MANAGING PROVENANCE FOR KNOWLEDGE DISCOVERY AND REUSE

by

David Allen Koop

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

Computer Science

School of Computing

The University of Utah

May 2012

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **David Allen Koop**

has been approved by the following supervisory committee members:

Juliana Freire	, Chair	**7/18/2011** Date Approved
Cláudio T. Silva	, Member	**7/18/2011** Date Approved
Valerio Pascucci	, Member	**7/18/2011** Date Approved
Susan Davidson	, Member	**7/18/2011** Date Approved
Matthias Troyer	, Member	**7/18/2011** Date Approved

and by **Alan Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Serving as a record of what happened during a scientific process, often computational, provenance has become an important piece of computing. The importance of archiving not only data and results but also the lineage of these entities has led to a variety of systems that capture provenance as well as models and schemas for this information. Despite significant work focused on obtaining and modeling provenance, there has been litte work on managing and using this information. Using the provenance from past work, it is possible to mine common computational structure or determine differences between executions. Such information can be used to suggest possible completions for partial workflows, summarize a set of approaches, or extend past work in new directions. These applications require infrastructure to support efficient queries and accessible reuse.

In order to support knowledge discovery and reuse from provenance information, the management of those data is important. One component of provenance is the specification of the computations; workflows provide structured abstractions of code and are commonly used for complex tasks. Using change-based provenance, it is possible to store large numbers of similar workflows compactly. This storage also allows efficient computation of differences between specifications. However, querying for specific structure across a large collection of workflows is difficult because comparing graphs depends on computing subgraph isomorphism which is NP-Complete. Graph indexing methods identify features that help distinguish graphs of a collection to filter results for a subgraph containment query and reduce the number of subgraph isomorphism computations. For provenance, this work extends these methods to work for more exploratory queries and collections with significant overlap. However, comparing workflow or provenance graphs may not require exact equality; a match between two graphs may allow paired nodes to be similar yet not equivalent. This work presents techniques to better correlate graphs to help summarize collections.

Using this infrastructure, provenance can be reused so that users can learn from their own and others' history. Just as textual search has been augmented with suggested completions based on past or common queries, provenance can be used to suggest how computations can be completed or which steps might connect to a given subworkflow. In addition, provenance can help further science by accelerating publication and reuse. By incorporating provenance into publications, authors can more easily integrate their results, and readers can more easily verify and repeat results. However, reusing past computations requires maintaining stronger associations with any input data and underlying code as well as providing paths for migrating old work to new hardware or algorithms.

This work presents a framework for maintaining data and code as well as supporting upgrades for workflow computations.

To my parents

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisors, Juliana Freire and Cláudio Silva, for their guidance, support, and direction throughout my work. I would also like to thank the other members of my committee for their help in advancing my work: Valerio Pascucci, Susan Davidson, and Matthias Troyer have given advice and helped organize my ideas.

I have had the opportunity to work with a number of talented collaborators, and I am grateful for their contributions that have aided this work. Thank you to all of the members of the VisTrails team, especially Erik Anderson, Steven Callahan, Tommy Ellqvist, Emanuele Santos, Carlos Scheidegger, and Huy Vo, who have helped to assist me with my work and troubleshoot bugs; it has been a joy to work with such a talented team. I would also like to thank Bela Bauer, Brigitte Surer, and the rest of the ALPS group, for their help in testing many of the techniques integrated with VisTrails. Other collaborators and co-authors, including Philippe Bonnet, Daniel Fink, Steve Kelling, and Jeff Morisette, have been influential in my understanding of the issues in computational science, reproducibility, and reuse.

There have also been a number of people and organizations that have helped shape and facilitate my work, and I am grateful for this support. Thanks to folks I have worked with at VisTrails, Inc., including Douglas Alves, Benjamin Burnett, and Ramesh Pinnamaneni, for their work and insight. Also, thank you to the staff at the School of Computing and Scientific Computing & Imaging Institute at the University of Utah who helped troubleshoot technical and procedural issues, dealt with scheduling and equipment needs, and made sure necessary paperwork was completed. Thank you to Microsoft Research for a summer internship to expand my understanding, and to everyone involved with provenance challenge workshops that helped grant me a broader understanding of the field.

Thank you to the various funding agencies that have made the research possible, including the Department of Energy SciDAC (VACET and SDM centers), and the National Science Foundation (grants IIS-0746500, CNS-0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096, CCF-0401498, OISE-0405402, CCF-0528201, CNS-0551724). Also, thank you to the students in the scientific visualization courses of 2007 and 2008 at the University of Utah, who provided the provenance information used in this work.

Thank you to my friends and family for their support. Thanks to all of the friends I've met in Utah as well as those from my years in Madison, Grand Rapids, and elsewhere. Specifically, thanks

to John, Steve, Erik, Joel, and Carlos, for dealing with rants and questions over lunch, as well as the disc golfers and ultimate players who have provided the excuse for a break from work. Thank you to my parents, Jan and Al, who always supported my education and continued to encourage my work throughout the years. Finally, thank you to Jen, my wife, for her support and understanding during this process.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Serving as a record of what happened during a scientific process, often computational, provenance has become an important piece of computing. The importance of archiving not only data and results but also the lineage of these entities has led to a variety of systems that capture provenance as well as models and schemas for this information. Despite significant work focused on obtaining and modeling provenance, there has been litte work on managing and using this information. Querying this information has been studied for feasibility and interoperability concerns, but applications that drive these queries have been limited. One of the applications for provenance is reproducibility—exactly replicating a process or computation. This work proposes reuse as an improved application, allowing provenance users to migrate work to new techiques or hardware and more easily extend published findings.

Provenance documents how something was accomplished; a collection of such information is thus extremely valuable in understanding solutions. Furthermore, using data mining, it is possible to determine similar solutions or common pieces of provenance information. Such parts can then be used to derive new complete or partial solutions. Note that an important component in such mining is the structure of the provenance; with more abstraction, it can be easier to locate patterns. In order to suggest relevant suggestions, existing structure can be used to index into a summary of collected provenance.

Along similar lines, while suggestions are targeted to help users for specific tasks, summaries of collections of provenance can be useful for browsing the information. Because provenance is often understood as a graph of dependencies, a textual summary of information is usually difficult to parse. On the other hand, visually a collection of graphs is complicated by the fact that comparing graphs is NP-Complete. This work presents algorithms to build summaries of collections of graphs. In addition, it demonstrates methods for editing these summaries by splitting and joining multinodes and multiedges.

To support queries, completions, and summaries for provenance, there must be infrastructure to support efficient access to the information. Indexing techniques are often used to speed queries over certain fields in databases, but because provenance information is stored as a graph, a query

can leverage both distinct node criteria and connectivity constraints. Thus, indexing must also encapsulate these features. However, because subgraph isomorphism is NP-Complete, even comparing two graphs in a collection to test their equivalence can be difficult. Existing techinques for graph indexing leverage discriminative subgraphs that help filter candidates, limiting the number of full verifications that need to be calculated via subgraph isomorphism calculations. However, provenance queries can be more vague, referencing only loosely-connected pieces of a subgraph, and may return large numbers of results. This work proposes a framework to adapt existing indexing techniques to make provenance queries more efficient.

A key concern in provenance is the data associated with the steps involved. For exploratory science, it is not always possible or efficient to curate data and ensure their longevity. At the same time, referencing data by filenames or URIs is problematic; a file can be moved or deleted, and linking provenance from outside the originating machine is difficult. This work proposes a framework for both identifying and managing the input and output data involved inline with provenance information.

For the goal of reuse—not simply reproducibility, it is important to have the ability to migrate and adapt documented processes to use new hardware or techniques. To identify possible incompatibilities and how the necessary changes may be completed, documenting version information is required. At the same time, the provenance of the changes themselves can be invaluable when diagnosing differences in results. Thus, provenance plays an important role in both allowing upgrades but also in documenting changes.

1.2 Thesis Statement

Techniques for the management and analysis of provenance enable applications for knowledge discovery and reuse by leveraging the information contained in provenance stores.

1.3 Dissertation Objectives

In this dissertation, we present a set of techniques for managing and analyzing provenance information as well as applications that use this framework to aid in future work. The goal is to use provenance, often viewed as archival data, to help develop solutions that use this information. The outline of this dissertation can be separated into four contributions:

- A method to suggest possible workflow completions using provenance information about previously constructed workflows [85]. This technique both offers starting points for novice users and reduces the effort for more experienced users in constructing workflows.
- A framework for indexing provenance information that permits more exploratory queries and supports queries that have large numbers of results. The techinques augment existing

graph indexing techniques by adding an extra layer to the index for more quickly locating large numbers of results as well as incorporating disconnected features.

- A technique to display a collection of graphs in a visual summary that allows discovery of similarities and differences. This can be used to display collections of provenance graphs so users can discover changes, and the summaries are editable so they can serve to develop reusable analogies that can be applied to other work.
- An infrastructure to support new modes of publication. To support work on executable papers and Web-based publications, it is necessary to maintain links to data [84] and support the longevity of provenance for later use through upgrades [86].

The rest of this dissertation is organized as follows. Chapter 2 reviews background on provenance and other work in this area. Chapter 3 describes VisComplete, a recommendation system for workflows that uses provenance information to derive completions. Chapter 4 describes an indexing scheme for querying provenance information. Then, Chapter 5 describes techniques for visualizing collections of graphs, including provenance graphs. Chapter 6 describes contributions that enabled greater reuse and longevity for publications. Finally, Chapter 7 presents conclusions and directions for future work.

CHAPTER 2

BACKGROUND

As this work relies on provenance, it is important to start by reviewing what constitutes provenance, how it is generated and captured, and what techniques exist to manipulate and access this information. The techinques and frameworks have been implemented or integrated into computational work, usually via an existing system. The VisTrails scientific workflow system has served as a testbed for this work, and both the system and workflows in general have prompted and aided many of the applications. However, the framework and algorithms are general and can be integrated with other systems. This chapter begins by defining provenance before describing scientific workflow systems and provenance capabilities. VisTrails is used as an example to highlight how provenance and scientific workflow systems are coupled.

2.1 Provenance

Provenance is the lineage or history of some object, including relationships to other objects that influence it. It can refer to the trail of ownership of a piece of artwork from painter to current owner, the steps in baking a cake from ingredient collection to finished product, or the processes involved in deriving a scientific result from experimental setup to analyses. While the term has not always been associated with science, the concepts are ingrained into both the work and mindset of scientists. Published results are derived from information about the exact procedures followed, captured data, annotations, and documented analyses. In addition, all of this information is documented in the publication so other scientists can validate procedures and reproduce and extend results. This provenance is often as important, if not more, than the results.

As computing resources are used for more tasks and data are stored in digital form, the pace of work has accelerated and the complexity of tasks has increased. Manually keeping track of all steps followed, parameters set, and data used is burdensome and prone to error. For computational tasks, it is more efficient to have computers record this information. Computational provenance, then, tracks the steps and data involved in some computational task. The provenance (also referred to as the audit trail, lineage, and pedigree) of a data product contains information about the process and data used to derive the product [47, 137]. It provides important documentation that is key to preserving the data, to determining its quality and authorship, and to reproducing as well as

validating the results. These are all important requirements of the scientific process.

The scope and granularity of provenance information vary based on the task and capture mechanism. For example, fine-grained information about the lineage database tuples can be captured and used to analyze query results [144]. For more general tasks, the granularity of provenance varies from a listing of all low-level system/kernel calls [49, 107] to abstracted workflow descriptions [81, 145, 153]. Note that low-level capture is more general but requires significant work to obtain a high-level description. More abstract provenance can be more easier understood but may lack some of the details.

Another classification for provenance information involves the type of information being collected. *Prospective provenance* captures the specification of a computational task (*i.e.,* a script or workflow)—it corresponds to the *steps that need to be followed* (or a recipe) to generate a data product or class of data products. *Retrospective provenance* captures the *steps that were executed* as well as information about the execution environment used to derive a specific data product—a detailed log of the execution of a computational task. Note that retrospective provenance can be captured for any task regardless of whether that task has prospective provenance. For example, information like which processes were run, who ran them, and how long they took, can be captured without knowing the sequence of steps ahead of runtime.

Provenance can also contain *user-defined information*, documentation that cannot be automatically captured but records important decisions and notes. These data are often captured in the form of annotations. Annotations can be added at different levels of granularity and associated with components of both prospective and retrospective provenance.

To investigate the capabilities of various systems, relationships between them, and models for storage, challenges were proposed and accomplished by a set of teams. The first challenge highlighted different methods for capturing and querying provenance information [121]. The second investigated interoperability of provenance models [122], and the third challenge focused on using the Open Provenance Model (OPM) as a model for exchanging provenance between systems [123]. As a very general model, OPM allows a variety of types of provenance information to be recorded without enforcing many constraints [105].

2.2 Scientific Workflow Systems

Computational tasks can be represented using a variety of mechanisms including computer programs, scripts, and workflows. They can also be constructed interactively using specialized tools (*e.g.,* ParaView [82] for scientific visualization, GenePattern [54] for biomedical research) that often have their own internal format to represent a task. Some complex computational tasks require that different tools be weaved together, including loosely-coupled resources, specialized libraries,

distributed computing infrastructure, and Web services. For example, to analyze the results of a CT scan, it may be necessary to preprocess the data with different parameters, visualize each result, and compare them. To ensure reproducibility of the entire task, it is beneficial to have a description that captures these steps and the different parameter values used.

Workflow and workflow-based systems have recently grown in popularity within the scientific community as a means to assemble complex processes [40, 46, 81, 104, 114, 138, 145, 149, 153, 156]. Not only do they support the automation of repetitive tasks, but they can also systematically capture provenance information for the derived data products [39]. Most workflow systems support provenance capture, although each adopts its own data and storage models [39, 47]. These range from specialized Semantic Web languages (*e.g.,* RDF and OWL) and XML dialects that are stored as files in the file system, to tables stored in relational databases.

A *workflow* describes a set of computations as well as an order for these computations. To simplify the presentation, we focus on dataflows; but note that our approach is applicable to more general workflow models. In a dataflow, computational flow is dictated by the data requirements of each computation. A dataflow is represented as a directed acyclic graph where nodes are the computational *modules* and edges denote the data dependencies as *connections* between the modules—an edge connects the *output port* of a module to an *input port* of another. Often, a module has a set of associated *parameters* that can control the specifics of one computation. Some workflows also utilize *subworkflows* where a single module is itself implemented by an underlying workflow.

Because workflows abstract computation, there must be an association between the module instances in a workflow and the underlying execution environment. This link is managed by the *module registry* which maps module identifiers to their implementations. For convenience and maintenance, related modules are often grouped together in *packages*. Thus, the module identifier may consist of package identifier, a module name, an optional namespace, and information about the version of the implementation. Version information can serve to inform us when implementations or interfaces in the environment change, and is part of provenance information.

Consider, for example, the VisTrails system [153]. In VisTrails, each module corresponds to a Python class that derives from a predefined base class. Users define custom behaviors by implementing a small set of methods. These, in turn, might run some code in a third-party library or invoke a remote procedure call via a Web service. The Python class also explicitly describes the interface of the module: the set of allowed input and output connections, given by the module's *ports*. A VisTrails package consists of a set of Python classes.

One of the benefits of workflow systems is that they lend themselves to visual programming environments. Those browsing a collection of workflows should be able to gain an idea of the computation from a depiction of the general structure without reading a long code listing. Connec-

tions show relationships between modules without the need to trace variable names, and parameter settings can be located with the modules they affect. This enables users to more quickly set parameter values, add computational modules, or delete extraneous analyses.

2.3 VisTrails

VisTrails (http://www.vistrails.org) is an open-source system that supports data exploration and visualization. It combines and substantially extends useful features of scientific workflow and visualization systems. Similar to scientific workflow systems [81, 117, 145, 156], VisTrails allows the specification of computational processes which integrate existing applications, loosely-coupled resources, and libraries according to a set of rules; and similar to visualization systems [71, 82, 91, 154], VisTrails makes advanced scientific and information visualization techniques available to users, allowing them to explore and compare different visual representations of their data. As a result, users can create complex workflows that encompass important steps of scientific discovery, from data gathering and manipulation, to complex analyses and visualizations, all integrated in one system.

A distinguishing feature of VisTrails is a comprehensive provenance infrastructure that transparently captures and maintains detailed history information about the steps followed and data derived in the course of an exploratory task [48]. Whereas workflows have been traditionally used to automate repetitive tasks, for applications that are exploratory in nature, such as simulations, data analysis, and visualization, very little is repeated—change is the norm. As a user generates and evaluates hypotheses about data under study, a series of different, albeit related, workflows are created as they are adjusted in an iterative process. VisTrails was designed to manage these rapidly-evolving workflows: it maintains provenance of data products (*e.g.,* visualizations, plots) of the workflows that derive these products, and their executions. The system also provides annotation capabilities that allow users to enrich the automatically captured provenance.

Besides enabling reproducible results, VisTrails leverages provenance information through a series of operations and intuitive user interfaces that help users to collaboratively analyze data. Notably, the system supports reflective reasoning by storing temporary results, by providing users the ability to examine the actions that led to a result and to follow chains of reasoning backward and forward [113]. Users can navigate workflow versions in an intuitive way, undo changes but not lose any results, visually compare multiple workflows, and show their results side-by-side in a visualization spreadsheet [17, 48, 135].

Because the need for data analysis and visualization is pervasive across disciplines, VisTrails was designed with usability and extensibility in mind. VisTrails addresses important usability issues that have hampered a wider adoption of workflow and visualization systems. To cater

to a broader set of users, including many who do not have programming expertise, it provides a series of operations and user interfaces that simplify workflow design and use, including the ability to create and refine workflows by analogy, to query workflows by example, and to suggest workflow completions as users interactively construct their workflows using a recommendation system [85, 130]. VisTrails is also linked to a new framework that allows the creation of custom applications that can be more easily deployed to (nonexpert) end users [127, 128]. The extensibility of VisTrails comes from an infrastructure that makes it simple for users to integrate tools and libraries, as well as to quickly prototype new functions. This has been instrumental to enable the use of the system in a wide range of application areas, including environmental sciences [16, 70], psychiatry [10], astronomy [147], cosmology [9] , high-energy physics [42], quantum physics [7], and molecular modeling [64].

CHAPTER 3

VISCOMPLETE: DATA-DRIVEN SUGGESTIONS FOR VISUALIZATION SYSTEMS

3.1 Introduction

Data exploration through visualization is an effective means to understand and obtain insights from large collections of data. Not surprisingly, visualization has grown into a mature area with an established research agenda [109], and a number of software systems have been developed that support the creation of complex visualizations [30, 71, 82, 83, 101, 114, 153, 154]. However, a wider adoption of visualization systems has been greatly hampered due to the fact that these systems are notoriously hard to use, in particular, for users who are not visualization experts.

Even for systems that have sophisticated visual programming interfaces, such as DX, AVS, and SCIRun, the path from the raw data to insightful visualizations is laborious and error-prone. Visual programming interfaces expose computational components as *modules* and allow the creation of complex visualization pipelines which combine these modules in a dataflow, where *connections* between modules express the flow of data through the pipeline. They have been shown to be useful for comparative visualization and efficient exploration of parameter spaces [17]. Through the use of a simple programming model (*i.e.,* dataflows) and by providing built-in constraint checking mechanisms (*e.g.,* that disallow a connection between incompatible module ports), they ease the creation of pipelines. Notwithstanding, without detailed knowledge of the underlying computational components, it is difficult to understand what series of modules and connections ought to be added to obtain a desired result. In essence, there is no "roadmap"; systems provide very little feedback to help the user figure out which modules can or should be added to the pipeline. A novice user (*i.e.,* an experienced programmer that is unfamiliar with the modules and the dataflow of the system), or even an advanced user performing a new task, often resorts to manually searching for existing pipelines to use as examples. These examples are then adapted and iteratively refined until a solution is found. Unfortunately, this manual, time-consuming process is the current standard for creating visualizations rather than the exception.

Recent work has shown that provenance information (the metadata required for reproducibility) can be used to simplify the process of pipeline creation by allowing pipelines to be refined and queried by example [130]. For example, a pipeline refinement can act as an analogy template for

creating new visualizations. This is a powerful tool and can be helpful in situations when the user knows in advance what they want the end result to be. However, during pipeline creation, it is not always the case that the user has an analogy template readily available for the visualization that is desired. In these cases, the user is relegated to manually searching for examples.

In this chapter, we present VisComplete, a system that aids users in the process of creating visualizations by using a database of previously created visualization pipelines. 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 quality and nature of the suggestions depend on the data from which they are derived. Whereas in a single-user environment, suggestions are derived based on pipelines created by a specific user, in a multi-user environment, the "wisdom of the crowds" can be leveraged to derive a richer set of suggestions that includes examples with which the user is not familiar. 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), and visualization creation (*e.g.,* ManyEyes). 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 auto-complete 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 3.1 shows an example of VisComplete incorporated into a visual programming interface and Figure 3.2 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. By modeling pipelines as graphs, we develop an algorithm for predicting likely completions that 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 chapter we focus on the use of VisComplete for visualization pipelines, the techniques we present can be applied to general workflows.

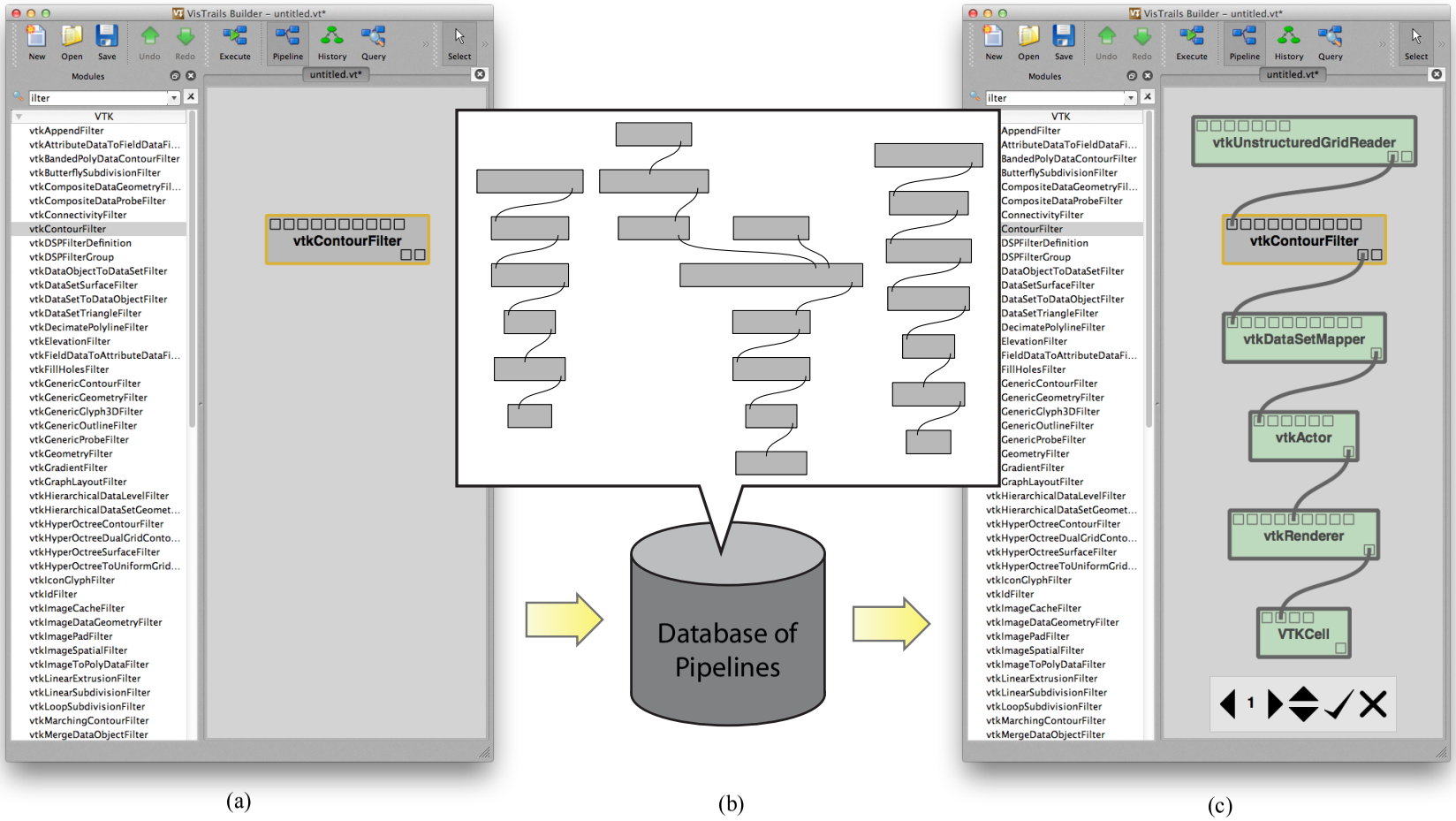


Figure 3.1: The VisComplete suggestion system and interface. (a) A user starts by adding a module to the pipeline. (b) The most likely completions are generated using indexed paths computed from a database of pipelines. (c) A suggested completion is presented to the user. The user can browse through suggestions using the interface and choose to accept or reject the completion.

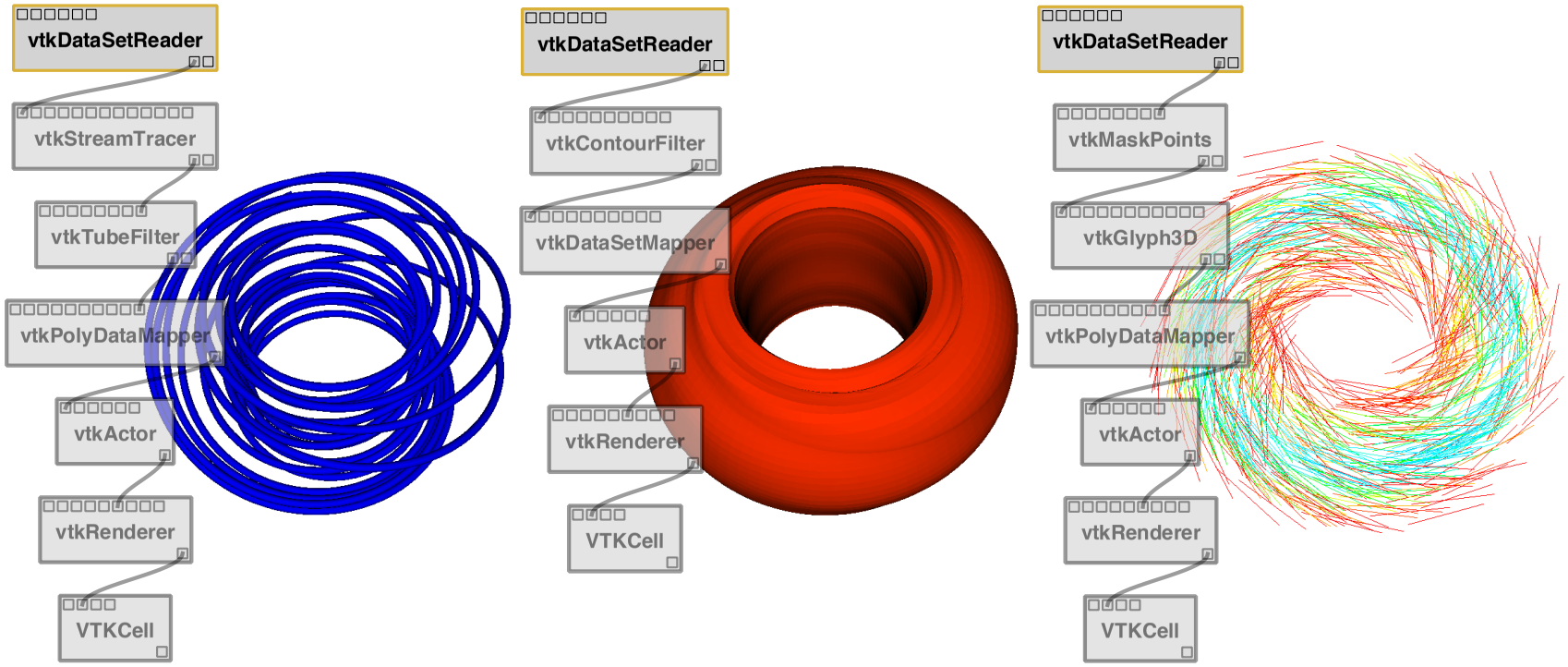


Figure 3.2: Three of the first four suggested completions for a "vtkDataSetReader" are shown along with corresponding visualizations. The visualizations were created using these completions for a time step of the Tokamak Reactor dataset that was not used in the training data.

The rest of this chapter is organized as follows. In Section 3.2, we discuss related work. In Section 3.3, we present the underlying formalism for generating pipeline suggestions, and in Section 3.4, we describe a practical implementation that has been integrated into the VisTrails system [153]. We then detail the use cases we envision in Section 3.5, report our experiments and results in Section 3.6, and provide a discussion of our algorithm in Section 3.7. We conclude in Section 3.8, where we outline directions for future work.

3.2 Related Work

Visualization systems have been successfully used to bring powerful visualization techniques to a wide audience. Seminal workflow-based visualization systems, such as AVS Explorer [154], Iris Explorer [111], and Visualization Data Explorer [71], have paved the way for more recent systems designed using an object-oriented approach such as SciRun [114] for computational steering and the Visualization Toolkit (VTK) [83] for visualization. Systems that incorporate standard point-and-click interfaces and operate on data at a larger scale, such as VisIt [30] and ParaView [82], still use workflows as their underlying execution engine. Development in workflow systems for visualization is ongoing, as seen in projects such as MeVisLab [102] for medical visualization and VisTrails [153] for incorporating existing visualization libraries with other tools in a provenance capturing framework. Our completion strategy can be combined with and enhance workflow and workflow-based visualization systems.

Recommendation systems have been used in different settings. Like VisComplete, these are based on methods that predict users' actions based solely on the history of their previous interactions [68]. Examples include Unix command-line prediction [87], prediction of Web requests [50, 112], and autocompletion systems such as IntelliSense [103]. Senay and Ignatius have proposed incorporating expert knowledge into a set of rules that allow automated suggestions for visualization construction [132], while Gilson *et al.* incorporate RDF-based ontologies into an information visualization tool [55]. However, these approaches necessarily require an expert that can encode the necessary knowledge into a rule set or an ontology.

Fu *et al.* [50] applied association rule mining [3] 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 are 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 [97] were introduced to allow users to explore the space of rendering parameters by suggesting a set of automatically generated thumbnails. Igarashi and Hughes [72] 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 [158] and Takahashi *et al.* [143] 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 [61] 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.* [90] 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.* [151] 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 [19, 76] provide the means for collaborative visualization from disjoint locations. Web sites such as Sens.us [63], Swivel [141], and ManyEyes [157] 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.

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

Pipelines are specified as graphs, where nodes represent modules (or processes) and edges determine how data flows through the modules. More formally, a *pipeline specification* 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 a complex 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.

3.3.1 Problem Definition

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.

Our solution to this problem consists of two main steps. First, we preprocess the collection of pipelines $\mathcal{G}$ and create $\mathcal{G}_{path}$, a compact representation of $\mathcal{G}$ that summarizes relationships between common structures (*i.e.,* sequences of modules) in the collection (Section 3.3.2). Given a partial pipeline p, completions are generated by querying $\mathcal{G}_{path}$ to identify modules and connections that have been used in conjunction with p in the collection (Section 3.3.3).

3.3.2 Mining Pipelines

To derive completions, we need to identify graph fragments that co-occur in the collection of pipelines $\mathcal{G}$. 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 $\mathcal{G}$, generating predictions based on subgraphs can be prohibitively expensive. Thus, instead of subgraphs, we use paths, *i.e.,* a linear sequence of connected modules. Specifically, we compute the frequencies for each path in $\mathcal{G}$. Completions are then determined by finding which path extensions are likely given the existing paths.

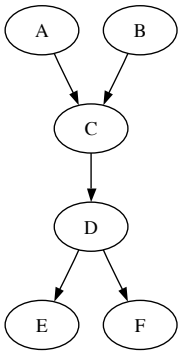
To efficiently derive completions from a collection of pipelines $\mathcal{G}$, we begin by generating a summary of all paths contained in the pipelines. Because completions are derived for a specific vertex v in a partial pipeline (we call this vertex the *completion anchor*), we extract all possible 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 3.6, 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 $\mathcal{G}_{path}$ 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 3.3. 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 3.3. 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 3.4). 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.



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 3.3: Deriving a path summary for the vertex D.

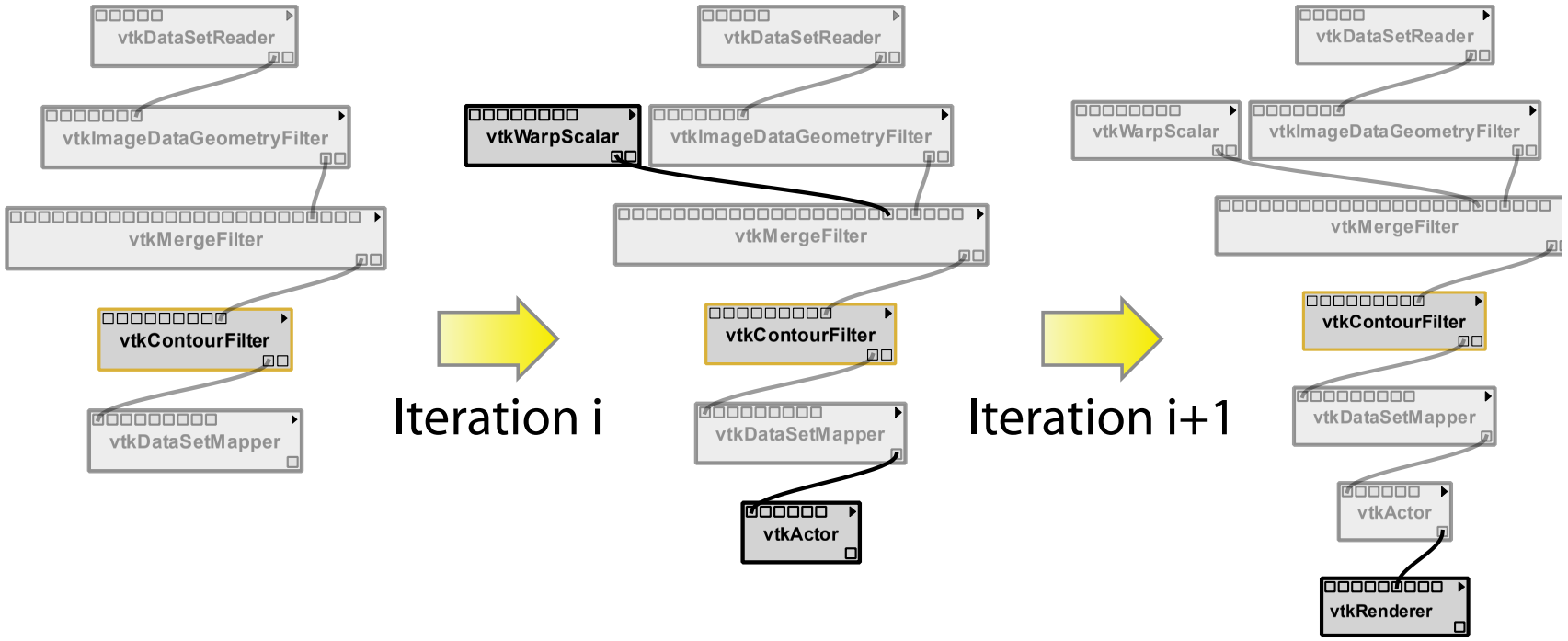


Figure 3.4: Predictions are iteratively refined. At each step, a prediction can be extended upstream and downstream; in the second step, the algorithm only suggests a downstream addition. Also, predictions in either direction may include branches in the pipeline, as shown in the center.

