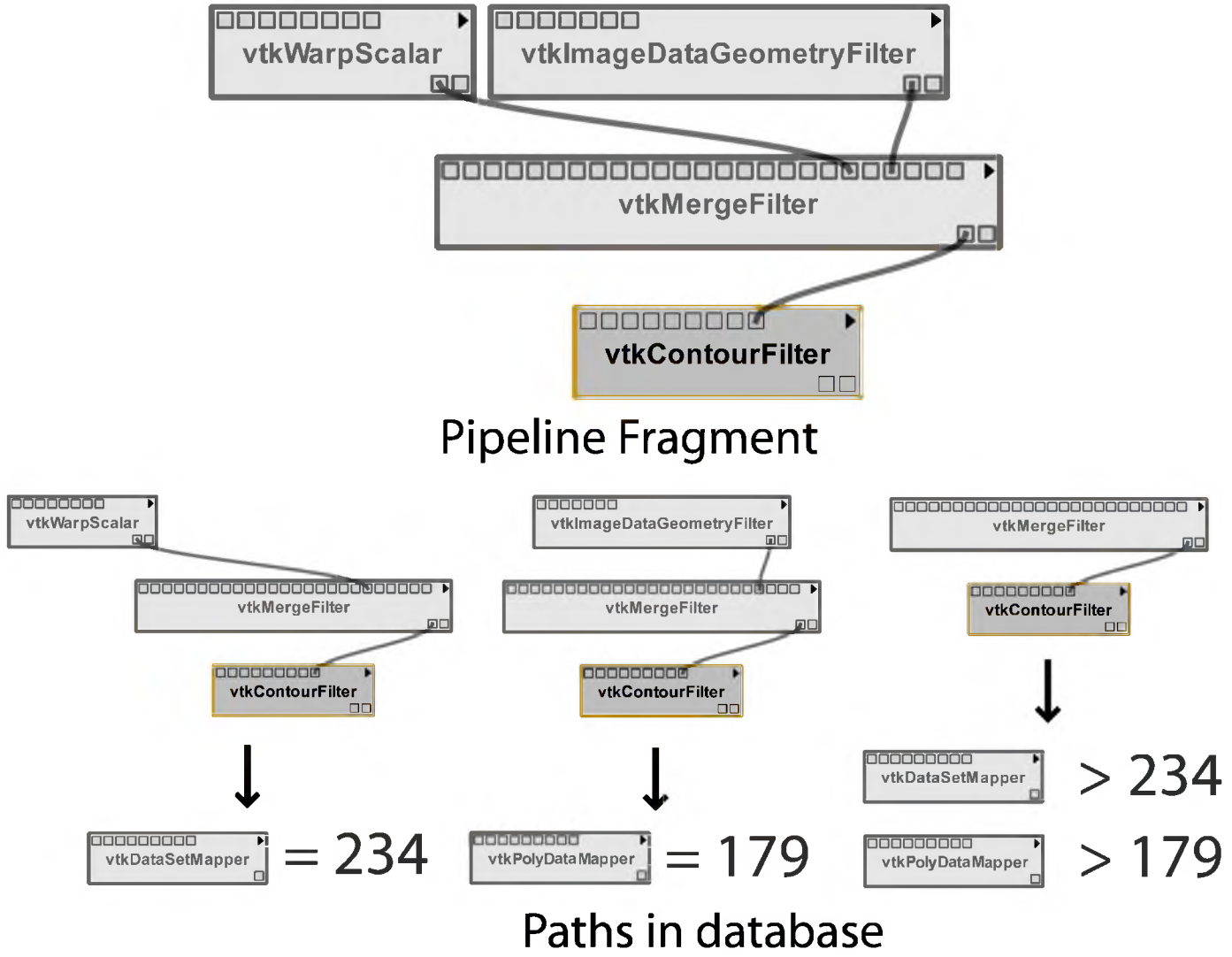


Figure 5.5: At each iteration, we examine all upstream paths to suggest a new downstream vertex. We select the vertex that has the largest frequency given all upstream paths. In this example, "vtkDataSetMapper" would be the selected addition.

formula takes this into account automatically. Specifically,

$$c(v) = \frac{\sum_{P \in upstream(v)} count(v \mid P)}{\sum_{P \in upstream(v)} count(P)} \tag{5.1}$$

Then, the confidence of a graph G is the product of the confidences of each of its vertices:

$$c(G) = \prod_{v \in G} c(v) \tag{5.2}$$

While each vertex confidence is not entirely independent, this measure gives a reasonable approximation for the total confidence of the graph. Because we perform our predictions iteratively, we calculate the confidence of the new prediction p_{i+1} as the product of the confidence of the old prediction p_i and the confidence of the new vertex v:

$$c(p_{i+1}) = c(p_i) \cdot c(v) \tag{5.3}$$

For computational stability, our implementation uses log-confidences so the products are actually sums.

Because we wish to derive predictions that are not just paths, our refinement step begins by identifying the vertex in the current prediction that we wish to extend our prediction from. Recall that we computed the average in- and out-degree for each vertex type in the mining step. Then, for each vertex, we can compute the difference between the average degree for its type and its current degree for the current prediction direction. We choose to extend completions at vertices where the current degree is much smaller than the average degree. We also incorporate this measure into our vertex confidence so that predictions that contain vertices with too many edges are ranked lower:

$$c_d(v) = c(v) + degree\text{-}difference(v) \tag{5.4}$$

We stop iteratively refining our predictions after a given number of steps or when no new predictions are generated. At this point, we sort all of the suggestions by confidence and return them. If we have too many suggestions, we can choose to prune our set of predictions at each step by eliminating those which fall below a certain threshold.

5.1.3 Biasing the Predictions

The prediction mechanism described above relies primarily on the frequency of paths to rank the predictions. There are, however, other factors that can be used to influence the ranking. For example, if a user has been working on volume rendering pipelines, completions that emphasize modules related to that technique could be ranked higher than those dealing with other techniques. In addition, some users will prefer certain completions over others because they more closely

mirror their own work or their own pipeline structures. Again, it makes sense to bias completions toward user preferences. We can adapt our algorithm to include such bias by incorporating a weighting factor in the confidence computation. Specifically, we adjust our counts by weighting the contribution of each path according to a pipeline importance factor determined by a user's preferences.

5.2 Implementation

Our implementation is split into three specific steps: determining when completion should be invoked, computing the set of possible completions, and presenting these suggestions to the user. Computing the possible completions requires the machinery developed in the previous section. The other steps are essential to make the approach usable. The interface, in particular, plays a significant role in allowing users to make use of suggestions while also being able to quickly dismiss them when they are not desired.

5.2.1 Triggering a Completion

We want to provide an environment where suggestions are offered automatically but do not interfere with a user's normal work patterns. There are two circumstances in pipeline creation where it makes sense to automatically trigger a completion: when a user adds a new module and when a user adds a new connection. In each of these cases, we are given new information about the pipeline structure that can be used to narrow down possible completions. Because users may also wish to invoke completion without modifying the pipeline, we also provide an explicit command to start the completion process.

In each of the triggering situations, we begin the suggestion process by identifying the modules that serve as anchors for the completions. For new connections, we use both of the newly connected modules, and for a user-requested completion, we use the selected module(s). However, when a user adds a new module, it is not connected to the rest of the existing pipeline. Thus, it can be difficult to offer meaningful suggestions since we have no surrounding structure to leverage. We address this issue by first finding the most probable connection to the existing pipeline, and then continue with the completion process.

Finding the initial connection for an added module may be difficult when there are multiple modules in the existing pipeline than can be connected to the new module. However, because visual programming interfaces allow users to drag and place new modules in the pipeline, we can use the initial position of the module to help infer a likely connection. To accomplish this, we compute the user's layout direction based on the existing pipeline, and locate the module that is nearest to the new module and can be connected to it.

5.2.2 Computing the Suggestions

As outlined in the previous section, we compute possible completions that emanate from a set of anchor modules in the existing pipeline using path summaries derived from a database of pipelines, and rank them by their confidence values. Depending on the anchor modules, a very large set of completions can be derived and a user is unlikely to examine a long list of suggestions. Therefore, we prune our predictions to avoid rare cases. This both speeds up computation and reduces the likelihood that we provide meaningless suggestions to the user. Specifically, because our predictions are refined iteratively, we prune a prediction if its confidence is significantly lower than its parent's confidence. Currently, this is implemented as a constant threshold, but we can use knowledge of the current distribution or iteration to improve our pruning.

VisComplete provides the user with suggestions that assist in the creation of the pipeline *structure*. Parameters are also essential components in visualizations, but because the choice of parameters is frequently data-dependent, we do not integrate parameter selection with VisComplete. Instead, we focus on helping users complete pipelines, and defer a discussion of the influence of parameters in visualizations to Chapter 7. Alternatively, existing systems for parameter explorations can be readily combined with the techniques presented in this chapter [10, 58, 59, 78]. It would certainly be beneficial to extend VisComplete to identify commonly used parameters that a user might consider exploring, but we leave this for future work.

5.2.3 The Suggestion Interface

In concert with our goal of unobtrusiveness, we provide an intuitive and efficient interface that enables users to explore the space of possible completions. Autocomplete interfaces for text generally show a set of possible completions in a 1D list that is refined as the user types. For pipelines, this task is more difficult because it is not feasible to show multiple completions at once, as this would result in visual clutter. The complexity of deriving the completion is also greater. For this reason, our interface is 2D: along one dimension, users can select from a list of full completions. The other dimension allows them to increase or decrease the extent of the completion.

Current text completion interfaces defer to the user by showing completions but allowing the user to continue to type if he does not wish to use the completions. We strive for similar behavior by automatically showing a completion along with a simple navigation panel when a completion is triggered. The user can choose to interact with the completion interface or disregard it completely by continuing to work, which will cause the completion interface to automatically disappear. The navigation interface contains a set of arrows for selecting different completions (left and right) and depths of the current completion (up and down). In addition, the rank of the current completion is displayed to assist in the navigation and accept and cancel buttons are provided (see Figure 5.1(c)).

All of these completion actions, along with the ability to start a new completion with a selected module, are also available in a menu and as shortcut keys.

The suggested completions appear in the interface as semitransparent modules and connections, so that they are easy to distinguish from the existing pipeline components. The suggested modules are also arranged in an intuitive way using a set of simple heuristics that respect the layout of the current pipeline. The first new suggested module is always placed near the anchor module. The offset of the new module from the anchor module is determined by averaging the direction and distance of each module in the existing pipeline. The offset for each additional suggested module is calculated by applying this same rule to the module it is appended to. Branches in the suggested completion are simply offset by a constant factor. These heuristics keep the spacing uniform and can handle upstream or downstream completions whether pipelines are built top-down or left-right.

5.3 Use Cases

We envision VisComplete being used in different ways to simplify the task of pipeline construction. In what follows, we discuss use cases which consider different types of tasks and different user experience levels. The types of tasks performed by a user can range from the very repetitive to the unique. Obviously, if the user performs tasks that are very similar to those in the database of pipelines, the completions that are suggested are very full—almost the entire pipeline can be created using one or two modules (see Figure 5.2 for examples). On the other hand, if the task that is being performed is not often repeated and nothing similar in the database can be found, VisComplete will only be able to assist with smaller portions of the pipeline at a time. This can still aid the user by showing the possible directions to proceed with pipeline construction, albeit at a smaller scale.

The experience level of users that could take advantage of VisComplete also varies. For a novice user, VisComplete replaces the process of searching for and tweaking an example that will perform their desired visualization. For example, a user who is new to VTK and desires to compute an isosurface of a volume might consult documentation to determine that a "vtkContourFilter" module is necessary and then search online for an example pipeline using this module. After downloading the example, they may be able to manipulate it to produce the desired visualization. Using VisComplete, this process is simplified— the user needs only to start the pipeline by adding a "vtkContourFilter" module and their pipeline will be constructed for them (see Figure 5.1). Multiple possible completions can easily be explored and unlike examples downloaded from the Web, VisComplete can customize the suggestions by providing completions that more closely reflect a specific user's previous or more current work.

For experienced users, VisComplete still offers substantial benefits. Because experts may not wish to see full pipelines as completions, the default depth of the completions can be adjusted

as a preference so that only minor modifications are suggested at each step. Thus, at the smallest completion scale, a user can leverage just the initial connection completion to automatically connect new modules to their pipeline. The user could also choose to ignore suggested completions as they add modules until the pipeline is specific enough to shrink the number of suggestions. Unlike the novice user who may iterate through many suggestions at each step, the experienced user will likely choose to ignore the suggestions until they provide the desired completion on the first try.

5.4 Evaluation

5.4.1 Data and Validation Process

To evaluate the effectiveness of our completion technique, we used a set containing 2875 visualization pipelines along with logs of the actions used to construct each pipeline. These pipelines were constructed by 30 students during a scientific visualization course.[1] Throughout the semester, the students were assigned five different tasks and carried them out using the VisTrails system, which captures detailed provenance of the pipeline design process: the series of the actions a user followed to create and refine a set of related pipelines [36].

The first four tasks were straightforward and required little experimentation, but the final task was open-ended; users were given a dataset without any restrictions on the use of available visualization techniques. As these users learned about various techniques over the semester, their proficiency in the area of visualization presumably progressed from a novice level toward the expert level.

To predict the performance gains VisComplete might attain, we created user models based on the provenance logs captured by VisTrails. User modeling has been used in the HCI community for many years [18, 19], and we employed a low-level model for our evaluation. Specifically, we assumed that at each step of the pipeline construction process, a VisComplete user would either modify the pipeline according to the current action from the log or select a completion that adds a part of the pipeline they would eventually need. We assumed that a user would examine at most ten completions and could select a subgraph of any of these suggestions.

Because VisComplete requires a collection of pipelines to derive suggestions, we divided our dataset into training and test sets. The training sets were used to construct the path summaries while the test sets were used with the user models to measure performance.

We note that this model presumes a user's foreknowledge of the completed pipeline, and this certainly is not always the case. Still, we believe this simple model approximates user behavior well enough to gauge performance. We also assumed a greedy approach in our model; a user would

[1] http://www.vistrails.org/index.php/SciVisFall2007

always take the largest completion that matched their final pipeline. Note that this might not always yield the best performance because the quality of the suggestions may improve as the pipeline is further specified.

5.4.2 Results

Figure 5.6 shows one of the test pipelines with the components that VisComplete could have completed highlighted along with its resulting visualization. To evaluate the situation where a set of users create pipelines that all tend to follow a similar template, we performed a leave-one-out test for each task in our dataset. Figure 5.7 shows that our suggestion algorithm could have eliminated over 50%, on average, of the pipeline construction operations for each task. Because Task 1 was more structured than the other tasks, it achieved a higher percentage of reduction. Because Task 4 was more open-ended, although the average percentage is also high, the results show a wider variation (between 30% and 75%). This indicates that the completion interface can be faster and more intuitive than manually choosing a template.

