Querying and Creating Visualizations by Analogy

Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, *Member, IEEE*, and Cláudio T. Silva, *Member, IEEE*

Abstract— While there have been advances in visualization systems, particularly in multi-view visualizations and visual exploration, the process of building visualizations remains a major bottleneck in data exploration. We show that provenance metadata collected during the creation of pipelines can be reused to suggest similar content in related visualizations and guide semi-automated changes. We introduce the idea of query-by-example in the context of an ensemble of visualizations, and the use of analogies as first-class operations in a system to guide scalable interactions. We describe an implementation of these techniques in VisTrails, a publicly-available, open-source system.

Index Terms—visualization systems, query-by-example, analogy

1 Introduction

Over the last 20 years, visualization research has emerged as an effective means to help scientists, engineers, and other professionals extract insight from raw data. Visualization techniques are key to understanding complex phenomena, and the field has grown into a mature area with an established research agenda [23]. Software systems have been developed that provide flexible frameworks for creating complex visualizations. These systems can be broadly classified as turnkey applications (e.g., ParaView, VisIt, Amira) [15, 5, 22] and dataflow-based systems (e.g., VTK, SCIRun, AVS, OpenDX) [27, 24, 11, 29]. In this paper, we focus on dataflow systems, since they are more general and often serve as the foundation of turnkey applications (e.g., both ParaView and VisIt are based on VTK).

Most dataflow-based systems have sophisticated user interfaces with visual programming capabilities that ease the creation of visualizations. Nonetheless, the path from "data to insight" requires a laborious, trial-and-error process, where users successively assemble, modify, and execute pipelines [30]. 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 3-D isosurface plots, 2-D 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.

In previous work, we proposed a new provenance model that uniformly captures changes to pipeline and parameter values during the course of data exploration [1, 4]. We showed that 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.

In this paper, 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 new framework that enables the effective reuse of this knowledge to aid both expert and non-expert users in performing data exploration through visualization.

The framework consists of two key components: an *intuitive interface for querying dataflows* and a *novel mechanism for semi-automatically creating visualizations by analogy*. The query interface supports both simple keyword-based and selection queries (e.g., find visualizations created by some user), as well as complex, structure-based queries (e.g., find visualizations that apply simplification before an isosurface computation for irregular grid data sets). The query engine is exposed to the user through an intuitive *query-by-example* interface whereby users query dataflows through the same familiar interface they use to create the dataflows (see Figure 1). This simple, yet powerful approach lets users easily search through a large number of visualizations and identify pipelines that satisfy user-defined criteria.

While the query interface allows users to identify pipelines (and sub-pipelines) that are relevant for a particular task, 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 2 illustrates, our 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. Together, these contributions are a step towards scalable pipeline development and refinement as an integral part of visualization systems.

Contributions and Outline. To the best of our knowledge, this is the first work that leverages provenance information to simplify and automate the construction of new visualizations. The paper is organized as follows. We review related work in Section 2. In Section 3, we define a set of basic operations over sets of dataflows. These operations include computing the difference between two pipelines, updating pipeline definitions, and matching similar pipelines. For the latter, we describe a new algorithm based on neighborhood similarities (Section 5.3). The basic operations are used both in the query-by-example interface and in creating visualizations by analogy, which are presented in Section 4. An implementation of the proposed framework is discussed in Section 5. In Section 6, we present case studies that illustrate how our new dataflow manipulations streamline the process of constructing visualizations, and provide scalable mechanisms for exploring a large number of visualizations. We discuss the potential impact of our work on existing visualization systems in Section 7. We conclude in Section 8 where we outline directions for future work.