3.3.3 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 Algorithm 1, we iteratively develop our list of predictions by adding new vertices using this criteria.

Algorithm 1: Generate Predictions

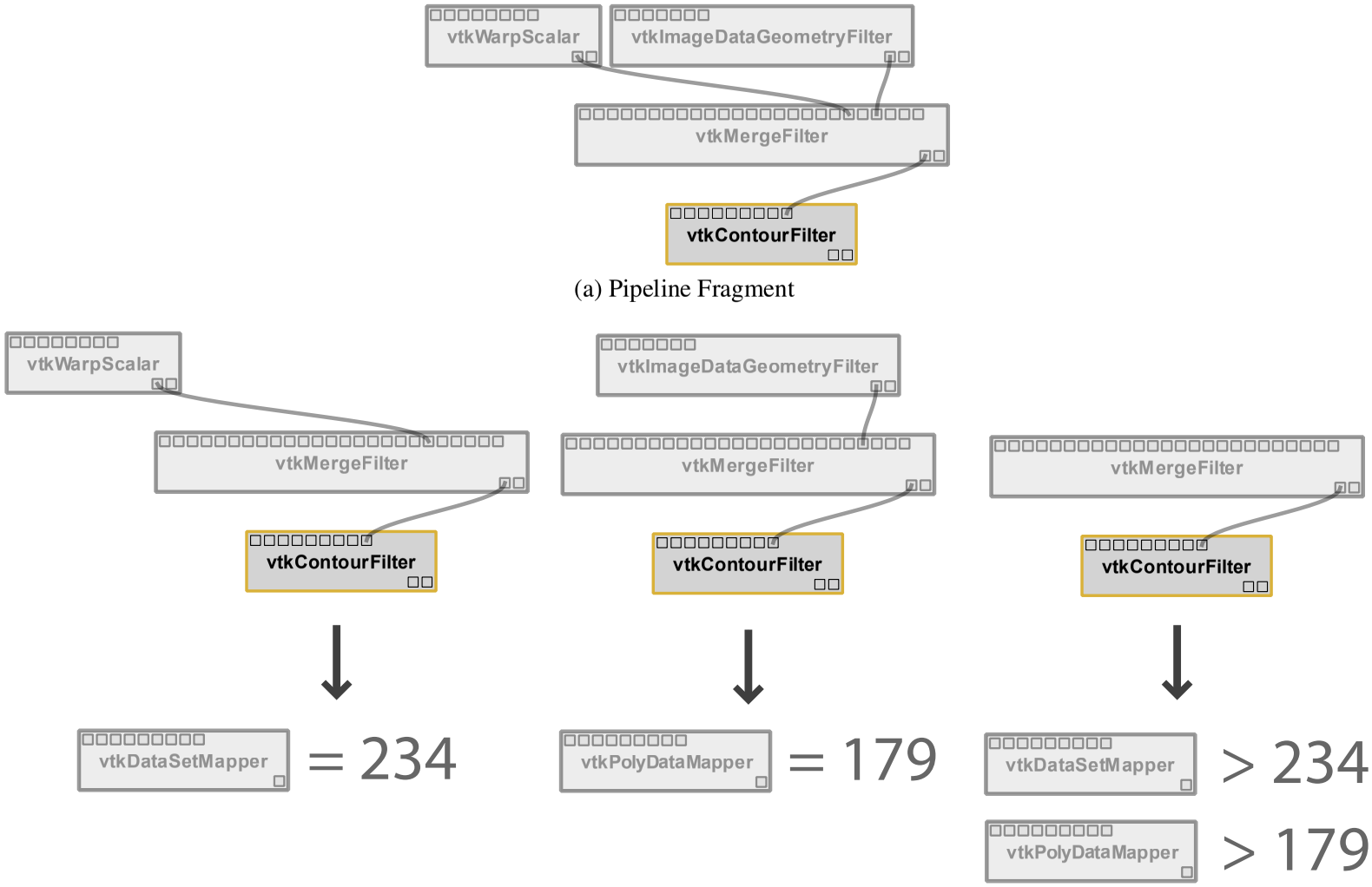
Input: A set of paths P
Output: A set of workflow completions $\mathcal{P}$
GENERATEPREDICTIONS(P)
(1) *possibles* $\leftarrow$ FIRSTPREDICTION(P)
(2) $\mathcal{P} \leftarrow [\,]$
(3) **while** $|$*possibles*$| > 0$
(4) **do** $p \leftarrow$ REMOVEFIRST(*possibles*)
(5) *newPossibles* $\leftarrow$ REFINE(p)
(6) **if** $|newPossibles| = 0$
(7) **then** $\mathcal{P} \leftarrow \mathcal{P} + p$
(8) **else** *possibles* $\leftarrow$ *possibles* + *newPossibles*

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 3.4 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 $\mathcal{G}_{path}$ are not normalized; we use the exact counts. Then, as illustrated by Figure 3.5, we combine the counts from each path. This means we do not need any weighting based on the frequency of paths; the formula takes this into account automatically. Specifically,

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



(a) Pipeline Fragment

(b) Paths Database

Figure 3.5: 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.

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

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

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:

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

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 from which we wish to extend our prediction. 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:

$$c_d(v) = c(v) + \textit{degree-difference}(v)$$

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.

3.3.4 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.

3.4 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.

3.4.1 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.

3.4.2 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 our technique. Instead, we focus on helping users complete pipelines, and direct them to existing techniques [17, 77, 78, 96] to explore the parameter space. Note that it might be beneficial to extend VisComplete to identify commonly used parameters that a user might consider exploring, but we leave this for future work.

3.4.3 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. Auto-complete interfaces for text generally show a set of possible completions in a one-dimensional 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 two-dimensional: users can select from a list of full completions and then 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 3.1(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 to which it is appended. 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.

3.5 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 3.2 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 3.1). 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.

3.6 Evaluation

3.6.1 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 [48].

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 [23, 24], 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 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.

---

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

3.6.2 Results

Figure 3.6 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 3.7 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 four other tasks, we expected that most users would be proficient using the tool and closer to the expert user described in Section 3.5. 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 3.8 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 3.9 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 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 chapter, with the exception of Figure 3.6,

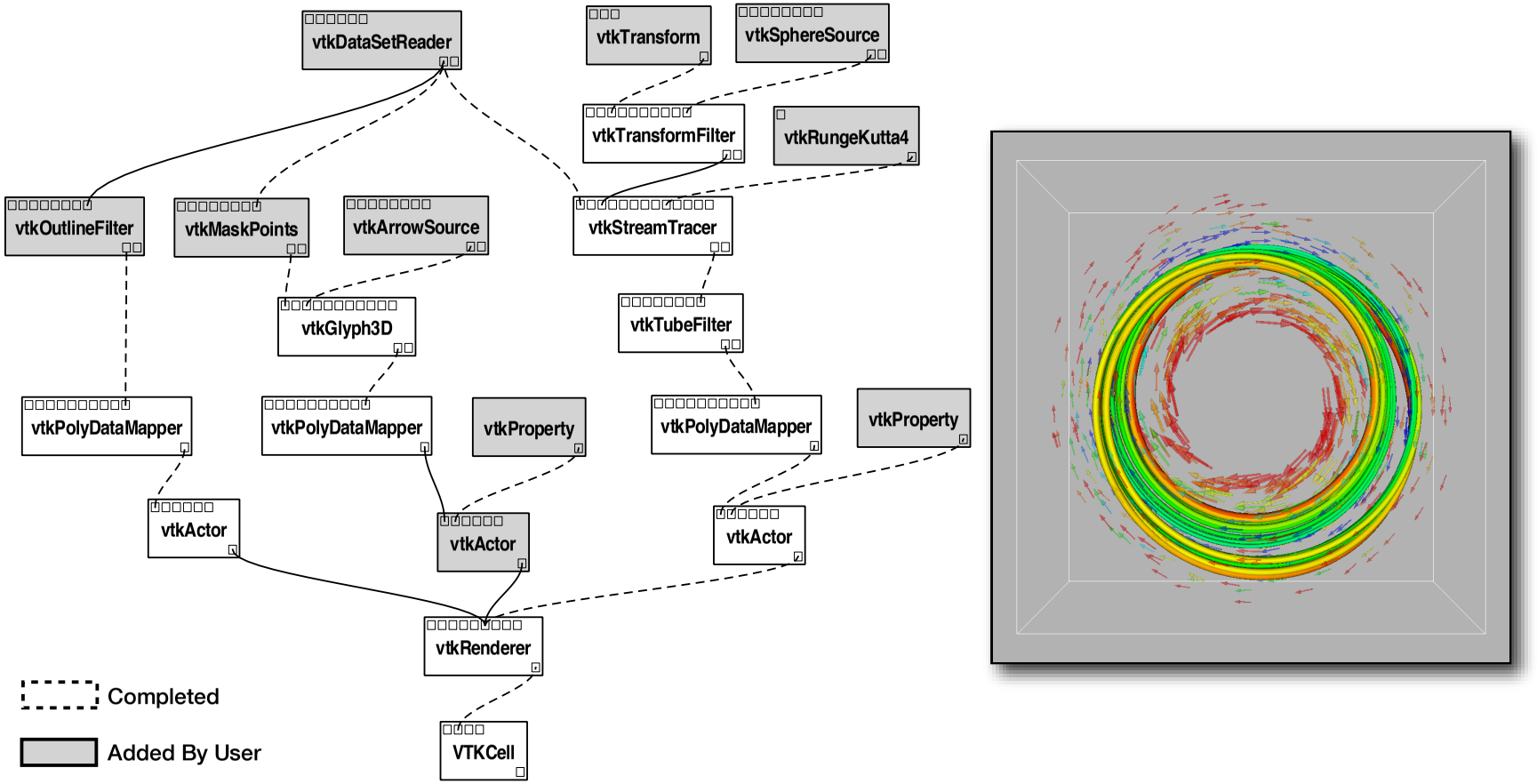


Figure 3.6: One of the test visualization pipelines applied to a time step of the Tokamak Reactor dataset. VisComplete could have made many completions that would have reduced the amount of time creating the pipeline. In this case, about half of the modules and completions could have been completed automatically.

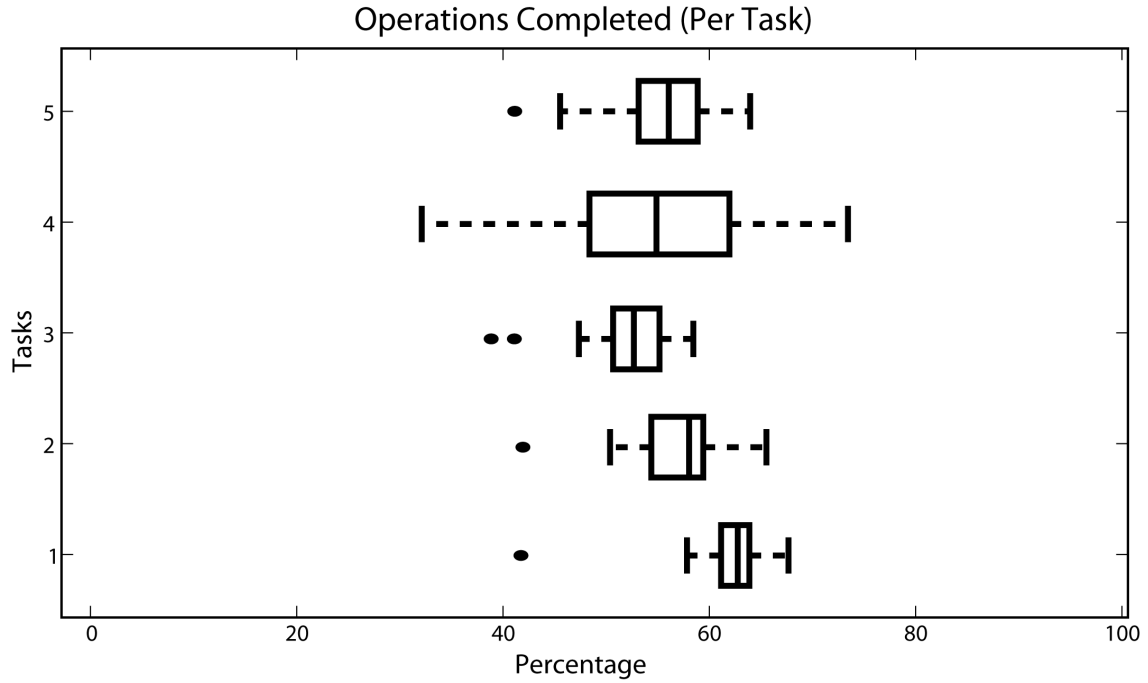


Figure 3.7: Box plot of the percentages of operations that could be completed per task (higher is better). The statistics were generated for each user by taking them out of the training data.

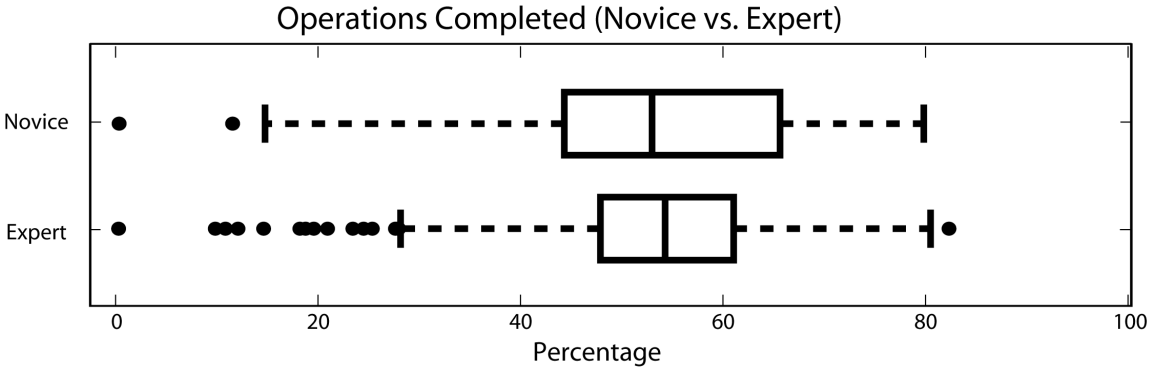


Figure 3.8: Box plot of the percentages of operations that could be completed given two types of tasks, novice and expert. The statistics were generated by evaluating the novice tasks using the expert tasks as training data (novice) and by evaluating the expert tasks using the novice tasks as training data (expert).

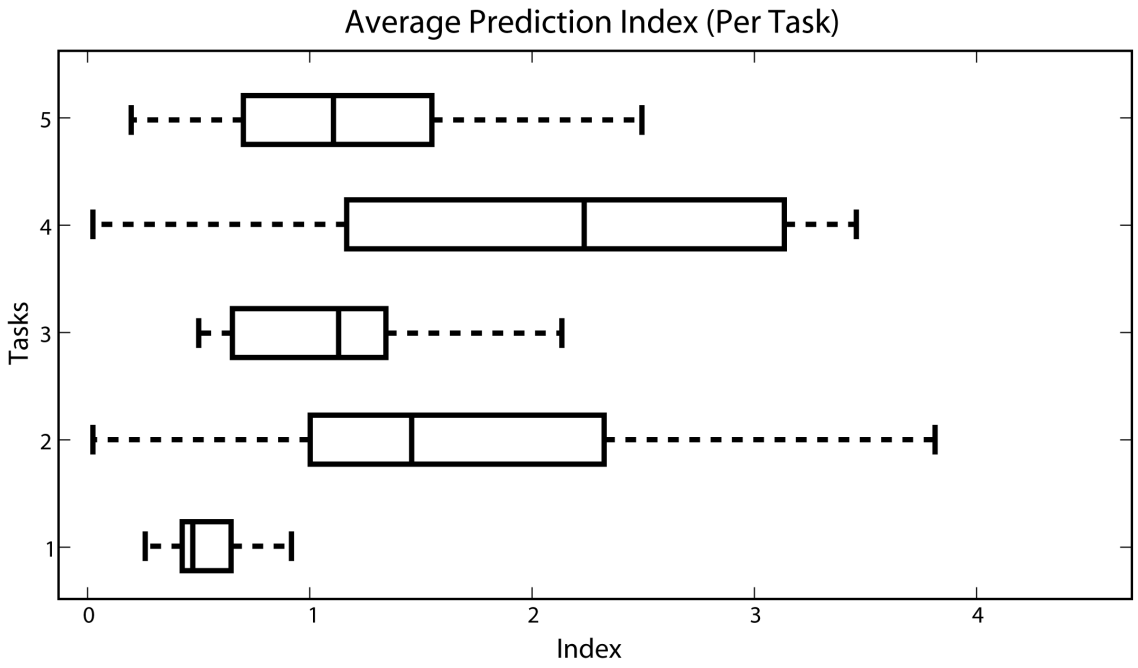


Figure 3.9: Box plot of the average prediction index that was used for the completions in Figure 3.7 (lower is better). These statistics provide a measure of how many suggestions the user would have to examine before the correct one was found.

used the entire collection of pipelines to generate predictions. Figure 3.6 used only the pipelines from Tasks 1–4.

3.7 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 [130] 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 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, Section 3.3). 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 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 [82], 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.

3.8 Summary

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.

CHAPTER 4

EFFICIENT EVALUATION OF EXPLORATORY QUERIES OVER PROVENANCE COLLECTIONS

4.1 Introduction

Increasingly, scientific exploration requires advanced computing capabilities to help researchers obtain insights into large datasets. The processes required to analyze and visualize data are often defined as workflows, which are iteratively refined as researchers formulate and test hypotheses. To manage these complex analyses, including the intermediate and final data products, workflow systems have been developed and track the provenance of the data products as well as of the workflow evolution [39, 47].

As the volume of provenance captured by these systems grows and is shared among users, new opportunities are created for knowledge reuse. Different kinds of queries can be posed against provenance [121]. Since workflow provenance can be represented as a graph [39], queries that seek the detailed derivation history of a given data product require that the provenance graph be recursively traversed (backwards), starting from the node that represents the data product. Another useful class of queries involve exploring the structure of the workflows that derive the data products. The workflows (and workflow traces) shared in provenance repositories expose users to examples of (sophisticated) uses of tools and libraries [33, 110]. By querying this information, users can leverage the collective wisdom it encodes. Not only can users find workflows that are relevant for a particular task and learn to assemble new workflows by example [18, 130, 131], but recommendation systems can be built to leverage this information to guide users in the workflow design process [85]. This is especially important given the fact that, despite the growing popularity of workflow systems, constructing workflows is often a challenging and time-consuming task. Detailed knowledge of the underlying computational components is necessary to determine what modules and connections ought to be added to obtain a desired result.

While there has been work on speeding up recursive queries over provenance graphs [65], the problem of evaluating structural queries has been largely overlooked. In this paper, we study the problem of efficiently evaluating structural queries that are exploratory in nature. Exploratory

queries are naturally expressed as simple graphs that may contain wildcards in constrast to standard containment queries like the one shown in Figure 4.1. For example, Figure 4.2 shows an exploratory query posed by a scientist interested in habitat modeling reports that were generated using the RandomForest model with a climate predictor layer. This query can be quickly defined without the need to understand exactly how the different components are connected. Queries with wildcards are useful to search for workflows (or subworkflows) that contain a given structural pattern, but can also be used and to identify possible directions for completing an unfinished workflow. For example, when a workflow designer is faced with a known input and desired output, it is helpful to identify different subworkflows that can connect the source and sink nodes of the the graph (see Section 4.5).

Although there has been substantial work on graph indexing techniques to speed up the evaluation of fully-specified structural queries [133, 166, 168], the same cannot be said of the problem of efficiently evaluating exploratory queries: Existing approaches have focused on connected-graph queries, not queries that are disconnected or contain wildcards. In addition, while the filtering step in these indexing schemes significantly reduces the number of required (and costly) subgraph isomorphism checks, vague queries often have a large number of answers, all of which must be verified through subgraph isomorphism. FG-Index introduced a verification-free indexing scheme to address this issue [29], but this comes at a cost: when the number of frequent subgraphs is large, the index may become prohibitively large.

We propose a flexible, two-level framework to support exploratory queries over provenance collections. Building on graph indexing techniques, we add 2-component frequent subgraphs to the index to support vague queries like those with wildcards and summary graphs to limit the time spent verifying candidate graphs after the filtering step. By augmenting the collection with summary graphs before constructing a discriminative index, we can process queries by verifying summary graphs first, reducing the total number of subgraph isomorphism checks required. We implemented a prototype mechanism and evaluated it on two large collections of provenance information.

This chapter is organized as follows. We review workflow definitions as well as graph terminology in Section 4.2 before introducing our indexing framework in Section 4.3. In Section 4.4, we detail our implementation, and Section 4.5 describes applying the framework to workflow completions. We evaluate our framework using provenance data from visualization and Yahoo! Pipes workflows in Section 4.6. We discuss extensions and limitations in Section 4.7 and review related work in Section 4.8 before concluding in Section 4.9.

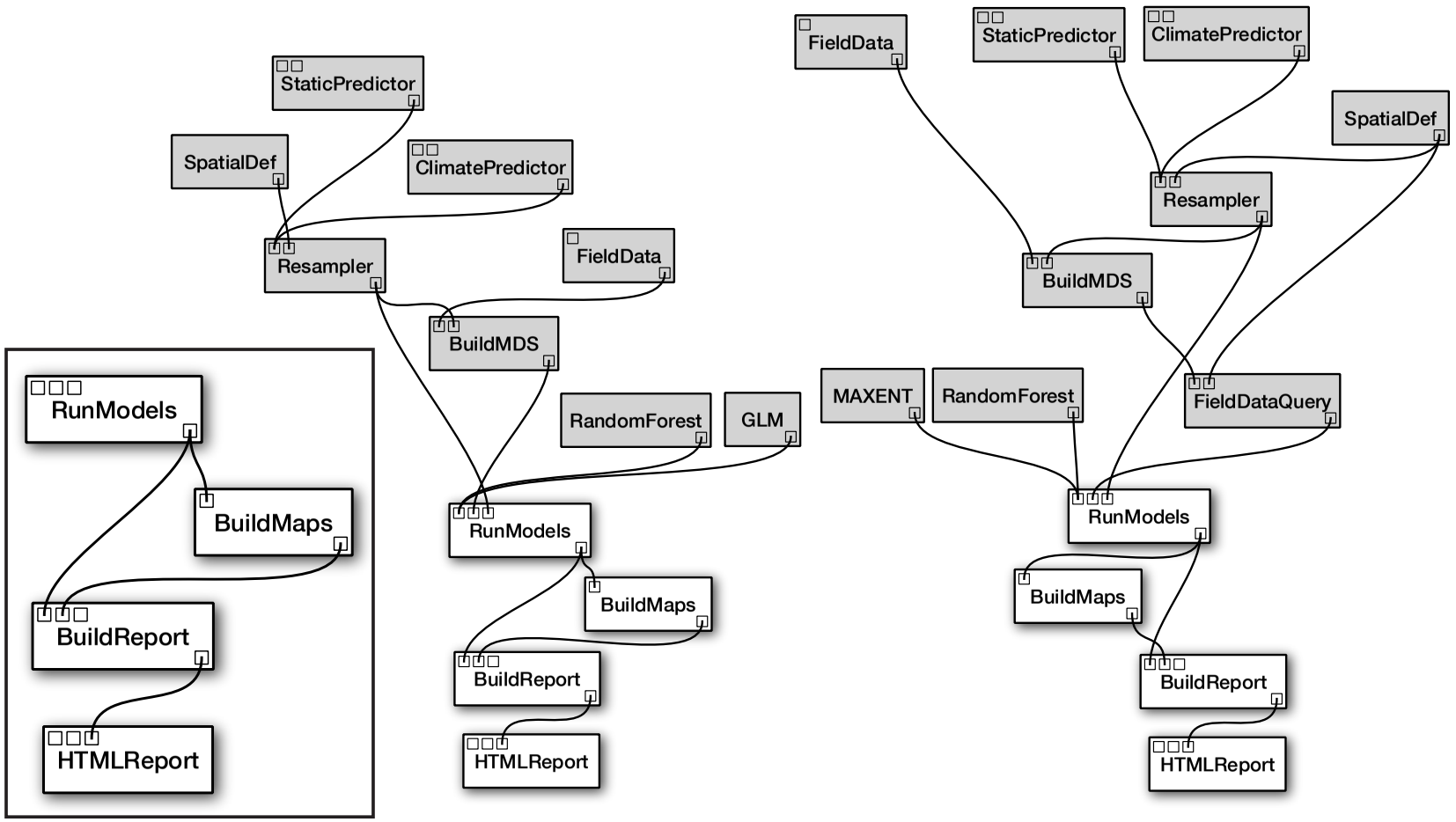


Figure 4.1: A standard containment query searches a collection to find workflows with the specified subgraph.

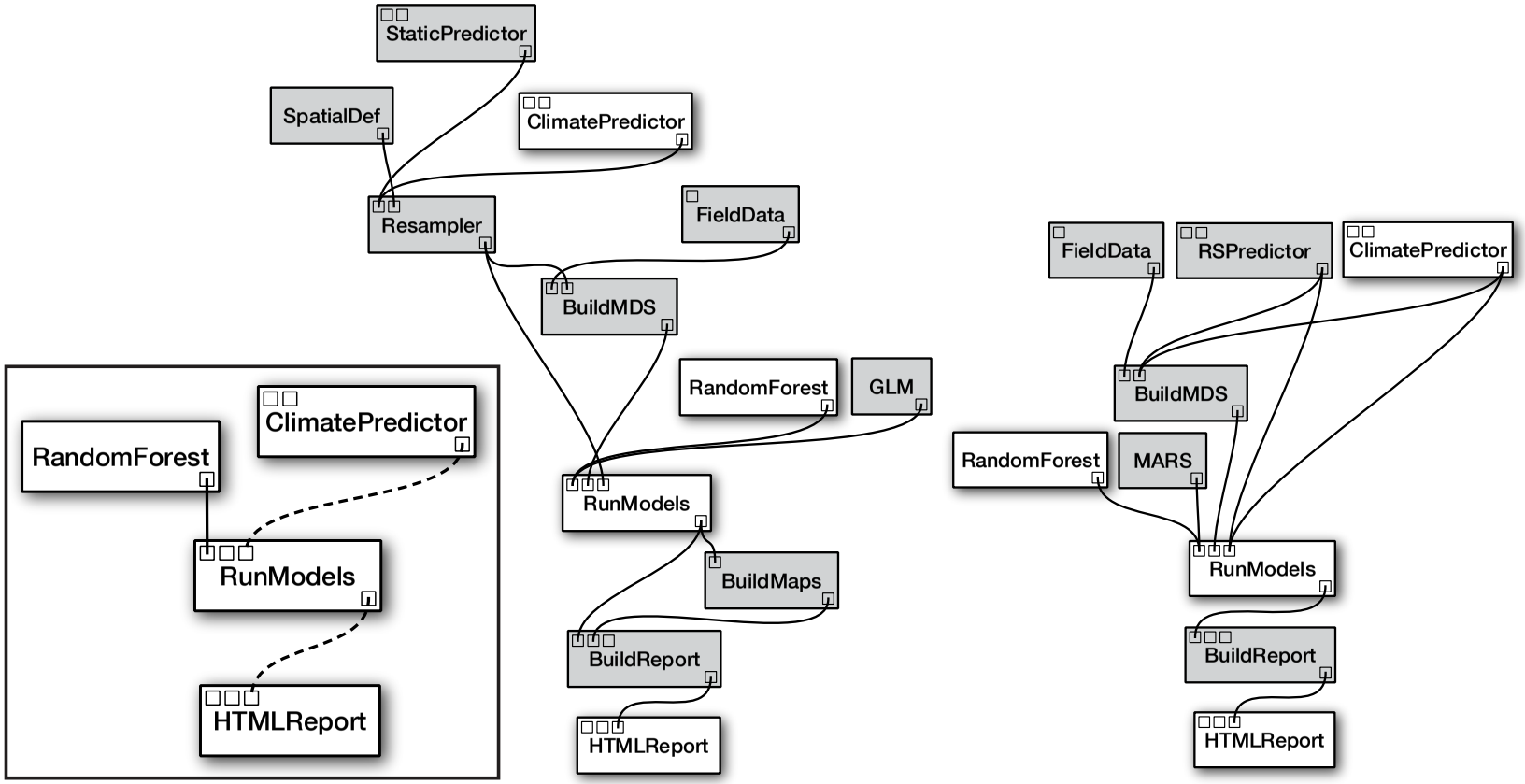


Figure 4.2: An exploratory query allows wildcards to permit less-specific queries. The dashed lines in the query are wildcard paths; each result must contain a path between the connected modules.

4.2 Background

Before presenting our indexing framework for exploratory queries over provenance collections, we will review terminology and definitions. Specifically, we wish to abstract these constructs to graphs in order to leverage and extend existing graph indexing techniques. We first review the correspondence between provenance and workflows, then define queries over workflow collections, and finally abstract this to graphs.

4.2.1 Provenance and Workflows

Provenance information is represented as a directed acyclic graph (DAG) encoding dependencies among computational steps. Similarly, workflows can also be represented as a graph specifying the order of computation, and most scientific workflows are dataflows which are also DAGs. Furthermore, when provenance is generated during the execution of a workflow, the provenance graph directly reflects the structure of the workflow. Thus, a query over provenance graphs (or parts of that query) can often be translated into a query over workflows. In many cases, the workflow specification is *shared* among several provenance traces derived from multiple executions of similar workflows. In addition, the workflow graph can be much more *compact* than the provenance graph, especially for workflows that include looping constructs. Thus, while our indexing framework can be directly applied to provenance graphs, it is usually more efficient to index the workflows behind the provenance graphs.

A *workflow* is a set of steps usually associated with some partial order. The steps followed can be controlled by their order, a set of logical constraints, or dictated by human input. A *dataflow* is a special kind of workflow that is a DAG.[1] In a dataflow, each node performs a computation and edges define the flow of data from the outputs of one node to the inputs of another [92]. While general workflows may contain cycles and explicit control constructs [1], their provenance can be represented as DAG—with loops unrolled and branches selected.

Formally, a workflow w is a set of computational modules linked by connections that define the flow of data from one module to another. This is often represented as a DAG whose vertices are modules and edges are connections. Each vertex and edge is distinguished with the *type* of module or connection it represents. For example, the center module in Figure 4.3 has the type `RunModels`, and the type of connection from it to the `BuildMDS` module is defined by the ports used to connect the modules.

[1] The dataflow model is the most prevalent model supported by scientific workflow systems.

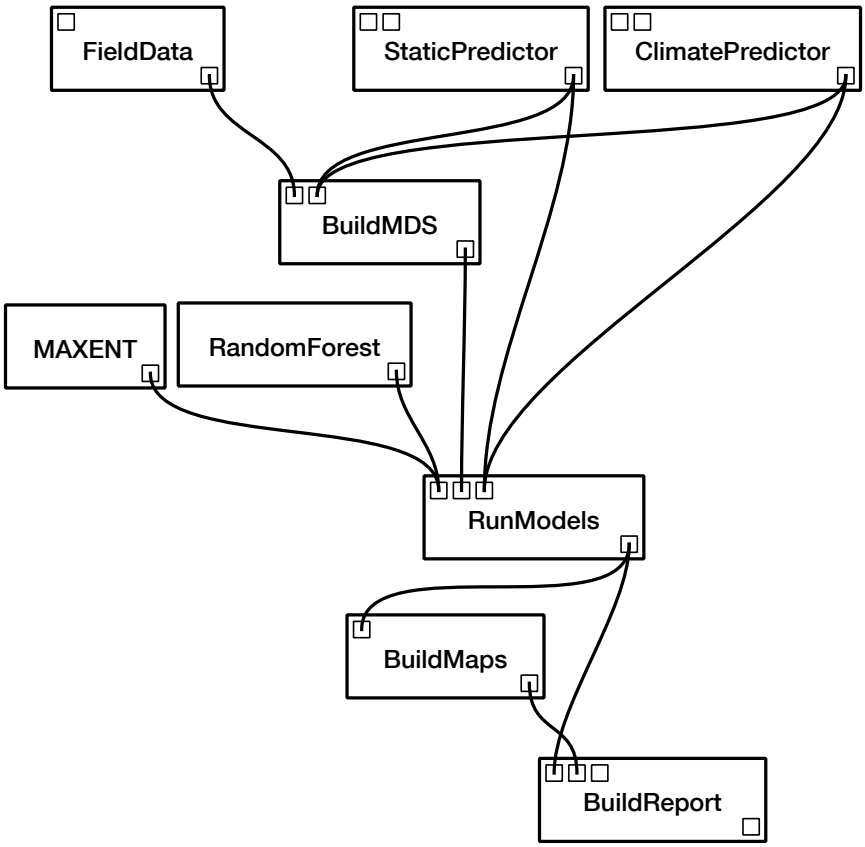


Figure 4.3: A representative workflow from a collection of workflows used for habitat modeling.

4.2.2 Queries Over Provenance Collections

A *provenance collection* consists of a set of provenance records. The collection may contain records generated from multiple executions of a single workflow, from a variety of workflows created as part of a collaborative scientific project, or from an entire database of workflows built by members of a scientific research group over a period of many years. Note that large, distributed collections implicitly contain a wealth of scientific information, cataloging different strategies, experimental approaches, and results. As described earlier, because provenance records often contain (or link to) the specifications of the workflows that were run, these collections often contain an embedded collection of workflows.

Some queries can be posed against a single workflow, others involve the differences between two workflows, but many are best answered by examining an entire collection. If a user wishes to know exactly which predictors in the workflow shown in Figure 4.3 affect the maps generated in the report, they need only analyze that single workflow. Another important type of query is identifying differences between pairs of workflows [15]. However, users are often interested in searching a collection of workflows to find those that exhibit specific behaviors. For example, a user may wish

to locate all workflows that run using a `RandomForest` module, a climate predictor, and that generate a report. We focus on this type of query. More formally, given a workflow collection $\mathcal{W}$ and a query q, we wish to find the subset $\mathcal{W}_q \subseteq \mathcal{W}$ such that every $w \in \mathcal{W}_q$ satisfies q.