Because it is much more likely that our collection will contain pipelines from a variety of tasks, we also evaluated two cases that examined the type of knowledge captured by the pipelines. Since Task 5 was more open-ended and completed after the other four tasks, we expected that most users would be proficient using the tool and closer to the expert user described in Section 5.3. We ran the completion results using Tasks 1 through 4 as the training data (2250 pipelines) and Task 5 (625 pipelines) as the test data to represent a case where novice users are helping expert users, but we also ran this test in reverse to determine if pipelines from expert users can aid beginners. Figure 5.8 shows that both tests achieved similar results; this implies that the variety of pipelines from the four novice tasks balanced the knowledge captured in the expert pipelines.

Our testing assumed that users would examine up to ten full completions before quitting. In reality, it is likely that users would give up even quicker. To evaluate how many predictions a user might need to examine before finding the desired completion, we recorded the index of the chosen completion in our tests. Figure 5.9 shows that the the chosen completion was almost always among the first four. Note that we excluded completions that only specified the connection between the new module and the existing pipeline because these trivial completions are possible at each prediction index.

Our results show that VisComplete can significantly reduce the number of operations required during pipeline construction. In addition, the completion percentages might be higher if our technique were available to the users because it would likely change user's work patterns. For example, a user might select a completion that contains most of the structure they require plus some extraneous components and then delete or replace the extra pieces. Such a completion would almost certainly

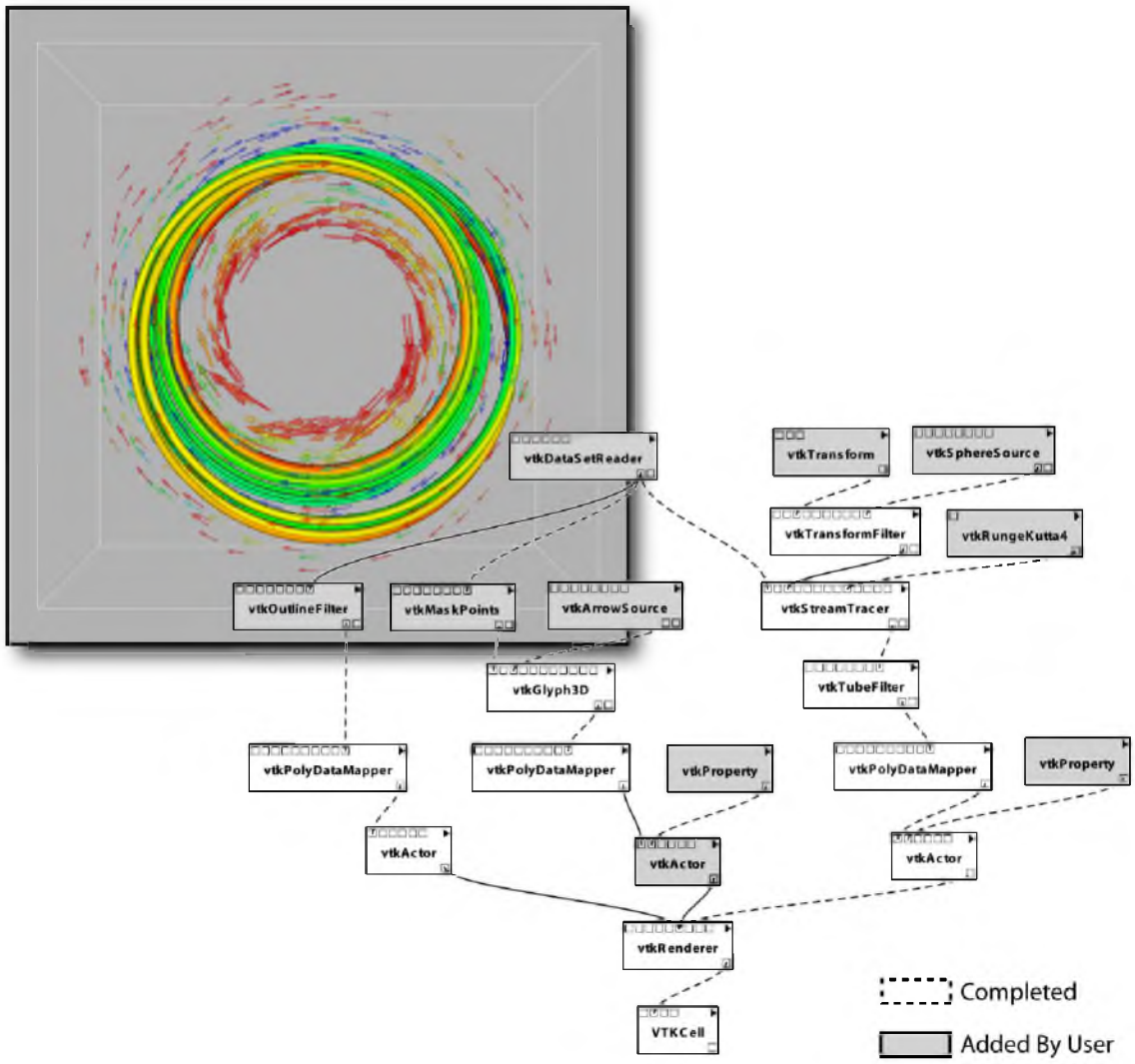


Figure 5.6: One of the test visualization pipelines applied to a time step of the Tokamak Reactor dataset. VisComplete could have made many completions that would have reduced the amount of time creating the pipeline. In this case, about half of the modules and completions could have been completed automatically.

save the user time but was not captured with our user model. Finally, the parameters (e.g., pruning threshold, degree weighting) for the completion algorithms were not tuned. We plan to evaluate these settings to possibly improve our results.

The completion examples shown in the figures of this paper, with the exception of Figure 5.6, used the entire collection of pipelines to generate predictions. Figure 5.6 used only the pipelines from Tasks 1–4.

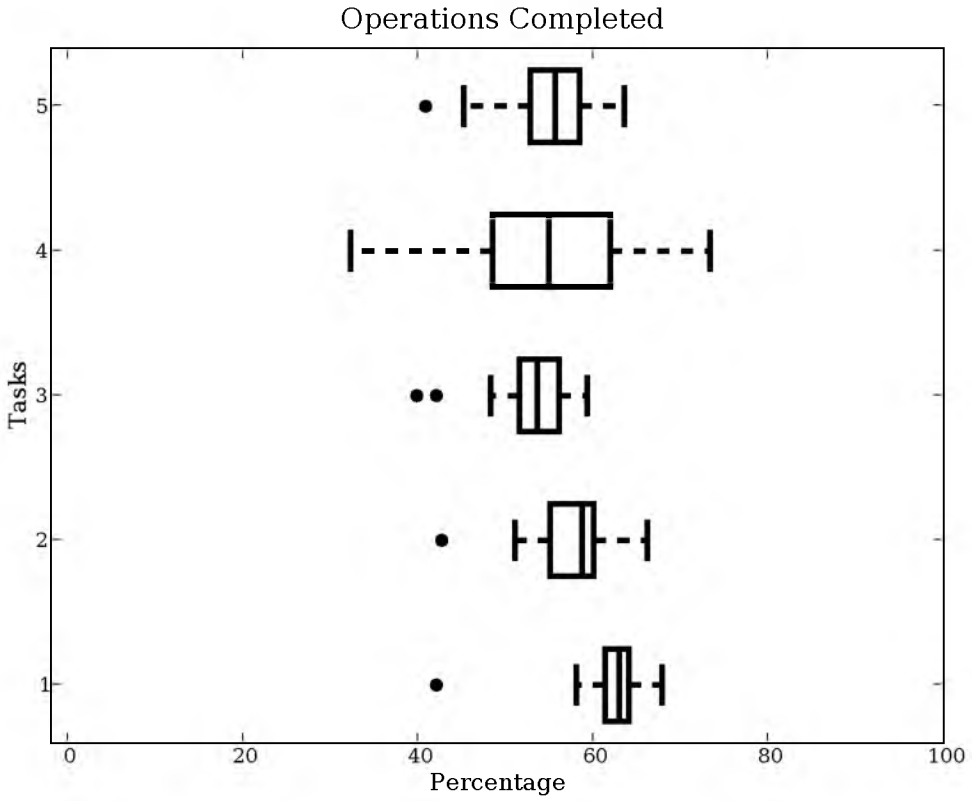


Figure 5.7: Box plot of the percentages of operations that could be completed per task (higher is better). The statistics were generated for each user by taking them out of the training data.

5.5 Discussion

To our knowledge, VisComplete is the first approach for automatically suggesting pipeline completions using a database of existing pipelines. As large volumes of data continue to be generated and stored and as analyses and visualizations grow in complexity, the creation of new content by consensus and the ability to learn by example are essential to enable a broader use of data analysis and visualization tools.

The major difference between our automatic pipeline completion technique and the related work on creating pipelines by analogy presented in Chapter 4 is that instead of using a single, known sequence of pipeline actions, our method uses an entire database of pipelines. Thus, instead of completing a pipeline based on a single example, VisComplete uses *many* examples. A second important difference is that instead of predicting a new set of actions, our method currently predicts new structure regardless of the ordering of the additions. This also means that VisComplete only adds to the structure while analogies will delete from the structure as well. By incorporating more

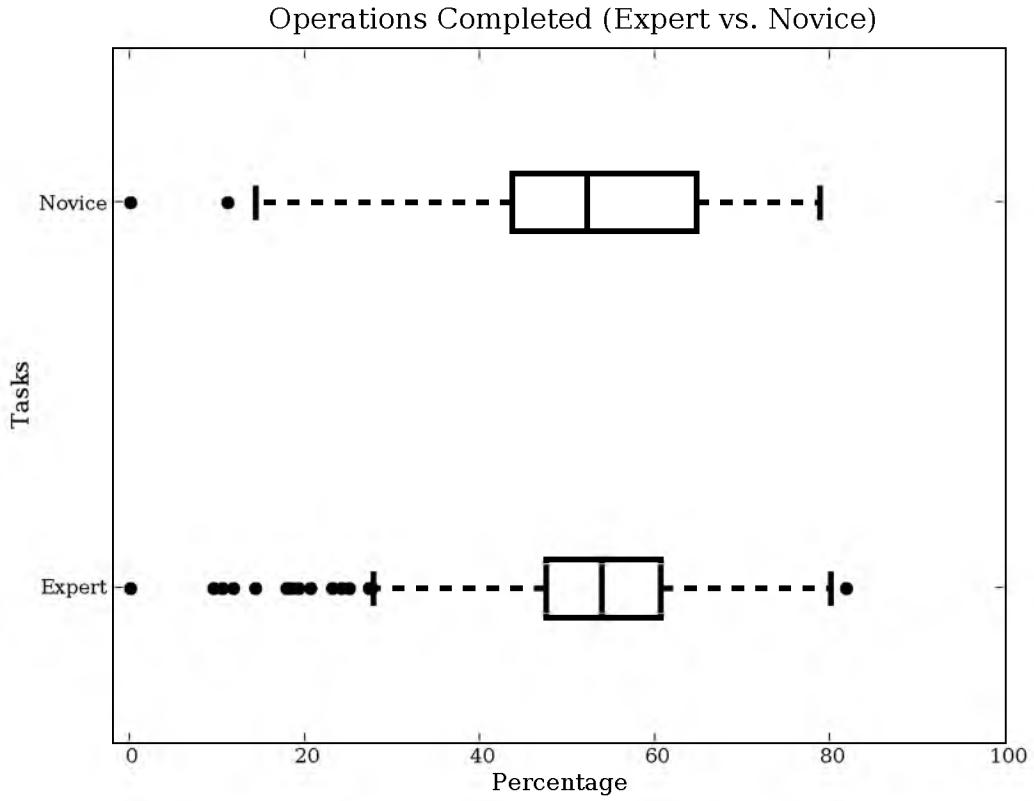


Figure 5.8: Box plot of the percentages of operations that could be completed given two types of tasks, novice and expert. The statistics were generated by evaluating the novice tasks using the expert tasks as training data (novice) and by evaluating the expert tasks using the novice tasks as training data (expert).

provenance information, as in analogies, VisComplete might be able to leverage more information about the order in which additions to a pipeline are made. This could improve the quality of the suggested completions.

We note that there will be situations where data about the types of completions that should occur are not available. Also, some suggestions might not correspond to the user's desires. If there are no completions, VisComplete will not derive any suggestions. If there are completions that do not help, the user can dismiss them by either continuing their normal work or by explicitly canceling completion. Currently, we determine the completions in an offline step (by precomputing the path summary as described in Section 5.1). We could update the path summary as new pipelines are added to the repository, incorporating new pipelines as they are created. In addition, we could learn from user feedback by, for example, allowing users to remove suggestions that they do not want to see again. Completions could be further refined by assigning greater weight to those that more

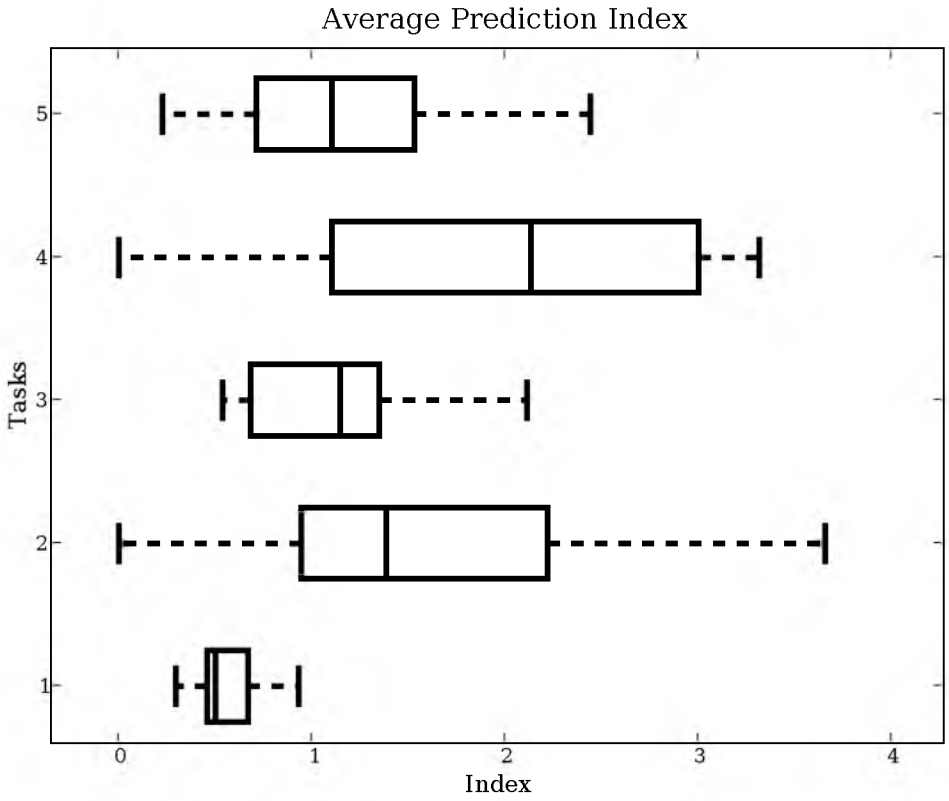


Figure 5.9: Box plot of the average prediction index that was used for the completions in Figure 5.7 (lower is better). These statistics provide a measure of how many suggestions the user would have to examine before the correct one was found.

closely mirror the current user's actions, even if they are not the most likely in the database.