2 Related Work

Visualization systems have been quite successful at bringing visualization to a greater audience. Seminal systems such as AVS Explorer and

- *Carlos E. Scheidegger, Huy T. Vo, David Koop and Cláudio T. Silva are with the Scientific Computing and Imaging (SCI) Institute at the University of Utah. email: {cscheid, hvo, dakoop, csilva}@sci.utah.edu.*
- *Juliana Freire is with the School of Computing at the University of Utah. email: {juliana}@cs.utah.edu*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 2 November 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

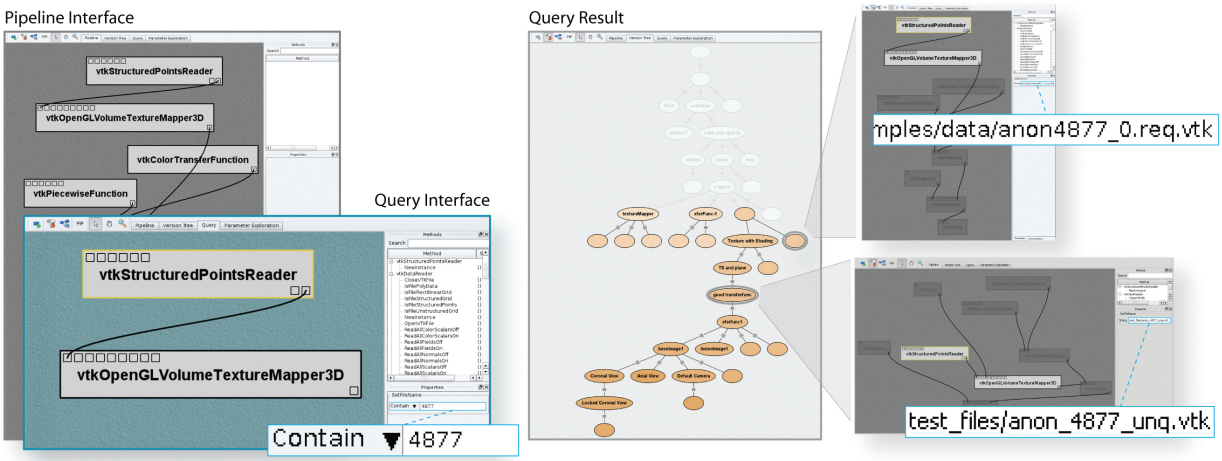

Fig. 1. Querying by example. The interface for building a query over an ensemble of pipelines is essentially the same as the one for constructing and updating pipelines. In fact, they work together: portions of a pipeline can become query templates by directly pasting them onto the Query Canvas. In this figure, the user is looking for a volume rendered image of a file whose name contains the string "4877". The system highlights the matches both at the visualization level (version tree, shown in the middle) and at the module level (shown in the right insets).

Data Explorer [29, 11] enabled domain scientists to create visualizations with minimal training and effort. The early success of these systems led to the development of several alternative approaches. SCIRun [24] focuses on computational steering: the intentional placement of visualization and human intervention in the process of generating simulations. The Visualization Toolkit [27] is a library that directly exposes a powerful dataflow API for several programming languages.

However, as scientific visualization becomes more widely used, several scalability issues have arisen, which range 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 [5, 3] 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.

Another 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 a given result was generated) and for the exploratory process, the sequence of steps followed to design and refine the workflows used to process the data [25, 26, 4]. Provenance mechanisms have also been proposed for visualization-specific systems. Kreuseler et al. [16] proposed a history mechanism for keeping track of parameter values in visual data mining, and Jankun-Kelly et al. [13] recently proposed a formal calculus for parameter changes. VisTrails uses a scheme that uniformly captures both parameter and pipeline changes [1, 4].

As multiple workflows are manipulated in exploratory processes, it is important to provide interfaces that allows users to compare their results. Jankun-Kelly and Ma [12] have proposed a spreadsheet-like interface for quickly exploring the parameter space of a visualization. In the area of user interfaces, Kurlander et al. [18, 17] have presented approaches to streamline the repetitive tasks users often face. The seminal example of a system which uses the same interface to both manipulate and query data is the Query-By-Example database language [31]. We propose a similar approach for querying visualization ensembles in Section 4.2. Graph searching and query languages have also been investigated in database systems [28].

The algorithm we describe for matching two pipelines is similar to a technique developed to match database schemas [21]. It is also reminiscent of well-known variations of PageRank, which is the basis for Google's successful ranking algorithm [2, 19]. Our visualization-by-analogy mechanism shares some of the objectives of programming-by-example techniques [20].