Like others [18, 85], we posit that a query can be represented as a workflow. The most basic type of workflow query is containment. Formally, a *workflow containment query* q is a workflow specification, and a workflow $w \in \mathcal{W}$ satisfies q if there exists an injective function f that maps modules in q to modules in w such that

- $\text{type}(m) = \text{type}(f(m))$, $m \in q$, $f(m) \in w$, and
- $c(m_1, m_2) \in q \implies \exists c'(f(m_1), f(m_2)) \in w$ and $\text{type}(c) = \text{type}(c')$

where $c(m_1, m_2)$ is a connection from module m_1 to module m_2. Thus, the query is satisfied when a workflow contains the query workflow. This type of query can be used when looking for a particular region of functionality; for example, searching for all workflows that run a predictor and resamples its results.[2]

The problem with these containment queries is that the user must know exactly what to look for—the exact module types and connectivity. We suggest a more powerful form of workflow queries where the query allows wildcards for module or connection types. This relaxation allow queries to specify existence of paths in addition to direct connections, and existence of a module rather than a specific module type. More formally, an *exploratory workflow query* is a partial workflow q, a workflow where modules and connections can have the wildcard type $*$ meaning any type of module or connectivity is allowed. Then, a workflow w satisfies the exploratory query q if there exists an injective function f such that

- $\text{type}(m) = *$ or $\text{type}(m) = \text{type}(f(m))$
- $\text{type}(c(m_1, m_2)) = * \implies \exists\, \text{path}(f(m_1), f(m_2)) \in w$
- $\text{type}(c) \neq * \implies \text{type}(c) = \text{type}(f(c))$

where $f(c) = f(c(m_1, m_2)) = c'(f(m_1), f(m_2))$. Note that exploratory queries offer far greater flexibility; users can query a collection without worrying about steps that are not important to their search. For example, a user may wish to find all workflows that use a `RandomForest` module and eventually output an HTML report that includes information from that model; whether or not `BuildMap` is used is not relevant to the user. In an exploratory query, wildcards can be used to indicate that a path must connect the two modules but with no restrictions on what modules that path connects. See Figures 4.1 and 4.2 for an example of the difference between containment and exploratory queries.

---

[2] Note that workflow queries may also include information about parameters: these can also be specified as part of the workflow.

4.2.3 Graphs and Isomorphisms

Because we wish to make use of existing graph indexing approaches, we propose a translation from workflow queries to queries over collections of graphs. Workflows can be naturally represented as labeled graphs whose vertices and edges are labeled by the module or connection type. Formally, a workflow w can be represented by the labeled graph $G(V, E)$ where each module in w is represented by a vertex in V and each connection is an edge in E. In addition, the labeling functions, $L_V(v)$ and $L_E(e)$ are defined as the types of the modules and connections, respectively. Then, a basic workflow query can be immediately translated into a subgraph isomorphism problem, and exploratory workflow queries can be translated to an extension of subgraph isomorphism involving wildcards.

Two graphs G and H are *isomorphic* if there exists a bijective function $f : V(G) \rightarrow V(H)$ such that for every edge $(v_i, v_j) \in E(G)$, there exists an edge $(f(v_i), f(v_j)) \in E(H)$ and vice versa. If G and H are labeled graphs, then f must also preserve labels: $L_V(v_i) = L_V(f(v_i))$ and $L_E((v_i, v_j)) = L_E((f(v_i), f(v_j)))$. If we relax f to be an injective function, then G is *subgraph isomorphic* to H, $G \subseteq H$, again with the same restrictions for labeled graphs.

Much of the existing graph indexing work has focused on speeding up graph containment queries: given a query graph Q, find all graphs G in the collection for which $Q \subseteq G$. This type of query is analogous to our workflow containment query, and thus these approaches do not support exploratory queries with wildcards. To extend these techniques, we must first extend the definition of subgraph isomorphism to incorporate wildcards.

A *wildcard graph* G^* is a labeled graph where any edge can have a special $*$ label that denotes a path (not necessarily a single connection) between two vertices. Then G^* is *wildcard subgraph isomorphic* to H if $G^* - \{e \mid L_E(e) = *, e \in E(G^*)\}$ (G^* excluding all wildcard edges) is subgraph isomorphic to H and for each wildcard edge (v_i, v_j), there exists a path from $f(v_i)$ to $f(v_j)$ such that no internal vertex in this path is in $f(V(G^*))$. Note that the restriction on the path ensures that a query graph where vertices are specified and not identified as path of a path cannot be used in a wildcard path.

4.3 Indexing Framework

With the abstraction of provenance and workflow queries over provenance collections to graph queries, we will propose extensions to existing graph indexing frameworks to support exploratory queries. The inherent graph structure in provenance queries means they are subject to theoretical constraints on subgraph isomorphism which is known to be NP-Complete [31]. Thus, doing a subgraph isomorphism check for each graph in the collection will not scale. We propose a two-level framework that extends existing graph indexing techniques by incorporating summary

graphs that capture verification-free subgraph isomorphisms and discriminative features defined over the extended provenance collection. Our goal is to reduce the number of total subgraph isomorphism checks while at the same time allowing *less-specific*, and thus more exploratory, queries. The framework is rooted in the observation that even if we cannot resolve a query without any verification as in [29], we can reduce the number of subgraph isomorphism computations by finding a subset of the result set with limited verification.

4.3.1 Standard Graph Indexing

Standard graph indexing seeks to make subgraph containment queries over collections of graphs more efficient by limiting the number of subgraph isomorphism checks. Indexing strategies have primarily fallen into two categories: feature-based methods (see *e.g.,* [133, 166, 168]) and hierarchical organization [62, 162]. We focus on feature-based methods because for exploratory searches, users are often querying for specific (and often disconnected) features.

Feature-based graph indexing identifies features that aid in distinguishing graphs in a collection from each other. Each feature is linked to the graphs that contain it, and all features are organized in a hierarchy according to feature size. Queries are evaluated by identifying a set of features contained by the query and computing the intersection of the graphs associated with each feature. A graph must contain the same set of features as the query, but this is not sufficient as the features do not necessarily uniquely identify a graph. Thus, we must check whether each candidate graph is subgraph-isomorphic to the query. The fewer isomorphisms we compute, the faster the query execution. Thus, we wish to find a set of features that minimizes the size of the candidate set.

4.3.1.1 Identifying Features

The first ingredient in graph indexing is identifying features that will help to differentiate the graphs in our collection. To minimize the size of the index, we wish to find a set of features that serves to filter the collection into small subsets of graphs such that the features are not redundant. Formally, given a graph collection $\mathcal{G}$, a subgraph H is *frequent* with respect to a threshold T if $|\sup(H)| \geqslant T$ where the *support* of a subgraph H is

$$\sup(H) = \{G \mid H \subseteq G \in \mathcal{G}\}$$

Note that if a query graph contains a given frequent subgraph H, we can immediately exclude all graphs in $\mathcal{G}$ that are not in $\sup(H)$.

For a frequent subgraph H, any subgraph of H is also frequent because any graph that contains H must also contain all subgraphs of H. This means that there may exist a large number of frequent subgraphs when a dataset has a large pattern that occurs frequently. More generally, when frequent

subgraphs have similar support values, they serve to prune nearly the same set of graphs. We desire to select a smaller set of frequent subgraphs that still provides good pruning power. This implies that selected feature subgraphs should not significantly overlap. Given the collection $\mathcal{G}$ and a set of subgraphs $\mathcal{F}$, a subgraph F is *discriminative* if

$$\sup(F) \gg \cap_{F' \in \mathcal{F}, F' \subseteq F} \sup(F')$$

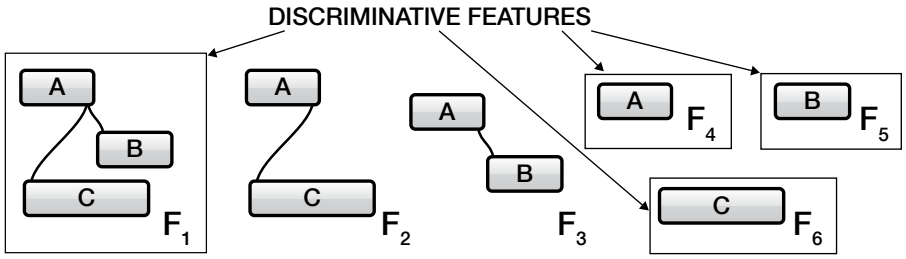
Figure 4.4 shows a set of frequent subgraphs, their respective supports, and the size of the intersection of the supports of their subgraphs. Note that F_2 and F_3 are well indexed by F_4, F_5, and F_6 and thus are not discriminative.

4.3.1.2 Index Construction and Query Processing

After identifying the discriminative features, we build an index by organizing the features into a hierarchy to facilitate apriori pruning. Note that this hierarchy may contain features that are not discriminative in order to simplify traversals during query processing. Because each feature is linked to a list of graphs that contain the feature, we can easily prune our search space for each feature in the query. A query is processed by starting with individual vertices and building features with increasing size by traversing the hierarchical index. Once we have the maximal features from the query, we intersect the lists of graphs associated with each of the features. The intersection of these graph lists forms the candidate set of graphs that may satisfy the query. Because we do not know if the candidates actually match the query, we must then verify each candidate by computing a subgraph isomorphism. Note that because subgraph isomorphism can be costly, it is important to have features which prune a large portion of the collection.

4.3.2 Wildcard Graph Indexing

Standard graph indexing techniques present two major issues when dealing with exploratory provenance queries. The first is that they usually assume that queries are connected graphs which is not necessarily the case when dealing with workflows. For example, suppose that a user wishes to find a workflow that uses a particular data source and produces a figure in a specific output format. In this case, the user does not care what the internals of the workflow are, so the standard containment query does not apply. A second issue is that answering queries with a large number of satisfying workflows may result in many subgraph isomorphism calculations. A vague query like one to find a common subworkflow might produce many candidates after filtering, all of which need to be verified. We introduce 2-component frequent subgraphs and summary graphs to address these issues.



$$|\sup(F_1)| = 10, |\sup(F_2)| = 32, |\sup(F_3)| = 35,$$
$$|\sup(F_4)| = 70, |\sup(F_5)| = 54, |\sup(F_6)| = 49$$
$$|\sup(F_4) \cap \sup(F_6)| = 36, |\sup(F_4) \cap \sup(F_5)| = 39,$$
$$|\sup(F_2) \cap \sup(F_3)| = 28$$

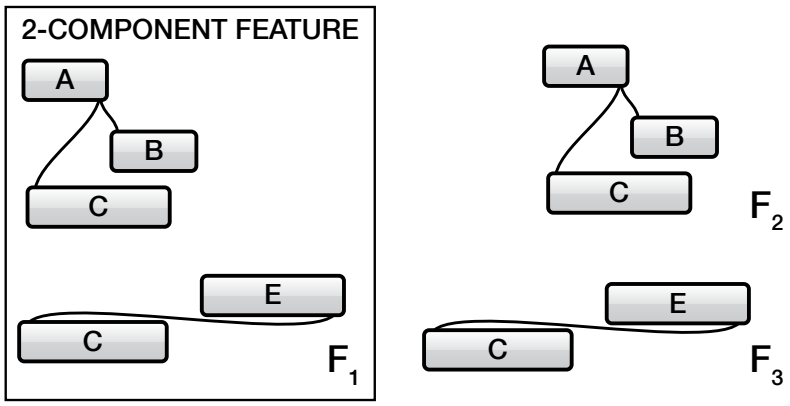
Figure 4.4: Because the graphs identified by a feature may also be identified by subgraphs of that feature, we choose *discriminative features* to be those whose subgraphs collectively identify many more graphs. For example, F_1 is selected because the graphs identified by the combination of F_2 and F_3 is $28 \gg 10$.

4.3.2.1 2-Component Frequent Subgraphs

Because exploratory queries frequently contain only pieces of a graph, we propose an indexing strategy that considers disconnected frequent subgraphs. Most existing frequent subgraph mining algorithms can be extended to also consider disconnected subgraphs. The problem with doing so is that the number of frequent subgraphs jumps exponentially. Any frequent subgraph with n vertices has on the order of 2^n possible disconnected frequent subgraphs that are also frequent. We can classify these disconnected subgraphs by the number of components. An m-component subgraph is a subgraph whose vertices can be partitioned into no fewer than m sets such that there does not exist any path from a vertex in one set to a vertex in another set. Including 2-component subgraphs in our set of frequent subgraphs only increases the number of frequent subgraphs by a quadratic amount. In addition, a frequent subgraph with more than n components contains $O(n^2)$ 2-component subgraphs so we still have a large number of features to help prune the search space. Figure 4.5 shows an example where the two-component subgraph F_1 filters many more graphs than F_2 and F_3. This usually occurs when the query identifies components as nonoverlapping, but many of the graphs indexed by the single-component features have them overlapping.

4.3.2.2 Summary Subgraphs

While frequent subgraphs prune the search space and help quickly locate graphs that may satisfy the query, we still need to check every graph that remains after pruning. This verification step involves the computation of a subgraph isomorphism, and this can be even more costly when



$$|\sup(F_1)| = 22, |\sup(F_2)| = 62,$$
$$|\sup(F_3)| = 57, |\sup(F_2) \cap \sup(F_3)| = 50$$

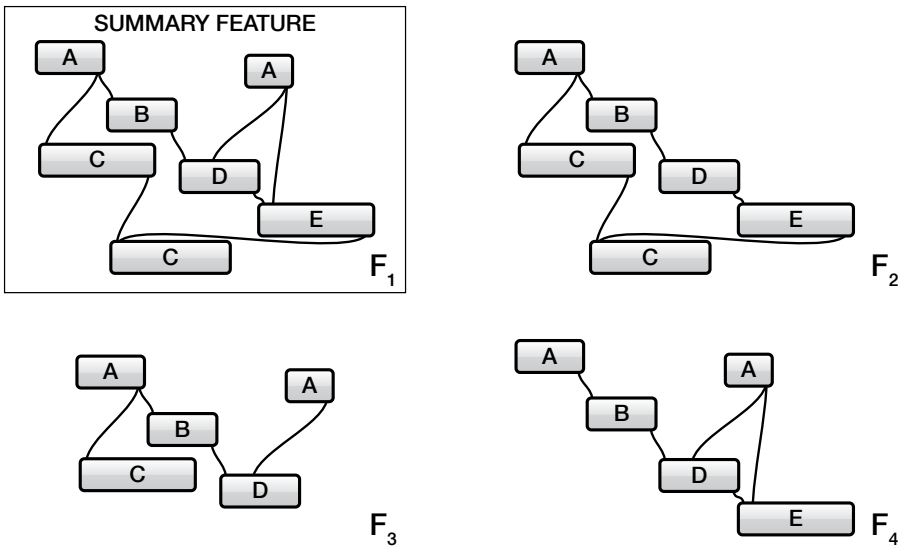
Figure 4.5: While the features F_2 and F_3 occur together often, they are usually disjoint as defined by the two-component feature F_1: $|\sup(F_1)| \ll |\sup(F_2) \cap \sup(F_3)|$.

wildcards are involved. Cheng *et al.* proposed FG-Index as a way to eliminate the verification step by noting that when the query is itself a frequent subgraph, the indexed graphs automatically satisfy the query [29]. While these verification-free answers are ideal, indexing all of the frequent subgraphs—not only discriminative ones—can lead to prohibitive index sizes.

We propose summary subgraphs as a scalable way to limit the number of verification steps. A summary subgraph F is linked to a subset of the graph collection where each graph G is a supergraph of F. Then, if a summary subgraph satisfies the query, we know that all of the graphs the summary subgraph indexes also satisfy the query. In addition, we will only include subgraphs that do not have immediate supergraphs that index a similar number of graphs. See Figure 4.6 for an example showing which subgraphs are selected as summary features. Formally, a subgraph F is a *summary subgraph* in a set of graphs $\mathcal{F}$ if for all $F' \in \mathcal{F}$, $F' \supseteq F$:

$$\sup(F) \ll \sup(F')$$

A summary subgraph is analogous to the δ-tolerance closed frequent subgraph [29], but we use them differently. When a query graph H is found to be a subgraph of a summary subgraph G, we know that all of the graphs that G indexes also satisfy H. Thus, if this single verification of $H \subseteq G$ succeeds, we avoid verifying *all* of the graphs G indexes. Note that H may satisfy other graphs; we leave the remaining graphs to either other summary graphs or basic verification using subgraph isomorphism. However, because mining features are required for feature-based indexing techniques, finding summary subgraphs takes minimal computation.



$$|\sup(F_1)| = 52, |\sup(F_2)| = 62,$$
$$|\sup(F_3)| = 57, |\sup(F_4)| = 60$$
$$|\sup(F_1) \cap \sup(F_2)| = 50, |\sup(F_1) \cap \sup(F_3)| = 51,$$
$$|\sup(F_1) \cap \sup(F_4)| = 50$$

Figure 4.6: Because each subgraph of a frequent subgraph is also frequent, we choose *summary features* to be those whose supergraphs have much smaller frequency.

4.3.2.3 Index Construction and Query Processing

Our index is composed of both summary and discriminative features. Both summary and discriminative features link to supergraphs of themselves that exist in the graph database, as illustrated in Figure 4.7. Because we need to identify the summary subgraphs during query processing just like any other candidate graph, our discriminative features will index to those graphs as well as those in the graph database. Additionally, after identifying the summary subgraphs, any graph indexing scheme can be applied to this extended graph database. Index construction begins by mining a set of connected and 2-component frequent subgraphs. Then, we identify the summary graphs and add them to the collection. Next, we create an index over the augmented collection; because we have already mined features, it is more efficient to use a feature-based scheme. As described earlier, discriminative features can be quickly extracted to index these graphs.

Query processing is similar to standard graph indexing schemes, except that we use 2-component frequent subgraphs to better filter candidates and shortcut the verification process using summary subgraphs. When pruning candidate graphs, we are able to use 2-component frequent subgraphs as features. In addition, our verification step begins by checking all summary graphs first, then

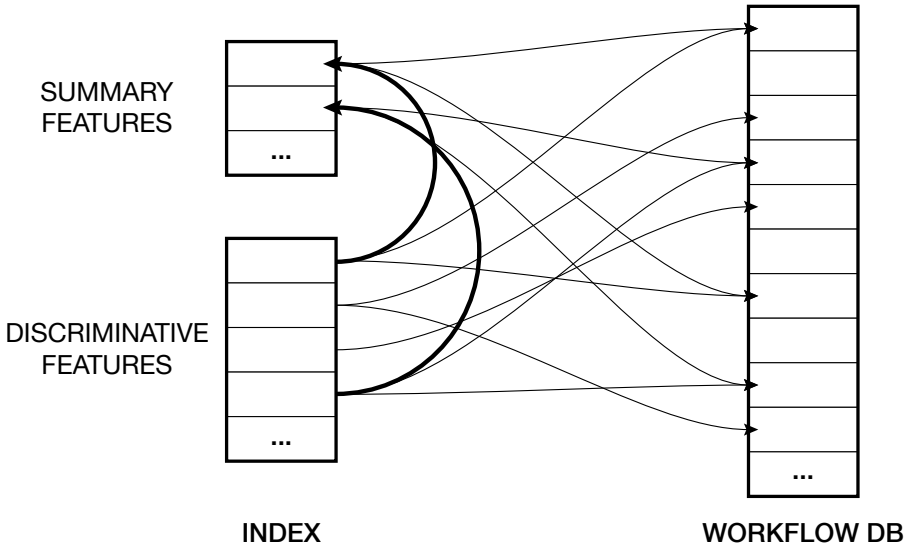


Figure 4.7: Our index has two tiers, the summary features which summarize frequent features and provide verification-free answers, and the discriminative features which point to both the original workflow database and the summary features. Note that for this illustration, many items have been omitted from the figure; in practice, each workflow is indexed by at least one discriminative feature.

verifying graphs that remain unverified by the summary graphs.

4.3.2.4 Verification

Given the summary graphs and discriminative index, our query processing proceeds like standard graph indexing with the exception that we choose to verify summary graphs *before* any of the graphs from our collection. As noted earlier, whenever a summary graph S satisfies the query graph, we immediately know that any graph indexed by S also satisfies the query graph. This means that we do not have to individually verify that entire subset of graphs. Note that if S does not satisfy the query, we cannot exclude the graphs indexed by S because summary graphs are inclusive rather than exclusive. However, we expect that frequent graphs will be summarized, and a query that has a candidate summary graph that does not verify will either be indexed by another summary graph or be infrequent.

4.4 Implementation

Our implementation of the index construction, query processing, and index maintenance is described in this section. Note that we do much of our processing by levels of the subgraph hierarchy.

4.4.1 Index Construction

As described earlier, there are two distinct steps in our index construction: summary graph selection and constructing an index for the collection augmented with the summary graphs. Summary graph selection requires frequent subgraph mining, and we also implement our discriminative index using frequent subgraphs. For that reason, we mine features from the entire collection and use those features for both steps. After mining, we choose summary graphs and construct a discriminative subgraph hierarchy over the augmented collection. See Figure 4.8 for an overview of the process.

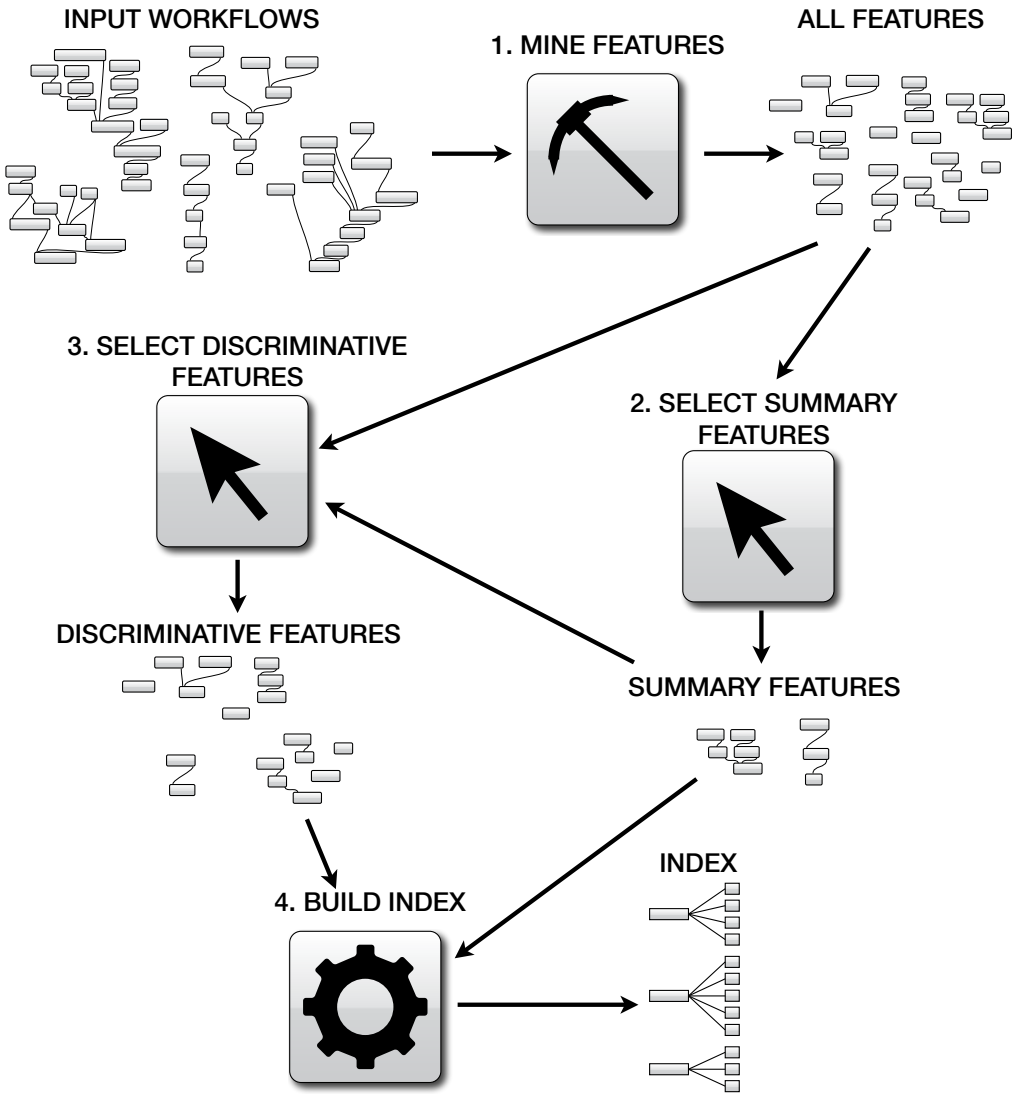


Figure 4.8: The construction of our index involves feature mining, followed by the identification of summary features, which are used to determine discriminative features and build the index.

4.4.1.1 Mining Frequent Subgraphs

Among existing subgraph mining algorithms, we have chosen to use the open-source implementation of gSpan [164] from Jahn and Kramer [74]. Because summary graph and discriminative feature selection involve comparing a graph against its super- or subgraphs, we maintain a directed acyclic graph $\mathcal{L}$ to manage these relationships. An *immediate subgraph* of a graph G is a subgraph G' where G' is missing exactly one of the edges from G (it may also be missing a vertex if the edge connects to a degree-one vertex). Then, as we mine features, we maintain links from every graph to its immediate subgraphs in $\mathcal{L}$. This will help us process graphs in levels where each level has all graphs with a set number of edges.

4.4.1.2 Generating 2-component Frequent Subgraphs

To mine 2-component subgraphs, we can either modify an existing algorithm like gSpan [164] or the Frequent Subgraph Miner [89] or process the set of connected frequent subgraphs and generate the 2-component subgraphs. For gSpan, the depth-first search can be amended to include a component number so that mining will consider disconnected subgraphs [165]. However, if we already have the set of connected frequent subgraphs, we can generate the 2-component frequent subgraphs by examining all pairs of frequent subgraphs, and checking whether the intersection of graphs matching both subgraphs contain them disjointly.

Given our subgraph relationship graph $\mathcal{L}$, we can build 2-component frequent subgraphs (2CFSGs) by checking pairs of subgraphs level-by-level up from single-vertex pairs. If any pair of subgraphs is not frequent, we need not continue to check supergraphs of that combination. Also, note that as we compute these 2CFSGs, we will update $\mathcal{L}$ to include the new subgraphs as well. The entire process is detailed in Algorithm 2.

4.4.1.3 Selecting Summary Graphs

To select summary graphs, we follow the principles outlined in Section 4.3.2, but again work on a level-by-level basis, in order to more easily determine whether a subgraph should be selected or not. We maintain the set of supergraphs for each subgraph in parent level, and work top-down. Then at any level, we can look up exactly what the cumulative supergraph support is without traversing the entire supergraph hierarchy. Algorithm 3 details this process.

4.4.1.4 Building the Discriminative Index

After selecting summary graphs, we build a discriminative index over the graph collection augmented with the summary graphs. As noted earlier, determining discriminative subgraphs is similar to selecting summary graphs except that we are concerned with exclusion here in contrast to

Algorithm 2: Mine Frequent Subgraphs

Input: A collection of workflows $\mathcal{G}$ and a threshold T
Output: A set $\mathcal{F}$ of frequent subgraphs of $\mathcal{W}$
MINESUBGRAPHS($\mathcal{G}$)
(1) $\mathcal{F} \leftarrow$ Run $\texttt{gSpan}(\mathcal{G}, T)$
(2) Add single-vertex features to $\mathcal{F}$
(3) *pairs* $\leftarrow$ all pairs of single-vertices from $\mathcal{F}$
(4) **while** *pairs* $\neq \varnothing$:
(5) **foreach** G_1, G_2 **in** *pairs*:
(6) *matches* $\leftarrow \sup(G_1) \cap \sup(G_2)$
(7) **if** $|$*matches*$| < T$
(8) **continue**
(9) $G' \leftarrow G(G_1, G_2)$
(10) **if** $|\{G \mid \text{VERIFY}(G', G)\}| < T$
(11) **continue**
(12) $\mathcal{F} \leftarrow \mathcal{F} + G'$
(13) $G_2^+ \leftarrow$ IMMEDIATESUPERGRAPHS(G_2)
(14) **foreach** $S_2 \in G_2^+$:
(15) *pairs* $\leftarrow$ *pairs* $+ (G_1, S_2)$
(16) $G_1^+ \leftarrow$ IMMEDIATESUPERGRAPHS(G_1)
(17) **foreach** $S_1 \in G_1^+$:
(18) *pairs* $\leftarrow$ *pairs* $+ (S_1, G_2)$

Algorithm 3: Select Summary Graphs

Input: A set $\mathcal{F}$ of frequent subgraphs of $\mathcal{W}$
Output: A subset of summary subgraphs $\mathcal{S}$
SELECTSUMMARYSUBGRAPHS($\mathcal{F}$)
(1) Sort $\mathcal{F}$ according to the number of edges in decreasing order
(2) **foreach** $G \in \mathcal{F}$:
(3) $G^+ \leftarrow$ IMMEDIATESUPERGRAPHS(G)
(4) **if** $G^+ = \varnothing$ or $|\cup_{G' \in G^+}$ *supports*$(G')| > T$
(5) $\mathcal{S} \leftarrow \mathcal{S} + G$
(6) *supports*$(G) \leftarrow \sup(G)$
(7) **else**
(8) *supports*$(G) \leftarrow |\cup_{G' \in G^+}$ *supports*$(G')|$

summary graphs which emphasize inclusion. Recall a subgraph is discriminative when its subgraphs that are also discriminative filter many fewer graphs. Thus, we work on a level-by-level basis, but work from the bottom-up instead of top-down as we do with the summary graphs. Algorithm 4 formally expresses this idea.

4.4.2 Query Processing

Given a workflow query q in graph form, we break the graph q into features. Note that these features may be 2CFSGs, so we begin with single-vertex features and grow them into progressively

Algorithm 4: Build Index

Input: A collection of workflows $\mathcal{W}$ and summary graphs $\mathcal{S}$
Output: An index $\mathcal{T}$ for the collection
BUILDINDEX($\mathcal{W}$, $\mathcal{S}$)
(1) $\mathcal{G} \leftarrow \mathcal{W} \cup \mathcal{S}$
(2) Sort $\mathcal{G}$ according to the number of edges in increasing order
(3) **foreach** $G \in \mathcal{G}$
(4) $G^- \leftarrow$ IMMEDIATESUBGRAPHS(G)
(5) **if** $G^- = \varnothing$ or $|\cap_{G' \in G^-} \mathit{supports}(G')| > T$
(6) $\mathcal{T} \leftarrow \mathcal{T} + G$
(7) $\mathit{supports}(G) \leftarrow \sup(G)$
(8) **else**
(9) $\mathit{supports}(G) \leftarrow \cap_{G' \in G^-} \mathit{supports}(G')$

larger (and possibly disconnected) features according to the tree we maintain as part of the discriminative index. When we have determined the maximal features, we compute the intersection of the sets of graphs linked to these features. Note that the resulting set of candidates $\mathcal{F}^*$ may contain graphs from the collection and summary graphs.

Next, we must verify the set of candidates. Recall that if a summary graph S satisfies the query, we immediately know that all of the graphs that S represents also satisfy the query. Thus, we may be able to avoid some individual verifications by checking summary graphs *first*. However, if a summary graph does not satisfy the query, we cannot assume that the graphs it indexes do not satisfy the query; another summary graph may verify them or we may have to check them individually. Algorithm 5 details this process. For workflow queries that do not involve wildcards, these verification steps are just subgraph isomorphism checks. However, wildcard queries require an extension of subgraph isomorphism. Figure 4.9 illustrates the entire process.

4.4.2.1 Wildcard Query Verification

Because wildcard queries are essentially disconnected graphs with special wildcard edges, we can delete the wildcard edges and run the query over the disconnected graphs. However, during verification, we need to evaluate whether the wildcard edges are satisfied by the candidate graph: we need to check the *wildcard subgraph isomorphism* problem defined in Section 4.2. Since we have chosen to mandate that paths between the two vertices on each side of the wildcard edge must not contain vertices already matched to the query graph, we cannot evaluate this query using standard transitive closure. Instead, we do this evaluation with a standard connectivity search on the candidate graph by excluding any vertices already matched, using a depth-first search. Note that this requires computing the matching generated from a subgraph isomorphism, and checking other subgraph isomorphism answers if a suitable answer has not been found.

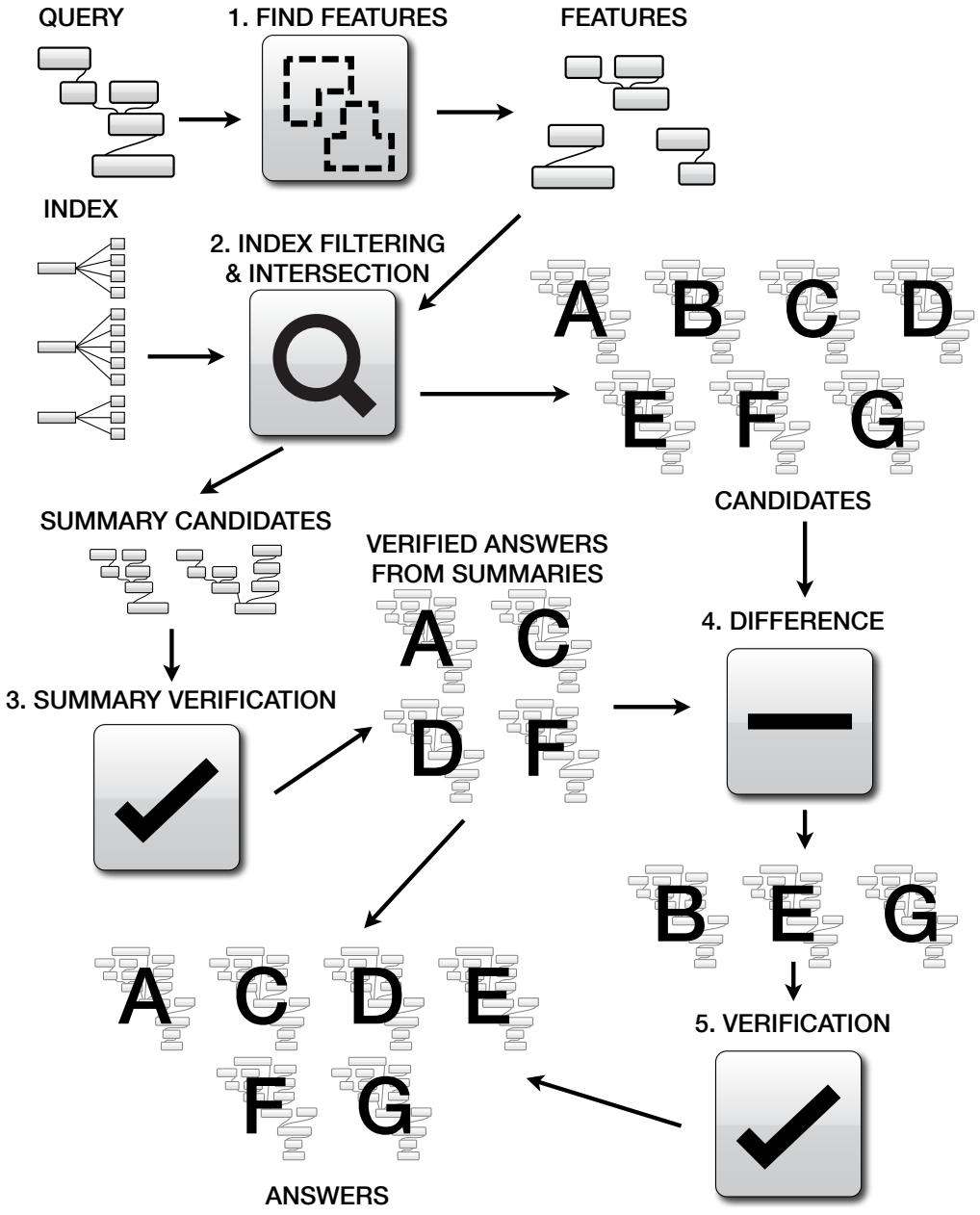


Figure 4.9: Query processing is faster because the discriminative index limits the number of candidates and summary graphs limit the number of computationally-expensive verifications.