One important aspect of our technique is that it leverages the visual programming environment available in many visualization systems. In fact, it would be difficult to offer suggestions without a visual environment in which to display the structural changes. In addition, the information for the completions comes from the fact that we have structural pipelines from previous work. Without an interface to construct pipeline structures, it would be more difficult to process the data used to generate completions. However, we should note that turnkey applications that are based on workflow systems, such as ParaView [62], may also be able to take advantage of completions in a more limited way by providing a more intelligent set of default settings for the user during their explorations.

5.6 Conclusions and Future Work

We have described VisComplete, a new method for aiding in the design of visualization pipelines that leverages a database of existing pipelines. We have demonstrated that suitable pipeline fragments can be computed from the database and used to complete new pipelines in real-time. Furthermore, we have shown how these completions can be presented to the user in an intuitive way that can potentially reduce the time required to create pipelines. Our results indicate that substantial effort can be saved using this method for both novice and expert users.

There are several areas of future work that we would like to pursue. As described above, we would like to update the database of pipelines incrementally, thus allowing the completions to be refined based on current information and feedback from the user. We plan to refine the quality of the results by formally investigating the confidence measure and its parameters. We would also like to explore suggesting finished pipelines from the database in addition to the constructed completions we currently generate. For finished pipelines, we could display not only the completed pipeline structure but also a thumbnail of the result from an execution of that pipeline. While the evaluation presented in Section 5.4 presents good preliminary evidence of the quality of the predictions generated, it critically does not consider the interplay between the usability of the user interface we designed for VisComplete and the effectiveness of the predictions. That is best addressed by a user study.

CHAPTER 6

THE VISTRAILS EXECUTION MODEL

This chapter describes the execution model used for pipelines in VisTrails. The execution engine in VisTrails was designed with the following goals in mind. First, the set of execution modules in VisTrails should be extensible, so that it is easy to include third-party functionality in VisTrails. Second, VisTrails should try to accommodate the styles used for commonly wrapping third-party scientific visualization and computation software. In particular, VisTrails was designed to easily support application libraries written in either of the following styles: precompiled binaries that are executed on a shell and use files as input/outputs, or C++/Java/Python class libraries that pass internal objects as input/output. Finally, because of its focus on exploratory data visualization and analysis, VisTrails should try to take advantage of the common pieces of pipelines executed in the process, to make the execution more efficient.

We satisfy the first requirement by allowing new classes to be included in VisTrails by what we call *module packages*. Concretely, these are user-defined Python modules. Inside these modules, the user defines custom classes that subclass from a basic VisTrails execution module simply called `Module`. These subclasses implement a simple interface that defines the appropriate custom behavior. We will describe this in detail in Section 6.1.

The second requirement means that VisTrails should be as lenient as possible when it comes to its data model. In particular, this means that the VisTrails dataflow model is somewhat different than architectures that provide a stark distinction between the data types being processed and the classes that perform the processing (typically called "filters"). In VisTrails, the module class hierarchy and the data class hierarchy are the same. The prototypical example is a module that might, for example, produce as output an object of its own type. This is very similar to the object type systems of Java and C++, and the choice was not accidental: it was very important for us to support automatic wrapping of large class libraries. For example, VisTrails users have automatically wrapped class libraries such as the ones in ITK and SCIRun. It is not clear that this would be possible with a more structured data model. We note, in addition, that the model that separates data and filter types is strictly included in our type model.

To satisfy the third requirement, we have designed and implemented a caching mechanism that reuses pipeline pieces from previous executions. In exploratory visualization, we expect similar pipelines to be executed in close succession, and so it is wasteful to require re-execution of identical fragments of a pipeline. Because we reuse previous execution results, we implicitly assume that modules are *functional*: given the same inputs, modules will produce the same outputs. This requirement imposes definite behavior restrictions on classes, but we believe they are reasonable. If we intend VisTrails to be a vehicle for disseminating scientific results and facilitate reproducibility, allowing modules to violate this requirement and behave in a nondeterministic way would undermine most of our other design goals. There exist, on the other hand, obvious situations where this behavior is unattainable. Some modules might be executed solely for the side effects that occur during its execution, not because of the data produced as a result of their execution (uploading a file to a remote server or saving a file to disk, for example). Other modules might use randomization, and their nondeterminism might be desirable. We discuss these issues in more detail in Section 6.2.1

The process provenance model described in Chapter 3 is appropriate for managing changes in the workflows. In exploratory analysis, however, the actual modules themselves might change in the course of an analysis. For example, bugs might be uncovered or modules might require new input parameters. These are changes in both the implementation and interface of `Module` subclasses. In Section 6.6, we describe how VisTrails allows changes to these modules to be instrumented such that workflows that use obsolete module versions are still at least partially supported.

6.1 The VisTrails `Module`

This section provides an overview of the implementation aspects of user-defined classes. A more developer-oriented explanation is provided in the VisTrails User's Guide [6].

Every module that is executed in VisTrails is an instance of a subclass of `Module`. This class defines the basic interface that user-defined modules implement. In the most basic setting, the user simply defines a new `compute` method, which is invoked after all the inputs of the module have been computed. These inputs modules are made available to the current module via its *input ports*, for which data can be requested. Hence, the `compute` methods for the modules in a pipeline are invoked sequentially. If a module does not spawn new threads, the execution will be inherently single-threaded. Notice, however, that the module inputs and outputs are full-fledged Python objects, and so it is easy to overcome this limitation by deferring the actual module computation to methods in the data types being passed by the pipeline.

To simplify the typechecking algorithms for input and output ports, we require that VisTrails modules to be *singly inherited* for the base class `Module`. The Python class declaration itself can

use multiple inheritance, but ultimately, there should be only one inheritance path to `Module` from any user-defined module.

6.1.1 Defining Inputs and Outputs

The input and output of a new module are explicitly defined by the class declaration itself, via the class variables `_input_ports` and `_output_ports`, which will hold a list of *port declarations*. Each port declaration specifies the data type supported by each port. These type declarations are used by the pipeline building stage to ensure that connections between an output and an input port are compatible. A connection is defined to be *compatible* when the output port's module type is a subtype of the input port's module type. This closely matches the Java and C++ type model for a parameters in a function call. In C++ and Java, the parameter used in the call must provide *at least* the interface required by the declared type. The function declaration in a C++ program corresponds to the input port of a VisTrails module: it is an interface requirement.

It is important to emphasize that the type-checking is only performed at pipeline building time. During the execution of a pipeline, VisTrails does not verify whether the output emitted by a module in a port actually obeys the specified interface. While this (deliberate) choice makes debugging of incorrectly programmed modules harder, modules are then free to break the type system if it becomes too restrictive. Since one of the stated goals of the VisTrails execution model is to be as lenient as possible in its data model, we see this as a compromise that allows for convenient design-time type checking while not necessarily restricting the flow of data that can be defined. In addition, the type checking in VisTrails can be overridden by declaring a port type to be of type `Variant`. This is declaration is interpreted to mean that static type-checking should be disabled for this port, and so every connection with it is allowed.

6.2 The VisTrails Cache Manager

The VisTrails Cache Manager schedules the execution of modules in VisTrails. From a high-level perspective, the goal of the Cache Manager is to analyze a visualization pipeline description to determine whether some of its fragments pipeline have already been executed. In that case, the Cache Manager simply reuses these pieces. In this session, we provide a more detailed description of the process.

Definition 1 *Given a visualization pipeline, its* defining graph *is the (directed acyclic) graph with one vertex for each module and an edge for each connection such that vertices and edges stand in one-to-one correspondence with the modules and connections.*

Definition 2 *Given two visualization pipelines P_1 and P_2, P_2 is a* subpipeline *of P_1 if the defining graph of P_2 is a subgraph of the defining graph of P_1.*

Definition 3 *Given the standard partial order $\preceq$ induced by the defining graph of a pipeline P, a pipeline* fragment *corresponding to a module $m \in M(P)$ and vertex v in the defining subgraph is the subpipeline P' consisting of all modules with the corresponding vertices v' such that $v' \preceq v$. The connection set of P' is one such that if there exists a connection between v_1 and v_2 in P, this connection is also present in P'.*

Combined with the assumption made by the VisTrails execution model that modules are functional, we can see that the output generated by any module being executed is completely determined by its pipeline fragment. The main mechanism behind the Cache Manager is a procedure that computes for each pipeline fragment a concise *signature*. When a new pipeline is scheduled for execution, the Cache Manager computes these signatures for all fragments, and checks against fragments that have already been executed. If the signatures match, a new module is not instantiated: the new execution pipeline simply *shares* the already existing fragment. In a sense, the cache acts as a *persistent* set of modules, and the Cache Manager decides when to add new modules to it and when to reuse parts of it. The procedure is shown in Figure 6.1.

Notice that the results generated by each pipeline fragment are not explicitly referenced in the cache. While the earliest implementation of this caching mechanism described by Bavoil et al. is effective for caching intermediate results for pipelines of VTK modules, the scheme described here is conceptually simpler and actually closer to the interpretation of the VisTrails cache as a tabled Datalog program given in the paper [10].

We now turn to the computation of the actual pipeline fragment signatures. As can be seen from Figure 6.1, these signatures are used as keys in a hash table of module references returned by

```
INSTANTIATE-PIPELINE(p̃)
1  s ← SIGNATURES(p̃)
2  m ← {}
3  for m̃ ∈ FIRING-ORDER(p̃)
4       do if s[m̃] ∈ cache
5            then m[m̃] ← cache[s[m̃]]
6            else new-module ← NEW-MODULE(m̃)
7                 CONNECT-MODULES(new-module, p̃, s)
8                 cache[s[m̃]] ← new-module
9  return m
```

Figure 6.1: Basic procedure to instantiate a set of execution modules given a description of a pipeline.

NEW-MODULE. More specifically, they are 256-bit digests produced by SHA-256, an algorithm designed to generate cryptographically safe hashes [26]. SHA-256 computes signatures from arbitrarily long strings. Notice, however, that a digest from SHA-256 is also a string whose signature can also be computed. We exploit this to build pipeline fragment signatures hierarchically: first, we build signatures for modules such that different modules will have different signatures. We then use these module signatures to compute signatures for the corresponding connections by concatenating the module signatures with the appropriate connection information, and computing the digest of this string.

If a module has no incoming connections, then the pipeline fragment signature is defined to be the module signature (in this case, we call the pipeline fragment "atomic"). The other pipeline fragment signatures are computed by structural induction on the depth of a module, where the depth d of a module m, $d(m)$ is defined to be the smallest integer such that $d(m) > d(m')$, for all $m' \prec m$. This way, a module of depth k has access to pipeline fragment signatures of depth $k-1$. Each of these pipeline fragments is connected to the module via some connection. We combine the signature of this connection with the pipeline fragment's to obtain a list of composite signatures, one for each pair. We now have to combine these signatures somehow. Notice that we have to find one consistent permutation of the list, such that if subsequently, a pipeline fragment contains the same connections and fragments, the computed signature has to be the same. We do this by simply sorting the list of signatures, and then concatenating them in order, together with the signature of the current module. The signature of this larger string is extremely likely to be unique because of the design of SHA-256 [26]. The pseudo-code for this procedure is given in Figure 6.2, and the result of a simple execution is illustrated in Figure 6.3.

6.2.1 Selectively Disabling the Cache

Essentially, every pipeline that is executed in VisTrails will, at some point, perform an operation with side effects. Most often, this side effect will be the data from a pipeline fragment being displayed on screen, but some pipelines will save files to disk, or upload the results to a remote server. These modules are *not functional*: different executions with the same data might yield observably different results for the user. Hence, the execution model in VisTrails needs to selectively disable the caching behavior for these modules. We solve this issue by adding a method to the interface of `Module` called `is_cacheable`. This method, by default, returns true inconditionally. When a pipeline gets scheduled for execution, VisTrails first scans the list of modules in the cache and calls `is_cacheable` for each of these. All the modules that return false are simply removed from the cache.

Another important feature of the caching mechanism in VisTrails is that user-defined modules

```
FRAGMENT-SIGNATURE(fragment)
 1  if IS-ATOMIC(fragment)
 2     then return MODULE-SIGNATURE(fragment)
 3     else fragment-ids ← {}
 4          for fragment ∈ UPSTREAM-FRAGMENTS(fragment)
 5              do id ← FRAGMENT-SIGNATURE(fragment)
 6                 conn ← FRAGMENT-CONNECTION(fragment)
 7                 conn-id ← CONNECTION-SIGNATURE(conn)
 8                 fragment-ids ← fragment-ids ∪ {id ⊕ conn-id}
 9          result ← 0
10          for id ∈ SORTED(fragment-ids)
11              do result ← result ⊕ id
12                 return result
```

Figure 6.2: Generation of a signature for a pipeline fragment. In the pseudocode, $\oplus$ denotes the generation of an SHA-256 digest from two other SHA-256 digests, as described in the text.

are free to define their own signatures. Modules that define their own signatures have an associated Python function that returns the correct signature that is declared at the level of a VisTrails package.

The hashing algorithm as defined above can distinguish pipeline fragments with different components. In some cases, however, some parameters or inputs do not actually affect the generated output. For example, some inputs might be used only for performance purposes. If these modules use the basic signature generating algorithm, the Cache Manager will assume that these modules generate different data, which will be potentially wasteful: if the results have been previously computed, parameters that control performance are irrelevant. A user-defined signature computation might allow a module to simply ignore these parameters when computing the signature.

6.3 Performance

The performance gains obtained by caching are heavily dependent on the set of vistrails used in a given session. For instance, for both the brain and visible human case studies (shown in Figures 6.4 and 2.1), where there is substantial overlap, caching leads to speedups that vary between 2 and 2.5 times, computed over the execution of all the visualizations. For the CORIE case study, on the other hand, there is no overlap among the vistrails. In this case, the caching mechanism actually incurs overhead. However, it is very small – under 1%.