3 Pipeline Operations

Below, we review some terminology and introduce basic pipeline operations that serve as the basis for query-by-example and visualization by analogy.

Definitions. A *visualization system* is a system that provides functionality for graphically displaying data according to a specific set of rules. The programmatic rules for displaying this data constitute a *pipeline*. Executing the pipeline in the visualization system produces a *visualization*. The pipeline is composed of *modules* which define specific operations and *connections* which specify the conceptual flow of data between modules. Each connection links an *output port* of one module (the *source*) with an *input port* of another module (the *destination*). Module state is represented by *module parameters*. We denote the set of all visualization pipelines as $\mathbb{V}$.

Operations as functions on $\mathbb{V}$. One important observation that we leverage throughout the text is that every operation performed on a pipeline (adding and deleting modules, connections and parameters, etc.) can be directly expressed as a (potentially partial) function $f : \mathbb{V} \to \mathbb{V}$. Many of our results depend on making these functions first-class elements in the visualization system.

3.1 Computing Pipeline Differences

Dataflow-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 define $\delta : \mathbb{V} \to \mathbb{V}$ as a function on the space of visualizations, and $\Delta : \mathbb{V} \times \mathbb{V} \to \delta$ as 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. In some sense, we would like to pick the δ_{ab} that minimally changes all other pipelines. We define the distance between p_a and p_b as the number of changes necessary to perform the transformation. We then look for the minimal set of operations that takes p_a to p_b. As we discuss in section 4.1, this is computationally impractical, so we relax the minimality requirement and instead use heuristics.

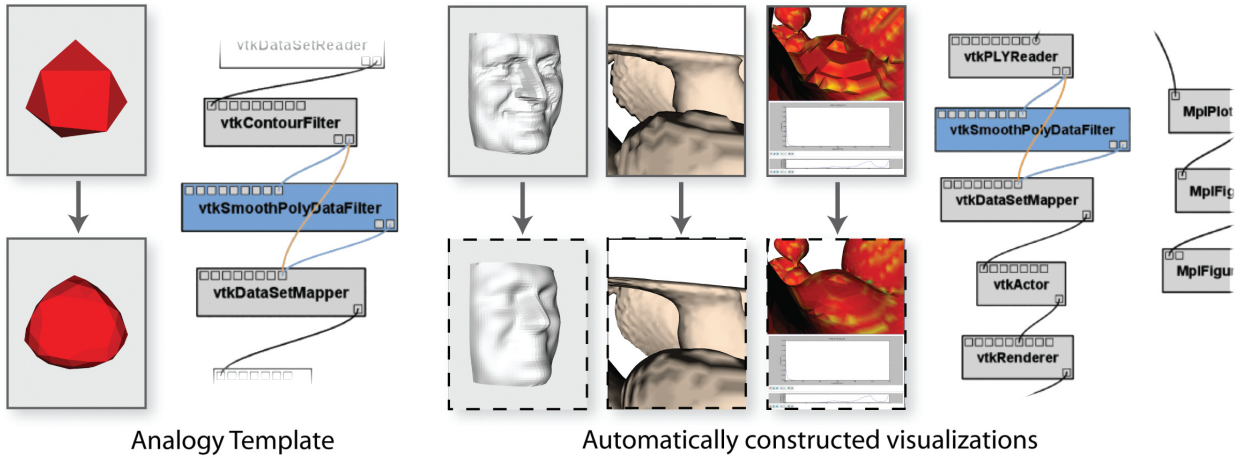

Fig. 2. Visualization by analogy. The user chooses a pair of visualizations to serve as an analogy template. In this case, the pair represents a change where a file downloaded from the WWW is smoothed. Then, the user chooses a set of other visualizations that will be used to derive new visualizations, 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.

We also 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. We demonstrate such an implementation in Section 5.

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 pipelines respect this property, and we would like Δ to be completely general. We start with a simple extension: if δ_{ab} exists, so should δ_{ba}. In fact, we would like

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

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,

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

From now on, we assume that any function that operates on $\mathbb{V}$ has an inverse (note that both functions might still be partial).