Algorithm 5: Process Query

Input: Query workflow Q, index $\mathcal{T}$, collection of summary graphs $\mathcal{S}$, collection of workflows $\mathcal{W}$
Output: A subset $\mathcal{A}$ of $\mathcal{W}$ that satisfy Q

PROCESSQUERY(Q, $\mathcal{T}$, $\mathcal{S}$, $\mathcal{W}$)
(1) $\mathcal{A} \leftarrow \mathcal{W}$
(2) *features* $\leftarrow$ single-vertex features of Q
(3) **foreach** $F \in$ *features*
(4) **if** $F \in \mathcal{T}$
(5) *features* $\leftarrow$ *features* +
(6) IMMEDIATESUPERGRAPHS(F, Q)
(7) $\mathcal{A} \leftarrow \mathcal{A} \cap \mathcal{T}(F)$
(8) Sort $\mathcal{A}$ so that all $G \in \mathcal{S}$ are first
(9) **foreach** $G \in \mathcal{A}$
(10) **if** VERIFY(Q, G)
(11) **if** $G \in \mathcal{S}$
(12) $\mathcal{A} \leftarrow \mathcal{A} + \mathcal{S}(G)$
(13) **else**
(14) $\mathcal{A} \leftarrow \mathcal{A} - G$
(15) $\mathcal{A} \leftarrow \mathcal{A} - \mathcal{S}$

4.4.3 Index Maintenance

Because selecting frequent features requires mining, we cannot expect to regenerate the entire index on the fly. Note, however, that both levels of the index can be updated to include new graphs using existing summary and discriminative features. Unfortunately, it is difficult to discover new frequent subgraphs after index creation as this requires recomputing the mining. As has been discussed in other work [166], we can wait until a certain number of graphs have been added or deleted and then recreate the index. In addition, we will always index base features like single vertices or edges to ensure that new graphs will appear in search results. Thus, while the quality of the index may degrade until it is recreated, it should not degrade too quickly. We may also choose to recreate the discriminative index without updating the summary subgraphs, or we can recalculate both levels of the index.

4.5 Workflow Completions

We can leverage exploratory workflow queries to suggest workflow completions, *i.e.,* how a partial workflow might be filled in. Similar to how textual completion works for programming environments [103] or search fields [57], workflow completion seeks to provide suggestions to users as they construct workflows. For example, in Figure 4.10, we show possible completions for a modeling workflow. These completions are derived from an existing provenance collection and aggregate workflow query results in order to rank putative completions. In [85], we showed that using automatically generated completions, the effort to create workflows is substantially reduced.

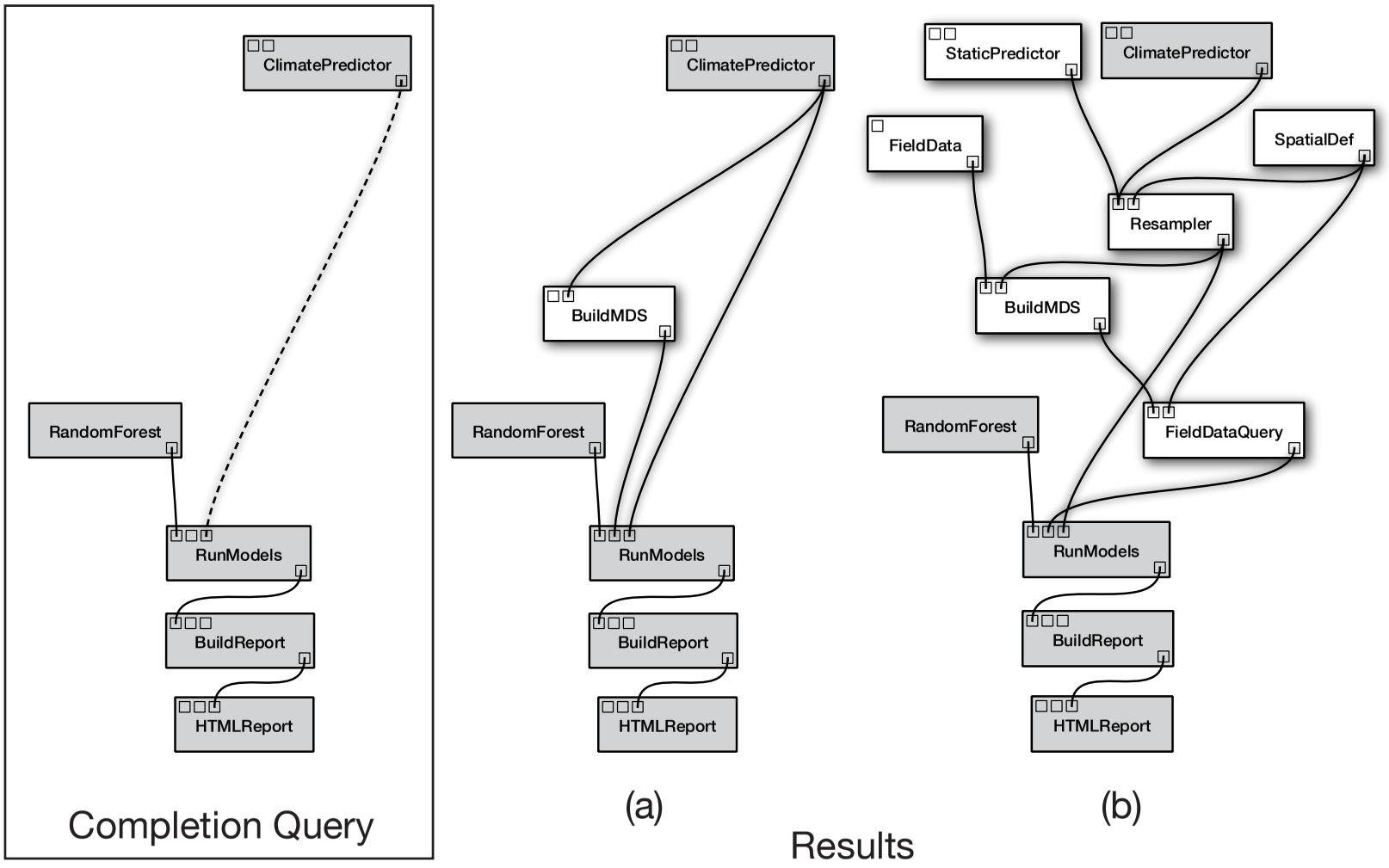


Figure 4.10: Workflow completions are generated from a completion query by replacing wildcards with modules and connections according to existing workflows in a collection.

In our experiments, we observed a reduction of over 50% in the the number of operations performed by users to construct workflows.

Formally, a *workflow completion* of a partial workflow w is a (nonpartial) workflow w' such that the workflow query represented by w is satisfied by w'. Thus, each wildcard module is matched to some module in the completion and each wildcard connection maps to some path in the completion. Note that a completion need not do more than satisfy each wildcard constraint and match non-wildcard modules and connections. Thus, completions may be simple, just filling in the minimum structure (see Figure 4.10(a)), or more complex, adding additional branches (see Figure 4.10(b)). Also, since a completion is the result of a workflow query, improvements in the efficiency of workflow queries translate directly to improvements in workflow completion.

4.5.1 Implementing Workflow Completions

We can augment our indexing framework to support workflow completions by capturing the paths that satisfy the connectivity constraints in wildcard queries. During wildcard query verification, we can capture these paths and suggest them as completion paths. Note that we can expand a completion path to include modules and connections attached to the path. However, we always add only vertices and edges that do not appear in the query graph (*i.e.,* the existing workflow). This ensures that the user does not see suggestions that reflect the workflow pieces they have already constructed.

Ideally, completions should be ranked based on their importance. If one completion occurs in hundreds of workflows and another occurs only once, we would like to present the more prevalent one first. Note that a match to a summary graph guarantees that a completion there occurs frequently. Thus, after verifying only the summary graphs, we might immediately present the user with suggestions for completing a workflow. We can continue to generate suggestions from the other graphs in the background, but allow the user to see the initial suggestions quickly.

4.6 Evaluation

4.6.1 Theoretical Costs

The total cost of a workflow query q using a standard graph index is

$$C(q) = C_f + |\mathcal{I}(q)|C_v$$

where C_f is the filtering cost, C_v is the cost of verification, and $\mathcal{I}(q)$ is the set of candidate graphs from the index given the query q. Our improved index splits subgraph isomorphism checks into two classes; we pay more up front in the hope of reducing the total number of verifications. Let

$\mathcal{S}(\mathcal{I}'(q))$ denote the set of summary graphs identified by the index for a query q and $\sup(\mathcal{S}^+)$ denote the workflows indexed by the subset that satisfies q. Then, the total cost is:

$$C(q) = C_f + |\mathcal{S}(\mathcal{I}'(q))|C_v + |\mathcal{I}'(q) - \sup(\mathcal{S}^+(\mathcal{I}'(q)))|C_v$$

Note that when $\sup(\mathcal{S}^+)$ is large, we avoid many verification steps as a single summary graph verification check suffices. In general, we seek to minimize $|\mathcal{I}'(Q)|$ while at the same time maximizing $|\sup(\mathcal{S}^+)|$. The worst case is when the index identifies a set of summary graphs, none of which satisfy the query ($|\sup(\mathcal{S}^+)| = 0$); we perform extra verifications but because they are all negative, we do not gain anything. However, these cases are rare as we expect frequent queries from users, and any candidate sets for nonfrequent queries should be limited by the discriminative features. In most cases, the summary graphs will provide fewer verification checks and thus faster query times.

4.6.2 Data Sets

We evaluated our techniques using two provenance collections. The first comes from a set of visualization workflows and the second from Yahoo! Pipes workflows [163]—both are sets of dataflows. The collection of 6,117 visualization workflows was generated over two years by 60 different users. The users were assigned specific tasks and generated workflows to solve these problems. As such, we expect some overlap in the overall structure of the workflows, although there should be some variation throughout. There were 150 different types of modules involved in these workflows. The second collection is a set of 40,505 Yahoo! Pipes workflows used to construct Web mashups. Here, the number of module types used was only 54, and users tended to follow very similar patterns in workflow development. As such, there were few frequent patterns but many occurrences of them.

We ran queries selected at random from the entire set of mined frequent subgraphs for these data sets. In order to test the effectiveness of the summary subgraphs and the addition of 2-component frequent subgraphs, we performed tests with both features enabled (S+2C), only summary graphs enabled (S), and both features disabled (Orig.). Note that when both are disabled, the framework is similar to gIndex [166]. For the Yahoo! Pipes data with features that occurred at least 500 times and the summary and discriminative thresholds also held at 500, we were able to compute answers to queries with an average of over 700 candidate graphs with only 20 isomorphism checks. For the visualization workflows, with the same parameters all at 200, we were able to compute answers to queries with an average of over 380 candidate graphs with only 64 checks.

Our prototype uses a `sqlite3` database and `python` code to construct the index and perform queries. Results for tests of the visualization dataset are shown in Figure 4.11a; note that for queries with few results, our technique performs slightly more verifications than the number of results,

but for queries with many results, the summary graphs help dramatically reduce the number of verifications. Figure 4.11b shows similar results for queries of frequent subgraphs in the pipes dataset.

We also performed tests with a range of thresholds for summary graph selection and discriminative feature selection. As expected, smaller thresholds produced better results in both cases. Figure 4.12a shows that decreasing the discriminative threshold significantly decreases the number of isomorphisms. Examining Table 4.1, we see that the number of graphs in the index does not increase significantly either. Figure 4.12b shows that decreasing the summary threshold also decreases the number of isomorphisms, although here we do not see as pronounced an effect. This can be explained by the fact that frequent subgraphs with no supergraphs are always included. Thus, the internal nodes, where the threshold matters, play a less significant role.

Figure 4.13 shows that the number of edges involved in the query has an effect on the mean ratio of the isomorphisms computed; smaller graphs are more often summarized which is to be expected. The subtraction of 2-connected features, however, does not follow the same pattern. In addition, the original feature-based strategy does not have a ratio of one because the summary graphs are computed as candidates but not verified.

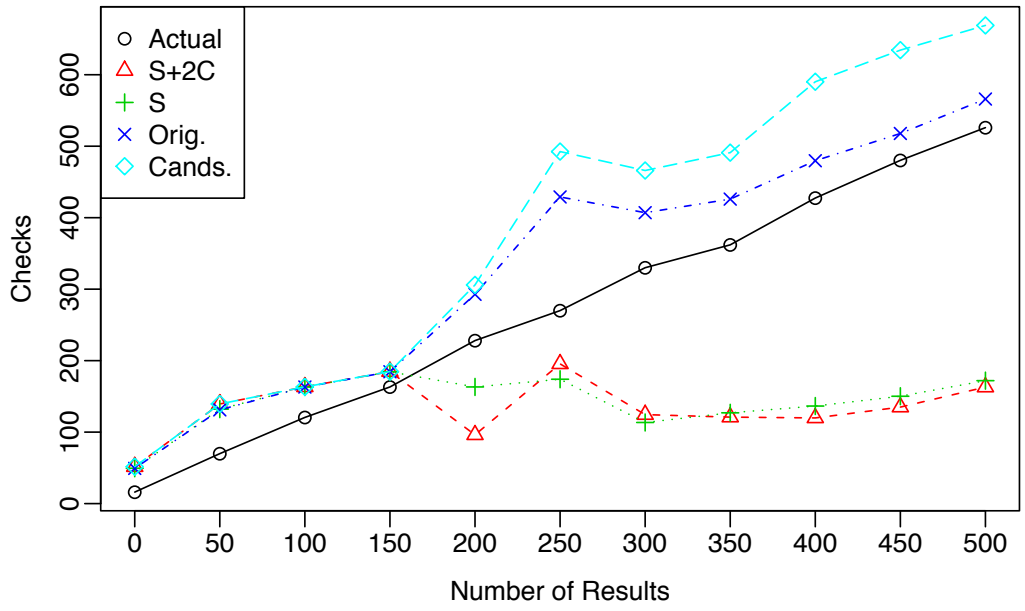
4.7 Discussion

4.7.1 Subworkflows

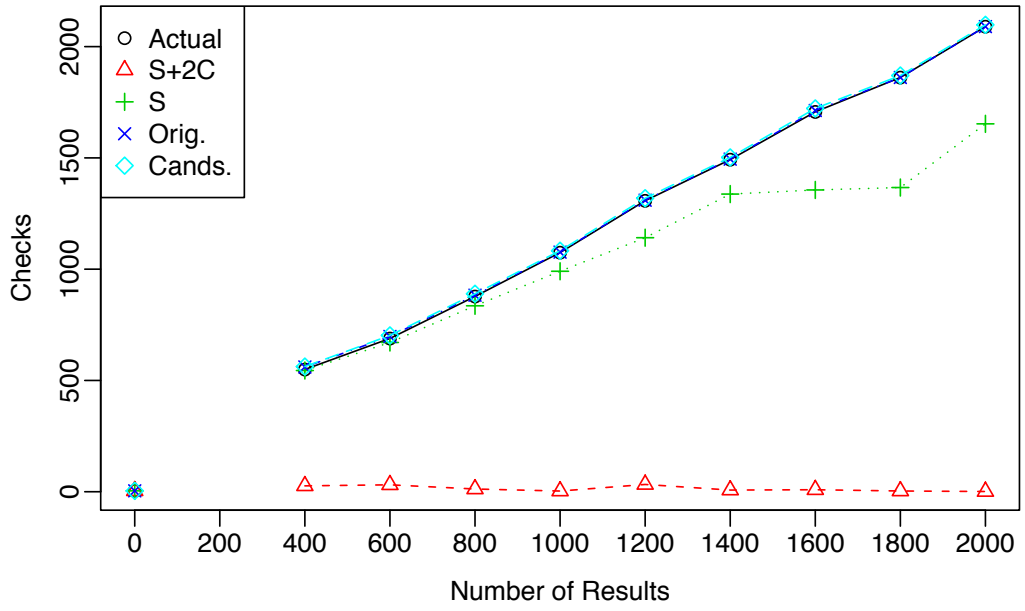
Many workflow systems offer operations to abstract parts of a workflow into subworkflow or compound modules, whose computation is the execution of another workflow. Note that a workflow can be expanded (or refined [18]) by replacing its subworkflow modules with the actual specifications for the subworkflows. If we add the restriction that subworkflows cannot be included circularly, we can always expand both query workflows and workflows in the collection. When indexing, we always index the fully-expanded workflow, and fully-expand a query before evaluating it. Note that in some cases, it may be worthwhile to also index unexpanded workflows, specifically when a subworkflow is very common, because it will reduce the time involved in mining.

4.7.2 Scalability

Since a given frequent subgraph has n connected subgraphs of its own, it may have $O(n^2)$ 2-component subgraphs. Thus, the number of 2-component frequent subgraphs can quickly become unreasonable. Note that a large pattern will generate many frequent subgraphs, but because most of them are summarized by the pattern, it is not necessary to mine all of them. If a 2-component subgraph has a supergraph that is a summary graph, and the difference in their supports is minimal, we need not worry about it or its supergraphs. A test using the visualization dataset confirmed

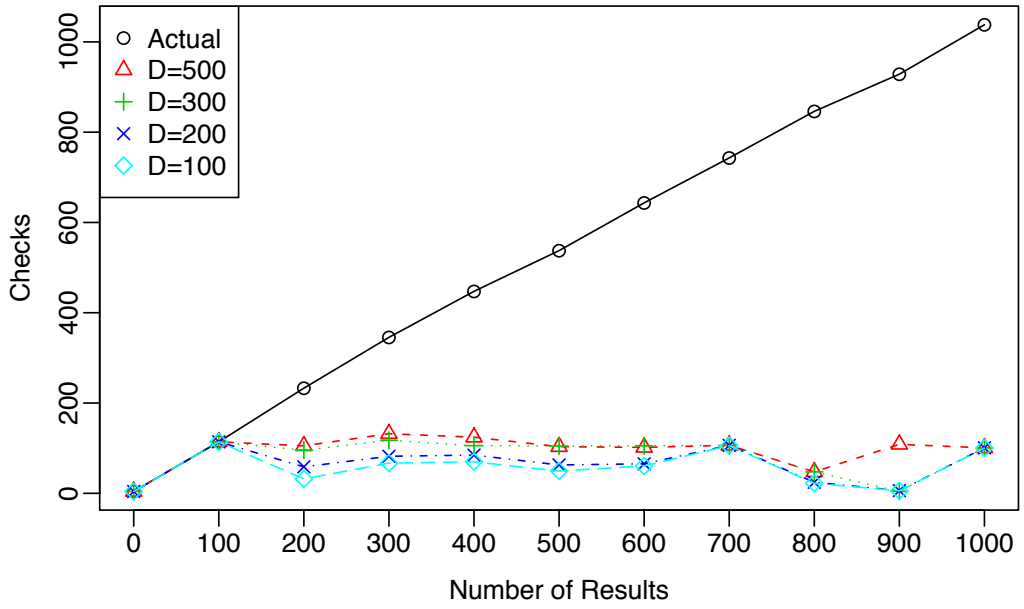


(a) Visualization Data

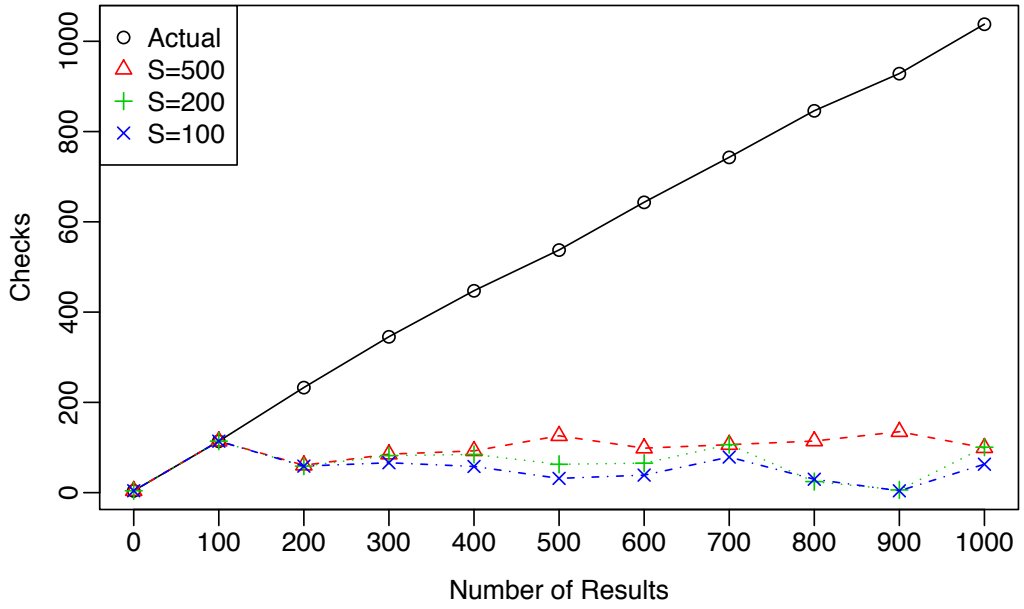


(b) Pipes Data

Figure 4.11: Comparison of the number of subgraph isomorphism verifications required for queries with different numbers of results across different indexing schemes. For both the visualization workflows (a) and the Yahoo! Pipes workflows (b), we used the proposed scheme having both summary features and 2-component subgraphs (S+2C), a scheme using only summary features (S), and the original feature-based indexing scheme (Orig.). The actual number of results is plotted as a baseline (Actual) as well as the number of candidates (including summary graphs) after filtering for the proposed scheme (Cands.).



(a) Discriminative Thresholds



(b) Summary Thresholds

Figure 4.12: The effect of varying the thresholds for identifying the (a) discriminative and (b) summary features for the proposed index.

Table 4.1: Statistics for various thresholds for the visualization dataset. Note that the number of summary graphs increases significantly as the threshold decreases.

Summary Threshold	100	100	100	100	200	200	200	200	500	500	500	500
Discriminative Threshold	500	300	200	100	500	300	200	100	500	300	200	100
# of Summary Graphs	5260	5260	5260	5260	2607	2607	2607	2607	1449	1449	1449	1449
# of Discriminative Graphs	237	248	261	291	234	244	258	289	233	246	259	284
# of Total Nodes in $\mathcal{T}$	330	370	431	840	353	411	454	691	343	675	488	622

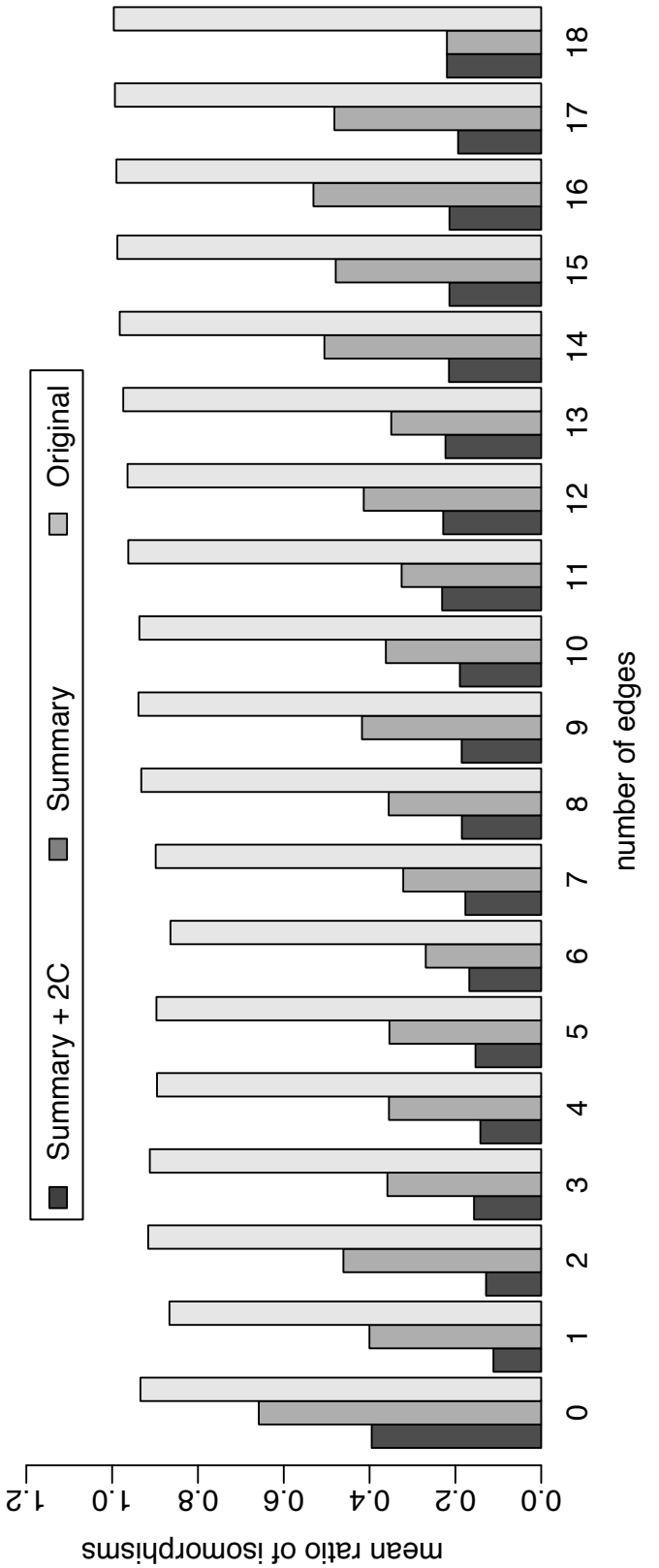


Figure 4.13: Mean ratio of the number of isomorphisms computed to the number of matching graphs according to the number of edges in the query graph, shown for the proposed scheme (Summary + 2C), only summary graphs (Summary), and the original feature-based scheme (Original).

that with 25,708 frequent connected subgraphs, we would have 304,626 frequent 2-component subgraphs, but running with the maximal heuristic, we can reduce this to 14,025 2-component subgraphs.

4.7.3 Parameters

Although our focus in this work is on structural queries, we can also integrate parameters and connection types into our approach. Connection types are edge labels, and we easily integrate this information into existing algorithms. We could encode parameter information as part of the module type and use the existing algorithms as well, but because parameters are more likely to vary across workflows, this will likely lead to a much larger index. Instead, we propose that parameters be indexed separately and queries processed by joining results from the parameter index with those from the structural index. This also allows for queries that involve comparisons of parameters (*e.g.,* $\textit{temperature} < 98.6$); an exact structural index where parameter values are baked into module types would not be very effective.

4.8 Related Work

Although there has been work on frameworks and interfaces for querying workflow collections [18, 130] and for using these collections to derive recommendations for users as they design workflows [85], not much attention has been given to performing these tasks efficiently. To the best of our knowledge, ours is the first proposal for an index structure that supports the efficient evaluation of exploratory queries over collections of workflows.

There is a substantial body of work on techniques for indexing graph databases to speed up subgraph queries. Much of that work has focused on mining and utilizing frequent features that serve to differentiate graphs. Index features range from single vertices and edges to paths and graphs. Zhao *et al.* [168] provide a discussion and empirical analysis of the role and granularity of paths, trees, and non-tree subgraphs. GraphGrep [133] uses paths as index elements, gIndex [166] uses discriminative subgraphs, and Tree + Δ [168] uses selective trees along with graphs generated on-demand. These techniques work in two separate steps. First, there is a filtering step that uses the index to prune graphs that do not match the query by intersecting the matches for each feature to generate a set of candidate graphs. Then, in the verification step, each candidate graph is checked via a subgraph isomorphism calculation against the input query. The goal of the pruning step is to minimize the number of subgraph isomorphism checks required to evaluate the query. Note that because of these checks, the query time scales according to the number of results. Chen *et al.* have attacked the problem of searching collections of large graphs by using randomized summaries [25]; note that their summaries each characterize a *single* graph while our summary features index multi-

ple graphs.

FG-Index [29] improves on the feature-based techniques by avoiding the expensive verification step when a query exactly matches an indexed feature. The FG-Index contains entries for all frequent subgraphs in a tree structure with an edge-set lookup. This was later extended to FG*-Index [28] which incorporated a feature-based lookup as well as on-demand indexing. The size of the FG-Index is dependent on the number of frequent subgraphs; thus, if a collection has a frequent subgraph G with 20 edges, all subgraphs of G (a set with close to 2^{20} elements) will also be frequent. Although it is inefficient to keep all such subgraphs indexed, by indexing only the largest subgraph, we can still quickly return answers for queries that are subgraphs of the "large" subgraphs.

Wildcard queries are also supported by XML query languages. A number of approaches have been proposed for XML indexing to efficiently support wildcard queries that have path expressions involving ancestor and descendant axes. These range from encoding parent-child and ancestor-descendant relationships via numbering schemes [94] and using structure-encoded sequences for documents [159] to indexing a subset of the data paths in an XML document [26], However, these techniques are specifically designed for tree models and cannot be directly applied to graphs.

4.9 Summary

We have presented a new indexing strategy to support exploratory queries over provenance collections. The proposed two-level indexing framework combines the pruning power of discriminative features of graphs with verification-free answers from indexed frequent subgraphs. Our results show that the addition of summary graphs and 2-component subgraphs significantly reduces the number of verification steps for queries that have many results. In fact, for the Yahoo! Pipes dataset, this number was nearly constant even as the number of results increased. Our framework is also flexible: the two enhancements are naturally orthogonal, and we can integrate either with other existing graph indexing schemes. In addition, we are able to apply our framework to suggestions for workflow completions. As the amount of provenance data continues to grow, understanding and using this information requires efficient support for the exploratory queries that users are interested in.

CHAPTER 5

VISUAL SUMMARIES FOR GRAPH COLLECTIONS

5.1 Introduction

From molecular structures to social networks and workflows, graph collections are widely available. While visualizing a single graph has been an important step in understanding these data, given large graph collections, it becomes crucial to analyze the differences and similarities in the collection. The questions of how a graph differs from some norm or where edges or nodes change across time are more important than being able to view each graph individually. We introduce *summary graphs* to synthesize collections of graphs to a single, interactive visualization. See Figure 5.1 for an example summary graph.

There are a variety of domains where graphs are common, and being able to better understand relationships between graphs is important. Molecules are well-known structures that can be represented as graphs, and understanding structural differences can help inform physical or biological processes. More recently, there has been a great deal of interest in social networks, and one might consider analyzing groups of users across different networks by comparing relationships. Metabolic pathways, the chemical interactions between enzymes and compounds in a cell, are also represented as networks, and pathways in different organisms often vary, allowing researchers to construct phylogenies using information about these differences. With structured computations like visualization pipelines or workflows, we also have a graph structure that can be analyzed for differences.

Graphs are usually best understood via visual encodings, and there has been significant work in algorithms to lay out and draw them. However, most of this work has focused on single graphs. There are also algorithms to calculate distances between graphs or find maximal common subgraphs. Here, most of this work has focused on pairwise comparisons of graphs. With more than two graphs, understanding the similarities and differences can be a challenge with pairwise techniques. In addition, these comparisons tend to be strict, with little flexibility when vertices or edges are not exactly equal. The ability to gather a slightly "fuzzy" mapping between graphs is important when summarization is the goal.

One of the major challenges in understanding relationships between graphs is that there is no

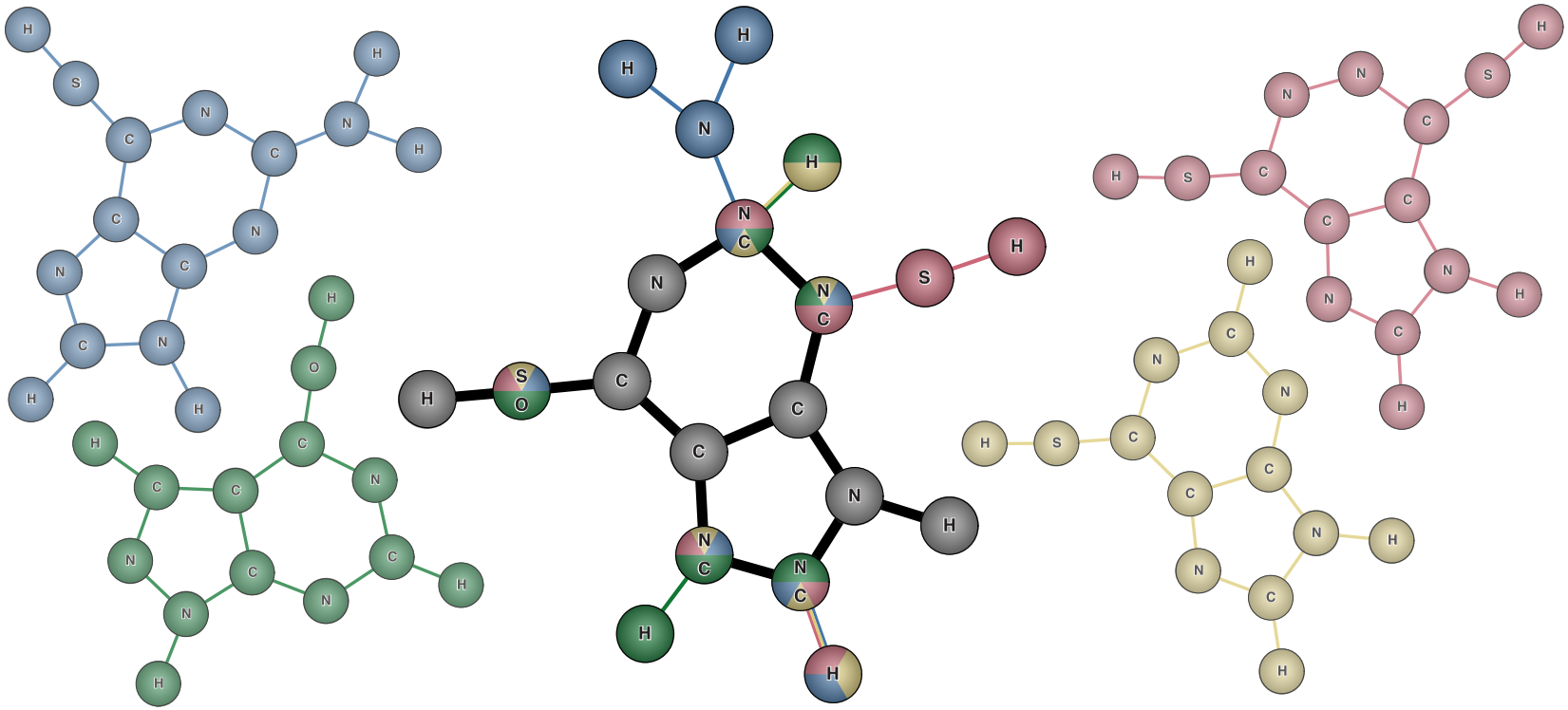


Figure 5.1: A summary graph constructed from four molecules. The supplemental video shows the order of summarization and how edit operations allow users to tune the visualization.

known computationally efficient method for comparing them. Testing whether one graph is an exact subgraph of another (subgraph isomorphism) is NP-Complete, and relaxing constraints to allow for greater freedom in any matching leads to even more possibilities. Unfortunately, exhaustive algorithms tend to take too long because of the number of possible alignments, and heuristic algorithms cannot be guaranteed to produce a good result. However, in many cases, a collection of graphs contains significant overlap between individual graphs and locating possible matches is less daunting than the general problem. We present a set of methods that are effective at matching pairs of graphs by solving the assignment problem on a matrix that encodes cost estimates for matching vertices. By first diffusing vertex similarities across a product graph, we can also integrate some global connectivity information.