Changes to class design and algorithms have the potential to improve the efficiency of the VCM. In particular, some techniques require substantial preprocessing steps to enable faster update times, e.g., the out-of-core isosurface technique of Chiang et al. [21]. In order to leverage these techniques with the *conventional* VTK pipeline, it is necessary to *explicitly* enforce the availability of the required index structures for subsequent operations – and this requires programming. In contrast,

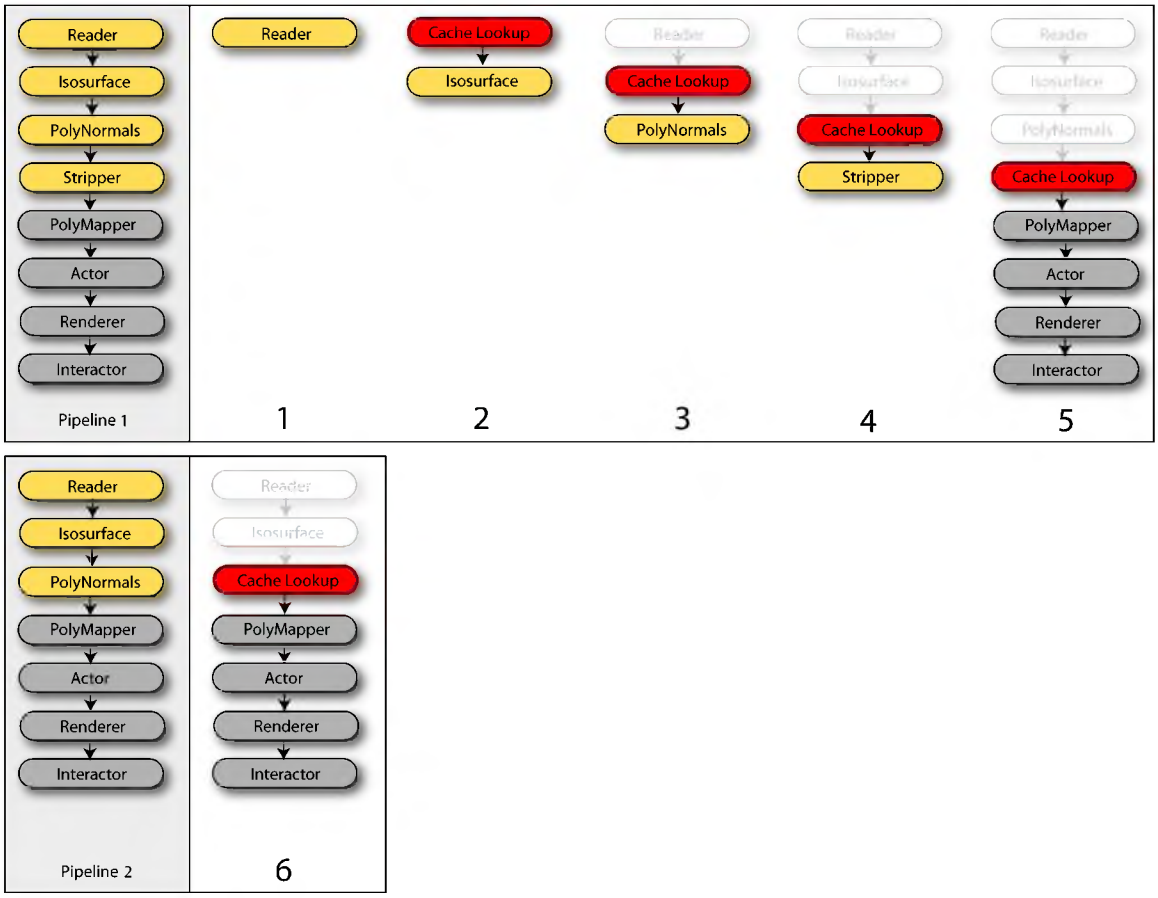


Figure 6.3: In order to execute the first pipeline, the cache manager first determines the data dependencies among its modules. It then decomposes it into a series of subnetworks that generate the intermediate results for this pipeline (steps 1–5). Each intermediate result is associated with a unique identifier in the cache. Gray nodes represent noncacheable modules; yellow nodes indicate cacheable modules; and red nodes indicate vistrail modules that are replaced with cache lookups. Ghosted modules are not present in the subnetworks, but they contribute to the construction of subnetwork cache keys. When the second pipeline is scheduled for execution (step 6), the results for the `Reader-Isosurface` fragment previously computed for Pipeline 1 (in step 4) are reused. Thus, Pipeline 2 requires fewer expensive computations.

the VCM achieves the same behavior in a transparent (and automatic) fashion.

Increasing the number of synchronized views in a spreadsheet has a direct effect on the rendering frame rate. The reason is obvious: all views share a single GPU and the overall frame rates depend highly on the *combined* complexity of the models being shown. In practice, if the frame rate is too slow to interact with a given group of views, we can disable synchronization and interact with a single view until the desired viewpoint and/or values are determined. At that time, we turn synchronization back on. A comprehensive solution to this problem requires the availability of time-critical dynamic level-of-detail rendering algorithms for each of the types of data supported by

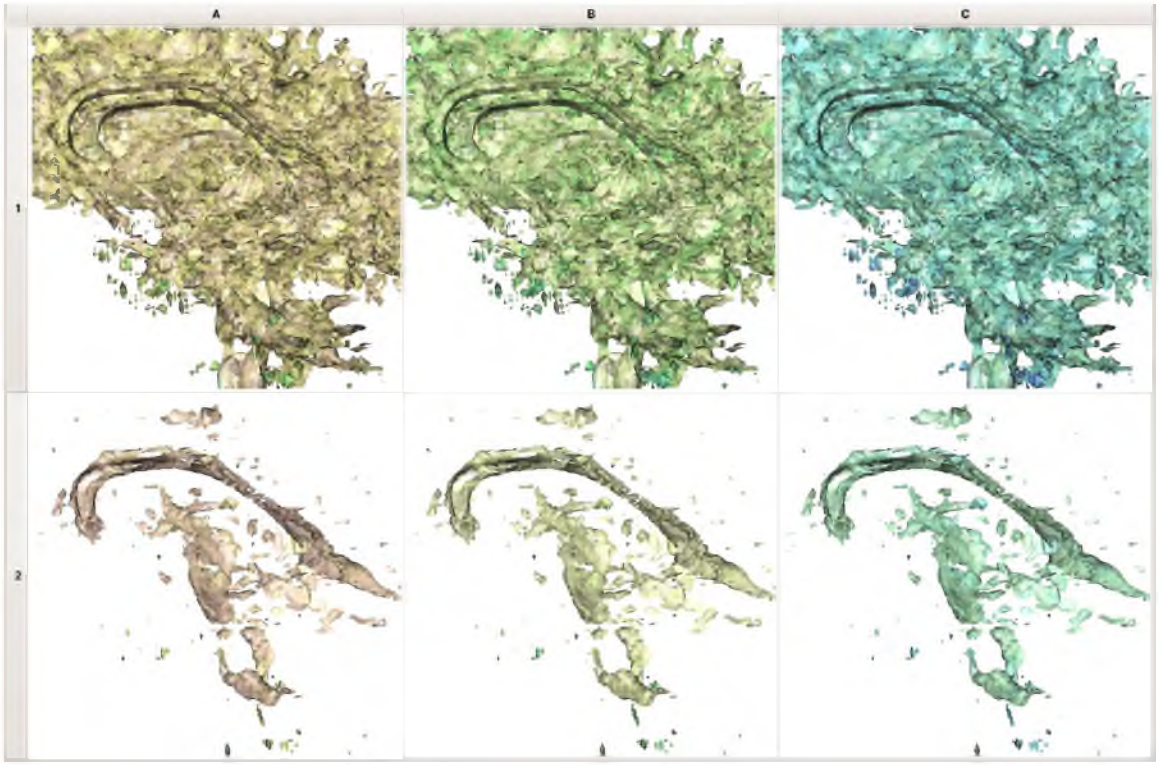


Figure 6.4: When executing many related visualization pipelines such as the ones shown in this parameter exploration, the efficiency gains of a caching scheme can be substantial.

the system. The development of these renderers is an active area of research [77]. In the context of visual programming systems for visualization, it would be interesting to explore data-driven approaches such as the one described in Chapter 5 to automatically suggest a more efficient version of a pipeline that is likely to not be particularly efficient.

6.3.1 Cache Replacement Strategy

As any caching system, VisTrails needs to apply some cache replacement policy. Since entries in the cache have different memory footprints and modules may have widely different execution times, traditional cache replacement techniques might be inappropriate. Because VisTrails modules are arbitrary Python objects, determining their memory usage is not trivial. For example, some modules might return indices into data produced by other modules. By simply inspecting the outputs produced, a memory manager might overestimate the amount of memory being used by a module. Our solution for this problem is to have a VisTrails object report its total memory allocated during this execution. We also note that using some operating system mechanism to inspect the memory usage before and after the module executes does not work in general, because libraries (or even the operating systems themselves) often perform their own heap management, including memory

preallocation.

The best the VisTrails cache manager can do, then, is to monitor the total memory usage as reported by the instantiated modules, and when this figure exceeds a user-specified amount (the current default setting in VisTrails is half the total main memory reported by the operating system), to purge the cache according to some strategy. Currently, when VisTrails reaches this limits, it purges the cache down to 50% memory utilization.

In traditional cache management algorithms, every object under management has the same size. In this context, the optimal strategy was formalized by Belady's theoretical MIN algorithm [11], which states that the optimal algorithm will remove the page which will not be used for the longest time in the future. Clearly, this algorithm cannot be implemented, but it serves as a comparison for other strategies. The popular LRU strategy (Least Recently Used) assumes that an object that has not been accessed recently will tend to stay in that way, thus trying to approximate MIN's behavior without actually peeking into the future.

The argument presented by Belady can be adapted to the slightly more complicated case in VisTrails, where different objects have different sizes. The goal of MIN's behavior is to free the page which wants to come back to the cache the least, or, equivalently, to find the "most secure spot" in the cache for new data. In a sense, it is trying to find the biggest chunk of "space-time" that is going to be available in the future. In the case of objects with different sizes, then, the ideal strategy is then to minimize, for each object, the product of object size and time until future usage. In the case of VisTrails, we define a variant of LRU that assumes that if an object was accessed t units of time ago, it will only be become needed t units of time in the future, and use that as the estimate for time until future usage. The worst-ranked objects are then removed until enough memory is again available.

An undesirable property of this scheme is that as time progresses, two objects that have not been touched in the cache might actually swap positions in the removal ordering: relatively large objects become progressively bad in the ordering as wall time progresses. This behavior is illustrated in Figure 6.5. Notice that this scheme also currently assumes that the time to "retrieve the object," that is, the time for a module to finish executing, is constant, regardless of what object it is. While this is obviously not the case, and although we can reasonably expect that a future execution will take about as long as the previous one has, a more complete study of caching strategies remains as future work. In fact, in early prototypes of VisTrails, the caching replacement strategy was to simply clear the cache every time the memory usage became unacceptably high, which provides evidence that thrashing the cache is not a critical issue.

Another feature that should be noted in the VisTrails cache replacement algorithm is that the re-

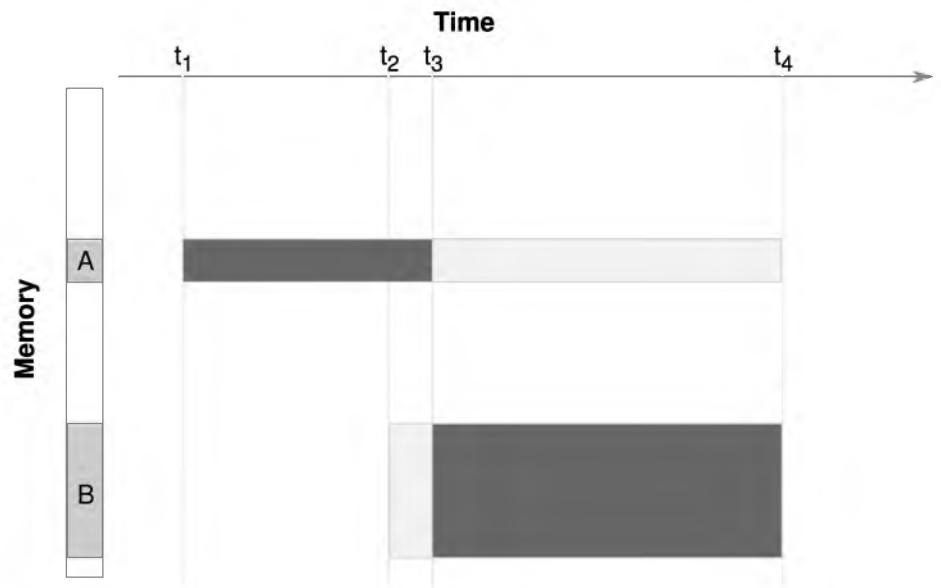


Figure 6.5: One of the properties of the cache replacement algorithm in VisTrails is that the ranking of the objects under consideration changes. Object A was last accessed at time t_1, and Object B, at t_2. At t_3, Object A fares worse than Object B (as illustrated by the area of the rectangle it spans). At t_4, however, Object B fares worse than Object A. In general, larger objects will rank progressively worse and tend to be removed from the cache.

moval of one module induces the removal of all modules that depend on it, because it is impossible, in general, for VisTrails to know if the data associated with a module being removed will still be used by downstream modules. Ideally, these downstream modules should also be included in the analysis of which object to remove, but for the sake of simplicity, we chose to ignore them.

6.4 Instantiating a VisTrails Workflow

From a user's perspective, it is convenient to present all of the state pertaining to a Module in a single location. In VisTrails, this state is shown in the Module Method pane, as shown in Figure 6.6. The method pane shows, for a single module, all the possible parameter names together with the appropriate type of data they accept so that the user can appropriately configure the module's behavior. Since most methods in VisTrails take a single parameter, in this section, we use the terms "parameter" and "method" interchangeably.