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.

3.2 Updating Pipelines

Finding differences is not only a useful technique for analyzing pipelines, but it can also be used to create new visualizations. The idea is similar to applying patches in software 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 δ can *fail* if an element of $D(\delta)$ does not exist in p_a. However, if we allow δ to continue despite one or more operations failing, we can still achieve a partial update.

3.3 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 the best 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

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

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 $(v_a, v_b) \in M$.

4 Scalable Pipeline Manipulation Primitives

Using the concepts defined in the previous section, we now introduce two new primitives for manipulating visualizations.

4.1 Complexity Analysis

The operations described in this section are theoretically hard to compute. Computing a minimal $\Delta(p_a, p_b)$, or matching two pipelines p_a and p_b, is, in general, as hard as solving a subgraph isomorphism.

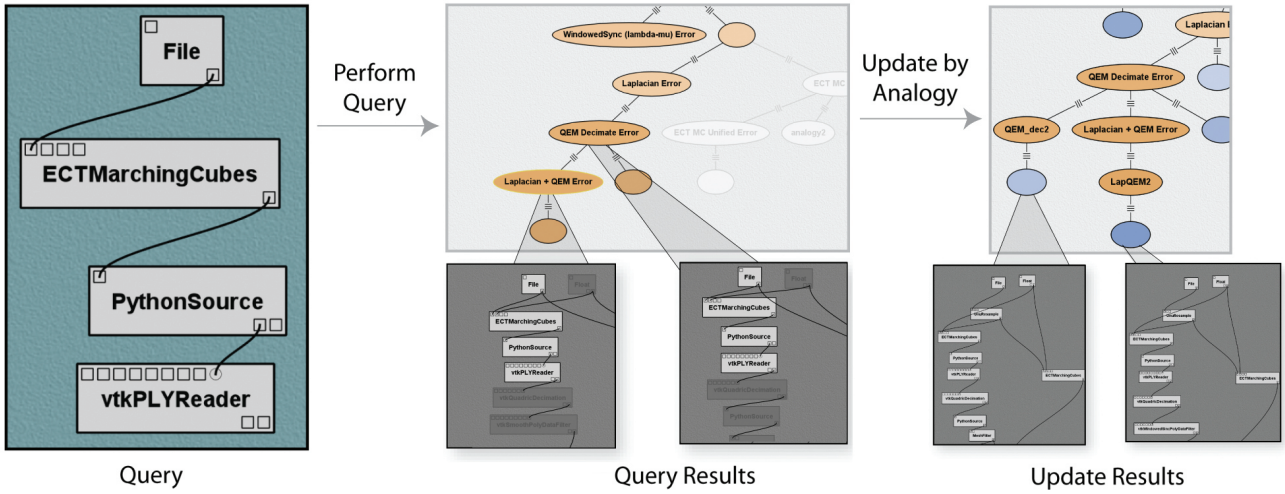

Fig. 3. Query-by-example and analogy-based updates provide a simple way for users to manipulate multiple pipelines simultaneously. In this example, the user selects parts of a query result and updates them all with an analogy that introduces a preprocessing module to the pipeline that supersamples the original dataset.

This problem is trivially reducible from the MAX-CLIQUE problem, a well-known NP-complete problem. Additionally, MAX-CLIQUE is a particularly hard problem: there is no approximation algorithm for it with a subpolynomial approximation factor [9]. Since we cannot get a good approximation, heuristics for both problems are well justified.

In this work, we make use of the information stored in δ functions both to reduce the search space and to increase the effectiveness of these heuristics.

4.2 Query-By-Example

Working with multiple visualizations is problematic if they have to be treated individually. In the process of visualizing different data sets, trying different techniques, and tweaking parameters, a user may create a large number of visualizations. It is clearly impractical to locate those that match certain criteria by examining each individually. To solve the problem of locating pipelines, we introduce the idea of *query-by-example* for visualizations. Instead of formulating the search criteria in a structured language, a user builds a pipeline fragment that contains the desired features. The exact same interface used in building a pipeline can be used for building a query, which means that a user familiar with building pipelines can easily query them. Figure 1 shows an excerpt of the query-by-example functionality.