Given approaches for comparing pairs of graphs, we can build a summary graph using hierarchical agglomeration. From two graphs, we can construct a simple summary by combining matched vertices and edges and connecting unmatched pieces of each graph. We recursively combine initial and intermediate graphs until we end up with a single summary. Note that it may become apparent that certain graphs do not share many (if any) similarities with the rest of the collection, and those might be disregarded. An initial clustering step can help organize a set of graphs into logical collections where summary graphs are appropriate.

Visualizing this computed summary graph should allow users to explore the similarities and differences between graphs. As such, it is important to be able to recognize which pieces are unique to a subset of graphs or common to most graphs. For large collections where a general understanding is desired, we use lightness to indicate the relative occurrences of graph elements. By highlighting a single graph with color in the context of the summary, one can understand how that graph relates to the collection. Projecting a selected subset of graphs and using color to differentiate them allows one to see how their specific relationship mirrors or differs from the summary. See Figure 5.2 for an example of how color is used in a summary graph.

Finally, while our computed summaries usually offer a good initial picture of the collection, we provide tools to allow users to interactively explore and update the summary graph. By ordering matches according to a confidence measure, a user can control how much summarization occurs. At a lower-level, users can select vertices and break or join them according to their wishes. Thus, blemishes in an initial configuration can be quickly remedied. At the same time, this guidance can be used to rerun the summarization process with a user's preferences. In addition, these operations are animated so a user can see exactly where nodes are split or joined as the layout is updated. This is useful when navigating the amount of summarization.

In this chapter, we formally define summary graphs and show how they can be constructed and visualized. Their construction is aided by an extension to an existing matching algorithm that

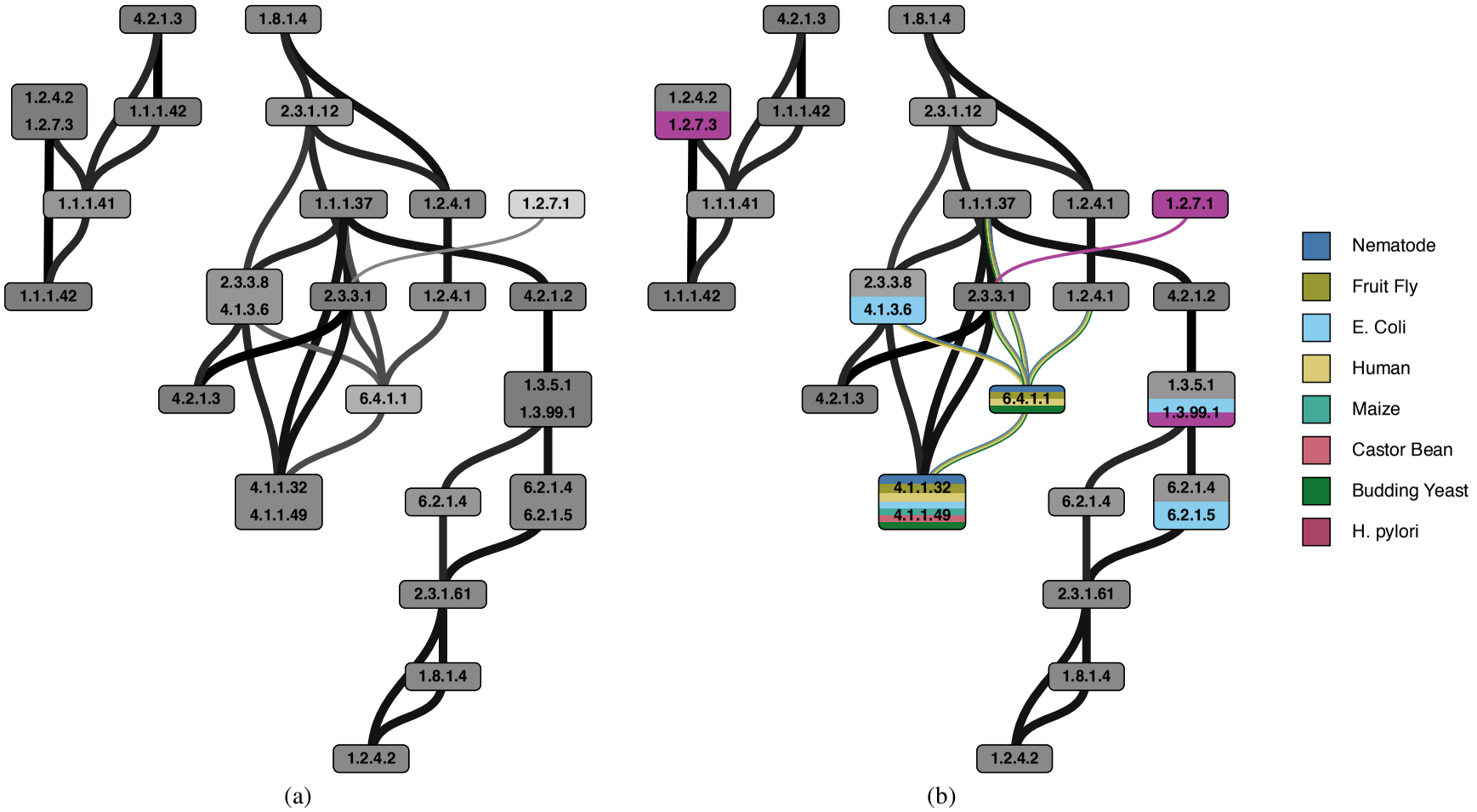


Figure 5.2: A summary graph of enzyme relation graphs from the citric acid cycle for eight organisms. Notice that color can highlight differences while levels of gray indicate how common a graph component is.

produces concise, inexact matchings. In addition, we present operations for interactively editing summary graphs as well as different modes for analyzing these visualizations. Finally, we present applications that demonstrate the utility of our technique.

We begin by reviewing related work, and then build the definitions and computational machinery for computing matches between pairs of graphs in Section 5.3. In Section 5.4, we formally define a summary graph and detail how to construct one from a graph collection. Next, we describe how we visualize and interact with summaries in Section 5.5. In Section 5.6, we provide case studies for uses of summary graphs, and we conclude in Section 5.7 with a discussion of both shortcomings and possible extensions.

5.2 Related Work

There has been substantial work in the area of graph visualization, ranging from layout algorithms to methods for visualizing large graphs to interacting with graphs. However, the problem of visualizing multiple graphs in a single view has been largely overlooked. Herman *et al.* provide a good summary of the early work on graph visualization and navigation [66].

There has been work on comparing and visualizing two or more trees at once. Furnas and Zacks suggested Multitrees as a way to integrate sets of hierarchical information [51]. The InfoVis 2003 Contest generated significant work in tree comparison [119]. Much of it was rooted in work done for consensus trees used for phylogenies in the biological community [2]. Munzer *et al.*'s TreeJuxtaposer tackled the problem of comparing large trees using focus+context and visibility criteria [108]. Tu and Shen showed how to encode changes in treemaps [152], and Graham and Kennedy used directed acyclic graphs to agglomerate multiple trees [58]. They also provide a comprehensive survey of work in the area of visualizing multiple trees [59]. Isenberg and Carpendale explored how collaborators can share comparison information [73]. There has been some work to compare *pairs* of graphs visually, including the visual diff for comparing two workflows by Freire *et al.* [48]. The difference between pairs provenance graphs was considered by Bao *et al.* [14].

In contrast with the approach described in this chapter that aims to summarize a collection of graphs, much of the work on graph summarization has focused on single, large graphs. Such graphs can be clustered or summarized with regions collapsed into smaller entities (*e.g.,* [44]). Other approaches have used topology [12] and interaction [52, 155] to better navigate graphs. In addition, edges can be bundled, allowing users to better identify connectivity when graphs have large numbers of edges [34, 69]. Level-of-detail can also be used to more efficiently navigate large graphs [13]. There are also techniques for multivariate graphs that focus on relationships between nodes [160]. We note that while our focus is on combining graphs into a single visualization, as summary graphs are graphs, this work can also be applied to them.

Also related to our work are techniques for computing matches between graphs. The diffusion matching we present is an extension of the similarity flooding work by Melnik *et al.* [100] and the analogy matching from Scheidegger *et al.* [130]. Our work, however, uses a different formation that is rooted in the concepts of graph edit distance. Riesen and Bunke suggest an approximation for graph edit distance that uses a solution to the assignment problem [124], and Zeng *et al.* use a similar formulation for graph searching [167]. We also use a solution to the assignment problem to solve our final matrix after performing a diffusion step. Heymans and Singh use another variant to compare metabolic pathways to generate phylogenetic trees [67].

5.3 Graph Matching

Generating a summary graph requires that common substructures in a given collection of graphs be merged. Our algorithm for constructing summary graphs depends on pairwise merges, and graph matching plays an important role in determining these merges. We are not, however, concerned with the strict, exact matching that is often considered in graph theory. Rather, we wish to find the best inexact matches so as to maintain a compact set of vertices and edges. When two graphs have nodes that are exactly the same and their neighborhoods are similar, it makes sense to match them. However, there are often nodes that, while not equal, are also very similar. This similarity may be due to the fact that they are known to be related or because they are used in similar contexts in their respective graphs. For example, in chemistry, it is known that sodium and potassium are very similar atoms as they appear in the same group in the periodic table. However, while sulfur and nitrogen are less similar, if they are bonded to similar neighborhoods, it may still be reasonable to match them.

Graph edit distance is a measure of graph similarity, and its delineation between substituting, adding, and deleting components provides a framework for identifying when a merge is appropriate. However, computing graph distance can be slow and approximations do not take into account global connectivity. Similarity flooding uses a Markov chain on a product of two graphs to diffuse similarities and determine matches with the influence of connectivity information [100]. We propose extensions to similarity flooding that incorporate concepts from edit distance approximations and provide information to consider when vertices are best left unmatched. We begin by defining graphs, graph matchings, and the quality of a match. After reviewing graph edit distance and similarity flooding, we present our enhanced matching algorithm.

5.3.1 Definitions

A *graph* $G = (V, E)$ is a set of vertices V and set of edges E where each edge $e = (v_1, v_2) \in E$ connects two vertices $v_1, v_2 \in V$. In an *undirected* graph, each edge is specified by an *unordered*

pair of vertices, and in a *directed* graph, the edges are specified by *ordered* pairs of vertices. Note that an undirected graph can be transformed into an equivalent directed graph by creating a pair of edges, one in each direction, for each edge. Because our graph comparisons rely on information contained at vertices or on edges, we define labels for each. A *labeled graph* is thus a graph with labeling functions $L_V : V \rightarrow L$ and $L_E : E \rightarrow L$ where L is a finite set of labels.

5.3.1.1 Matching

There are several ways to define a matching between two graphs, ranging from exact equality to any one-to-one mapping between the two. Graph isomorphism involves finding a mapping from one graph to the other such that each vertex and edge is matched, meaning each pair of labels corresponds and the vertices of each edge also match. Subgraph isomorphism allows for an injective map from a smaller graph to a larger one where the larger graph is allowed to have unmatched vertices and edges. The decision version of the subgraph isomorphism problem has been shown to be NP-Complete [31], making an efficient exact solution to this problem unlikely. Also note that both of these problems consider exact matches on vertex and edge labels. We would like to allow more flexibility by also matching vertices and edges that have similar, but not equivalent, labels. Furthermore, we also wish to permit unmatched vertices in both graphs as some nodes are best unmatched.

More formally, a graph isomorphism between two graphs $G = (V, E)$ and $G' = (V', E')$ is a bijection h between their vertices V and V' such that for each edge $e = (v_1, v_2) \in E$, there exists an edge $e' = (v'_1, v'_2) \in E'$ such that $h(v_1) = v'_1$ and $h(v_2) = v'_2$. For labeled graphs, we also have the restriction that $L_V(v) = L_{V'}(h(v))$ and $L_E((v_1, v_2)) = L_{E'}((h(v_1), h(v_2)))$. We wish to allow any mapping, regardless of whether it matches all vertices or if there are label mismatches. Thus, a *graph matching* for two graphs G and G' is any map $h : V \rightarrow V'$. Note that this definition allows a wide variety of matchings, ranging from an empty map to a full isomorphism. Since there can be multiple valid graph isomorphisms for two graphs, there are many more of these general matchings.

Because of the number of possible matchings, we have some qualitative measure to evaluate their utility. Any such measure is based on functions that define the similarity between nodes and edges. These functions may be, most strictly, based only on the data represented by the node or edge, but this is often relaxed to allow a comparison of neighborhood structure. Such generalized scoring functions allow improved measures of similarity in a more global setting. Formally, a vertex similarity function S_V can be defined for two graphs $G = (V, E, L_V, L_E)$ and $G' = (V', E', L_{V'}, L_{E'})$ as $S_V : V \times V' \rightarrow \mathbb{R}$. An edge similarity function S_E can be similarly defined. Strict similarity functions may be defined without considering the vertices themselves, only the labels. Note that a similarity function can also be represented inversely as cost; components with

high similarity have low cost.

Using these measures, which may be domain-specific, we can define the score of a graph matching. Given $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, S_V, and S_E, we define the *score* of a graph matching h as

$$\text{score}(h) = \sum_{v \in V_1} S_V(v, h(v)) + \sum_{e \in E_1} S_E(e, h(e))$$

Then, a *maximal graph matching* is a graph matching h where $\text{score}(h)$ is maximized. Note that there cannot be any claim of uniqueness; for example, a graph with any symmetry will usually have at least two matchings that attain the same, maximal score.

5.3.1.2 Graph Edit Distance

While similarity functions can help guide users to the best correspondence between a pair of graphs, note that for functions that define strictly positive scores, it is always best to match as many vertices and edges as possible. Allowing negative scores or using thresholding [100] can aid in discarding poor matches, but note that it can be instructive to have some measure of when it is best not to match nodes. Graph edit distance establishes a cost for transforming one graph into another. Such a transformation consists of three types of operations: substitutions, additions, and deletions. This allows a direct comparison between the cost of substituting a vertex in one graph for another versus the cost of removing the vertex and adding the other. In the context of graph matching, this means we can identify when to leave vertices or edges unmatched. We will show that we can use a solution to graph edit distance to construct a maximal graph matching.

Formally, given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ and cost functions $C_{va} : V_2 \to \mathbb{R}$, $C_{vd} : V_1 \to \mathbb{R}$, $C_{vs} : V_1 \times V_2 \to \mathbb{R}$, $C_{ea} : E_2 \to \mathbb{R}$, $C_{ed} : E_1 \to \mathbb{R}$, $C_{es} : E_1 \times E_2 \to \mathbb{R}$, the *graph edit distance* between G_1 and G_2 is

$$\min_{V_a, V_d, V_s, E_a, E_d, E_s} \sum_{v \in V_a} C_{va}(v) + \sum_{v \in V_d} C_{vd}(v) + \sum_{v \in V_s} C_{vs}(v) + \\ + \sum_{e \in E_a} C_{ea}(e) + \sum_{e \in E_d} C_{ed}(e) + \sum_{e \in E_s} C_{es}(e)$$

The solution to this minimization is not straightforward because the selection of which vertices are substituted affects the selection of edge actions. Thus, E_a, E_d, and E_s are not independent of V_a, V_d, and V_s.

However, note that if, during the computation of graph edit distance, we track the composition of $V_a, V_d, V_s, E_a, E_d, E_s$, we can construct a graph matching. V_s and E_s define exactly which nodes and edges, respectively, should be matched. Note that this does not match our original definition of maximal because we now have some measure of when components should be left unmatched. This

suggests an extension to our similarity functions: expand the domain to include unmatched entities. Let ϵ represent leaving a vertex unpaired. Then, our vertex similarity function becomes

$$S_V : V_1 \cup \{\epsilon\} \times V_2 \cup \{\epsilon\} \rightarrow \mathbb{R}$$

and the edge similarity function can be extended in the same manner. Note that $S(\epsilon, \epsilon)$ does not play a role in the computation. Then, vertex additions are scored by $S(\epsilon, v_2)$ and vertex deletions by $S(v_1, \epsilon)$. This allows us to directly compare $S(v_1, v_2)$ with $S(v_1, \epsilon) + S(\epsilon, v_2)$.

5.3.2 Computing Graph Edit Distance

Graph edit distance is also NP-Complete so an efficient solution for it is unlikely [167]. In practice, for small graphs, A* can be used to evaluate the entire space of possible solutions. For larger graphs, we can use heuristics that rely on estimates to prune the search space.

5.3.2.1 A* Search

To compute graph edit distance, we can explore the space of all possible vertex matchings, since they will induce edge costs [124]. Note that for any vertex matching, we can choose substitution or an add/delete pair based on which operation(s) have a lower cost. Then, we can iteratively compute the cost of a node as the cost of its parent plus the cost of matching the two vertices indicated by the node. This cost includes both the node similarity and the costs of the induced edge operations. The number of leaves in this search tree is $n!/(n-m)!$ for $m < n$, meaning that there is an exponential number of cost computations.

Because of the high computational cost, it is impractical to compute the full A* tree for anything other than small graphs. As with other search algorithms, we can use pruning strategies to help cut down these costs. The key ingredient is being able to determine which branches are promising, usually accomplished by estimating the remaining cost down the tree. Given such a heuristic, we can evaluate all active nodes by estimating the remaining cost, and keep only the top (or top-k in beam search). This reduces the number of computations to k per level, for a total of $k \cdot \min\{m, n\}$, which will clearly be faster.

5.3.2.2 Edit Distance and the Assignment Problem

For these algorithms to be practical, we must have a reasonable estimate for graph edit distance. If we discard costs for edges, we only need consider the assignment of vertices. For the substitution costs for vertices in two graphs G_1 and G_2, we have the following matrix:

$$\begin{pmatrix} s_{00} & s_{01} & \cdots & s_{0n} \\ s_{10} & s_{11} & \cdots & s_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m0} & s_{m1} & \cdots & s_{mn} \end{pmatrix}$$

where s_{ij} is the cost of substituting $v_j \in G_2$ for $v_i \in G_1$. Because each node can be matched to at most one other node, we can select at most one entry from each row and at most one from each column but want minimal cost. Furthermore, a full assignment for an $m \times n$ matrix requires that we have all rows in columns in the smaller dimension covered. This is exactly the assignment problem [22], and it has a polynomial time solution via the Hungarian algorithm [88].

One issue is that we would like to include the add and delete costs as well. Riesen and Bunke proposed a matrix that encodes these costs and still allows for a solution via the Hungarian algorithm [124]. The matrix has four blocks

$$\left(\begin{array}{cccc|cccc} s_{00} & s_{01} & \cdots & s_{0n} & a_0 & \infty & \cdots & \infty \\ s_{10} & s_{11} & \cdots & s_{1n} & \infty & a_1 & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{m0} & s_{m1} & \cdots & s_{mn} & \infty & \infty & \cdots & a_m \\ \hline d_0 & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & d_1 & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \cdots & d_n & 0 & 0 & \cdots & 0 \end{array}\right) \tag{5.1}$$

where the upper-left block contains all substitution costs (s_{ij}), the lower-left and upper-right blocks contain add/delete costs (a_k and d_ℓ), and the lower-right block is free. The trade-off between substitution and addition/deletion is enforced by setting off-diagonal entries in the upper-right and lower-left blocks to ∞, meaning that the assignment must, for each left column, either select a substitution or a deletion, and for each upper row, either select a substitution or an addition. Because we must assign each row/column (we can pad to a square matrix for rectangular matrices), we can obtain a minimal cost for vertex operations.

5.3.2.3 Including Neighborhood Information

While we can efficiently solve the vertex assignment problem given vertex similarity functions, such a solution does not take into account any edge information. This is problematic, as Figure 5.3b shows, because connectivity is a major component of a graph; otherwise we are simply matching sets of vertices. Connectivity can also influence when vertices should be matched. For example, when evaluating whether a vertex in one graph matches one in another, it is reasonable to consider a correspondence if all of the neighbors in the first graph correspond to the neighbors in the second graph.

However, note that for our vertex assignment matrix, there is no restriction that the costs must correspond to the vertex similarity function. We can choose to define the entries in the cost matrix as a combination of the vertex similarity cost and an estimate of the cost involved in matching

neighborhoods. Instead, we can add the costs for matching adjacent edges of each pair of vertices [124]. For such a pair, the best correspondence for edges can also be solved via a solution to the assignment problem for the edge sets. Note that this will produce a *locally* optimal cost, and because the constraints are not global, the local costs cannot usually be reconciled. An edge correspondence for one pair of vertices may pair two edges differently than a similar correspondence for an adjacent pair of vertices. Figure 5.3c shows that the detected asymmetry between the A and E nodes does not propagate to correctly pair the C vertices.

5.3.3 Diffusion Matching

In order to incorporate global connectivity into the vertex assignment matrix, we need greater diffusion of similarity. An iterative refinement like similarity flooding [100] allows individual similarities to influence the entire graph. This refinement works by constructing a product graph whose vertices are all pairs of nodes from the original graphs and whose edges indicate correlations between pairs. One issue with the original approach is that it uses thresholding to determine when nodes should be unmatched; we improve this by integrating both the assignment matrix solution from graph edit distance and skip nodes, additional vertices added to the product graph that track scores for keeping vertices unmatched.

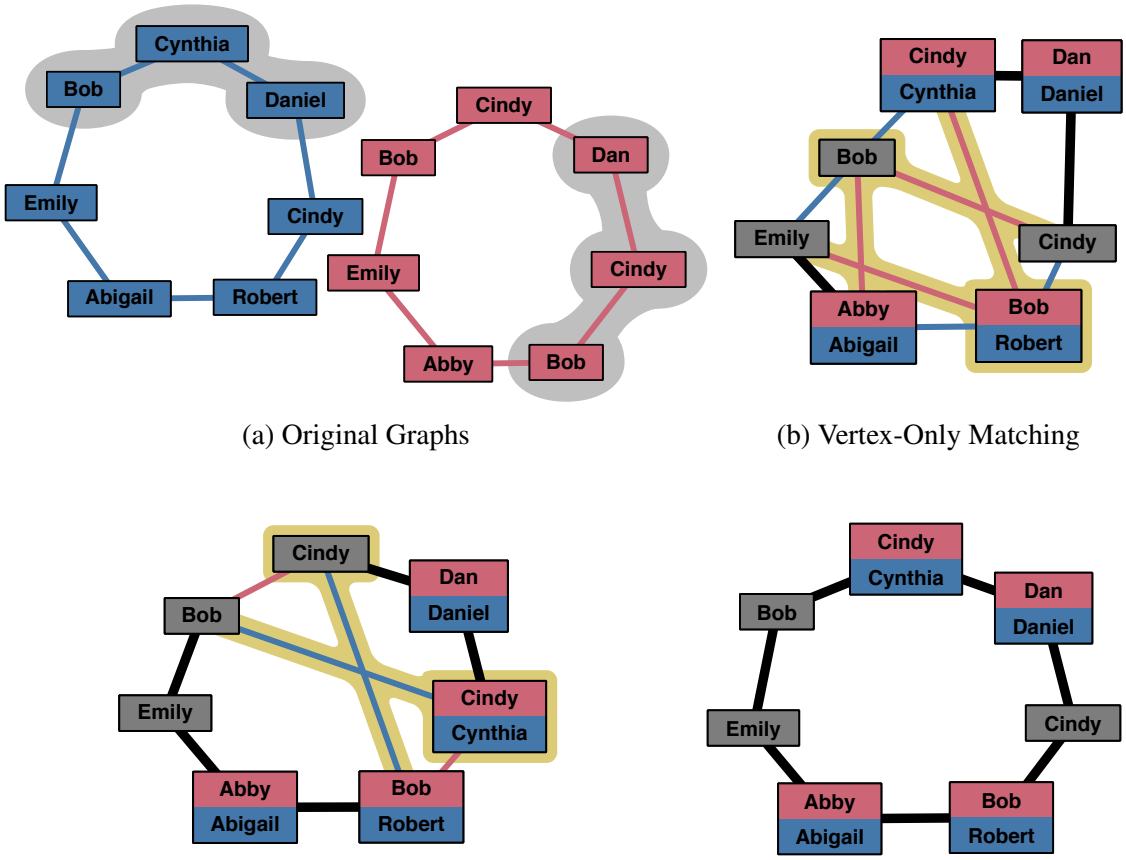
5.3.3.1 Similarity Flooding

To begin, we create a product graph so that we can diffuse costs between vertex pairs using the combined connectivity. Given graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, the *direct product graph* $G^*(V^*, E^*) = G_1 \times G_2$ where $V^* = V_1 \times V_2$ and

$$E^* = \{((u_1, u_2), (v_1, v_2)) \mid (u_1, v_1) \in E_1, (u_2, v_2) \in E_2\}$$

Note that this graph differs from the Cartesian product defined in other settings (*e.g.,* [75]), where we have a union of edge sets instead of an intersection; diffusion should only occur when an edge exists in *both* graphs. Also, for directed graphs, we must preserve order so we will not create a product edge from the simple edges (u_1, v_1) to (u_2, v_2) if $(u_1, u_2) \in E_1$ and $(v_2, v_1) \in E_2$. That said, these edges are *undirected*; paired edges indicate a connection between vertices and the direction of that edge is irrelevant for diffusion.

Then, for each edge $e^* \in G^*$, we need to assign a *diffusion score* S^* that represents how well-linked the new product vertices are. More accurately, we need to decide how costs should propagate from a product vertex to its neighbors. We can calculate this from some combination of S_V and S_E. When edges are dissimilar, S_E is small, and we do not wish to propagate as much similarity between the connected nodes. From these scores, we construct an adjacency matrix $A(G^*)$ for the



(a) Original Graphs

(b) Vertex-Only Matching

(c) Neighborhood Matching

(d) Global Matching

Figure 5.3: A comparison of graph-matching algorithms when run on the same two starting graphs, shown in (a); mismatched vertices and edges in (b) and (c) are highlighted. A vertex-only matching (b) has issues with mismatched edges (*e.g.,* the two Bob nodes from the red graph match the Bob and Robert nodes from the blue graph equally well). A neighborhood matching will correct some errors because it takes into account neighboring nodes, but it will not propagate this information to other nodes. For example, the neighborhoods of the two Cynthia/Cindy nodes in each graph (one neighborhood from each graph is highlighted in (a)) will match equally well and may cause an edge mismatch as shown in (c). Global methods, like A* search and diffusion matching seek to resolve such problems, leading to matchings without mismatched edges (d).

product graph. Each entry (i, j) corresponds to a pair of paired vertices, $((s, t), (u, v))$, and we assign its value to the diffusion score $S^*((s, t), (u, v))$. We normalize A so that the sum of each row is one; when a row is all-zero, we can factor that row out.

For each vertex $v^* = (v_1, v_2) \in G^*$, we can define a similarity score via $S_V(v_1, v_2)$. Computing this for each pair, we end up with a vector c which is again normalized. This will allow us to infuse each step of the diffusion with vertex scores. Let π_k measure the pairwise similarity and α be a parameter that controls the amount of diffusion versus initial similarity. Then, we have the following formula from [130]:

$$\pi_{k+1} = \alpha A(G)\pi_k + (1 - \alpha)c(G) = M\pi_k$$

which corresponds to the power method for eigenvector computation, according to the structure of M. Thus, this iterative process will converge to π_∞ which corresponds to the similarities after diffusion.

The output similarity vector π_k can be reshaped to a vertex assignment matrix, and we can proceed in a similar fashion to that outlined for graph edit distance. Note that because of the diffusion, this matrix contains scores that better incorporate connectivity information. Figure 5.3d shows that this approach allows us to compute a better match than a vertex-only or neighborhood-based matching.

5.3.3.2 Scoring Unmatched Nodes

In the usual product graph employed in similarity flooding, we have no way to determine if a node would be best unmatched. Thus, if we use the output vertex assignment matrix directly, we will attempt to pair all vertices. We can prune all matches that fall below a certain threshold or relative threshold [100], but note that we have no score on which to base this decision. Our approach extends the original product graph with *skip nodes*, nodes that represent unpaired vertices from the original graphs. They can be integrated into the normal diffusion scheme, and their final scores can be used to decide whether a substitution is more optimal.

We construct an *augmented product graph* $\hat{G}$ from G_1 and G_2 which is a product graph with additional vertices $(v, \epsilon), v \in V_1$ and $(\epsilon, v), v \in V_2$, and additional *directed* edges

- $((u_1, u_2), (v_1, \epsilon)), u_1, v_1 \in V_1, u_2 \in V_2 \cup \epsilon, (u_1, v_1) \in E_1$
- $((u_1, u_2), (\epsilon, v_2)), u_1 \in V_1 \cup \epsilon, u_2, v_2 \in V_2, (u_2, v_2) \in E_2$
- $((u_1, \epsilon), (v_1, v_2)), u_1, v_1 \in V_1, u_2 \in V_2, (u_1, v_1) \in E_1$
- $((\epsilon, u_2), (v_1, v_2)), u_1 \in V_1, u_2, v_2 \in V_2.(u_2, v_2) \in E_2$

Note that we do not add (ϵ, ϵ), and edges between skip nodes are relegated to those from the same original graph that are connected. In addition, edges to and from a skip node to the original graph are allowed when there are edges in a graph with the specified vertex. Note that the direction of the

edges is important when determining the adjacency matrix A. We can use information about other edges on the way into a skip node, but the probability of taking a possible edge back to the original product graph must be uniform.

Given our augmented similarity functions S_V^* and S_E^*, we can determine initial scores for unmatched vertices and construct a slightly larger adjacency matrix. The augmented vertex similarity function allows us to bias particular nodes to stay unmatched. For example, for molecules, we might define the similarity of hydrogen (usually a node with a single edge) with ϵ to be higher so that we end up merging too many nodes. Both our adjacency matrix A and similarity vector c are of larger dimension to accommodate the new nodes and connections, but our computation proceeds as before, producing a new vertex similarity matrix. Now, however, we can solve the assignment problem on the matrix shown in Equation 5.1 since we have scores for keeping vertices unmatched. Finally, because this is still essentially a matrix of vertex similiarities, we can incorporate the same neighborhood information used in graph edit distance to amplify vertex similarities.

5.4 Summary Graphs

A *summary graph* for a collection of graphs is a graph where each vertex (or edge) represents one or more vertices (or edges) from the graphs it summarizes. More formally, given a collection of graphs $\{G_1 = (V_1, E_1), \ldots, G_n = (V_n, E_n)\}$, a summary graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph such that there exists a surjective map $M : \cup_{i \in [1,n]} V_i \to \mathcal{V}$ where

- For each $v \in V_i$, there exists a vertex $v^* \in \mathcal{V}$ such that $M(v) = v^*$.
- For each $e = (v_1, v_2) \in E_i$, there exists an edge $e^* = (v_1^*, v_2^*) \in \mathcal{E}$ such that $M(v_1) = v_1^*$ and $M(v_2) = v_2^*$.
- For each $v^* \in \mathcal{V}$, there exists $v_i \in V_i$ such that $M(v^*) = v$.
- For each $(v_1^*, v_2^*) \in \mathcal{E}$, there exists $v_1 \in V_i, v_2 \in V_i$ such that $(v_1, v_2) \in E_i$.

As with graph matchings, there are many M which satisfy the criteria, and therefore, there are many summary graphs for a given collection. For example, M might map every vertex to a distinct vertex in the summary graph, generating a summary graph that is the union of the individual graphs from the collection. Thus, while we can be assured of the existence of a summary graph for any collection, it is important that we have a method to compare and evaluate these summaries.

A summary graph of two graphs can be generated from a graph matching in a straightforward manner. If nodes are matched, we create a new compound node; if not, we create individual nodes representing the unmatched nodes. More formally, given a graph matching $h : V \to V'$ for graphs G, G', we can create a summary graph $\mathcal{G}$ by defining the surjective map $M : V \cup V' \to \mathcal{V}$ as

$$M(v) = \begin{cases} (v, h(v)) & v \in V \text{ and } v \text{ is in the domain of } h \\ v & v \in V \text{ when } v \text{ is not in the domain of } h \\ v & v \in V' \text{ and } v \text{ is not in the range of } h \end{cases}$$

With a higher-dimensional matching relation, we could compute a summary graph from it in a similar manner.

As mentioned earlier, the ability to measure the quality of a summary graph is important for generating meaningful and succinct summary graphs. We extend the notion of similarity (Section 5.3) to define a *maximal* summary graph. However, we must first extend our similarity functions to compare more than two simple vertices or edges. We define a *compound* vertex as a vertex in a summary graph that represents one or more vertices (each of which may also be compound) from the graphsx being summarized. A *compound* edge is similarly defined as representing one or more edges from the graph collection.

5.4.1 Compound Similarity Scoring

With only two graphs, the initial similarity of two vertices is obtained via our user-defined similarity function. However, when the two vertices being compared are from summary graphs, this similarity can be computed in various ways. Formally, we have the node similarity function S_v^*, and two compound vertices v and v', each from different intermediate summary graphs. We know that v and v' both represent a nonempty set of vertices from the initial graph collections: $v = (v_{i_1}, \ldots v_{i_m})$ and $v' = (v_{j_1}, \ldots v_{j_n})$. Note that m need not be equal to n. We wish to define $S_V : \mathcal{V} \to \mathbb{R}$, and we can choose to compute the overall similarity as an average of all pairs, the sum of similarities for the best matches, or even the single best similarity. Because this may again be domain-specific, we allow users to define the combination method. In general, an overall average would seem to make the most sense as we are effectively determining the similarity of an entire set of vertices.

5.4.2 Construction

Solutions to the assignment problem become NP-Hard in dimensions higher than two [22]. Thus, extended graph matching approaches to compute the best matching for n graphs become impractical. Instead, we use hierarchical agglomeration to build the summary graph using a series of pairwise matchings. At each step, we combine two graphs by computing their matching and build a summary graph according to the matching. Recall that hierarchical clustering requires similarities for each pair of graphs. Because computing matchings for all pairs is inefficient, we use heuristics based on feature vectors of the node data to order the agglomeration. Given this ordering, we compute each of the matchings in the hierarchy to construct a summary graph. Note that for some collections, combining all graphs may not be desirable if the clusters are far apart.

A summary graph contains data from all of the graphs it represents, and its vertex and edge similarities are a combination of the comparisons of the individual data elements using the combined similarity scoring. Each time we merge two graphs, we create a new graph with compound vertices and edges. Each label becomes a combination of the labels from the individual vertices; note that we can eliminate duplicates for succinctness. From the map M for vertices, we infer the edges based on those correspondences; when two pairs of vertices are matched and there is an edge in both of the input graphs, we create an edge in the summary graph.