Note that values set in this pane are determined by the user's interaction with the GUI prior to the execution of the workflow, and are, in a sense, "constant": they cannot depend on data results upstream of the module. At the same time, it makes little sense to disallow values of these methods to be computed by upstream modules. The implementation of Module classes in VisTrails should,

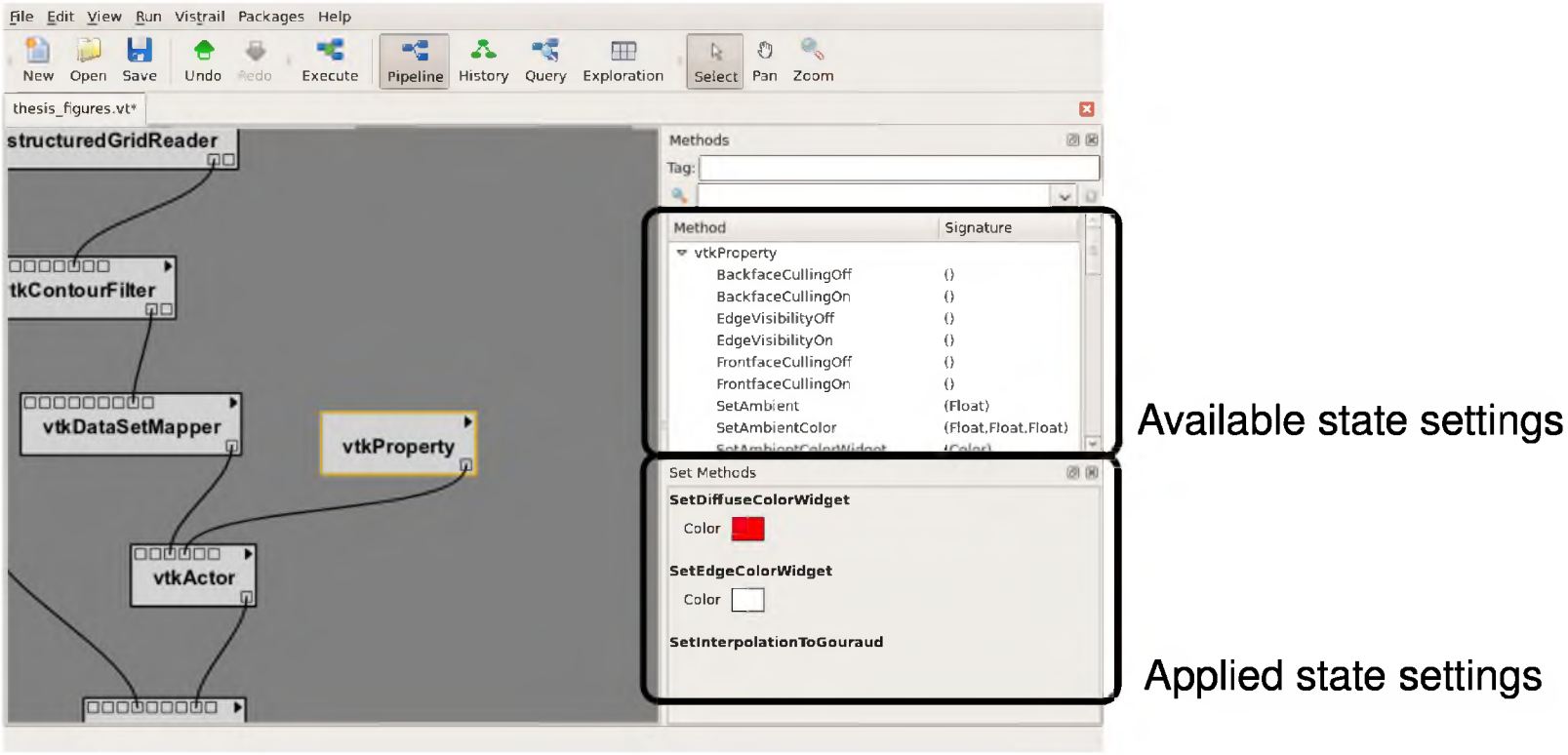


Figure 6.6: The state settings for a particular module are shown in the Module Method pane.

then, satisfy the following requirements:

1. Module state that can sensibly be manually entered should be explicitly listed in the GUI so users can set method values;

2. These methods should also be used as regular input ports in the case that the value needs to be computed by upstream modules; and

3. The API for accessing module inputs during module executions should make as few distinctions as possible between state set as method and inputs set through input ports.

The solution we have designed in VisTrails is to fix special input and output types to be the only possible candidates for the Method pane. In particular, when the designer of a module specifies the possible input and output ports (as previously described in Section 6.1.1), only the input types that are subclasses of `Constant` will be shown in the Method pane. This has several advantages. The `Constant` class defines an interface that its subclasses must implement which includes functions to create GUI widgets for the user to select appropriate values and serialization of the the data type's values. This allows module designers to include custom GUIs for complex data, such as the transfer functions present in modern volume visualization tools [63]. In the workflow view, only the port types that are not subclasses of `Constant` show up as input ports by default. This is simply meant to avoid clutter, and all ports can be made visible by an easily accessible menu option. We have found out, in practice, that most of the time `Constant` port types are indeed used for constant data.

This allows us to completely remove the distinction between method state and connection inputs at the API level. When a workflow is scheduled for execution, all methods set in a module are instantiated as regular modules of the specified type, and connected to the appropriate input port. At the API level, then, methods essentially do not exist. The instantiation is illustrated in Figure 6.7.

6.5 Managing User Interactions

The process provenance model presented in Chapter 3 appropriately manages and stores the branching exploration history of a user's session in VisTrails. Much of the exploration in a visualization session, however, happens interactively, *after* the user has executed the workflow. This exploration can consist, for example, of picking the correct camera angle or volume clipping plane. Jon Claerbout, one of the earliest scientists to have tackled the problem of reproducibility of computational results, states that "dependence on an interactive program can be a form of slavery (nonreproducible research)" [23]. However, it is impossible to argue with the power of interactive techniques in exploratory data analysis. In fact, it could be argued that interactivity is one of the

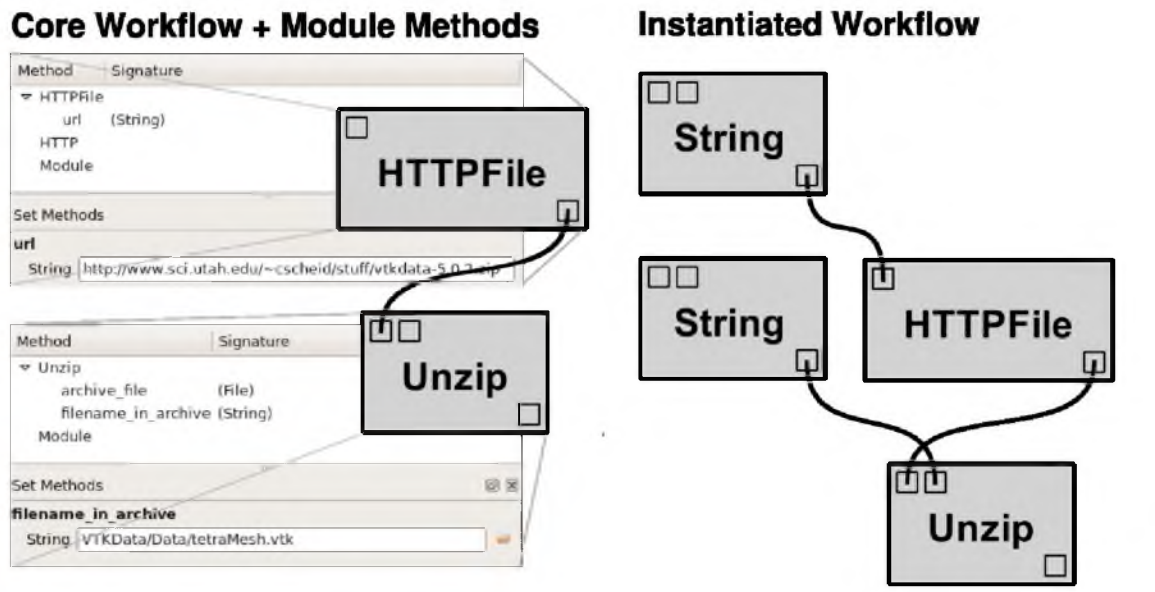


Figure 6.7: When a workflow is scheduled for execution, the methods of a module are instantiated as additional executable modules. This simplifies the implementation of `Module` subclasses by VisTrails users, and the interplay of method specification and caching functionality.

most important aspects of the exploration process. In this section, we show how VisTrails supports capturing this aspect of the visualization exploration process in the provenance model.

Jankun-Kelly, Ma and Gertz propose the *pset-calculus* ("pset" stands for parameter set) for this type of situation [59]. We could have implemented their technique in VisTrails as a way to factor the exploration process in two phases: before workflow execution, and after. We decided against it for several reasons. First, our solution expresses this step of the exploration process directly in terms of VisTrails actions, and we see the uniformity of the resulting model as an advantage. Second, while it is the case that most of the exploration process in this stage happens by changing parameters, there still exist some user interactions that are better modeled by adding extra modules and connections to the underlying workflow (specifying a camera angle when none existed before is an example that requires a new module in the workflow). This means that a solution based purely on tracking parameters would fail to capture part of the possible exploration.

The solution we adopt is to allow user-defined executed modules to post events from the execution objects back to the provenance tracking engine. These events contain the actions necessary to bring the executed workflow to the one currently expressed after the execution. These actions are then simply added to the version tree. Care must be taken, however, in the case that many modules in a single workflow generate the same signature and are thus mapped to a single object in the cache. In this case, the action by a single executed object needs to be translated into many

different actions, to ensure that all the objects in the workflow are updates correctly. It would be hard, in general, to guarantee that the actions posted by the execution object actually correspond to the change performed by the user, and so VisTrails assumes the action specifications are, in fact, accurate.

6.6 Managing Changing `Module` Specifications

In the course of using VisTrails for data analysis and exploration, sometimes a user will have to change some of the modules in the VisTrails packages she might have created, or some of the system libraries. The reasons for this are the same as the reasons for changing a workflow during the course of the exploration. The same motivations that we previously gave for the need to keep the process provenance then clearly apply for the underlying libraries and VisTrails packages. More importantly, the danger of *not doing so* is the same – for example, it is hard to enforce reproducibility of results, and also harder to effectively share the workflows. In fact, in the course of one of the evaluation studies we performed with VisTrails (described in Chapter 8), one expert reviewer pointed out that versioning of external packages was one aspect of the process provenance that VisTrails simply did not keep track of.

While we would ideally like to have a system in which every possible dependency is tracked, this would require a significantly different approach to the one we have taken with VisTrails. To give but one example, even current operating systems have fairly complex system-level mechanisms for resolving different library versions, such as GNU ld's *version scripts* [44], so it seems that on a fundamental level, shared library versioning is still very much an open problem. In the current version of VisTrails, then, there is no support for concurrent versions of shared libraries to be present at any given time. Still, modules can output the library versions that they use to the execution log. This way, if two execution runs produce unexpectedly different results in two different environments, it is possible to at least compare the execution runs to check whether different library versions can explain the change. Currently, it is the responsibility of the module designer to report the appropriate logging information, but there are possible alternatives, such as the OS-level system call tracking in PASS [91].

Where VisTrails can fare better, however, is in tracking the changes of the `Module` subclasses specification themselves in VisTrails packages. Our solution relies on the user package exposing hooks for resolving a failed lookup of an old module version, and replacing actions that refer to these modules with new ones. Such lookups can happen in only certain situations.

Whenever a new workflow version v is requested, we first compile a single action from the composition of all the actions necessary to change an older workflow to v (this older workflow is potentially one of the checkpoints described in Section 3.4). This compilation step is simply a

variant of dead-code elimination, where VisTrails removes canceling pairs of add-delete actions. We collect all the remaining actions that add new workflow pieces, such as modules, connections and parameters. All additions that refer to current valid package versions are performed as usual. Then, for each module whose version is outdated, VisTrails calls the responsible package and expects back a list of actions meant to encode an updated, equivalent pipeline fragment. Parameters need to be treated slightly differently, because their record in the vistrail schema include a serialized version of the value. In the case of parameters, the package is responsible for translating the serialization of the old module version to a serialization of the current module version.

This translation happens for all necessary pieces, and then these new actions are added to the vistrail, together with actions that delete all the old versions. This way, the version tree stores a record of this workflow update. This action is annotated as "updateVersion" to facilitate future queries. If the package is unable to perform this translation, it can simply raise an exception that will be captured by the system.

Requesting that packages return actions which encode the behavior of legacy versions requires extra work by the package developer, but it has advantages over possible alternatives. For example, the standard alternative is to keep around older versions of the Python classes and treat new versions as entirely different modules without any relation between them. Our proposed solution actually encodes the relationship between the old versions and the new versions in the version tree.

CHAPTER 7

INCORPORATING PARAMETER SETTINGS

The two techniques presented in Chapters 4 and 5 use the graph structure of visualization pipelines together with the process provenance to provide the user with higher-level primitives in which to interact with VisTrails. While these *structural* techniques are not possible with the previous state-of-the-art in visualization exploration techniques such as the work in Design Galleries [81] and the p-set calculus of Jankun-Kelly et al. [59], it is necessary to note that both visualization-by-analogy and VisComplete are, in fact, oblivious to the actual parameters used in a visualization. In this chapter, we outline a few possible mechanisms to provide parameter suggestion support to general pipelines in VisTrails. While these have not been currently implemented, they are avenues for future work.

7.1 Parameter PCA

One of the basic themes in scientific visualization and computer graphics is the search for a good set of parameters for a desired result, whether it is a colormap, a streamline seeding point, a camera angle or a transfer function. Spreadsheet-based techniques such as presented by Jankun-Kelly et al. [58] and the parameter explorations as described in Section 3.8.2 are effective in covering a wide swath of a local neighborhood in parameter space. However, they by definition ignore any interesting features of the visualization function when generating these slices through parameter space.

Design Galleries [81], in contrast, provide a very effective way to navigate large parameter spaces via a combination of stochastic sampling of the parameter space and a nonlinear dimensionality reduction technique such as Isomap [118] or LLE [106], *on the output data generated*. Design Galleries are general, simple and effective. However, they provide no information about the input parameter space. There is no relation between the sets of input parameters that generated the image and the display presented to the user by the method. The biggest disadvantage of this is, of course, that it provides very limited insight about which parameter settings brought about the change in the

result. If the goal of the exploration is to increase the insight about the data, this situation is also not ideal.

One alternative is to pay closer attention to the relationship between the space of input parameters and the space of outputs. Let us call our visualization technique f, and say that f maps from I to O. If we impose a manifold structure in both input and output spaces, we can talk about properties of the local structure around points $p \in I$ and $f(p) \in O$. In particular, we know that the derivative operator is a linear map from the tangent space of p to the tangent space of $f(p)$, and so we can now use this information to enrich parameter explorations. We illustrate this in Figure 7.1.

For example, by choosing an appropriate metric for the input and output spaces, we can meaningfully talk about which parameter changes around p created the biggest changes in $f(p)$ by simply computing PCA [47] on the vectors of the tangent space of $f(p)$ and then computing the pullbacks of the principal components through the derivative operator. This would combine the output-sensitivity of Design Galleries with the explanation power of parameter slices. More concretely, it would allow one to design a graphical user interface where it is possible to navigate a high-dimensional space of parameters by always picking a locally best set of changes to the current parameter. It would also naturally serve as a form of "visualization sensitivity," showing whether the results of a visualization might change drastically with a small change in the inputs. We speculate that other applications of this additional structure are also possible, such as parallel transport of preset paths through the parameter space.

7.2 Learning Good Parameter Values

This section really could use a description of supervised learning and kernel tricks in general.

While a system built on the ideas presented above is likely to provide an attractive interface for navigating the space of visualizations, it would say nothing about the *quality* of the generated explorations. Nevertheless, it is obvious that the right choice of parameters has a great influence in the quality of the resulting visualizations, and a system that could successfully predict (even if partially) whether a certain visualization is good could potentially discard much of the parameter space, making navigation much easier.

There has been a fair amount of recent work that aims to characterize the quality of a visualization by maximizing an appropriate metric. In particular, researchers have look at interpreting a figure as an information channel, and formulating the problem of picking the best parameters (for example, the viewpoint) as one of maximizing the entropy of the symbol list [95, 127]. Alternatively, some authors have used supervised learning techniques from the machine learning community, such as training a support vector machine (SVM), to classify whether 3D objects are upright [39]. This is the approach we intend to use to learn good parameter values.