Our algorithm is based on the observation that searching all pipelines for a given pattern is equivalent to determining whether a candidate pipeline matches the pattern. Once a query, represented as a pipeline fragment, is constructed, we can use the pipeline matching algorithm on each candidate pipeline to determine if it satisfies the query. Depending on user preferences, we can require an exact or an approximate match. While each element of the query pipeline (modules, connections, parameters, etc.) needs to be included in the match, a candidate pipeline that contains *more* elements than those in the query pipeline still satisfies the query.

It should be noted that differences can help optimize our matching. For example, suppose that we have a given query pipeline p_q and two candidate pipelines p_a and p_b. If we find that p_a satisfies the query, and we know δ_{ab}, we can check to see if the domain context $D(\delta_{ab})$ contains any elements that matched p_q. If it does not, we know that p_b also matches. Similarly, if p_a does not match p_q and $R(\delta_{ab})$ does not contain necessary elements for matching p_q, we know that p_b will not satisfy the query. Thus, we can determine all pipelines that satisfy our query by iteratively matching and updating the matches based on differences.

4.3 Visualization by Analogy

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. Figure 2 illustrates the process of creating visualizations by analogy.

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 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.

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 3.1, 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, as described in Section 3.3. 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 $p_d = \delta^*_{cb}(p_c)$

5 Implementation

To implement the scalable manipulation primitives introduced in Section 4, we use the freely available VisTrails system. VisTrails automatically captures the evolution of workflows which allows straightforward implementations of the pipeline operations presented in Section 3. We provide a quick overview of some key concepts in VisTrails; the reader is referred to [1, 4] for more details.

In VisTrails, as a user constructs a visualization, the entire history of manipulation is transparently stored in the *version tree* (the term *vistrail* is used interchangeably). Each action f that modifies the pipeline (e.g., adding or deleting a module, connecting modules, or changing

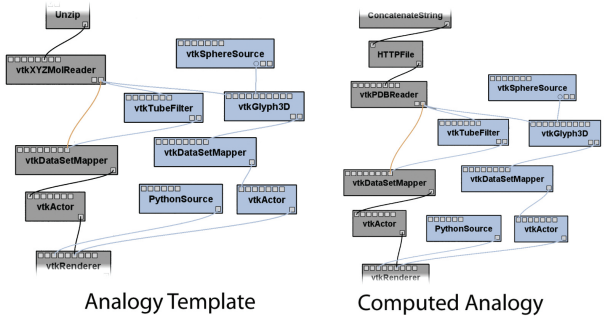

Fig. 4. Example of an analogy between pipelines where there is no perfect module matching. The difference in the left pipeline pair is transferred to the right pipeline pair. Note, however, that the module names are not the same—the system must find the most likely pairing based on the similarity measure described in Section 5.3.

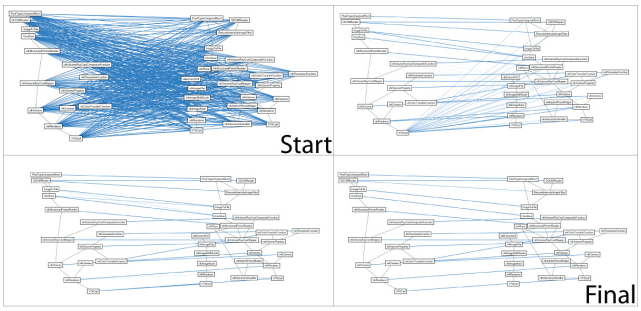

Fig. 5. Example matching generated by the pipeline matching algorithm. Thicker edges correspond to stronger correspondences. Notice that the correspondences get progressively better as the algorithm iterates. This matching corresponds to Example 2 in Section 6.

a parameter) is represented explicitly as a function $f : \mathbb{V} \to \mathbb{V}$, where $\mathbb{V}$ is the space of all possible visualizations. A pipeline is then the composition of these functions and is materialized by applying the resulting function to the empty visualization.