From the assignment matrix, we can generate a node ordering for each match. In order to generate a global sequence of *node* merges, we use the graph ordering in combination with the node orderings for each graph merge. Note that each node merge is pairwise so it may take multiple merges to create the final summary vertex. Thus, we must enforce a tree ordering so that, even if the score for merging a node is maximal, we delay adding it until its dependent merges have been accomplished. From only this tree, we can generate any level of summarization. As detailed in Section 5.5.2, this feature is useful for interacting with the summary graph.

5.5 Visualizing and Interacting with Graph Summaries

While constructing a summary graph is the initial step toward understanding a collection of graphs, visualizing and interacting with this new summary is critical to obtaining insight. We built the GraphSum system to help users explore summary graphs. There are a variety of visual cues that we can use to represent features of a summary graph with respect to the graph collection. From node labels to edge thickness, we can both capture common information and unique features. At the same time, the ability for users to interact with and edit the summaries is also important. Beginning with layout and initial display, we have developed a variety of tools that allow users to interactively compare and edit summary graphs.

5.5.1 Layout and Display

One key concern when generating a summary graph is the layout of the result. This graph is a combination of graphs from the input collection that may or may not have existing layouts. We use the dot and neato algorithms from the graphviz library [53] to layout directed and undirected graphs, respectively. We try to preserve layouts by incorporating existing position information, although since the algorithms we use are not intended for dynamic layouts, the layouts will change. We use animation to show how these changes occur.

Recall that a node of a summary graph can represent nodes from one or more of the input graphs. Each of these nodes can have a separate label, although there is often overlap in the set of labels. We can use both the cardinality and labels of the nodes represented to render a summary node. When

displaying a node, we can use saturation or lightness to indicate the number of nodes summarized. When a summary node represents nodes with different labels, we can also display each label. Color can be used to indicate which nodes match graphs from the initial collection.

As with summary nodes, a summary edge can encode an edge from one or more input graphs. Here, the thickness or opacity/saturation of the edge can be used to indicate how many graphs in which the edge appears. In addition, edge labels can be used as text or to generate styled edges (*e.g.,* via colors) linked to a legend. Some styles might be extended to encode the contents of a summary edge.

Of course, when there exist domain-specific techniques for visualizing graphs, it is useful to utilize them by extending the original technique to capture summary information. At the same time, a summary graph could violate constraints of the technique; the summary "operation" is not closed. For example, a summary graph of a set of planar graphs may not itself be planar. Thus, any layout or rendering that depends on planarity could not be extended to work in general.

5.5.2 Controlling the Amount of Summarization

Recall that the key method for condensing graphs is merging nodes and edges when they match. Thus, in order to control the amount of summarization, we can manipulate the number of merges that take place. Because we have a measure of the similarity between nodes, we can order merges according to this measure. For a single merge step, this is straightforward. However, we would like to be able to explore and control the node merges over the entire hierarchy. Although we have scores for the merges at each step, in this situation, we have an added constraint: merging a previously merged node with another node cannot be allowed if the first merge has not occurred. For this reason, we must construct an ordering that takes both measures into account; we cannot simply order by count. From this ordering, we allow the user to control the amount of summarization via a linear control. The user selects the index of the order, and we construct the summary graph based on all of the merges. See Figure 5.4 for an example of this control.

To construct the global order, we need the forest that corresponds to all merges that we allow from our summary graph construction. Each node in the forest is a node resulting from a merge of two nodes. Then, for each node, we have a weight that corresponds to the measure of similarity from the matching. A node is a leaf if it is a merge of two nodes from the initial graphs. To compute the order, we use an iterative algorithm with a priority queue. The queue is seeded with all of the leaves of the tree and ordered by the similarity measure. At each step, the highest scoring node is removed from the queue, added to the final order, and if all siblings of the selected node have already been added to the final order, the parent node is inserted to the priority queue. For example, the summary graph shown in Figure 5.5 is summarized by the global order tree shown in Figure 5.6.

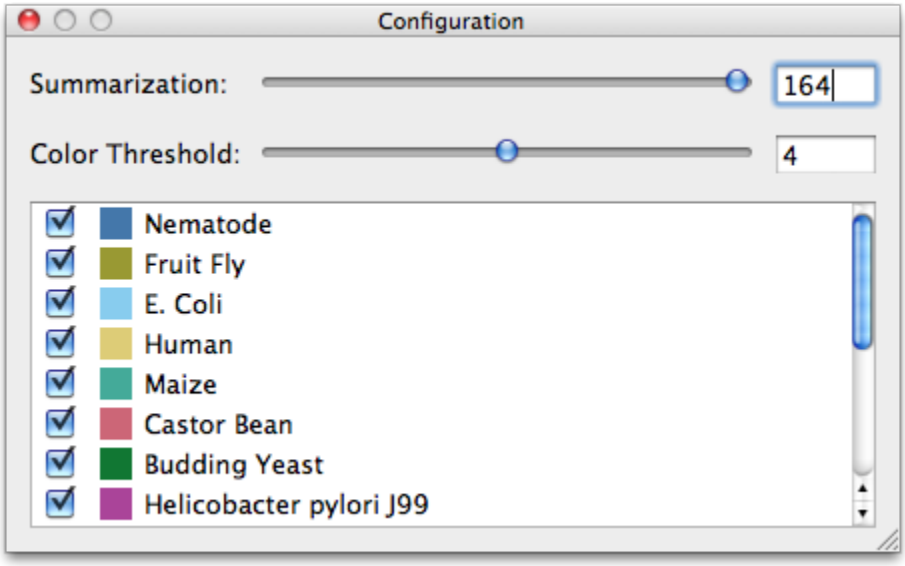


Figure 5.4: The settings for GraphSum allows users to control summarization, adjust vertex and edge coloring, and toggle the display of individual graphs.

5.5.3 Color

Color provides a significant visual cue, and we take advantage of it to compress the information from many graphs into our more compact summary graph. We can highlight both nodes and edges by a representative color for the original graph to which they belong. However, as Figure 5.7a shows, too much color can over-saturate the visualization and provide a more complex display to understand. Using a threshold to determine when color should be used, we can highlight unique differences, as shown in Figure 5.2b which only uses color when the feature is seen in the selected original graphs. Users can also focus on these graphs specifically by hiding other graphs (Figure 5.7c). The threshold and selection controls are available via the settings interface shown in Figure 5.4.

5.5.4 Manipulating the Summary Graph

Because we are using a heuristic approach to compute the summary graphs, we cannot expect that summary graphs will always conform to a user's preference. For this reason, we provide the ability to control parameters including the amount of compacting the algorithm does and the amount to which vertex degree should influence diffusion. In addition, we provide the ability to explicitly define which nodes should be joined or split. We provide three operations for users to manipulate nodes:

1. **Break.** Split each selected vertex by creating two new compound vertices that split the data from the vertex.

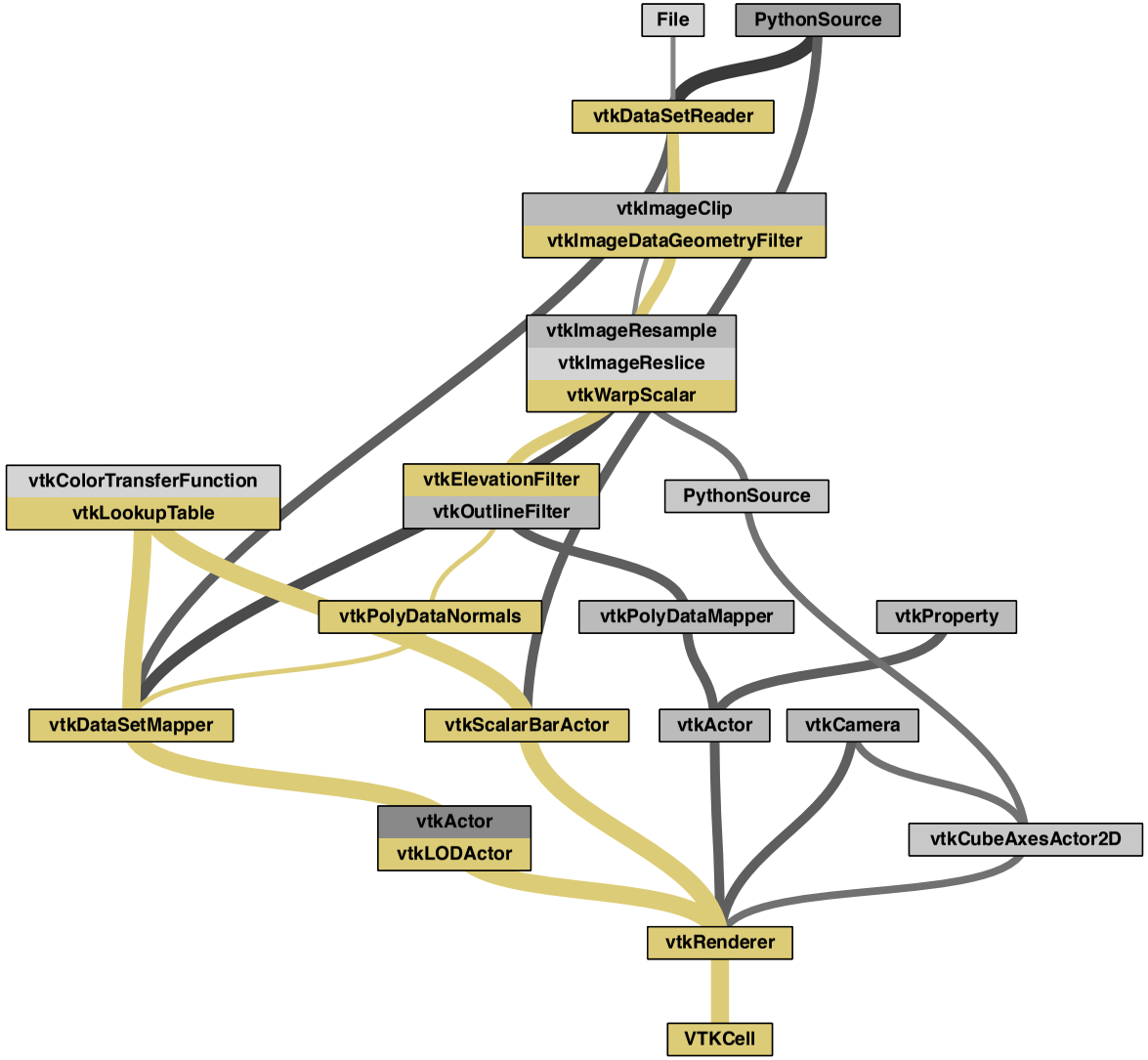
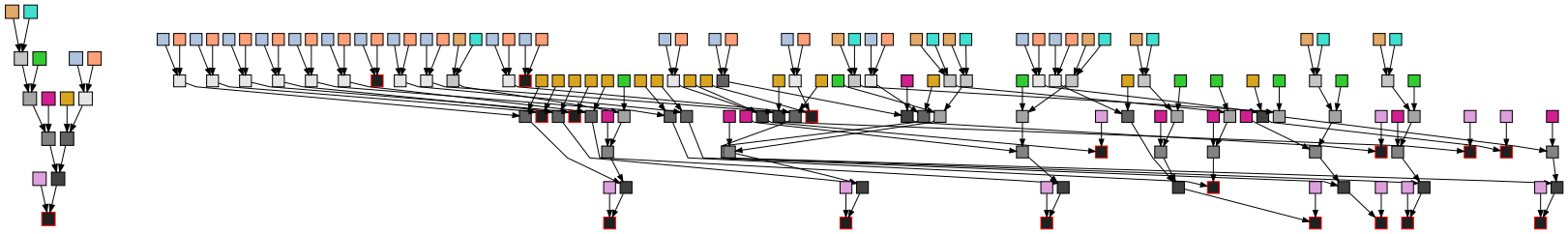
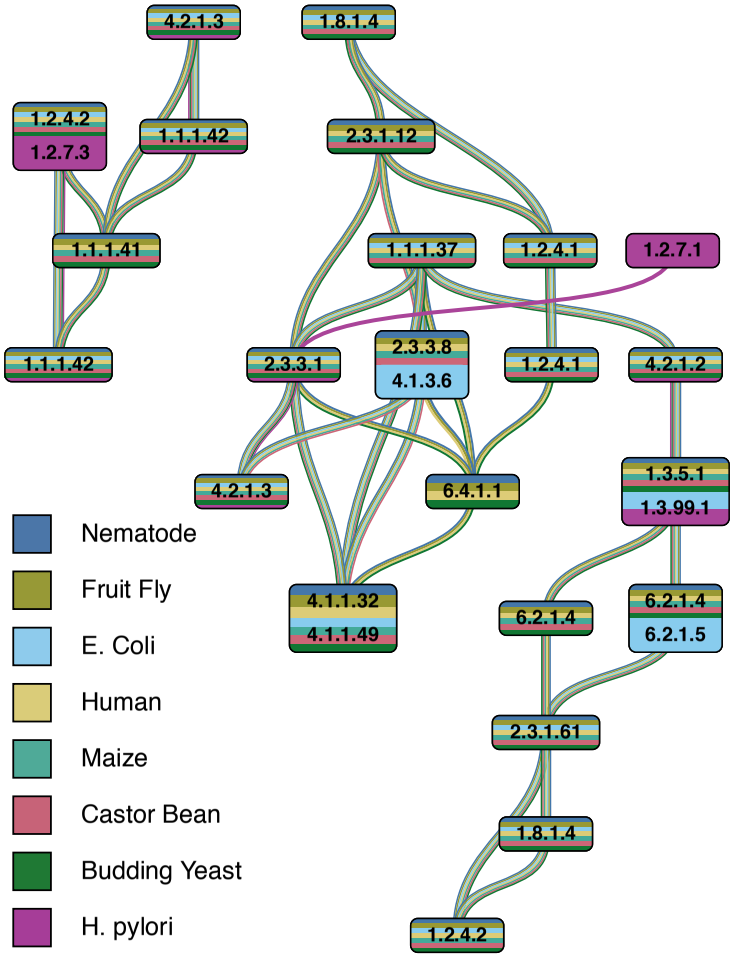


Figure 5.5: A summary of eight graphs representing visualization workflows generated by different students for a specific homework problem. A single student's work is highlighted in the context of the other graph, allows specific comparisons with the group as a whole.



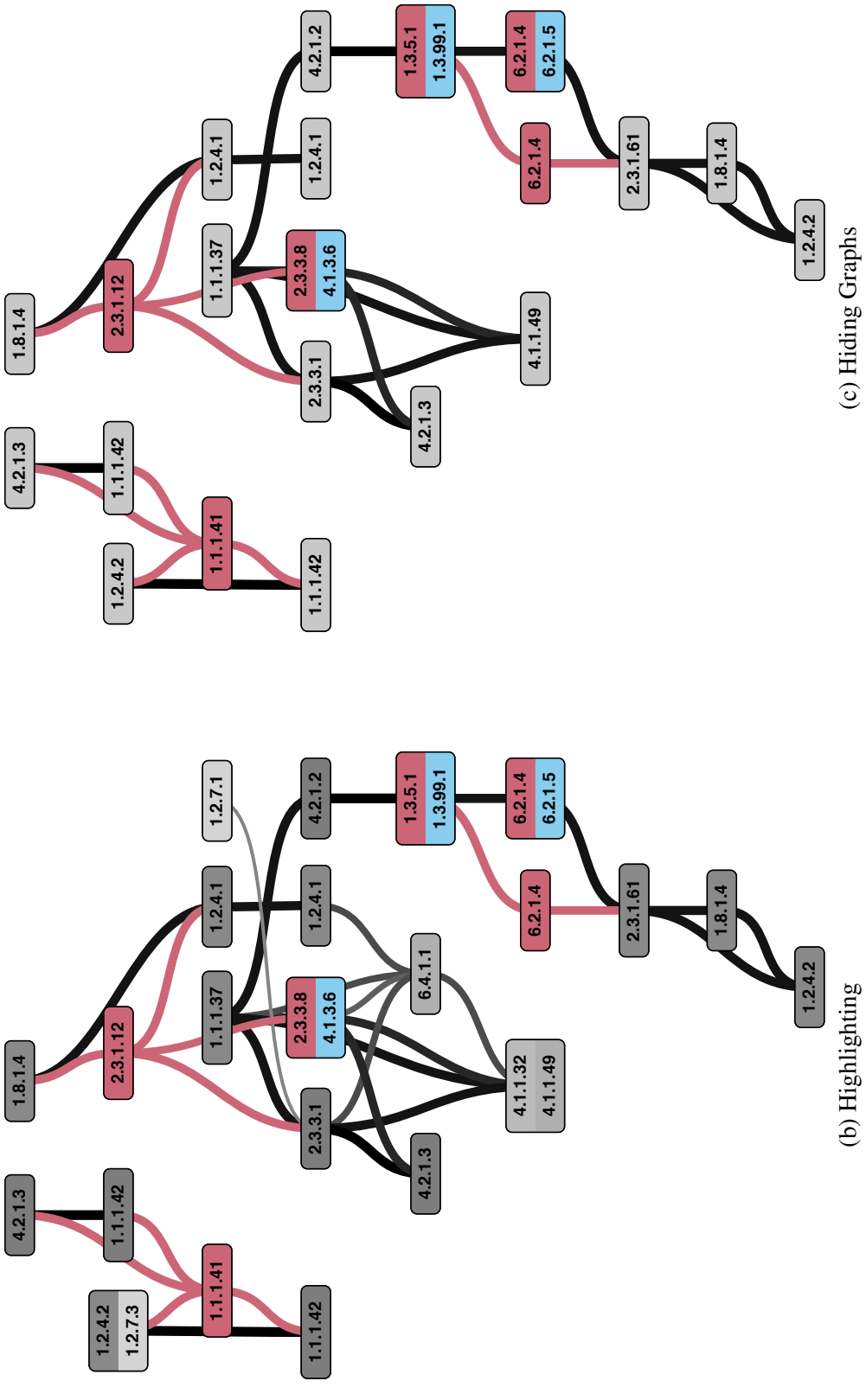
(a) Merge Order (b) Node Merge Order

Figure 5.6: We define an initial ordering to merge graphs to create the summary graph, but after each merge, we use the similarity scores for individual nodes to order the individual node merges. The figure shows this node merge ordering for the workflow summary graph shown in Figure 5.5. This ordering provides a natural method for navigating the amount of summarization in a linear fashion. Note that any dependent merges must occur before a given merge.



(a) All Colors

Figure 5.7: A summary graph of enzyme relation graphs from the citric acid cycle for eight organisms. We can show all colors to identify which entities appear in each graph (a). Our GraphSum application allows a user to highlight a subset of the graphs to better show individual differences (b). Users can also hide the other graphs to show only the similarities and differences between the selected graphs (c).



(b) Highlighting

(c) Hiding Graphs

Figure 5.7: Continued.

2. **Join.** Join the selected vertices by creating a new compound vertex that combines the data from each selected vertex.
3. **Reinforce.** Does not change the current graph but specifies that the selected vertices should remained merged in further operations.

Note that because of the diffusion, rerunning the algorithm conditioned to the nodes that were selected means that every blemish need not be corrected. Often joining or splitting single nodes can help influence the other desired results. An example is shown in Figure 5.8.

An initial approach for controlling merges can be accomplished by manipulating the order of summarization. If we remove a merge, we effectively split the node. If we choose to join two nodes, we can introduce a new join. Note that for both of these operations, we have constraints on the insert into the order from the tree; we cannot introduce a merge before the nodes to be merged have been merged. However, we do not have any measure on the quality of the merge. We currently introduce the merge into the order at the currently selected level of summarization. Another option might be to take the score of such a merge from the computed scores.

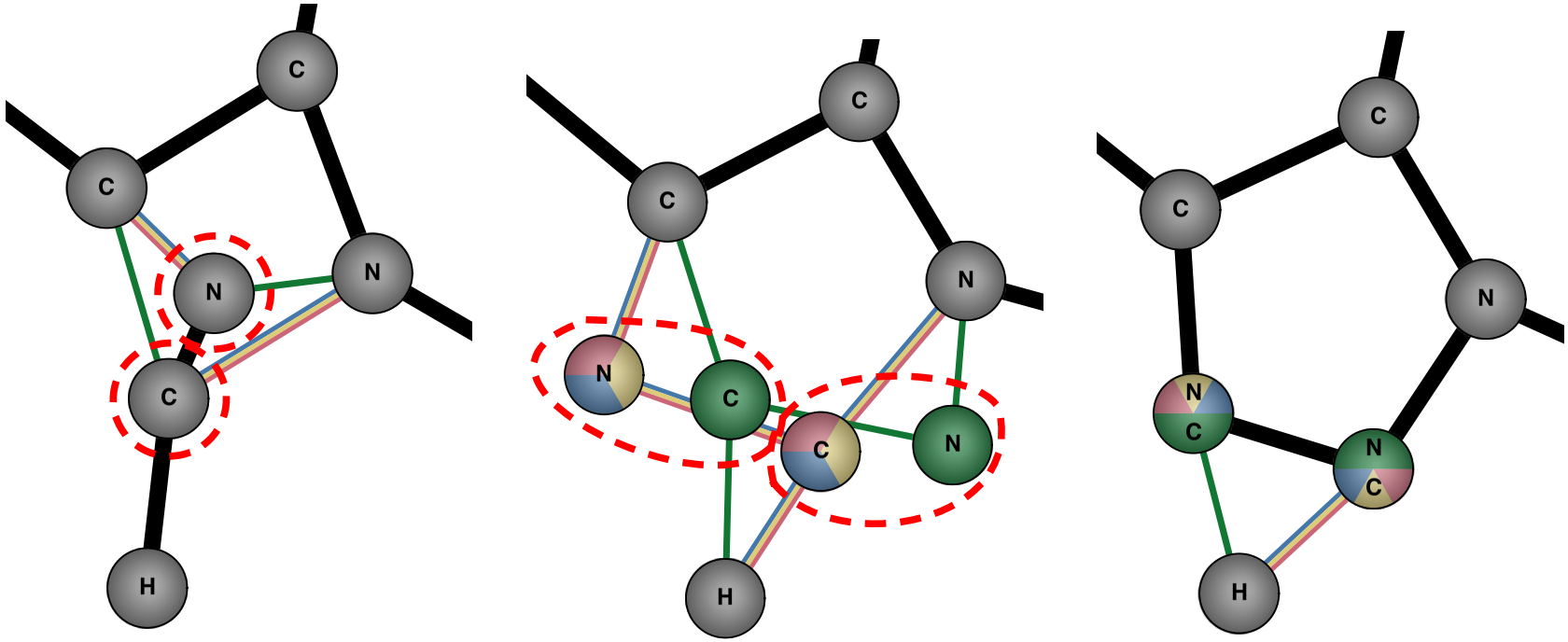
We use animation to point users to the specific changes. Specifically, when changing the amount of summarization, the animation can highlight the splits or merges. In addition, it can help provide continuity when the layout of the graphs changes. For nodes that are not split or merged, we use linear interpolation between a node's old location and its new location. When a node is split, we show the new nodes as they move from the original position to their new positions. When a node is joined, we show the old nodes as they move from their original positions to their new common position.

5.6 Case Studies

5.6.1 Metabolic Pathways

In biology, the pathways that drive life by chemical processes are important components in our understanding of cells. During each process, compounds (metabolites) are processed into other compounds via different enzymes and substrates. Even for the same process (*e.g.,* glycolysis or the citric acid cycle), the components involved can vary across organisms. Understanding these differences can help in comparing organisms, especially with respect to their evolutionary history. Each pathway can be abstracted to only the orders of different enzymes to simplify this process [67]. Enzymes can be identified by their EC numbers, four-number tuples separated by dots, which are organized hierarchically [161]. Thus, the similarity between enzymes can be determined based on how many numbers match (*e.g.,* 1.2.3.4 and 1.2.3.5 are quite similar while 4.3.2.1 and 4.5.6.7 are close to completely different).

In our examples, we used the Kyoto Encyclopedia of Genes and Genomes (KEGG) database



(a) Original Graph (b) Broken Nodes (c) Joined Nodes

Figure 5.8: A piece of a molecular summary graph that shows how edit operations can be used to transform one summary into another via two break operations, (a) to (b), followed by two join operations, (b) to (c). With editing operations, the user can decide how the summary is best presented.

which provides a vast array of information on metabolic pathways in addition to other genetic data [80]. Specifically, they store known pathway information for many different organisms. We used data from the Citric Acid Cycle for eight organisms, ranging from bacteria to humans, to generate a graph summary. After recovering the enzymes and their relationships, we looked up the EC numbers, again using KEGG, to generate labelled graphs. Figure 5.2 shows an example of a resulting summary graph. Note that complex organisms share most enzyme relationships, while simpler organisms like yeast and E. coli have many differences from the common components in the summary. This visualization has the potential to help scientists determine evolutionary relationships, including full phylogenies, via greater information than a single number that measures similarity as in [67].

5.6.2 Visualization Pipelines

For programs that have a modular structure, it is often useful to visually examine this structure. Visualization pipelines or workflows are a good example of structured programs: they are defined as graphs, where modules correspond to computational functions and edges represent how data flow through the modules. Given a collection of visualization pipelines, being able to summarize them can help users better understand existing approaches. For example, one can examine different contexts where volume rendering is used, and even contrast these against other techniques such as isosurface extraction. An important application in this context is in teaching. Given that we can summarize a set of pipelines that are produced by different students as a solution to some visualization assignment, we can derive both a "consensus" solution as well as identify the unique differences in individual solutions.

We investigated a set of visualization pipelines built with the VTK [83] modules in the VisTrails system [153]. The pipelines were built by students to accomplish a well-defined homework task, so we can expect significant overlap in their solutions. However, we would also expect that some students will use different approaches to obtain their results. In order to compare modules (our vertices), we used the VTK inheritance hierarchy and module interfaces to estimate similarity. For example, `vtkStructuredPointsReader` and `vtkDataSetReader` both inherit from `vtkDataReader` and share similar ports so they have a greater similarity than `vtkActor` and `vtkMapper`, which do not share a common base class. Figure 5.5 shows a summary graph for eight different visualization workflows. We can see that most students followed the same core structure, but the filters used varied and some added additional features. The summary graph combines similar modules like `vtkActor` and `vtkLODActor` to condense the number of nodes, but with color, we can see which student(s) used the LOD version.

5.6.3 Molecular Structures

Understanding the differences between molecules can be important in a variety of fields from pharmaceuticals to physics. Molecules are composed of atoms and the chemical bonds between them. With similarity graphs, we can understand the similarities and differences in families of molecules. One application of this is searching for substructures that may be important in combating diseases. For example, knowing that a certain class of molecules shows an ability to resist or attack viruses or bacteria can aid researchers searching for clues to produce vaccines or drugs.

We examined the NIH's AIDS Antiviral Screen dataset of chemical structures for compounds that have been checked for evidence of anti-HIV activity.[1] The dataset contains the molecular description of 42,390 such compounds, including all atoms and bonds. Note that unlike metabolic pathways and visualization pipelines, molecules are *undirected* graphs. This allowed us to test the impact of directed edges on our algorithm; note that this allows more alignments. In order to search for similar molecules, we used a feature vector containing the atom counts. From this rough clustering, we were able to test groups of similar molecules by building summary graphs. Our similarity function for nodes ranked exact matches highest, but also ranked atoms in the same group in the periodic table higher than those in other groups. Some groups produced poor results because the connectivity structure was very different despite a similarity in atom counts, which was not unexpected. Figure 5.1 shows an example of one of the summary graphs obtained.

5.7 Discussion

Like any graph visualization, a summary works best in conditions where the graphs that are involved are somewhat compact and sparse, allowing a user to see node labels and individual edges. There are methods to draw more compact representations of graphs by condensing regions of graphs, and those same techniques can be utilized for summary graphs as well. In addition, specific regions of the summary may correspond to a specific subset of the collection, and we can use this information to guide this condensation.

5.7.1 Overlaps

A second condition that can be used to evaluate the utility of the summary is the amount of overlap between the set of input graphs. Intuitively, graphs that are very dissimilar cannot be nicely summarized and much of their display will be independent. Those which contain large proportions of similar nodes are more likely to produce more informative summaries. We propose a measure of the amount of *overlap* in a graph collection in a pairwise manner. Recall that for each pair of

[1] `http://dtp.nci.nih.gov/docs/aids/aids_data.html`

graphs, we can compute the overall score between two graphs. From the entire hierarchy, we can compute an overall score that represents an overlap between graphs. The higher the overlap score, the better the summarization. This overlap score can also be used to determine whether adding a given graph to an existing summary is worthwhile.

One method for dealing with a collection where the overall overlap is low is to separate it into smaller subcollections and then construct summaries. Clustering could be used for the initial binning and the summaries created for each cluster. Note that there is little to be gained from a summary of two graphs that have a very small overlap, so it makes sense to construct separate overlaps.

5.7.2 Multi-Edge Graphs

For graphs that allow multiple edges defined between the same vertices (and in the same direction for directed graphs), we can extend our definition of summary graph to allow multiple edges between nodes. Note that most generally, we can allow arbitrarily many edges between pairs of vertices of the summary graphs as we do not place any restrictions on how vertices must be summarized. To accomplish this, we can solve an assignment problem for matching edges between graphs; when edges correspond, we can merge them; when they do not, we leave them separate. In some cases, to reduce clutter, it may be reasonable to force edges to always match, meaning the number of edges between two vertices in the summary graph will not exceed the largest cardinality of edges between individual vertices in the original graphs.

5.7.3 Scoring

The scoring of nodes (and edges) plays an important role in the quality of the summary graphs. If the scoring functions are poor, we cannot expect that the summary graphs will be well-collected. The default score is one of equality of the data contained in the nodes. However, there are many cases where this scoring can be improved to take into account known similarities. For example, it is known that atoms in the same group (family) share similar properties and should be considered more similar than two atoms from different groups. Such information can be encoded into a node similarity function for atoms.

5.7.4 How Much Summarization?

Note that the maximal summary graph need not reflect the best visual summary. Consider a dangling node that attaches to different nodes on each graph. While we can merge that node to create a more succinct display, it may be more intuitive for a user to see these nodes separately.

CHAPTER 6

SUPPORTING REPRODUCIBLE AND REUSABLE PUBLICATIONS

6.1 Bridging Workflow and Data Provenance Using Strong Links

As the volume of data generated by scientific experiments and analyses grows, it has become increasingly important to capture the connection between the derived data and the processes as well as parameters used to derive the data. Not surprisingly, the ability to capture the provenance of data products has been a major drive for a wide adoption of scientific workflow systems [38, 39, 47]. By tracking workflow execution, it is possible to determine how an output is derived, be it a data file, an image, or an interactive visualization.

However, the common practice of connecting workflows and data products through file names has important limitations. Consider, for example, a workflow that runs a simulation and outputs a file with a visualization of the simulation results. If the workflow outputs an image file to the filesystem, any future run will overwrite that image file. If different parameters are used, or the simulation code is improved and the updated workflow is run, the original image is lost. If that image file were managed with a version control system, the user could retrieve the old version from the repository. However, if the user reverts the output image to the original version, how does she know how it was created? Since there is no explicit link between the workflow instance (*i.e.,* the workflow specification, parameters, and input files) and the different versions of its output, determining their provenance is challenging. If we examine the provenance logs for the workflow runs, we will see that there are two runs that create the specified image file, one with the older simulation routine and the second with the newer one. We may be able to check timestamps in order to guess, but this is far from ideal. This problem is compounded when computations take place in multiple systems, and recording the complete provenance requires tying together multiple workflows through their outputs and inputs. As files are overwritten, renamed, or moved, provenance information may be lost or become invalid. As a result, maintaining an accurate provenance graph which ties processes and the data they manipulate requires a time-consuming and error-prone process.

While version control systems effectively track changes to files, such systems can only determine that changes have occurred, not how they came about. Provenance-enabled workflow systems,

on the other hand, are able to capture how changes came about but do not provide a systematic mechanism for maintaining data provenance in a persistent fashion, *i.e.,* given a file it may not be possible to determine which workflow instance generated it. We posit that a *tighter integration between scientific workflows and file management is necessary to enable the systematic maintenance of data provenance*.

In this section, we propose a new framework which, by coupling workflow provenance with the versioning of data produced and consumed by workflows, captures the actual changes to data as well as detailed information about how those changes came about. A persistent store for the data ensures that old inputs and results can be retrieved, and we can tie each version of a result to the provenance that details how the result was generated. We introduce the notion of a *strong link* which reliably captures the connection between a workflow instance and data it derives, and describe an algorithm for generating these links. Instead of relying on the user or ad-hoc approaches to automatically derive file names, strong links are identifiers derived from the file content, the workflow specification, and any parameters. As a result, they accurately and reliably tie a given workflow instance and its input and derived data.

Besides simplifying the process of maintaining data provenance, this approach has several benefits. By automatically capturing versions of data, it seamlessly supports exploratory tasks without requiring users to curate the data (*e.g.,* managing the file names). It also provides a general mechanism for the persistent caching of both intermediate and final results—this is in contrast to previous approaches which supported only in-memory caching [8, 17]. The caching mechanism can be used not only to speed up workflow execution, but also to support check-pointing for long-running computations. In addition, the use of a managed data repository allows the creation of workflows that are location agnostic: unlike workflows that point to files in the filesystem, workflows can be shared and run in multiple environments unchanged. Last but not least, our approach is general and can be combined with existing workflow systems. We describe our implementation in the VisTrails system and present a case-study, where the persistent data provenance infrastructure was deployed in a real application: managing data products in the context of the ALPS project [5].[1]

We begin by introducing our persistence scheme in Section 6.1.1, and then show how it can be applied to support data provenance in Section 6.1.2. In Section 6.1.3, we describe how our approach can be used to extend workflow caching strategies and for publishing scientific results. In Section 6.1.4, we describe how managed repositories can be shared among multiple users for both data access and caching. We describe an implementation of our scheme in Section 6.1.5, and describe its use in the ALPS project in Section 6.1.6. We highlight related work in Section 6.1.7

[1] `http://alps.comp-phys.org`

before concluding with future directions in Section 6.1.8.

6.1.1 Persisting Data Provenance Links

By integrating file management and version control with workflows, we aim to maintain stronger provenance by referencing data in a versioned, managed repository instead of via file paths (see Figure 6.1). This repository stores input, output, and intermediate data products, and can be used to facilitate caching and data sharing.[2] Similar to version control systems, this repository stores multiple versions of files, but to connect to workflow provenance information, it also contains metadata that represent identity and annotations.

Our approach to this problem is user-driven. As a user designs a workflow, she can specify which results (or input data) should be persisted in the repository. As we describe in Section 6.1.5, a possible implementation is to provide special workflow modules that can be connected to the output ports of modules whose results should be persisted. When users run workflows using data from the repository, we can ensure that future provenance queries can not only identify the data involved in the computations but also retrieve the actual data. In addition, given provenance of a workflow execution, we can reproduce it using the exact versions of the data used in the original execution. In these provenance applications, there is no need to archive data according to specific path-name conventions or remember to keep each separate version of the input data. Also, the automatic and transparent identification and versioning require little user involvement in maintaining these stronger links.