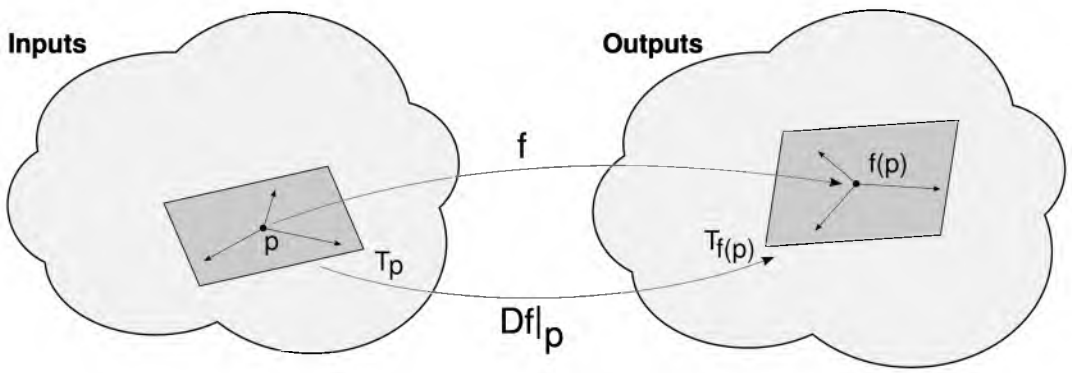


Figure 7.1: Illustration of the configuration around neighborhoods of input p and output $f(p)$. The derivative $Df|_p$ is the linear map that connects small changes of p to small changes of $f(p)$ (vectors in the tangent space of p and $f(p)$, respectively) . In particular, a *pullback* of a vector v in $T(f(p))$ is a vector v' in $T(p)$ such that $Df|_p(v') = v$.

The most commonly support vector machine receives a set of points in space, together with a binary classification on these points (hence the name "supervised learning"). Mathematically, these support vector machines look for a hyperplane that is maximally separated from any instances — SVMs are also known as maximum-margin classifiers [47]. Training an SVM involves solving (via preprocessing) a semidefinite quadratic program. While this is not very efficient, the actual classification of a point is typically much faster, because only a few training points are actually used in the computation: these are the "support vectors," and are points close to the separating hyperplane.

For the same reason argued in the previous section, a linear hyperplane will never be a good separator for the typical visualization techniques used in practice. The standard solution is to map the points (nonlinearly) to some *feature space*, and find the linear hyperplane on this extended space. This is known as the "kernel trick," since it is possible to formulate the SVM process by never referring to the point positions in space directly, but only by the inner product defined by the space. One of the most commonly used kernels in practice is the *gaussian kernel*, which is what we will adopt here.

7.2.1 Feature Engineering

The fundamental problem in our case, then, is one of *feature engineering*: how to define a meaningful set of features that will be used for a learning algorithm in a workflow system? Our solution is to require, for modules that want to have associated classifiers associated, that they be

able to compute a *metric d* for the inputs they take, and an *influence scale* σ. With these two components, we can define a kernel for that space as

$$K(x,y) = \exp\left(\frac{m(x,y)}{\sigma^2}\right) \tag{7.1}$$

That $K(x,y)$ is indeed an inner product is not trivial to prove, and not particularly important for our applications, but the series expansion argument is available in Minh et al. [88]. With that in hand, we now need to find a way to generate training sets.

7.2.2 Generating Training Data

To generate training data, we intend to use the process provenance data collected from users willing to donate their visualizations and data. One possible dataset comes from the vistrails provided by the students of the Scientific Visualization course used in Chapter 5. To navigate different parameters for the training data, we can use the technique described in the previous section.

To determine whether a particular visualization is good or bad, we can use a system similar to Louis von Ahn's Games With a Purpose [126], by asking two people to correctly guess which is the visualization that *the other person* will prefer, using similar techniques as described in that paper to avoid cheating. After presenting a sufficiently large set of visualization/parameter pairs to users, we simply group the ones that received a majority of positive votes as "good," and similarly for "bad." Notice that in the case of a workflow used during training, more than one module my have associated classifiers, and so the "good" or "bad" vote will be cast to all the modules in the workflow. After the per-module SVMs have been trained, we can easily use them to classify potential visualizations by simply combining the classifiers of each module in the workflow and present them to the user.

CHAPTER 8

EVALUATION

This chapter will present some studies we have performed to evaluate the effectivess of the design choices behind VisTrails. We first describe the two evaluations of the system as a whole we have performed on separate occasions. We then discuss further evaluation studies and related issues. Evaluating the effectiveness of a system such as VisTrails is not trivial. In particular, it is damaging to the field in general to perform user studies that measure overly simplistic characteristics of human behavior using the system. For example, while it is possible to design a study to determine which of two graphical interfaces are faster for a user, it is much harder to measure whether a system allows a user to better understand the data being analyzed. It is especially perilous to fall under the trap of assuming that the attributes we can measure are the ones by which we should evaluate a system. It is important to remember, then, that the design goals behind VisTrails are not simply to help users create visualizations faster. The evaluations we have performed are aimed at understanding how users interact with the novel features rather than determining quantitative measures about the system performance.

8.1 Expert Review

We have solicited user input about VisTrails on two separate occasions. Our stated goals in developing the system were to facilitate exploratory data analysis, and the reproducibility of execution runs and processes. We first asked for a handful of collaborators to write about their use of VisTrails and their impressions of the system, in the spirit of expert evaluations [119]. We explicitly solicited input about "usefulness, importance, extensibility and scalability" of the provenance tracking mechanism described in Section 3 and the execution model described in Section 6.

The respondents (four in total) were generally pleased with the ease of adding new modules to VisTrails. One of the respondents, however, claimed the execution mechanism was entirely ill-suited for his problem domain (analysis of high-energy physics experimental data). On the other hand, this same respondent claimed that the history mechanism was important enough in his case that it warranted using VisTrails with a custom replacement of the execution environment:

> I am using VisTrails to develop a workflow management system for doing Experimental High Energy Physics (HEP). In particular, I am using the workflow assembly interface and the history tracking mechanism, replacing the interpreter, adding custom modules and dropping the spreadsheet and using a custom made 'scientific notebook' instead. The custom interpreter allows for asynchronous processing of the workflow with incremental updates of the results in order to give fast feedback to the physicist. The custom modules correspond to very fine grained programming structures equivalent to 'for' loops, 'if' blocks, etc. This fine granularity allows us to use VisTrails' provenance tracking to record all the filtering changes typically done during an HEP analysis.

About provenance management, this same respondent pinpointed one of his perceived main weaknesses of the system:

> I feel tracking of provenance is an essential part of scientific research and the HEP field does not have any tools to aid with provenance tracking. However, to me provenance tracking is about 'what you did' while what VisTrails does is even better, which is 'replay what you did and use that as a starting point for new changes'. It is the latter ability which enticed me to use VisTrails. Given that VisTrails tracks all modifications done to a module, the system is extremely extensible, assuming that all modules are defined such that they contain no internal 'hysteresis'. However, there is one change that VisTrails does not have a mechanism for, which is versioning of the external packages.

This is a critical issue that we have only started to address with VisTrails and whose difficulties we have discussed in Section 6.6.

Another point generally made by respondents is that, while they prefer the provenance tracking of VisTrails to a manual management of the exploration data, the presentation of this history is still somewhat cumbersome:

> The importance of the provided provenance system relies on the fact that the possibilities of parameter exploration for this task are enormous, and often I am creating new workflows to create a different type of visualization or analysis. The fact that VisTrails has a supportive provenance system allows me to explore different alternatives with more freedom, since I know that the different alternatives that I pursued before can be easily recovered. I wish there was a better way to clear paths that I do not want anymore, other than starting a new vistrail and copying what I want.

> Although necessary, the history management interface is not necessarially [sic] ideal. In my projects, it is rare that I go back to earlier versions. While I know that the entire progression from beginning to end is stored, I do not need to see old versions as it tends to clutter the already full screen space. However, by using the search-and-refine functionality, my working set of versions is significantly reduced to a manageable subset.

> The idea of representing the history as a graph works well for me. My only problem is how quickly the graph grows. To me, part of the problem is that activities which I think are unimportant to the physics [...] are represented on the graph and items of extreme importance (e.g., a workflow which I actually ran and got results from) are only shown on the 'reduced' graph if I manually name them.

8.2 User Survey

More recently, we have asked a wider audience to fill a survey on their usage of VisTrails. In particular, we have asked the users about which features in VisTrails they commonly use, and the results are summarized in Table 8.1. We note that the completion mechanism of Chapter 5 is not included in the current release of VisTrails, and so does not appear in the question.

Even if somewhat disappointingly, we note that most of the "advanced" features in VisTrails, such as the visualization by analogies or the Visual Differences, are used much less often than (say) the extensibility capability in VisTrails via user-defined packages. We have not approached the particular respondents to ask for reasons while that might be so, but intend to do it after we stop collecting answers for the survey.

On the other hand, the process provenance tracking mechanism seems to be widely used. In order to understand better how often people are using the branching history capabilities, we decided to ask how many different versions in the version tree a user would tag at any given time. Even though we have the vistrails collected from the Scientific Visualization courses taught using VisTrails, we asked other users this question explicitly, since our assignments required them to tag different versions and so would bias the responses. The results are shown in Figure 8.1. Based on those responses and the ones about text querying from Table 8.1, we speculate that users are, at least in part, using the provenance management in a regular basis.

8.3 Discussion

We believe that a comprehensive and persistent tracking mechanism is very valuable to users of data analysis in general, and that appropriately managing this history is one of the outstanding problems in the design of visualization systems. At the same time, we note that more dedicated user studies are still necessary to effectively evaluate the utility of the techniques presented here.

However, we argue that traditional evaluation methodologies might not be directly applicable in this situation. In particular, some of the scenarios in which we believe the history tracking of VisTrails is most useful are also the ones where users already have developed their own (however ad-hoc, cumbersome and inefficient) schemes for managing their explorations. During the preparation of one of the early drafts of the user survey described above, we had some respondents fill out the survey in our presence to ensure, among other things, that the question wording was clear. When asked about his answer with respect of number of tagged versions (only one per vistrail), he agreed that the version tree might be useful, but he was used to saving different versions as separate files in the file system. The problem in designing an evaluation methodology in our case, then, is in how to appropriately control for these situations.

Table 8.1: Summary of VisTrails features usage by frequency. The question asked was: "How often do you use any of these particular features in VisTrails?" There were, at the time of writing, 24 respondents.

	Never	Used once or twice	Every five sessions	Every session	More than once a session
Version Tree	20.8%	16.7%	0%	20.8%	41.7%
Parameter Exploration	8.3%	16.7%	45.8%	20.8%	8.3%
Text Querying	41.7%	29.2%	12.5%	0%	16.7%
Query-by-Example	62.5%	29.2%	4.2%	4.2%	0%
Analogies	79.2%	16.7%	4.2%	0%	0%
Visual Diff	54.2%	25.0%	16.7%	4.2%	0%
User-Defined Packages	45.8%	16.7%	0%	8.3%	29.2%

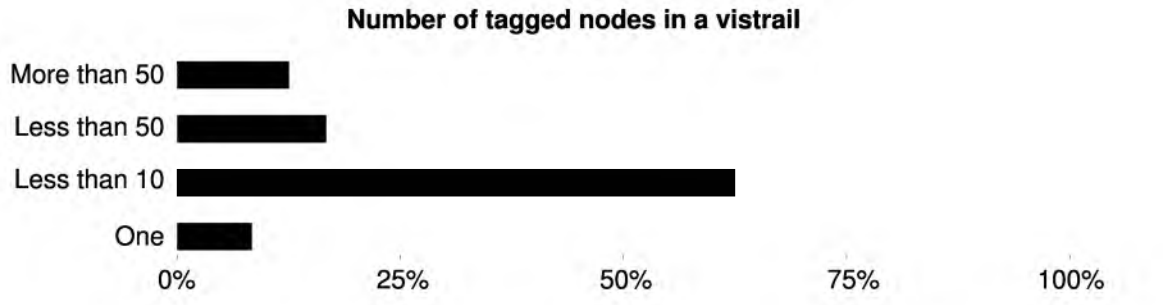


Figure 8.1: Answers to the question "In any given vistrail (one .vt file), how many different versions do usually you tag? That is, how many different workflow versions do you regularly use in a vistrail?"

One possible avenue for investigation is to instrument VTK, the visualization library packaged by default in VisTrails. VTK already has functionality to inspect a pipeline built using their libraries, so one could use borrow this to submit all executed pipelines during the programming assignments of a scientific visualization course. By carefully designing the assignments such that they are possible both in plain VTK and VisTrails, it might be possible to compare the results, in terms of time to completion, number of attempted visualizations, and so forth.

However, we would like to point out at least one problem with such quantitative analyses, which is that to perform such analyses essentially reduces the evaluation of a system to whether it is possible to finish a task faster. Although this is certainly an attractive feature of VisTrails, we would argue that the provenance tracking mechanism would still be desirable (or even necessary), *even if it were to cause users to be slower*. Compare the situation to performing physical experiments without extensive annotations in a lab notebook — it would surely let one finish experiments faster.

We argue that some infrastructure such as proposed here should exist by default whenever at all practical, and not only when it makes people work more efficiently. We simply do not currently know how to properly formulate this in user studies and experiments.

CHAPTER 9

AN OVERVIEW OF THE VISTRAILS ARCHITECTURE

In this chapter, I will present a high-level description of the architecture of VisTrails. This chapter clarifies which parts of VisTrails are involved in the different features presented in this thesis, such as the Visual Differences presented in Chapter 3, and visualization by analogy, presented in Chapter 4. This is intended to highlight some of the decisions behind the actual implementation of the system, and also to show programmers one possible way of implementing the techniques previously discussed.

Although the early prototypes were written in C++, fairly early in the development, VisTrails was ported to Python. It currently runs on Python 2.5 and later. The decision to use Python was mostly borne out of the perception that Python is quickly becoming a universal modern glue language for scientific software, much in the same way that TCL was in the late 1980s [98]. Many libraries written in different languages such as Fortran, C, and C++ use Python bindings as a way to provide scripting capabilities. Since one of the stated goals for VisTrails is to facilitate the orchestration of many different software libraries in workflows, a pure Python implementation would make this much easier. In particular, Python has dynamic code loading features similar to the ones seen in LISP environments, while having a much bigger developer community, and an extremely rich standard library.

VisTrails uses Qt [25] as its GUI toolkit, and, in particular, its PyQt Python bindings [75]. It currently has about 67k lines of manually-written code, 30k lines of included VisTrails packages, and 100k lines of automatically generated code from the `VisTrails db` component (see Section 9.3).

The provenance model described in Chapter 3, the visualization-by-analogy capability of Chapter 4, and the data-driven recommendation system of Chapter 5 were all primary efforts of this author. A substantial amount of the development behind VisTrails itself, however, involved a larger group of individuals. The development of the current source base started in late 2005, and involved three students working on it for a large portion of their time over the course of a year or so. After the basic infrastructure was stabilized, the group was increased to about ten people, with two people

being part of critical development efforts, ranging from substantial day-to-day maintenance tasks and the preparation of new releases to new features (most notably the `VisTrails db` component, a completely overhauled storage system developed by a different student). The remaining developers made other contributions such as third-party packages and extensions for VisTrails. These include a website where it is possible to browse and edit VisTrails workflows, and LaTeX extensions to facilitate the reproducibility of printed figures.

Figure 9.1 illustrates the main components and their associated subsystems. We now turn to a description of each of these components separately.