5.1 Pipeline Differences

In a vistrail, the straightforward application of the action-based formalism allows the computation of simple differences. When $p_a < p_b$, $\Delta(p_a, p_b)$ is the sequence of actions from p_a to p_b which can be read directly from the vistrail. In addition, we have implemented the inverse operation of f for each type of operation in VisTrails so δ_{ba} is also easily constructed. However, it is likely that we wish to compute a difference between pipelines that are not related in such a simple manner. Specifically, suppose that $p_a \not< p_b$ and $p_b \not< p_a$. Note that there exists some p_c (possibly the empty pipeline, which is in general the least common ancestor of both p_a and p_b) such that $p_c < p_a$ and $p_c < p_b$. Then,

$$\delta_{ab} = \delta_{ac}\delta_{cb} = \delta_{ca}^{-1}\delta_{cb}$$

Thus, we can find $\Delta(p_i, p_j)$ for any two pipelines, even if they are not directly related.

5.2 Pipeline Updates

Pipeline updates are also easily computed by taking the action-based representation of a pipeline and appending the new actions given by a δ. Although it is always possible to append the new actions in a vistrail, the resulting sequence of actions may be invalid. As noted earlier, this update can fail if the domain context of δ does match p_a. More specifically, each operation in δ can succeed or fail based on whether the elements to be modified or deleted exist in p_a.

5.3 Pipeline Matching

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 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 [6] between nodes may succeed in finding a map between nodes, we would like to enforce some connectivity 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 strikes a balance between the locality of pairwise compatibility and the overall similarity of the neighborhood. This definition seems circular, but, surprisingly, it leads to a very simple and elegant matching technique based on the dominant eigenvector of a Markov chain [19].

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:

$$\begin{aligned} \pi_{k+1} &= \alpha A(G)\pi_k + (1-\alpha)c(G) \\ &= M_G \pi_k \end{aligned} \tag{1}$$

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 1 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

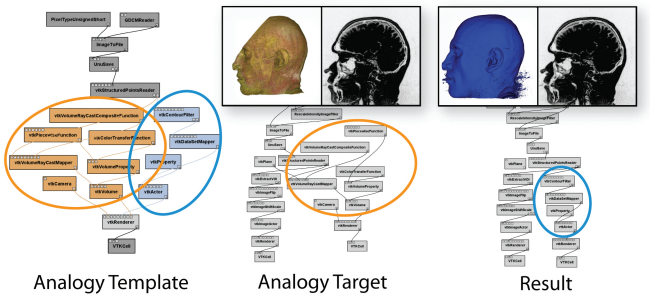

Fig. 7. Switching the rendering technique by analogy. The analogy template on the left specifies that volume rendering modules should be replaced by isosurfacing ones. The analogy target and the resulting pipeline are shown, together with the resulting visualization.

$(1, \alpha, \alpha^2, \cdots)$ [19], and so the iteration is exactly the power method [8] 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 5 illustrates how the matchings are refined as the mapping algorithm iterates.

5.4 Query-by-example

Recall that the benefit of query-by-example is that users do not have to learn a query language or a new interface to find matching pipelines. Our implementation presents the same interface used in building a pipeline as it does for querying an ensemble of pipelines, as shown in Figure 1. A user constructs a query pipeline by dragging modules from a list of available modules or copying and pasting pieces of existing pipelines. Parameters and connections can also be specified in a similar manner. When the user executes the query, the system searches the current version tree for all pipelines that match that query.

As discussed in Section 4, we want to find pipelines that contain the query pipeline. Currently, this matching is computed on a per pipeline basis. Specifically, for each pipeline, we topologically sort the vertices of the graph induced by the pipeline and match the vertices of the query graph. If all vertices match, we return the candidate pipeline as a match. All matches are selected and highlighted in the version tree so that users can quickly see query results. Selecting a version will display the pipeline with the portion of the pipeline that matched the query highlighted.

5.5 Visualization by analogy

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 applied to another pipeline. 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 but even easier to use. 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 mirrors the algorithm described in Section 4.3. 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 in Section 5.3 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.