In what follows, we start by describing a scheme to derive reliable and representative ids for linking data products and their provenance. We also present the file-management infrastructure and the attributes we maintain in the managed repository, the differences in our storage depending on the role of the data, and how data should be updated and stored. Note that while we discuss *file* management, the techniques described can be easily extended to directories as well.

6.1.1.1 Deriving Strong Links

Our approach to deriving strong links was inspired by the in-memory caching mechanism proposed by Bavoil *et al.* [17] and the content hashing used in version control systems including `git` [56]. We use the signatures of workflows to identify intermediate and output data derived by the workflows, and content hashing to identify input data.

The central idea of caching in workflow systems is that any self-contained piece of a computation can be reused if the computation is deterministic and its structure, input data, and parameters do

[2] In the remainder of the text, we use the terms “repository” and “managed store” interchangeably.

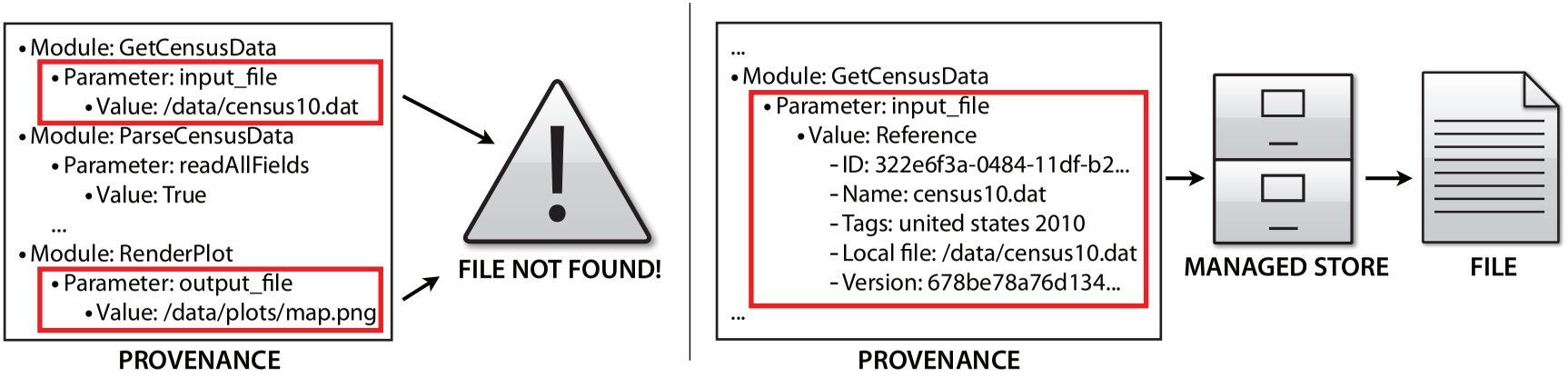


Figure 6.1: When provenance information references file-system paths, there is no guarantee those files will not be moved or modified. We propose references that are linked to a persistent repository which maintains that data and with hashing and versioning allows for querying, reuse, and data lineage.

not change. For dataflows, we can formalize this concept by defining the *upstream subworkflow* of a module m in a workflow W as the subgraph induced by all modules $u \in W$ for which there exists a path from u to m in W (including m itself). Note that the existence of such a path implies that the results of u may have an effect on the computation of m. Then, if any module or connection in the upstream workflow of m changes, we must recompute m. Conversely, if the upstream workflow does not change, we need not recompute m, and can reuse results from a previous execution. Thus, for any other workflow W' that contains an upstream subworkflow U that also exists in W, we can reuse the intermediate results of U from W in W'.

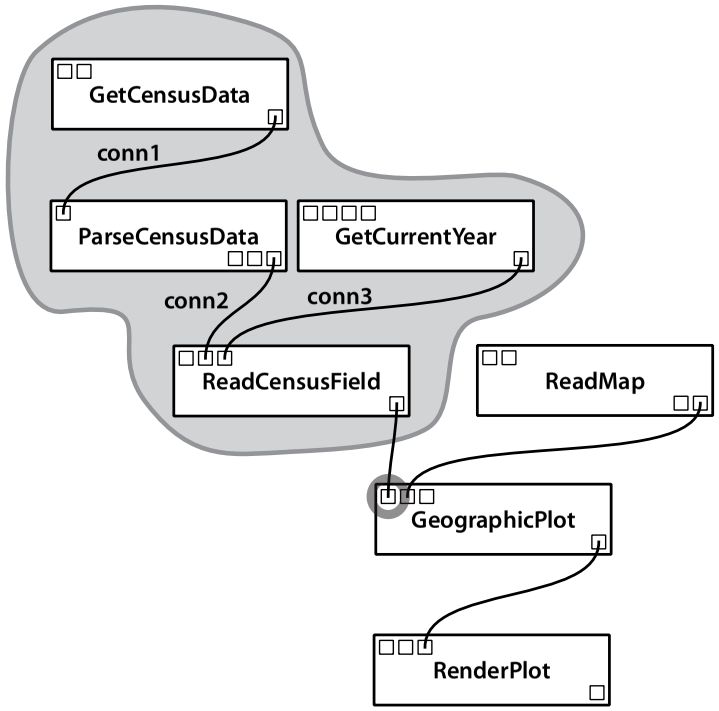
Caching thus requires the identification of equivalent subworkflows. This would be expensive if we needed to perform graph matching, but we instead use a recursive serialization of the upstream workflow that allows us to quickly check the cache. We define the default *label* of a module m, $\ell(m)$, as the serialization of its type and parameter values ordered by parameter name. Note that individual module types can override this default label to better capture module state; for example, a module linked to a specific file would define its label based on the contents of the file—if that file changes, the label changes. Similarly, the *label* of a connection c, $\ell(c)$, is the serialization of the types of the ports it connects. Then, a canonical serialization of the upstream subworkflow of a module m is defined recursively as

$$S(m) = \ell(m) + \bigplus_{c \in \mathrm{UC}(m)} S(\mathit{source}(c)) + \ell(c)$$

where $\mathrm{UC}(m)$ is the set of upstream connections into m sorted by $\ell(c)$, *source* returns the source (upstream) module of the given connection, and $+$ is concatenation. The *upstream signature* is the SHA1 hash of this serialization.

Figure 6.2 shows an example workflow and the serialization of the upstream subworkflow of the `ReadCensusField` module. Note that the upstream subworkflow will not always be a tree, but the recursive serialization always branches like it is. This allows two topologically different upstream subworkflows to have the same signature, but when this happens, the computations must be identical. For example, consider a subworkflow with a single module m that connects upstream to two other modules. Whether those two modules connect upstream to a single module n or to two identical modules that both do the same computation as n will not affect the downstream computation of m. In addition, by using memoization, we can keep this computation efficient despite the added branching.

For input files, we define the signature as the hash of its contents. Then, if the file's contents change, its signature changes even though its path or other identifying information may not. Note that we store the content hash separately as well so a file that is the output of one workflow and



Module	Serialization
S(GetCensusData)	ℓ(GetCensusData)
S(GetCurrentYear)	ℓ(GetCurrentYear)
S(ParseCensusData)	ℓ(ParseCensusData) + S(GetCensusData) + ℓ(conn1)
S(ReadCensusField)	ℓ(ReadCensusField) + S(ParseCensusData) + ℓ(conn2) + S(GetCurrentYear) + ℓ(conn3)

Figure 6.2: The *upstream signature* S(M) for a module is calculated recursively as the signature of the module concatenated with the upstream signatures of the upstream subworkflow for each port and the signature of the connection.

the input of another can be identified in both ways. Thus, the signature provides a strong link that contains a precise and accurate representation of the workflow fragment that derived a given result. As we describe in Section 6.1.2, we use this signature as the means to link a data product to the computation that derived it.

6.1.1.2 File Management

We are concerned with three roles for files in workflows: inputs, outputs, and intermediate data. Note that a single file may fill different roles depending on the workflow in which it is used; an output from one workflow may be used as the input to another. Thus, the distinction between roles does not affect the use of data in any situation, but rather determines what metadata can be captured, stored, and utilized. An output file can store information about the process that created its contents but an input file selected from the filesystem cannot. Similarly, an intermediate file need not be annotated at all if it is used for caching, but files that are to be used again should be named and tagged to allow users to query for them.

Each file in the repository is uniquely identified by a combination of an id and a version string, and annotated with user-defined and workflow-generated information including its signature and content hash. By allowing a collection of files to share the same id, a reference to that id can be configured to always retrieve the latest version. This is helpful to a user who wishes to run a workflow with the latest version of a data set but does not wish to manually configure which is the latest version. On the other hand, reproducing a workflow execution exactly requires a particular version of the data, and thus identifying data by both the id and version guarantees that the exact data will be retrieved.

6.1.1.2.1 Input files. An input file must reference an existing file, whether it is already in the managed store or only in the local system. Upon selection, we either use the existing identifier (from the store), or create a new unique identifier for the data. Note that we can detect whether the contents of a file already exist in the repository by computing the hash and checking for that hash in the repository. By default, changing the contents of the file creates a new version while changing the selected file creates a new id and version. Users can configure this behavior if necessary.

6.1.1.2.2 Output files. The main difference between output and input files is that input files are not affected by changes in the rest of the workflow. For outputs, any changes to the part of the workflow that is upstream of the output file may affect its contents. In addition, it is less clear when an output is a new entity and when it is a new version of an existing entity. When only parameters change, the output is likely a tweaked version of the original, but when the input files are switched, the output is more likely new. By default, we create new versions for each execution but allow users to change this behavior in order to version the outputs. Like inputs, output files can be both stored

in the persistent store for future reference and use and saved to a local file for immediate use or inspection.

6.1.1.2.3 Intermediate files. An intermediate file is exactly the same as an output file except that it is not a sink of the workflow; execution continues after the file is serialized and the contents are used in further calculations. Such files can be used as checkpoints for debugging, but they can also be used to cache computational results in order to speed further analyses. Note that an intermediate file need not be manually annotated or named; it is defined by its signature–the serialization of the upstream subworkflow.

6.1.1.2.4 Customization. It may be necessary for users to configure the behavior of the persistence of files in the store in order to link similar files or maintain separate identities for data products. By selecting an existing reference and linking it to a local file, a user can tie the reference to a new local file. In addition, users can decide whether files are only persisted in the managed store or if they are also saved to local files. If they use a local file, they can configure whether the contents of the file should take precedence or whether a new version should always be obtained from the repository. Similarly, if the local file contents change, a user can choose whether those changes should always be persisted to the managed store.

6.1.2 Linking Provenance

Below we discuss how we exploit the strong provenance links to answer important queries. We also suggest how stronger links from data to provenance can be accomplished. With the advent of extensible file formats (*e.g.,* HDF5[3]), it is possible to include direct links to provenance or even the provenance itself with the data. Finally, we present an application of the improved results from provenance queries in publishing scientific results.

6.1.2.1 Algorithms for Querying Linked Provenance

Perhaps the most basic provenance query is one that retrieves the lineage of a data product, specifically what input data and computations contributed to the result [121]. With only the provenance of the execution, a user may find the path to an input but even if a file still exists at that location, there is no guarantee it has not been modified. To protect against such problems, users store the exact data used with the provenance, manually archive all of the data, or add archival as part of the workflow process [106]. With our file management scheme, we can store the id, version, and content hash of any input as part of the provenance. Then, for lineage queries, we can return references that can be accessed from the provenance store using the id and version and verified using

[3] `http://www.hdfgroup.org/HDF5`

the content hash. Most workflow systems that support provenance capture also provide support for determining lineage queries [121].

Note that the content hash also gives us a way to locate the provenance of files that are unmanaged and may have been moved to a different location or had their names changed. We begin by hashing the contents of the file, then query the managed store for this content hash. The resulting entries have ids and versions for which we can then search our provenance. Because the provenance contains these stronger references, we can also identify and return the input data via the managed store. An outline of this algorithm is shown in Figure 6.3.

Because we abstract workflows from a specific filesystem, the provenance of the workflow executions can be tied directly to the exact inputs and outputs. This ensures better reproducibility because the exact content can be retrieved; with links to the file names, we have no guarantee that the files at those locations were unchanged. To reproduce a workflow execution, we retrieve the workflow specification and execute it using the data pointed to by the managed file references. Recall, however, that some workflow specifications may include only data identifiers and not the versions of the data used. This allows a user to rerun a workflow with the latest data which is not what we desire for reproduction. Thus, we need to examine the provenance for the execution, retrieve the exact version specified by the provenance and modify the specification.

Another provenance query that our strong links solves is the lineage of data when the input of one workflow is the output of another. In the Second Provenance Challenge [122], teams were asked to answer provenance queries from outputs that were the result of running data through three consecutive workflows. One issue was the identification of data as they were transferred from the

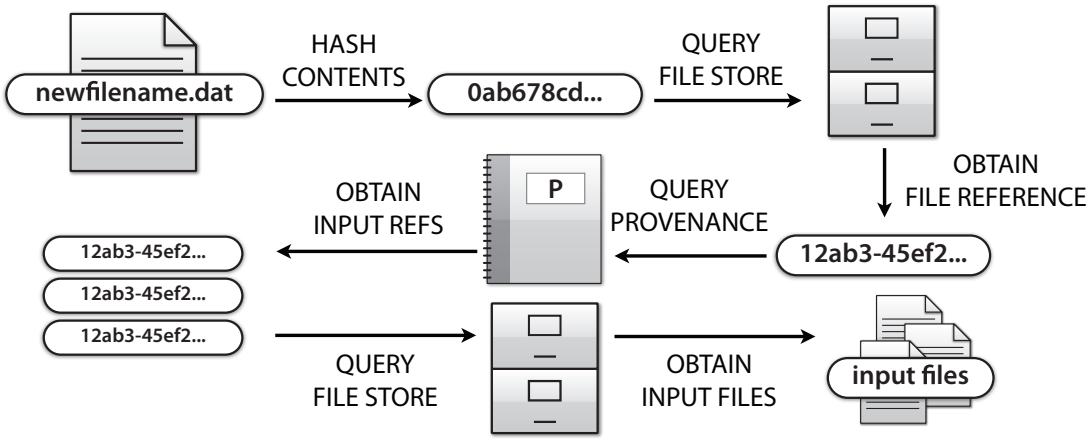


Figure 6.3: Given a file which has been moved and renamed, we can use the managed file store and provenance to first locate the managed copy, and we can locate the original input files as well.

output of one workflow to the input of another. With the managed store, we allow users to designate inputs as the output of another workflow by assigning them the same id. Thus, when the first workflow changes, the second workflow will incorporate the changed results. Even if users do not use the same identifiers, we can perform provenance queries using the content hashes to link data across different workflows.

6.1.2.2 Embedding Provenance with Data

We have demonstrated methods to find the provenance of data by searching a provenance store for the hash of a given file. However, such methods depend on access to the provenance store. An alternative approach is to embed provenance with the data itself. With many file formats including HDF5 supporting annotations, it is possible to embed provenance information or links to provenance with the data. In directories of data, we can add an additional file that contains the same information. Then, verifying data or regenerating a data product can be accomplished by examining the provenance stored with the data.

We have developed a schema that allows a user to either link to or directly encode provenance information in a file. Information represented in this schema can be serialized to XML and embedded in an existing file or saved to a separate file. Figure 6.4 shows an example of a workflow using this schema. While a provenance link can refer to a local file, we provide support for accessing a central repository of provenance information. With a central repository, if the file is transferred to a different user or machine, the link remains valid. With a local reference, it will be more difficult to link back to provenance information.

6.1.3 Using Strong Links

6.1.3.1 Caching

Caching the intermediate results of workflow computations is useful to avoid redundant computations. If a user executes a workflow, we can reuse any intermediate results from that first execution in future executions [17]. Using our file management for intermediate files, we are able to add support for caching *files* to existing in-memory caching which means that cached data can be persisted across sessions. With this extension, we can also consider how to share cached data between different users as well. We begin by reviewing the in-memory workflow caching algorithm and then introduce an extension for caching across sessions using the managed file store.

6.1.3.1.1 In-memory caching. Using the upstream signatures, we build a cache by labeling each intermediate result with its upstream signature. Dataflow computation proceeds in a bottom-up fashion; a sink (a module with no outgoing connections) requests data from all of its inputs which may in turn request data from their inputs and so on. Our caching algorithm works by hijacking this

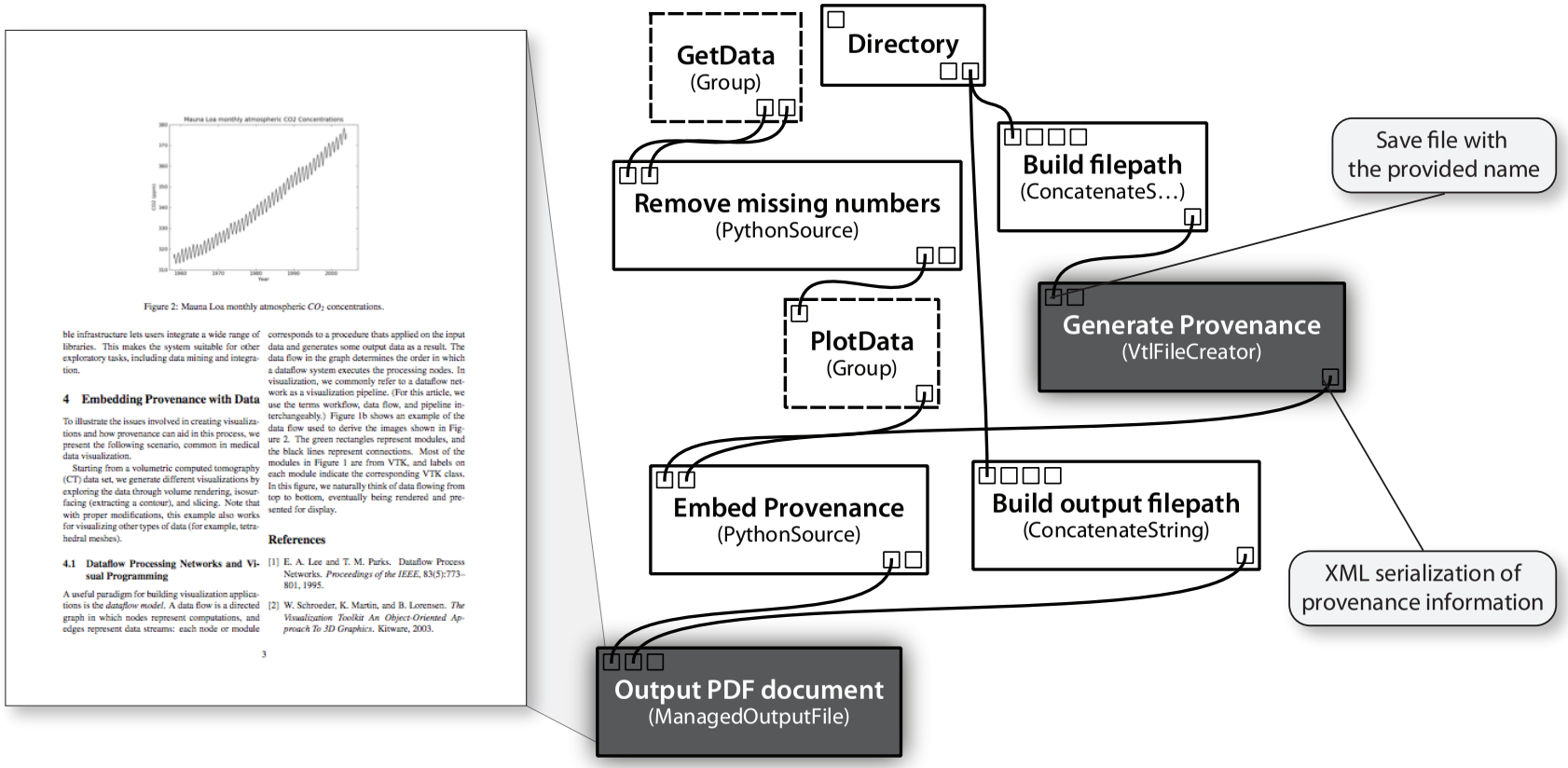


Figure 6.4: Embedding provenance with data: provenance can be either saved to a separate file or serialized to XML and embedded in an existing file.

request for data and checking if the upstream subworkflow has already been calculated, returning the result from the cache when it exists instead of doing the computations.

Before executing any workflow, we compute the upstream signatures for each module in the workflow. Note that the recursive computation of all signatures is easily memoized. During workflow execution, before a module is set to execute, we check if that module's upstream signature exists in the cache. If it does, we return the result from the cache. If not, we proceed with the computation, and after the module finishes executing, we store the results of the computation in the cache labeled by the upstream signature.

There are some modules that may not perform deterministic calculations. We allow the module developer to designate such modules as noncacheable. After a workflow with one or more such modules executes, we immediately remove all modules downstream of such a module from the cache.

6.1.3.1.2 Persistent caching. We can extend the in-memory caching techniques to persist results to disk, allowing users to cache results across sessions or share intermediate results. Note that we need a serialization of the results of any module type in order to mirror the entire in-memory cache. In addition, saving every intermediate result to disk can needlessly slow down computation. For these reasons, we have developed persistent caching as a user-driven technique for intermediate files. We allow the user to connect a new module to the workflow that designates that the upstream subworkflow of the module should be cached. For non-cached computation, this module receives a file and passes it downstream. However, when the module finds that the signature associated with a needed file exists in the cache, it retrieves the linked file without doing the upstream calculations.

This allows any module with serializable results to be persisted in a disk-based cache, but we can improve this process using the file management scheme described in Section 6.1.1.2. Using this scheme, additions to the cache are managed as intermediate files and cache lookup is a simple query to the store. In addition, users need not identify or in any way configure the intermediate files used for caching; the store assigns identity and stores signature information automatically. When the upstream workflow of the caching module changes, the cache lookup fails, and the store adds a new version of the intermediate file. Thus, a user does not lose any intermediate results when exploring different workflow configurations.

6.1.3.2 Publishing

When publishing scientific results, it is important to describe the lineage of a result. Providing data sets and computer code allows scientists to verify and reproduce published results and to conduct alternate analyses. In the past years, interest in this subject has increased in different communities which led to different approaches for publishing scientific results (see [45] for an

overview). Our schema for embedding provenance with data can be combined with these approaches. In particular, it simplifies the process of packaging workflows and results for publication. In addition, we have also implemented a solution that allows users to create documents whose digital artifacts (*e.g.,* figures) include a deep caption: detailed provenance information which contains the specification of the computational process (or workflow) and associated parameters used to produce the artifact [126].

6.1.4 Sharing Data

We have shown that maintaining workflow data in a managed store allows us to quickly locate existing data, store accurate provenance, and cache intermediate results across sessions. Additional benefits can be gained from having multiple users share the repository. For example, if one user has run a time-intensive step of a calculation, making that result available to other users allows them to proceed with later steps without each recomputing the same result. Similarly, if one user has already added a specific file to the store, other users with access to that store can access the data without locating and copying the same data. Below, we describe both centralized and decentralized approaches for sharing managed data across systems, and note that the advantages and disadvantages mirror those encountered with version control systems.

6.1.4.1 Centralized Storage

With a central store, users may either read and write directly to a common repository or transfer data between a local repository and a central repository. If users have access to a common disk, it may be possible to simply store all managed files and metadata in a single store on that disk. Then, all users will access the same repository and automatically have access to each other's input, output, and intermediate files. However, this solution may become impractical for large numbers of users. A second problem is that whenever users do not have access to that disk, they are unable to access their managed data.

When a central store is added to individual local repositories, a user will always have access to the local repository but can also retrieve from and add to a central repository. This allows a set of geographically distant users to share common data. In addition, it allows users to maintain and access local data even when disconnected from the central store. However, we maintain an extra copy of the data in this case, and there may be overhead in transferring files, especially if the distance from the central store is far. In addition, it requires building and maintaining infrastructure.

6.1.4.2 Decentralized Storage

In a decentralized approach, users would advertise their data and allow other users to transfer data directly from their repository. A search for a particular piece of data by, for example, name or signature, would query individual systems instead of one store. If the desired file is found, it is transferred directly from the source location to the requesting user. Thus, unlike with the central store, data are only transferred when they are needed. Combined with P2P approaches, the transfer may be distributed over several machines. However, if a particular machine is offline, the data generated on that machine may not be available.

A hybrid approach that supports a central table of files but decentralized storage would allow users to locate files even if they were not currently accessible. Users would not push data to or pull files from the repository but rather register the available files as they are added and whenever those data are requested, directly transfer it to the requesting machine.

6.1.5 Implementation

We added file management to the VisTrails system [153] by introducing a new package that included module types for input, output, and intermediate files and directories. The package also includes code to set up the managed store as well as navigate and update it through configuration dialogs. Our goal was to add this support in a way that changes little in workflow structure while providing ways for users to directly locate and identify data during workflow construction. Thus, users that normally only configure the path of an input file can do exactly the same for a managed input file module. In addition, adding an output file has fewer requirements; a user only needs to connect the data to be persisted to a managed output file module. The system generates unique ids and signatures automatically. At the same time, we provide methods for annotating data and configuring their storage and use.

The interface of our prototype implementation is shown in Figure 6.5. We define three new module types for files: `ManagedInputFile`, `ManagedOutputFile`, and `ManagedIntermediateFile` and their equivalents for directories. As described in Section 6.1.1.2, all share a common set of attributes and options. The key difference between inputs and outputs (or intermediates) is that outputs have a workflow-dependent signature. Thus, an input file needs to be manually identified by the user while an output file can be totally identified by its upstream signature.

A user can select a file by either referencing an existing identifier or by creating a new reference. When referencing a file that already exists in the managed store, the user can search the repository for metadata including name, tags, user id, date added, or a specific id or version. When creating a new reference, the user may provide a name and tagging information, and for input files, the local file that contains those data.

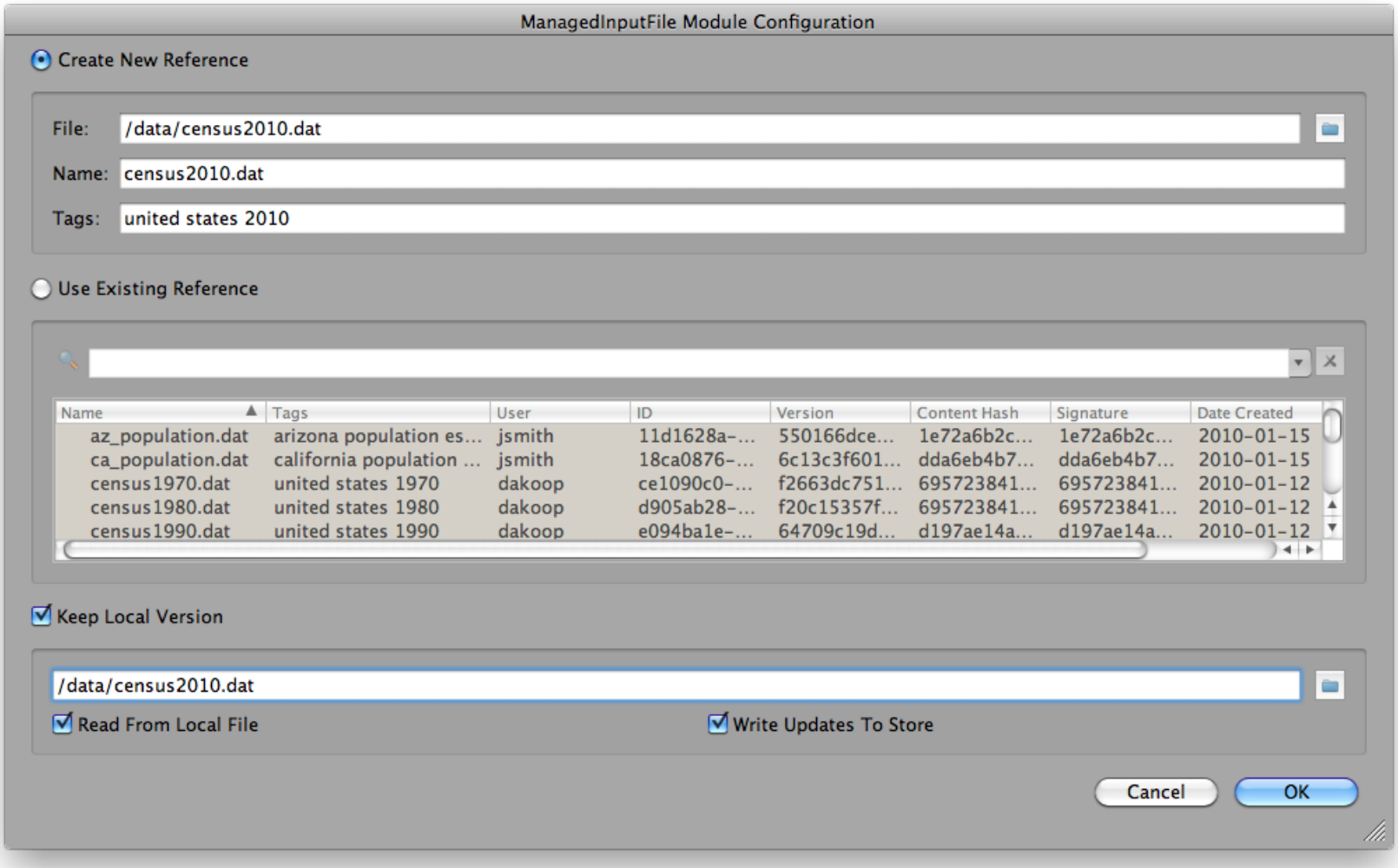


Figure 6.5: The ManagedInputFile configuration allows the user to choose to create a new reference from a file on his local filesystem or use an existing reference from the managed store.

By default, an identifier for an input file changes when a new local path is selected but does not change if the contents of the file change. In the second case, we maintain versions of the data, but update the "current version" whenever the contents changes. Thus, any user that wishes to use those data in another workflow will always get the latest data by referencing that identifier. Note that users may choose to link data to an existing reference even if that reference was initially linked to different data.

6.1.5.1 Storing Data

We use the `git` version control system [56] to manage files because it stores content independent of filesystem structure, and an `SQLite` database[4] to store its metadata. Thus, when the managed store is initialized for the first time, we create a `git` repository along with a database to store file information. While a reference is created and annotated during workflow design, the data are not persisted until execution. Upon execution, we save the file in the repository with its id (a UUID) as its name. We use `git` to add and commit the version of the file, and retrieve the content hash (`git` uses SHA1) and the version id (a SHA1 hash of the commit). Then, we update the database with the id, version, content hash, signature (if applicable), name, tags, user, and modification date.

6.1.5.2 Finding Data

In order to locate existing data, we provide methods to match content hashes and signatures as well as query the store for specific metadata like name or tag information. When a user selects a file, we can check the repository to see if that content has already been added by querying the database for the selected file's hash. If it does exist, we can prompt the user to reuse the existing reference. Additionally, when we execute a workflow, we can check to see if an intermediate file's signature matches one that already exists; if so, we can reuse that file instead of computing the upstream workflow. Finally, the configuration for managed file selection includes a free-text query field for the managed file database. A user can query for a specific name or tag to locate matching files that can be used as references. This is accomplished by querying the `SQLite` database and retrieving the matching id and, optionally, version.

---

[4] `http://www.sqlite.org`

6.1.6 ALPS Case Study

We have used the file management solution implementation for VisTrails with the ALPS project[5] (Algorithms and Libraries for Physics Simulations) [5]. ALPS is an open source software package that makes modern, high-performance algorithms for the simulation of quantum systems available to experimental and theoretical condensed-matter physicists. Typically, a simulation with ALPS consists of three steps:

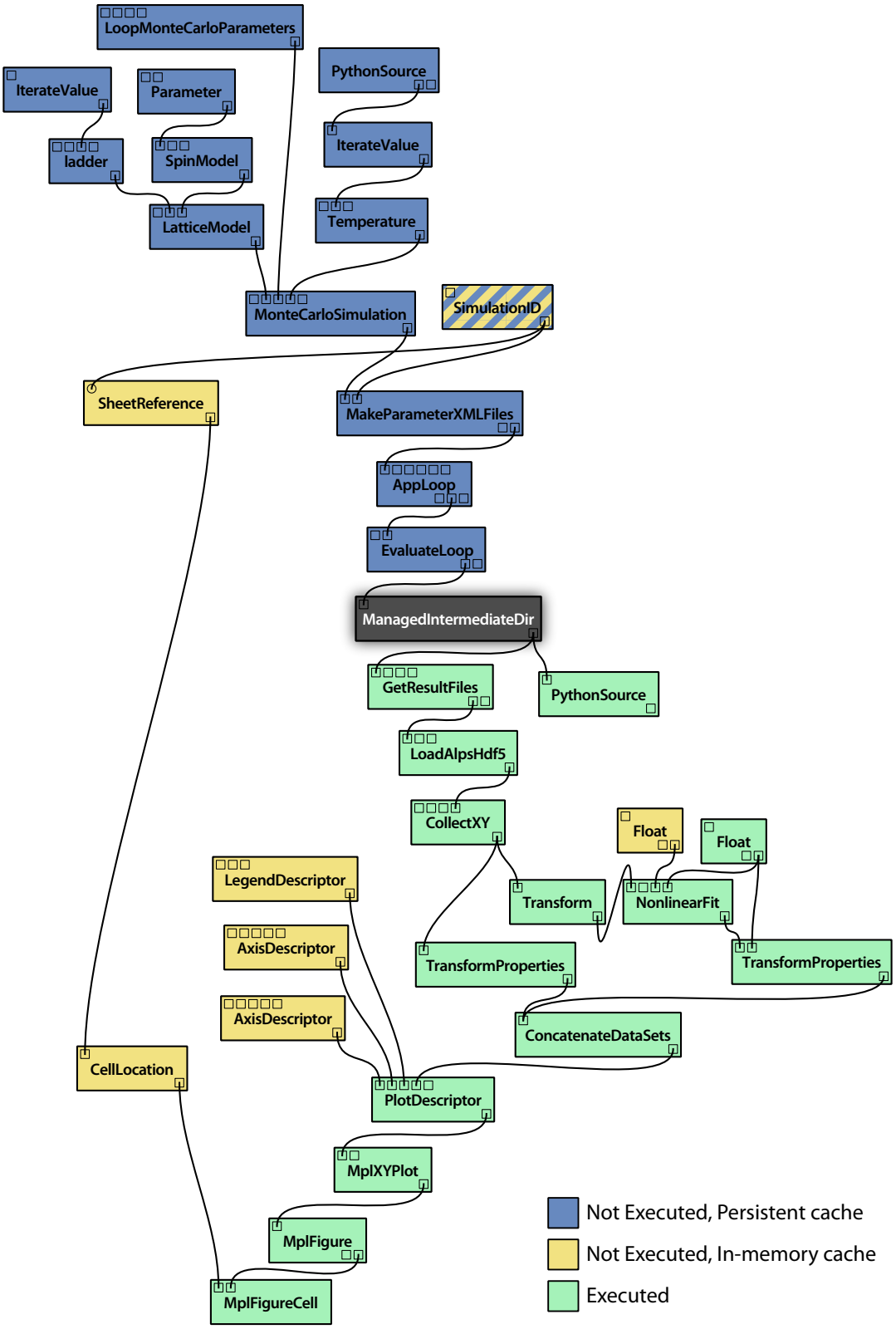
- Preparing the input files describing the model to be simulated.
- Simulating the model using one of the ALPS programs. Such a simulation can take between minutes on a laptop for very small test cases and weeks on large compute clusters or supercomputers for demanding applications.
- Collaboratively evaluating the "raw" simulation output by exploring and analyzing the data, comparing it to experimental data, and creating figures.