9.1 The `VisTrails core` Component

The `VisTrails core` component contains the pieces of the VisTrails code that handle the data structures necessary for capturing and representing the three major aspects of VisTrails: the capturing of the process provenance, the interaction with workflows, and the execution of these workflows. Each of these tasks is conceptually separated in a separate layer, naturally called the History, Workflow and Execution layers. Most features of VisTrails will interact with at least one

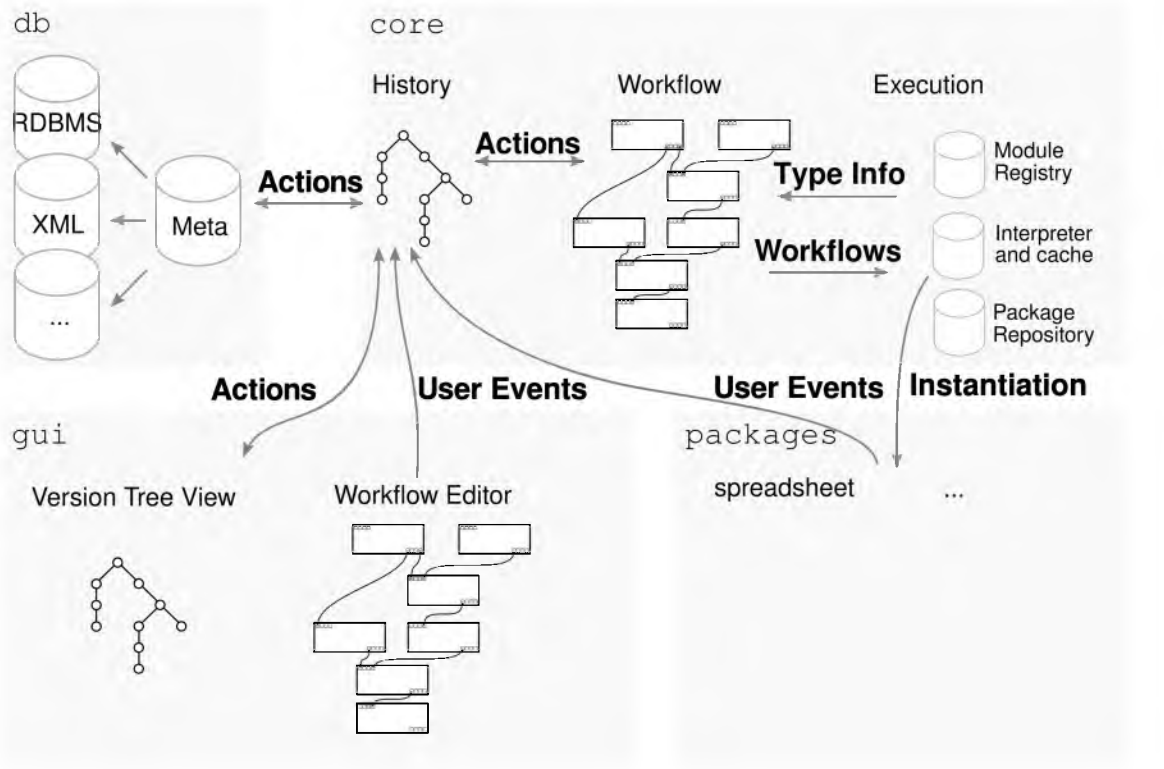


Figure 9.1: A high-level view of the VisTrails architecture. The main application is comprised of three high-level, mostly independent components: `VisTrails core`, `VisTrails db`, and `VisTrails gui`. The arrows indicate the major interactions between the subsystems.

of these core layers. Most importantly, `VisTrails core` completely provides a fairly complete abstraction over the change-based process provenance mechanism described in Chapter 3. There are a few advantages to this layering. First, all interaction with the underlying storage (the `VisTrails db` component) is done by `VisTrails core` and is invisible to applications. This separation allows the different `VisTrails db` schemes to be completely irrelevant to the rest of the application. In addition, it allows VisTrails features to operate on the level to which they are more naturally suited. For example, Visual Differences are more easily described as a sequence of vistrail actions, but better presented as two workflows overlaid on top of each other. As should be expected, then, the Visual Difference tool interacts only with the History and Workflow layer of `VisTrails core`.

The following sections present a more detailed description of each layer in `VisTrails core`.

9.1.1 The History Layer

All the algorithms related to the change-based model proposed in Chapter 3 are implemented in the History layer of `VisTrails core`. The basic data structure of this layer is the vistrail itself, which, as we have previously described, is a rooted tree of changes to individual pieces of a workflow. As such, any given workflow itself is never manipulated by the History layer.

The History layer provides a point of access to the actions performed by the user in the course of building their workflows. The most important objects in the layer are instances of the `Action` class, whose main functionality is to effect some specified change in a workflow. `Action` objects can be composed to create more complex actions that might represent a sequence of user actions. These objects can be inspected, so algorithms can reason about the effect that the actions will perform. As described in Chapter 4, this is used to compute the *context* of an analogy. This inspection is also used to prepare the Visual Difference display, as shown in Chapter 3. In addition, it also provides the creation of new actions that might not have been generated by the user. For example, the History layer is responsible for creating the actions that have changed context during the calculations involved in building a new visualization by analogy, also as described in Chapter 4.

9.1.2 The Workflow Layer

The Workflow layer controls access and methods of a single workflow. The main objects in the Workflow layer are, perhaps obviously, instances of the `Workflow` class. This is the class of objects on which `Actions` act, and it represents the static information about a workflow to be executed in VisTrails. A `Workflow` object is a collection of `Module` and `Connection` objects representing a directed acyclic graph. `Module` objects contain enough information to uniquely identify them, which is in our case, the identifier of the package that defines the module, a module name and an optional namespace. The behavior of modules can be determined by *parameters* that are sometimes

fixed ahead of execution (for example, the file name used by a data loading module would be a parameter). Each parameter is completely specified by the name of the module's method that sets the parameter and the value of the parameter. A `Module` then contains a list of `Parameter` objects. Similarly, each `Connection` object contains a name of the input and output ports that it is connecting.

It is important to notice that while parameters have a constant value, the methods that they call will many times have values determined by other pieces of the workflow. Imagine, for example, a workflow that performs automatic material surface extraction from a scalar field. One can imagine that one piece of the workflow will compute the scalar corresponding to the interface, while another module will actually extract the isosurface. Because of that, every parameter that can be set manually by typing in the value can also be set by the workflow execution. The distinction between regular input ports and parameter settings, then, should only exist in the Workflow layer. As described in Section 6.4, the interpreter's instantiation of a workflow (described in Section 9.1.3) involves compiling away the parameters into additional modules and connections, such that for the actual Python modules, there is no visible distinction between the two cases, making for a simpler implementation of user-defined modules.

In general, the Workflow layer has no notion of *dynamic*, or runtime, information of the modules. In that sense, it is similar to a syntax tree in a compiler implementation. It also has no information about the (static) subtyping relationship of the modules in VisTrails. All this semantic information is delegated to the Execution layer.

9.1.3 The Execution Layer

The Execution layer is responsible for managing the data and algorithms related to the semantic content of modules in VisTrails. It is the layer that will perform the instantiation of a particular workflow description (in the form of a `Workflow` object) from the Workflow layer into actual executable Python. It also contains the typechecking rules for determining which connections are allowed. It consists essentially of a workflow interpreter, a module registry and a package repository.

The module registry contains all information about the executable modules currently installed in a running VisTrails process, including pointers to the actual Python classes corresponding to the workflow modules. Because new user-defined modules have to explicitly describe their parent classes in their definition, the module registry can perform the necessary typechecking of connections, as described in Section 6.1.1.

The VisTrails interpreter is explained in significant detail in Chapter 6, so we restrict ourselves to discussing issues relevant to the overall architecture of VisTrails. As described in Section 6.5, it is possible for a running workflow to document changes in the state of the workflow (the standard

example being an important camera angle found through interactive exploration). When such a change is reported to VisTrails, it first goes to the interpreter. This provides a point of separation between the interface that is visible to designers of user-defined modules and the VisTrails internal structures, allowing us to change these without requiring users to have to fix their source code.

9.1.3.1 The VisTrails Package Repository

Basic to the infrastructure VisTrails provides, the workflows created in the system will sometimes involve many different software packages. A typical example comes from a paper comparing different techniques for tetrahedral mesh generation [74]. A vistrail from such a comparison will reference modules from many different packages, each of which with their own system library dependencies. Since one of the stated goals of VisTrails is the facilitation of sharing results and reproducibility, this is a serious problem: if users have easy access to the workflow specifications in VisTrails but cannot actually run them because of the myriad dependencies, the utility of the system is compromised.

In order to alleviate this problem, VisTrails provides an interface to access a *package repository*. This repository contains a catalog of user-defined packages and pointers to the necessary installation files. When a workflow that uses a missing user-defined module is referenced, the module registry requests the appropriate module package to be installed. This way, developers of user-defined modules need only to submit the module packages they want to make available for other VisTrails users.

This alleviates the problem of distributing the Python source files that define the modules that comprise the VisTrails workflows. However, many of these Python source files depend themselves on other libraries that might not be VisTrails module packages. Instead of requiring the user to find, compile and install the required binaries, VisTrails provides a simple wrapper around system-level software managers present in modern operating systems such as YUM in Fedora GNU/Linux [2] and APT in Ubuntu GNU/Linux [5] and Debian GNU/Linux [1]. Module packages in VisTrails explicitly write out the required system-level software, and VisTrails invokes the necessary installations. Workflows that exclusively use packages with this functionality are, then, very easy to share as far as dependencies are concerned: VisTrails transparently identifies and installs all necessary binaries.

This functionality is quite simple from a user's perspective, as can be seen in Figure 9.2. Because different operating systems have different names for these software packages, and because the name "package" was already in use in VisTrails, we call these system-level packages *bundles*. Support for bundles in different operating systems can be added by writing a small amount of code that detects an operating system and mediates the calls to the installation procedures. Currently, VisTrails

```
from core.bundles import py_import

# "regular" python would simply be
# import vtk
vtk = py_import('vtk', {'linux-ubuntu': 'python-vtk',
                        'linux-fedora': 'vtk-python'})
```

Figure 9.2: A snippet of code showing how package developers provide support for automatic dependency installation of software. VisTrails takes advantage of the combination of dynamic loading and reflection in Python to provide simple access to the underlying operating system software managers. Developers explicitly state the dependencies in their Python code. VisTrails then queries the system and, if necessary, requests the appropriate software installation.

supports Fedora GNU/Linux and Ubuntu GNU/Linux. The idea of the VisTrails package repository and the bundle system is similar to the PLaneT system available for the PLT Scheme programming language [82].

9.2 The `VisTrails gui` Component

The `VisTrails gui` component comprises all portions of VisTrails that receive direct user input. It presents information from `VisTrails core` and translates user interaction events into actions that represent the user's exploration process. GUI events that involve changing workflow objects get translated to actions that are directly sent to the history layer in `VisTrails core` and incorporated in the user's exploration process.

In VisTrails, the undo and redo commands are entirely delegated to `VisTrails core`, and interpreted in the following way: an undo command triggers a step "up" in the version tree, together with pushing an identifier for the undone version to a private stack. The redo command simply pops versions off this stack, and anytime the user navigates the version tree explicitly by clicking on versions, the redo stack is simply cleared.

9.2.1 GUI Updates During Workflow Execution

During the execution of a workflow, VisTrails provides visual feedback of the modules that are currently being executed. The base `Module` class includes a set of procedure hooks to notify beginning of module execution, progress, and end. These are used in the main application to provide visual feedback of the execution in the workflow view. However, in the execution of VisTrails workflows in server modes, they can be used to, for example, update a webpage with the progress of long-running workflows. It should be noted that these updates require establishing a mapping relation between modules being executed and modules in a `VisTrails core` representation of a workflow. As described in Chapter 6, multiple modules in a workflow view might coalesce into

a single invocation of a module if they generate the same signature. This mapping needs to be correctly accounted for.

9.3 The `VisTrails db` component

The internal database architecture implemented in VisTrails goes beyond the scope of this document. The relevant algorithms, features and techniques are all part of David Koop's doctoral studies. We describe it only in as much detail as necessary to clarify our decisions for the rest of VisTrails's architecture. Its most important feature is a description of the storage scheme that is agnostic to the actual underlying technology used to save and load vistrails. This way, VisTrails can switch between a file-based local storage scheme to a more advanced, multi-user RDBMS engine with very limited changes to the rest of the application. In addition, `VisTrails db` houses the technology to make graph-structural indexing and searching practical in VisTrails.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

This thesis presented the design and implementation of some main aspects of VisTrails. The original goal of the system was to explore the interplay between a consistent and thorough tracking of the process provenance behind scientific visualizations in a visual programming environment and the reproducibility of these visualizations. The system has, however, naturally grown into a full-fledged environment for managing rapidly-changing workflows in exploratory tasks.

One of the most important high-level conclusions from this work is that reproducibility fundamentally requires a whole-system approach - it only takes one aspect of the process management to be missing to make results not reproducible. In the current version of VisTrails, the weakest aspect is the inability to allow different module package versions to exist concurrently. If one vistrail contains different module versions, at least one of them will not be exactly reproducible. While the approach described in Section 6.6 is an acceptable remedial technique, it would be preferable to be able to execute legacy versions directly. However, this requires a fundamental change in the way VisTrails works — imagine, for example, that legacy libraries might require older kernel versions. Even though recent virtualization techniques might make this possible, it is far from a simple issue, and one that we leave as future work.

While the execution engine of VisTrails is powerful enough for many interesting workflows to be executed, there is no question that computer science is quickly moving to a many-core, parallel environment. Automatic parallelization of execution runs will have a rippling effect throughout most of the system, from how to properly log such an execution to how to manage libraries and data across different systems to caching implications. We remain hopeful that the rich literature in distributed programming languages and operating systems will provide guidance in this process.

The history data itself has been shown to be very useful to present interesting applications in Chapters 4 and 5. Still, we can see many future avenues that include further new applications of the data. To give but one example, it should be possible to use different user's histories to compare their exploration processes, and suggest portions of the visualization space that they might have missed.

Alternatively, a deployment of a history tracking mechanism in a larger institution might be useful to automatically suggest users that are working on similar visualizations.

The visual representation of the history tree as currently portrayed in VisTrails is by no means the only possible alternative. It would be interesting to investigate other visual metaphors, perhaps using additional information from the exploration process and the space of visualizations itself.

We believe that the automatic, permanent capture of the exploration history is only going to make more sense, as storage and computing power becomes ever cheaper, but humans interact with computers and data at the same speed. One area of research that we have not touched here, but that is one that can be critically important for a wider adoption of the provenance mechanisms we suggest here, is a careful study of the implications of a permanent document history in terms of security and privacy. Similarly, it is important to understand the limitations of the suggested model under data loss (or restricted access). Questions exist such as whether it is possible to recover pieces of the exploration process, or which parts are safely erasable. While these questions have started to receive attention in the context of keeping provenance of data products, an investigation of the consequences of the change-based representation advocated here remains as future work.