6 Case Studies

We present three examples that illustrate the proposed primitives.

Example 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 (highlighted in Figure 3) 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.

Example 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 7 renders an ITK [10] scalar field in VTK [27], using the Teem tools [14] 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.

Example 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 6, 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

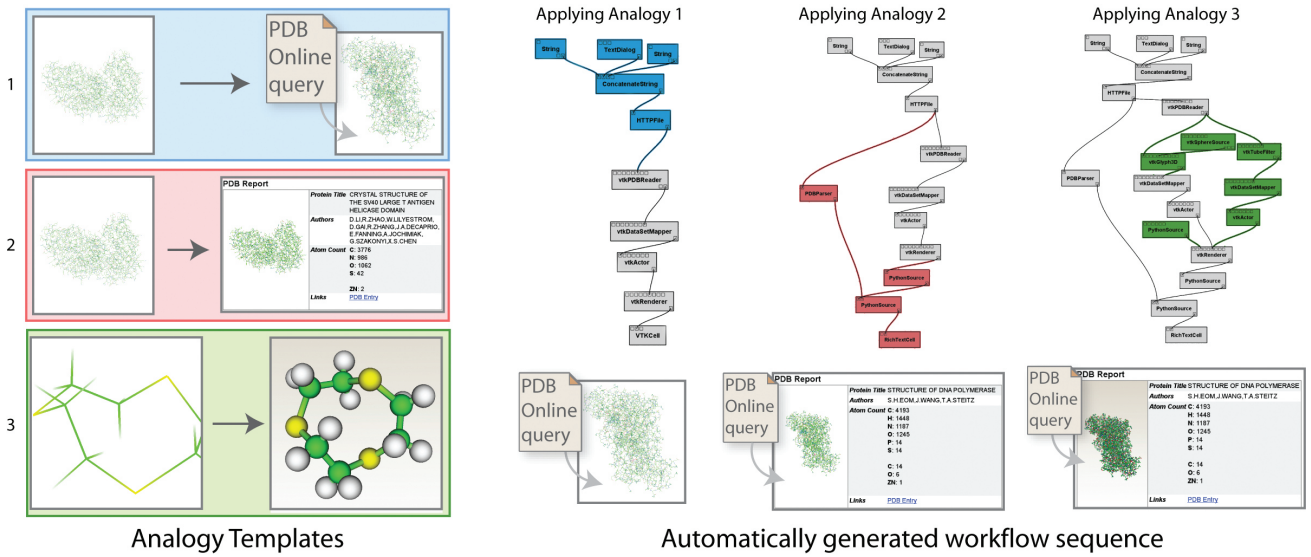

Fig. 6. Creating complex pipelines by chaining simple analogies. From three simple examples, the user creates a complex visualization that creates a web page with enhanced molecule rendering, whose results are fetched from the Protein Database, an online macromolecular database.

data from an online database, create an enhanced visualization for that protein, and finally publish the results as an HTML report.

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.

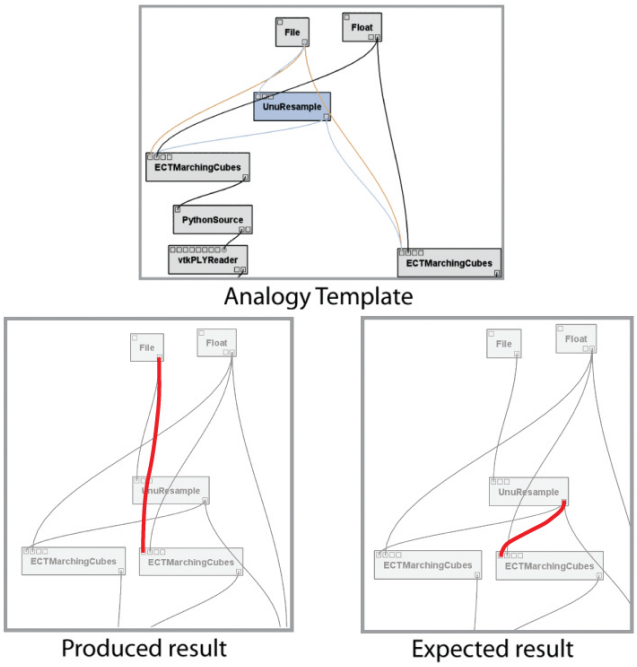

Fig. 8. A situation where creating pipelines by analogy fails. The intended effect when defining the analogy was to replace the raw file with a preprocessing step. Note, however, that there still is one lingering connection, highlighted in red.