REFERENCES

[1] The Debian GNU/Linux operating system. http://www.debian.org.

[2] The Fedora Project. http://fedoraproject.org.

[3] *Git's User Manual.*

[4] matplotlib. Available at http://matplotlib.sourceforge.net.

[5] The Ubuntu GNU/Linux operating system. http://www.ubuntu.com.

[6] *The VisTrails User's Guide, Version 1.3.* Available at http://sourceforge.net/projects/vistrails/files/vistrails/vistrails-usersguide-1.3-rev198.pdf. Last accessed on December 14th, 2009.

[7] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting With Computers*, 4(3):317–342, 1992.

[8] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, 1993.

[9] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In *International Provenance and Annotation Workshop (IPAW)*, pages 118–132, 2006.

[10] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. Vistrails: Enabling interactive, multiple-view visualizations. In *IEEE Visualization*, pages 135–142, 2005.

[11] Laszlo Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[12] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. on CHI*, 1(3):269–294, 1994.

[13] Wes Bethel, Cristina Siegerist, John Shalf, Praveenkumar Shetty, T. J. Jankun-Kelly, Oliver Kreylos, and Kwan-Liu Ma. VisPortal: Deploying grid-enabled visualization tools through a web-portal interface. In *Third Annual Workshop on Advanced Collaborative Environments*, 2003.

[14] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[15] Ken Brodlie, David Duce, Julian Gallop, Musbah Sagar, Jeremy Walton, and Jason Wood. Visualization in grid computing environments. In *IEEE Visualization*, pages 155–162, 2004.

[16] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. Managing the evolution of dataflows with visTrails *(Extended Abstract)*. In *IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow)*, 2006.

[17] Steven P. Callahan, Juliana Freire, Carlos E. Scheidegger, Claudio T. Silva, and Huy T. Vo. Towards provenance-enabling paraview. In *Proceedings of IPAW*, 2008.

[18] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Commun. ACM*, 23(7):396–410, 1980.

[19] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1983.

[20] Ed Huai-hsin Chi, Phillip Barry, John Riedl, , and Joseph Konstan. Principles for information visualization spreadsheets. *IEEE Computer Graphics and Applications*, 18(4):30–38, 1998.

[21] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization*, pages 167–174, 1998.

[22] Hank Childs, Eric S. Brugger, Kathleen S. Bonnell, Jeremy S Meredith, Mark Miller, Brad J Whitlock, and Nelson Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization*, pages 190–198, 2005.

[23] Jon Claerbout. Reproducible computational research: A history of hurdles, mostly overcome. Available on http://sepwww.stanford.edu/data/media/public/sep//jon/reproducible.html. Last accessed on July 29th, 2009.

[24] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 26. MIT Press, 2001.

[25] Nokia Corporation. Qt: A cross-platform application and ui framework. Available at http://qt.nokia.com. Last accessed on August 12th, 2009.

[26] Quynh Dang. NIST special publication 800-107: Recommendation for applications using approved hash algorithms. NIST.

[27] Mark Derthick and Steven F. Roth. Enhancing data exploration with a branching history of user operations. *Knowledge-Based Systems*, 14(1):65–74, March 2001.

[28] Andrew Dolgert, Lawrence Gibbons, Christopher D. Jones, Valentin Kuznetsov, Mirek Riedewald, Daniel Riley, Gregory J. Sharp, and Peter Wittich. Provenance in high-energy physics workflows. *Computing in Science and Engineering*, 10(3):22–29, 2008.

[29] James R. Driscoll, Neil Sarnak, Daniel Sleator, and Robert Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1), 1989.

[30] W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. A temporal model for multi-level undo and redo. *CHI Letters*, 2(2), 2000.

[31] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2), June 1989.

[32] Tommy Ellkvist, David Koop, Erik W. Anderson, Juliana Freire, and Claudio T. Silva. Using provenance to support real-time collaborative design of workflows. In *Proceedings of IPAW*, 2008.

[33] Per Cederqvist et al. *Version Management with CVS*. Available at http://ximbiot.com/cvs/manual/. Last accessed on July 31th, 2009.

[34] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *Proceedings of CIDR*, 2003.

[35] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[36] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18, 2006. Invited paper.

[37] James Frew, 2006. Private communication.

[38] James Frew and Rajendra Bose. Earth system science workbench: A data management infrastructure for earth science products. pages 180–189, 2001.

[39] Hongbo Fu, Daniel Cohen-Or, Gideon Dror, and Alla Sheffer. Upright orientation of man-made objects. *ACM Trans. Graph.*, 27(3), 2008.

[40] Xiaobin Fu, Jay Budzik, and Kristian J. Hammond. Mining navigation history for recommendation. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 106–112, 2000.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[42] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[43] O. Gilson, N. Silva, P. Grant, M. Chen, and J. Rocha. VizThis: Rule-based semantically assisted information visualization. Poster, in Proceedings of SWUI 2006, 2006.

[44] The GNU Project. *GNU binutils 2.19 Manual*. Available at http://sourceware.org/binutils/docs-2.19/. Last accessed on August 19th, 2009.

[45] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[46] J. Hastad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, (182):105–142, 1999.

[47] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, July 2003.

[48] James Hays and Alexei A Efros. Scene completion using millions of photographs. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 26(3), 2007.

[49] Jeffrey Heer, Jock D. Mackinlay, Chris Stolte, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Trans. Vis. Comp. Graph.*, 2008.

[50] Jeffrey Heer, Fernanda B. Viégas, and Martin Wattenberg. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)*, pages 1029–1038, 2007.

[51] Haym Hirsh, Chumki Basu, and Brian D. Davison. Learning to personalize. *Communications of ACM*, 43(8):102–106, 2000.

[52] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, 2nd. edition, 2005.

[53] IBM. OpenDX. http://www.research.ibm.com/dx.

[54] Takeo Igarashi and John F. Hughes. A suggestive interface for 3d drawing. In *ACM symposium on User interface software and technology (UIST)*, pages 173–181, 2001.

[55] Apple Inc. Spotlight. Available at http://www.apple.com/macosx/what-is-macosx/#spotlight. Last accessed on July 30th, 2009.

[56] VisTrails Inc. Provenance explorer for autodesk maya. Available at http://www.vistrails.com/maya.html.

[57] T. J. Jankun-Kelly, Oliver Kreylos, Kwan-Liu Ma, Bernd Hamann, Kenneth I. Joy, John M. Shalf, and E. Wes Bethel. Deploying web-based visual exploration tools on the grid. *IEEE Computer Graphics and Applications*, 23(2):40–50, 2003.

[58] T. J. Jankun-Kelly and Kwan-Liu Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, July/September 2001.

[59] T. J. Jankun-Kelly, Kwan-Liu Ma, and Michael Gertz. A model and framework for visualization exploration. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):357–369, March/April 2007.

[60] Haim Kaplan. *Handbook on Data Structures and Applications*, chapter Persistent Data Structures. CRC Press, 1995.

[61] Gordon Kindlmann. Teem. http://teem.sourceforge.net.

[62] Kitware. Paraview. http://www.paraview.org.

[63] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multi-dimensional transfer functions for interactive volume rendering. *IEEE Trans. Vis. and Comp. Graph.*, 8(3):270–285, July 2002.

[64] Benjamin Korvemaker and Russell Greiner. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 230–235, 2000.

[65] Matthias Kreuseler, Thomas Nocke, and Heidrun Schumann. A history mechanism for visual data mining. In *Proceedings of IEEE Information Visualization Symposium*, pages 49–56, 2004.

[66] David Kurlander and Eric A. Bier. Graphical search and replace. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 113–120, New York, NY, USA, 1988. ACM Press.

[67] David Kurlander and Steven Feiner. A history-based macro by example system. In *UIST '92: Proceedings of the 5th annual ACM symposium on User interface software and technology*, pages 99–106, New York, NY, USA, 1992. ACM Press.

[68] Jean-François Lalonde, Derek Hoiem, Alexei A. Efros, Carsten Rother, John Winn, and Antonio Criminisi. Photo clip art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, 26(3), 2007.

[69] Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

[70] George B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1):50–87, 1986.

[71] Mark Levoy. Spreadsheet for images. In *Proceedings of SIGGRAPH*, pages 139–146, 1994.

[72] Henry Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[73] L. Lins, D. Koop, E. Anderson, S. P. Callahan, E. Santos, C. Scheidegger, J. Freire, and C. Silva. Examining statistics of workflow evolution provenance: A first study. In *Proceedings of the 20th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2008.

[74] Mario Lizier, Jason Shepherd, Luiz Gustavo Nonato, Joao Comba, and Claudio Silva. Comparing techniques for tetrahedral mesh generation. In *Proceedings of the In Inaugural International Conference of the Engineering Mechanics Institute*, 2008.

[75] Riverbank Computing Ltd. Pyqt: a set of python bindings for qt. Available at http://www.riverbankcomputing.co.uk/software/pyqt/intro. Last accessed on August 12th, 2009.

[76] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[77] David Luebke, Martin Reddy, Jonathan Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level Of Detail for 3D Graphics*. Morgan Kaufmann, 2002.

[78] Kwan-Liu Ma. Visualizing visualizations: User interfaces for managing and exploring scientific visualization data. *IEEE Comput. Graph. Appl.*, 20(5):16–19, 2000.

[79] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, April 1986.

[80] Rob MacLeod. SCIRun/BioPSE: Integrated problem solving environment for bioelectric field problems and visualization. In *Proceedings of the Int. Symp. on Biomed. Imag.*, pages 640–643, 2004.

[81] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, W. Ryall, J. Seims, and S. Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In *ACM SIGGRAPH*, pages 389–400, 1997.

[82] Jacob Matthews. PLaneT: Automatic package distribution for PLT Scheme. http://docs.plt-scheme.org/planet/index.html (last accessed on July 24th, 2009).

[83] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, pages 787–795, 2004.

[84] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 117–128, 2002.

[85] Mercury Computer Systems. Amira. http://www.amiravis.com.

[86] MeVis Research. MeVisLab. http://www.mevislab.de.

[87] Microsoft. Intellisense. http://msdn.microsoft.com/en-us/library/hcw1s69b(VS.80).aspx.

[88] Ha Quang Minh, P. Niyogi, , and Y. Yao. Mercer's theorem, feature maps, and smoothing. In *Proceedings of Computational Learning Theory (COLT)*, 2006.

[89] Luc Moreau and Bertram Ludäscher et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 2007.

[90] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers, and Patrick Paulson. The Open Provenance Model, http://eprints.ecs.soton.ac.uk/14979/1/opm.pdf, 2008.

[91] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, , and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[92] Tamara Munzner, Chris Johnson, Robert Moorhead, Hanspeter Pfister, Penny Rheingans, and Terry S. Yoo. NIH-NSF visualization research challenges report summary. *IEEE Computer Graphics and Applications*, 26(2):20–24, 2006.

[93] Brad A. Myers and David S. Kosbie. Reusable hierarchical command objects. In *Proceedings of the ACM Conference on Computer Human Interaction (CHI)*, pages 260–267, 1996.

[94] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, Sept.-Oct. 2003.

[95] Polonsky O., Patane G., Biasotti S., Gotsman C., and Spagnuolo M. What's in an image: Towards the computation of the "best" view of an object. *The Visual Computer*, 21(8):840–847, 2005.

[96] Thomas Oinn, Mark Greenwood, Matthew Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

[97] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. Available at http://mercurial.selenic.com/wiki/MercurialBook. Last accessed on July 31st, 2009.

[98] John K. Ousterhout. Tcl: An embeddable command language. Technical Report UCB/CSD-89-541, EECS Department, University of California, Berkeley, Nov 1989.

[99] Steven G. Parker and Christopher R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 1995.

[100] Provenance challenge. http://twiki.ipaw.info/bin/view/Challenge.

[101] Wolfram Research. Mathematica 7.

[102] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, (74):358–366, 1953.

[103] Jonathan C. Roberts. On Encouraging Multiple Views for Visualization. In Ebad Banissi, Farzad Khosrowshahi, and Muhammad Sarfraz, editors, *IV'98 - Proceedings International Conference on Information Visualization*, pages 8–14. IEEE Computer Society, July 1998.

[104] Jonathan C. Roberts. Multiple-View and Multiform Visualization. volume 3960, pages 176–185. IS&T and SPIE, January 2000.

[105] David Roundy. Darcs. http://www.darcs.net.

[106] Sam T. Roweis and Lawrence K. Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, 2000.

[107] Szymon Rusinkiewicz. trimesh2. Available at http://www.cs.princeton.edu/gfx/proj/trimesh2/.

[108] Emanuele Santos, Lauro Lins, James P. Ahrens, Juliana Freire, and Claudio T. Silva. VisMashup: Streamlining the creation of custom visualization applications (to appear). *IEEE Trans. Vis. Comp. Graph.*, 2009.

[109] Carlos Scheidegger, David Koop, Emanuele Santos, Huy Vo, Steven Callahan, Juliana Freire, and Claudio Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 2007.

[110] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit*. Kitware Inc, 2007.

[111] Higmet Senay and Eve Ignatius. A knowledge-based system for visualization design. *IEEE Comp. Graph. and Appl.*, 14(6), 1994.

[112] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 3rd Edition*. McGraw-Hill Book Company, 1997.

[113] Chris Stolte, Diane Tang, and Pat Hanrahan. Multiscale visualization using data cubes. In *Proceedings of the IEEE Information Visualization Conference*, 2002.

[114] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. Comp. Graph.*, 8(1):1–14, January–March 2002.

[115] Etzard Stolte, Christoph von Praun, Gustavo Alonso, and Thomas R. Gross. Scientific data repositories: Designing for a moving target. In *Proceedings of ACM SIGMOD*, pages 349–360, 2003.

[116] Swivel. http://www.swivel.com.

[117] S. Takahashi, I. Fijishiro, Y. Takeshima, and T Nishita. A feature-driven approach to locating optimal viewpoints for volume visualization. In *IEEE Visualization*, pages 495–502, 2005.

[118] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323, 2000.

[119] M. Tory and T. Moller. Evaluating visualizations: do expert reviews work? *Computer Graphics and Applications, IEEE*, 25(5):8–11, Sept.-Oct. 2005.

[120] Steve Tsang, Ravin Balakrishnan, Karan Singh, and Abhishek Ranjan. A suggestive interface for image guided 3d sketching. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)*, pages 591–598, 2004.

[121] University of Utah. VisTrails. http://www.vistrails.org.

[122] Craig Upson et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[123] Jarke J. van Wijk. The value of visualization. In *Proceedings of IEEE Visualization*, pages 79–86, Minneapolis, MN, October 2005.

[124] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. ManyEyes: a site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of InfoVis)*, 13(6):1121–1128, 2007.

[125] Jeffrey Scott Vitter. Us&r: A new framework for redoing (extended abstract). *SIGPLAN Not.*, 19(5):168–176, 1984.

[126] Luis von Ahn and Laura Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, 2008.

[127] Udeepta D. Vordoloi and Han-Wei Shen. View selection for volume rendering. In *IEEE Visualization*, pages 487–494, 2005.

[128] Yiya Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, May 1988.