7 Discussion

We argue that both query-by-example and visualization by analogy are useful operations that provide efficient solutions for what are otherwise manual, time-consuming tasks. The basic operations introduced in Section 3 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 [7]), 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 multiple results. Figure 8 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. as an avenue for future work. Furthermore, our current implementation finds the best mapping in a greedy fashion, on a per-module basis. 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. Query-by-example represents an intuitive way for users to query pipelines. One could imagine a querying tool that narrows results as the query is built (e.g., similar to auto-completion). Also, to extend our analogy tool, users' input could be used to guide 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.

8 Conclusions and Future Work

We have described a new framework that leverages visualization provenance to simplify the construction of new visualizations. This framework provides scalable and easy-to-use primitives for querying pipeline ensembles and for creating multiple visualizations by analogy. We have also proposed efficient algorithms and intuitive interfaces for realizing these primitives in a visualization system.

There are many avenues for future work. The use of domain-specific distance measures between pipelines and modules may be useful for customizing analogy generation in some domains (for example, for transfer function design and comparison). We are currently investigating machine learning techniques for automatically determining common pipeline operations on a large database of visualizations, allowing templates to also be determined automatically.

Acknowledgments

We acknowledge the generous help of many colleagues and collaborators. Suresh Venkatasubramanian helped with discussions on graph matching and complexity. Erik Anderson and João Comba graciously provided their vistrails for this work. Chems Touati and Steven Callahan helped produce the video and figures. This work uses a number of existing open-source software and data repositories, including Teem (http://teem.sourceforge.net), VTK (http://www.vtk.org), ITK (http://www.itk.org), trimesh2 (http://www.cs.princeton.edu/gfx/proj/trimesh2/), and the RCSB Protein Database. This work was funded by the National Science Foundation, the Department of Energy, and an IBM Faculty Award.

References

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

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

[3] K. Brodlie, D. Duce, J. Gallop, M. Sagar, J. Walton, and J. Wood. Visualization in grid computing environments. In *Proceedings of IEEE Visualization*, pages 155–162, 2004.

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

[5] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization*, pages 190–198, 2005.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 26. MIT Press, 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*, chapter 5. Addison-Wesley, 1995.

[8] G. H. Golub and C. F. V. Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, MD, USA, 3rd. edition, 1996.

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

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

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

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

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

[14] G. Kindlmann. Teem. http://teem.sourceforge.net.

[15] Kitware. ParaView. http://www.paraview.org.

[16] M. Kreuseler, T. Nocke, and H. Schumann. A history mechanism for visual data mining. In *Proceedings of IEEE Information Visualization Symposium*, pages 49–56, 2004.

[17] D. Kurlander and E. A. Bier. Graphical search and replace. In *Proceedings of SIGGRAPH 1988*, pages 113–120, 1988.

[18] D. Kurlander and S. Feiner. A history-based macro by example system. In *Proceedings of UIST 1992*, pages 99–106, 1992.

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

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

[21] S. Melnik, H. Garcia-Molina, and E. 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.

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

[23] T. Munzner, C. Johnson, R. Moorhead, H. Pfister, P. Rheingans, and T. S. Yoo. NIH-NSF visualization research challenges report summary. *IEEE Computer Graphics and Applications*, 26(2):20–24, 2006.

[24] S. G. Parker and C. 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.

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

[26] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 2007. To appear.

[27] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Kitware Inc, 2007.

[28] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2002.

[29] C. Upson, J. Thomas Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[30] J. J. van Wijk. The value of visualization. In *Proceedings of IEEE Visualization*, pages 79–86, 2005.

[31] M. Zloof. Query-by-example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.