In one specific use case, we have simulated a quantum Heisenberg spin ladder, a model for quasi-one-dimensional copper oxide materials where magnetic excitations are suppressed at low temperature by an energy gap Δ [37]. The purpose of the simulation is to determine this gap Δ by calculating the magnetic susceptibility χ as a function of the temperature T and fitting it to the expression $\chi(T) \sim \frac{1}{\sqrt{T}} \exp^{-\Delta/T}$ [150]. We first use the "looper" program [146] of ALPS to calculate $\chi(T)$ and then use the exploration features of VisTrails to explore the data and find the optimal range $[T_{min}, T_{max}]$ for the nonlinear fit. The results of this exploration are shown in Figure 6.6.

Persistent caching and provenance adds a number of important advantages for the ALPS users:

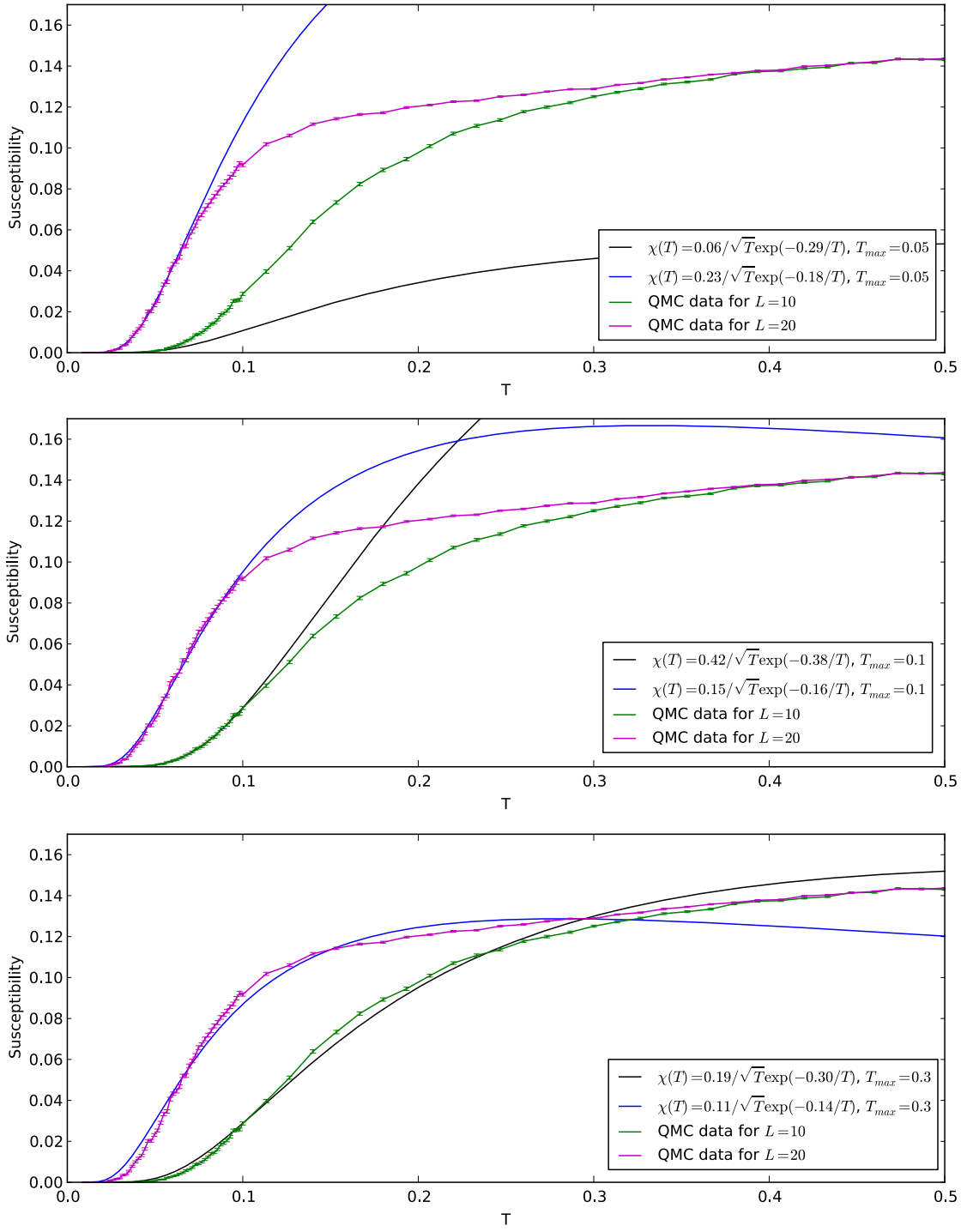
- Caching persistent files on a shared filesystem means that after one physicist runs the simulation, her colleague can modify the evaluation part of the workflow and explore the data without having to redo the time-intensive simulation.
- Identifying the cached files with the workflow signature avoids potentially critical mistakes of using old simulation results when input parameters to the simulation change. In our experience, simulations have often been recomputed only to ensure that the data have been produced with the latest version of codes and input files.
- Embedding provenance information in the data and figures gives immediate access to the provenance including any aspect of the simulation a physicist might wish to know. Since most projects involve collaborations with other scientists—often at different institutions—facilitating the exchange of data is very valuable. A common source of confusion is incom-

[5] `http://alps.comp-phys.org/`



(a) Workflow

Figure 6.6: An ALPS workflow colored by execution status (a). Blue modules were not executed since intermediate data existed in persistent storage; yellow modules were cached in memory; and green modules were executed. The results of a parameter exploration of the fitting range (b).



(b) Results

Figure 6.6: Continued.

plete documentation of data sent to collaborators. Embedded provenance information has been invaluable in making remote collaborations more efficient.

- Decoupling the executions of different parts of the workflows using persistent data enables physicists to explore data without the need to always rerun the entire workflow—while still having the workflow provenance accessible when needed.

In Figure 6.6, we show one ALPS workflow along with plots resulting from an exploration of the fitting range parameter. The modules colored in blue, including the time-consuming simulation module "AppLoop", were not run when this workflow was executed to create the plots because the output of the simulation had previously been persistently cached. Only the evaluation part of the workflow was re-executed when the fit range $[T_{min}, T_{max}]$ was modified. Note that the `SimulationID` module is striped blue and yellow; this is because it has two outgoing connections, one used in a file stored in the persistent cache and the other as part of a computation using the in-memory cache. Changing its value or structure would thus invalidate both cached results and all others downstream.

6.1.7 Related Work

Data provenance consists of the trail of processing steps and inputs that led to the creations of a given data object. Tracking changes to files and entire directory structures is well-studied, and version control systems have been developed exactly for this purpose [36, 140]. However, such systems can only determine that changes have occurred, not how they came about. More recently, version control systems that focus on tracking content and directory structure separately have been developed (see *e.g.,* [56]). Such systems identify files with hashing, and if duplicate files exist, the content is stored only once in the repository.

A number of workflow systems have been developed to help automate and manage complex calculations. The structure and abstraction provided by such systems have made them appealing to a wide assortment of scientific domains. Many of these systems [81, 145, 153] have included provenance capture to document the process and data used to derive data products [38, 47]. Standard provenance captured by these systems, however, is not sufficient to identify exactly which workflow generated a specific file. In fact, in recent exercises to investigate requirements for querying and integrating provenance information, the lack of effective means to identify intermediate and final results of workflows has been identified as an important challenge in provenance management [105, 122, 123]

Techniques have been developed to track provenance in databases [27]. These track fine-grained provenance, *i.e.,* changes to individual data items. In contrast, our approach is targeted to (whole) files. In future work, we plan to investigate how we can adapt our system to utilize database

provenance given encapsulated changes.

There is a significant amount of work with workflows that access and maintain curated data. In these cases, the provided ids or URIs are usually guaranteed to exist, and thus provenance information with them. Plale *et al.* have examined the issues involved in maintaining and cataloging large meteorological data, and noted the importance of allowing users to search and access this data [120]. Simmhan *et al.* have proposed data valets as a workflow-based method for facilitating the management of stores on the Cloud [136]. Note that if data for computations come from or are persisted to a curated source, a separate managed store is not required to ensure access to those files. However, maintaining local copies of these files does allow users to run workflows even when they cannot connect to the store.

For curated scientific data, the identification of those data is important. There are standards for such identification including LSID [125] and DOI [116]. Our primary goal is orthogonal to these: we aim to maintain strong links between data and their provenance. We are not concerned with registering ids for our local persistent stores and use UUIDs to identify data. Identifying data by content hashes has been accomplished using the MD5 and SHA1 hashes. Hashing has also been used in the context of secure provenance to maintain the confidentiality and integrity of provenance [60]. We use hashing to both identify and search for content as well as compute signatures for upstream subworkflows.

The problem with maintaining the data with workflows has been examined before. Some systems have provided specific modules for file management as part of workflow execution [106]. For example, after generating a data product, the result is not only displayed but also archived in a specific location or disk. This approach works well for static workflows, but for exploratory tasks, archival is not often included. The `cacher` package for R[6] provides a way to export verifiable statistical analysis and data in a tamper-proof scheme that utilizes hashing [118].

While we developed our store to aid users who use local files as data sources, our discussion of sharing the data in these stores overlaps many issues that have been considered. There already exist a number of solutions for managing scientific data on the grid and in cloud environments. GridFTP [6] and storage resource managers [134] have been developed to efficiently access data sets by utilizing networked resources. Such solutions can help provide faster access to data and infrastructure for transferring data across persistent stores.

---

[6] `http://www.r-project.org`

6.1.8 Summary

We have presented file management infrastructure that can be integrated with workflow systems to provide strong links to data in provenance information. In addition, we have discussed how such links can be used to solve provenance queries, facilitate persistent caching, and impact scientific publishing. Finally, we have described our implementation of this system in VisTrails and its use in the ALPS project.

One important aspect that we have not addressed is how the persistent store should be managed. In theory, keeping all of the data manipulated by workflows would ensure full reproducibility, but this is impractical for large amounts of data. In future work, we plan to investigate different strategies for determining when data can be purged from the store; for example, cached data that has not been annotated. While our current implementation supports a rich class of queries over the information in the repository, we would also like to support queries that involve workflow specification and the data involved—for example, finding a workflow with a `ParseCensusData` module that accesses the `census2010.dat` file.

Another area for future study is the automatic identification of intermediate files for caching. While users can identify important way points, it can be tedious to add such modules to a large collection of workflows. By examining the timestamps of module execution in provenance, we may be able to determine which steps are time-intensive and could benefit from caching. Also, the size of the intermediate result may also be important; if a large file is generated by a time-intensive step, but the next step strips unneeded information away, it may be more efficient to store the file after the extra information has been removed.

6.2 The Provenance of Workflow Upgrades

As tools that capture and utilize provenance are accepted by the scientific community, they must provide capabilities for supporting reproducibility as systems evolve. Like any information stored or archived, it is important that provenance be usable both for reproducing prior work and migrating that work to new environments. Just as word processing applications allow users to load old versions of documents and convert them to newer versions and data processing libraries provide migration paths for older formats, provenance-enabled tools should provide paths to upgrade information to match newer software or systems. Furthermore, it is important to capture and understand the changes that were made in order to run a previous computation in a new environment. One goal in documenting provenance is that users can more easily verify and extend existing work. If a given computation cannot be translated to newer systems or software versions, extensions become more difficult.

Workflow systems have made significant strides in allowing users to quickly compose a variety

of tools while automatically capturing provenance information during workflow creation and execution [39, 47]. Such systems enforce a structure on computations so that each workflow step is easily identifiable. Unfortunately, while these systems provide interfaces to a variety of routines and libraries, they are limited in their ability to upgrade workflows when the underlying routines or their interfaces are updated. It is well-known that software tends to *age* [115]. As requirements change, so do implementations and interfaces. This is more starkly obvious in the case of workflows, where different software tools from a variety of different sources need to be orchestrated. Figure 6.7 shows an example of different modifications that can be applied to workflow modules, including the addition of new parameters, the merger of two modules, and the replacement of the underlying computation. Still, many workflow systems do store information about the versions of routines as provenance. We seek to use this information to design schemes that allow users to migrate their work as newer algorithms and systems are developed.

There are two major approaches when dealing with upgraded software components and the documents or applications that utilize them. It is often important to maintain old versions of libraries and routines for existing applications that rely on them. In this case, an upgrade to a library should not replace the existing version but rather augment existing versions. Such an approach is common in system libraries and Web services where deleting previous versions can render existing code unexecutable. However, when we can safely upgrade the document or application to match the new interfaces, we might modify the object to utilize the new version. This second approach is more often used for documents than for existing applications or code, because there exists an application that can upgrade old versions. While the first approach is important to ensure that the original work can be replicated, because workflows are only loosely coupled to their implementations and live in the context of a workflow system, this second approach is sensible for them. Furthermore, as Figure 6.8 illustrates, by capturing the provenance of the upgrades, we know exactly what has been changed from the original version and how it might be reverted.

In order to accomplish the goal of upgrading an existing workflow, we must solve the challenges of detecting when upgrades are necessary and applicable, as well as dealing with routines (modules) from disparate sources. Because workflow systems often store information about the modules included in workflows, it is possible to detect when the current implementation of a module differs from one that was previously used. However, since they come from different sources, each source may define or release upgrades differently. Thus, we cannot hope to upgrade workflows atomically; we need to consider specific concerns from each source. Finally, while some upgrades may be automated or specified by a developer, others may require user intervention. When the user needs to be in the loop, it is important to make the process less tedious and error-prone.

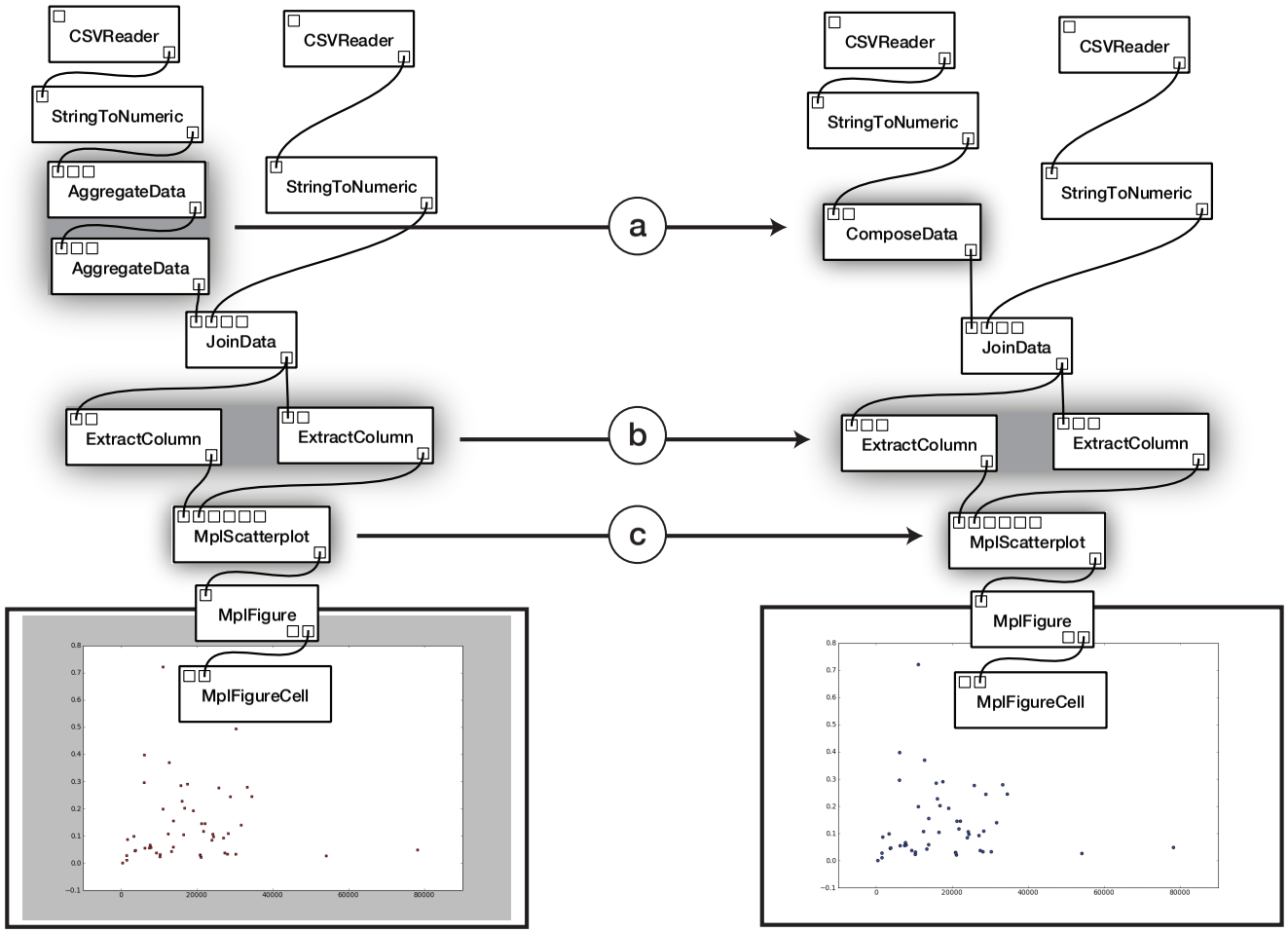
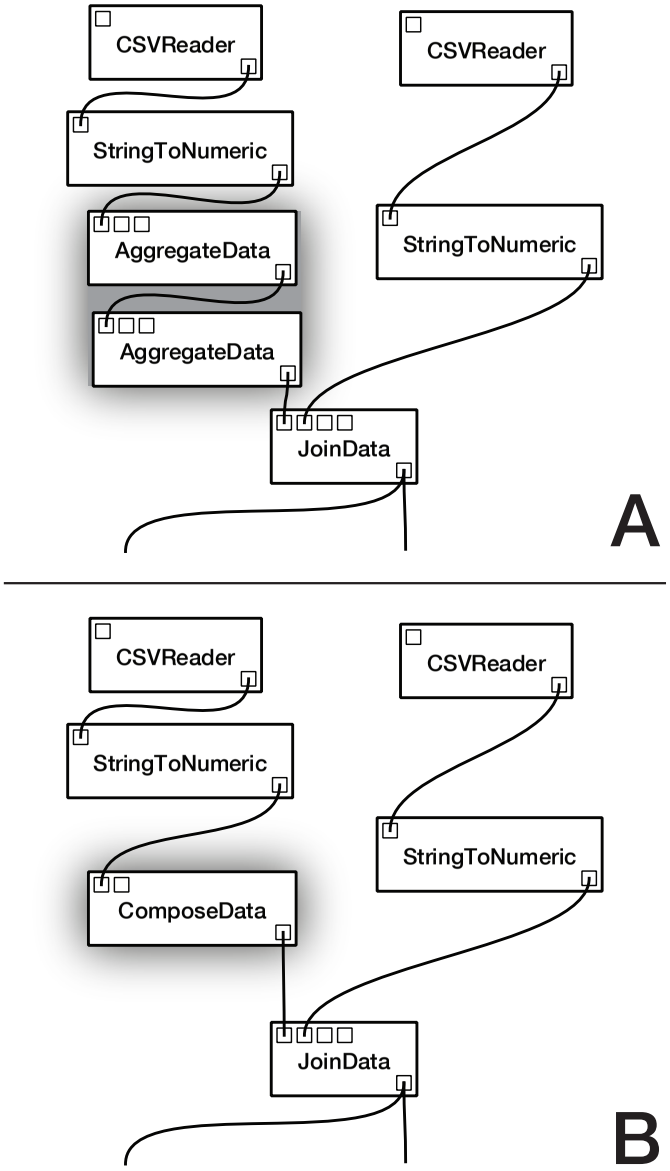


Figure 6.7: A workflow comparing road maintenance and number of miles of road by state before and after upgrading two packages. In (a), the `AggregateData` module has been replaced, and the developer has specified an upgrade to combine multiple aggregation steps into a single `ComposeData` module. In (b), the interface of `ExtractColumn` has been updated to offer a new parameter. Finally, in (c), the interface of the plotting mechanism has not changed, but the implementation of that module has, as evidenced by the difference in the background of the resulting plots.



Upgrade Provenance (A → B)
delete connection `StringToNumeric` → `AggregateData`
delete connection `AggregateData` → `AggregateData`
delete module `AggregateData` version 1.0.4
delete connection `AggregateData` → `JoinData`
delete module `AggregateData` version 1.0.4
add module `ComposeData` version 1.1.0
add connection `StringToNumeric` → `ComposeData`
add connection `ComposeData` → `JoinData`

Execution Provenance (A)
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
→ **execute module `AggregateData` version 1.0.4**
→ **execute module `AggregateData` version 1.0.4**
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
execute module `JoinData` version 1.0.0
...

Execution Provenance (B)
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
→ **execute module `ComposeData` version 1.1.0**
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
execute module `JoinData` version 1.0.0
...

Figure 6.8: On the right, we show the provenance of upgrading workflow (A) to the updated workflow (B). Besides the provenance of the upgrade, here we show the provenance of the executions of both (A) and (B). Note that version information is maintained in both forms of provenance.

We propose a routine for detecting when a workflow is incompatible with current installed software and approaches for both automated and user-defined upgrades. Our automated algorithm combines default routines for cases when only implementations changes with developer-defined routines, and uses a piecewise algorithm to process all components from each package at once. This allows complex upgrades, like replacing a subworkflow containing three modules with a single module. For user-defined upgrades, we suggest how a user might define a single upgrade once and apply it automatically to a collection of workflows. Finally, we discuss how upgrades should be considered as an integral part of the information currently managed by provenance-enabled systems. It is critical that we can determine what steps may have led to an upgraded workflow producing different results from the original. We describe our implementation of this upgrade framework in the VisTrails system [153].

6.2.1 Workflow Upgrades

6.2.1.0.1 Incompatible workflows. After a workflow is created, changes to the underlying implementation of one or more of its modules may make the workflow *incompatible*. Figure 6.9 shows an incompatible and a valid version of a workflow. Because module registry information is usually not serialized with each workflow, it can be difficult for users to define upgrades for obsolete workflows. As shown in the figure, although we may lack the appropriate code to execute a module or display the complete set of input and output interfaces for a module, we can display each module with the subset of ports identified by connections in the workflow. This is useful to allow users to edit incompatible workflows in order to make them compatible with their current environment.

6.2.1.0.2 Provenance of module implementation. Workflow systems offer mechanisms for capturing provenance information both about the evolution of the workflow itself and each execution of a workflow [47, 48, 129]. Information about the implementations used for each workflow module may be stored together with either evolution or execution provenance (*i.e.,*, the execution log). However, note that if the interface for a module changes, it will often require a change in the workflow specification. Thus, while the execution provenance may contain information about the versions used to achieve a result, any change in the interface of a module may make reproducibility via execution provenance alone difficult. By storing information about the implementations (like versions of each module) as evolution provenance, we can connect the original workflow to all upgraded versions.

Workflow evolution can be captured via *change-based provenance* [48], where every modification applied to a workflow is recorded. The set of changes is represented as a tree where nodes correspond to workflow versions and an edge between two nodes corresponds to the difference between the two corresponding workflow specifications.

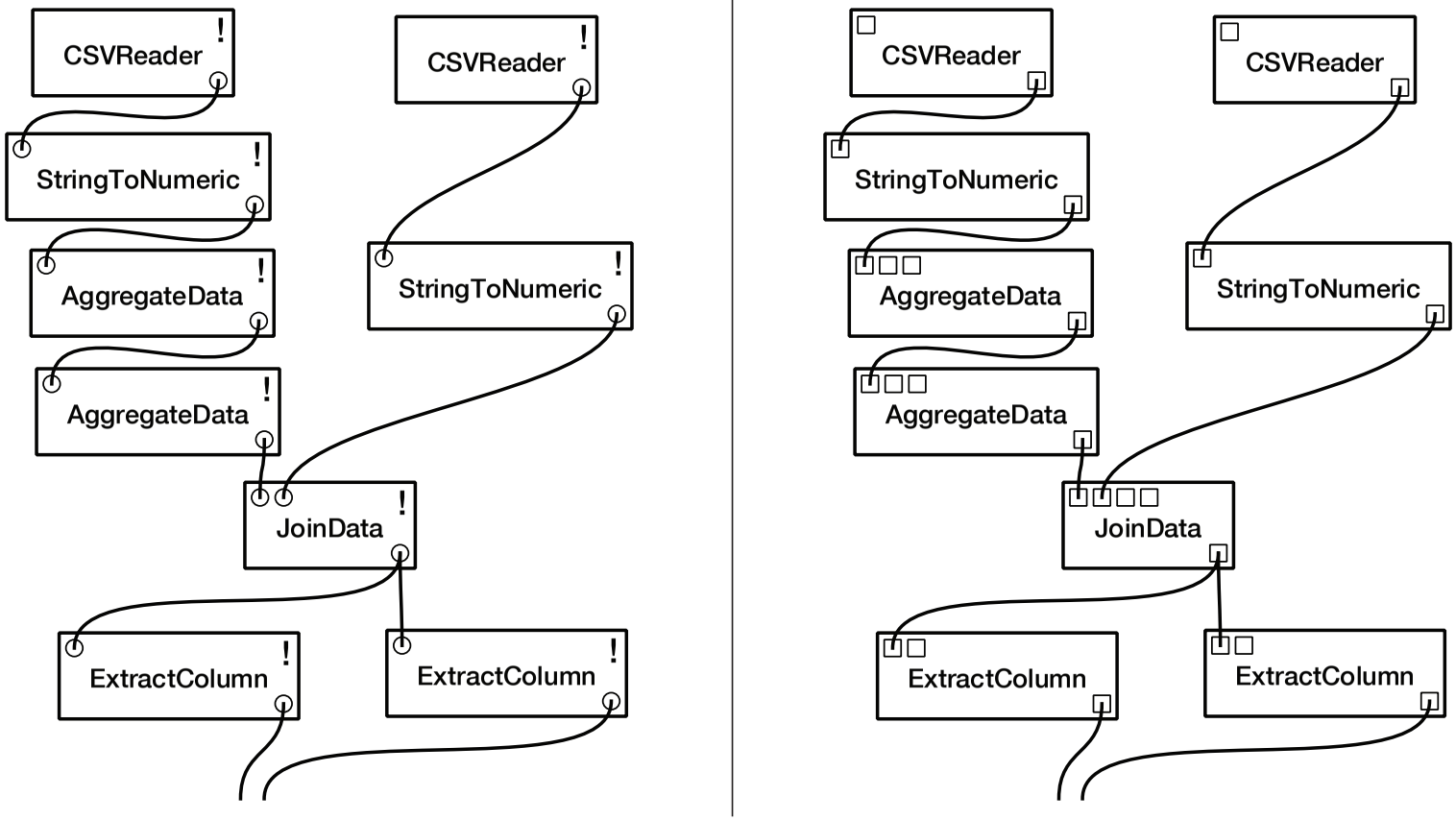


Figure 6.9: Incompatible (left) and valid (right) versions of a workflow. In an incompatible workflow, the implementation of modules is missing, and thus, no information is available about the input and output ports of these modules.

Any workflow instance can be reconstructed by applying the entire sequence of change operations from the root node to the current version. For upgrades, we can leverage this approach to record the set of changes necessary to update an old workflow to a new version. Note that these changes define the difference between the two versions, so our provenance will maintain an explicit definition of the upgrade for reference and comparison.

6.2.1.1 Detecting the Need for Upgrades

To support upgrades, workflow systems must provide developers with facilities to develop and maintain different versions of modules (and packages) as well as detect and process inconsistencies when workflows created with older versions of modules are materialized. First, it is important to have a mechanism for identifying a group of modules (*e.g.*,, using a group key), as well as a version indicator or some other method like content-hashing that can be used to identify when module implementations may have changed. Ideally, any version identifier of a module should reflect the version of the code or underlying libraries. In fact, we may be able to aid developers by signaling when their code has changed, alerting them to the need to change the version. Alternately, developers might link version identifiers to revisions of their code as defined in a version control system.

Second, we need to tackle the problem of identifying when and where upgrades might be necessary. Upon opening a workflow, the system needs to check that the modules specified are consistent with the implementation defined by the module registry. As discussed earlier, this usually involves checking version identifiers but could also be based on actual code. If there are inconsistencies, we need to identify the type of discrepancy; the workflow may specify an obsolete version of a module, a newer version, or perhaps the module may not exist in the current registry. In all of these cases, we need to reconcile the workflow to the current environment.

6.2.1.2 Processing Upgrades

We wish to allow developers to specify upgrade paths but also provide automated routines when upgrades are trivial and allow users to override the specified paths. The package developer can specify how a specific module is to be upgraded in all contexts. If that is not possible or the information is not available for a given module, we can attempt to automatically upgrade a module by replacing the old version with a new version of the same module. A third method for upgrading a workflow is to display the obsolete modules and let the user replace them directly. Our upgrade framework leverages all three approaches. It starts with developer-specified changes, provides default, automated upgrades if the developer has not provided them, and allows the user to choose to accept the upgrade, modify it, or design their own.

6.2.1.2.3 Developer-defined upgrades. Because the modules of any workflow may originate from a number of different packages, we cannot assume that a global procedure can upgrade the entire workflow. Instead, we allow developers to specify upgrade routines for each package. Specifically, we allow them to write a method which accepts the workflow and the list of incompatible modules. A module may be incompatible because it no longer exists in the package or its version is different from the implementation currently in the registry. A developer needs to implement solutions to handle both of these situations, but the system can provide utility routines to minimize the effort necessary for some types of changes. In addition, there may be cases where the developer wants to replace entire subworkflows with different ones. Changes in the specification of parameter values may also require upgrade logic. For example, an old version of module may have taken the color specification as four integers in the [0,255] range, but the new version requires floats in the [0.0, 1.0] range. Such conversions can be developer-specified so that the a user need not modify their workflows in order for them to work with new package versions. Note that developer-specified upgrades may need to be aware of the initial version of a module. For example, if version 0.1 of a module has a certain parameter, version 0.2 removes it, and version 1.0 adds it back, the upgrade from 0.1 to 1.0 will be necessarily different than the upgrade from 0.2 to 1.0.

6.2.1.2.4 Automatic upgrades. We can attempt to automate upgrades by replacing the original module with a new version of the same module. For any module that needs an upgrade, we check the registry for a module that shares the same identifying information (excluding version) and use that module instead. Note that it is necessary to recreate all incoming and outgoing connections because the old module is deleted and a new module is added. If an upgraded module renames or removes a port, it is not possible to complete the upgrade. We can either continue with other upgrades and notify the user, or rollback all changes and alert the user. Also note that if two connected modules both require upgrades, we will end up deleting and adding at least one of the connections twice, once for the first module replacement and again for the second module upgrade. Finally, we need to transfer parameters to the new version in a similar procedure as that used with connections. See Figure 6.10 for an example of an automatic upgrade.

6.2.1.2.5 User-assisted upgrades. While we hope that automatic and developer-specified upgrades will account for most of the cases, they may fail for complicated situations. In addition, a developer may not specify all upgrade paths or a user may desire greater control over the changes. In such a scenario, we need to display the old, incompatible pipeline, highlight modules that are out-of-date, and allow the user to perform standard pipeline manipulations. One problem is that, because we may not have access to the version of the package that was used to create the workflow, we may not be able to display the module correctly for the user to interact with. With VisTrails,

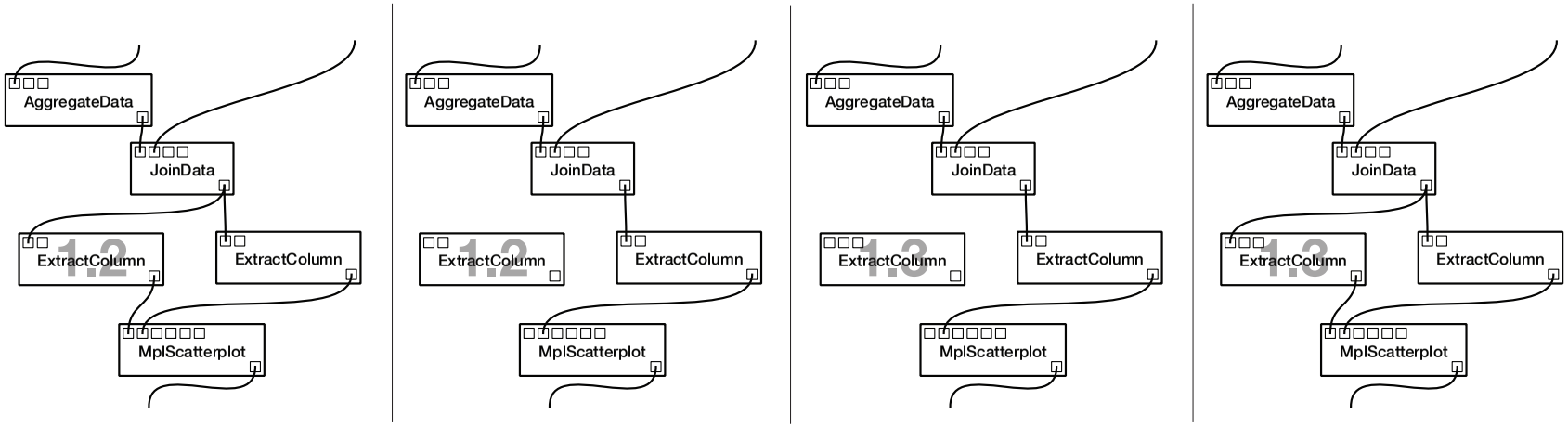


Figure 6.10: Upgrading a single module automatically involves deleting all connections, replacing the module with the new version, and finally adding the connections back.

we can display the basic graph connectivity as shown in Figure 6.9, but we may not have entire module specification. Our display is therefore a "recovery mode" where the workflow is shown but cannot be executed or interacted with in the same way as a valid version. Once users replace all old modules with current versions, they will able to execute the workflow and interact with it. We can aid users by providing high-level actions that allow them to, for example, replace an incompatible module with a new, valid one.

In addition, while users may be willing to perform one or two upgrades manually, it would be helpful if we are able to aid users by automating future upgrades based on those they have already defined. Workflow analogies provide this functionality by allowing users to select existing actions including upgrades and apply them to other workflows [130]. Because analogies compute a soft matching between starting workflows, they can be applied to a variety of different workflows. Thus, for a large collection of workflows, a user may define a few upgrades and compute the rest automatically using these analogies.

6.2.1.3 Provenance Concerns

Given a data product, we cannot hope to reproduce or extend the data product without knowing its provenance—how it was generated. If our provenance information includes information about the versions of the modules used, we can use that to drive upgrades. Note that without version information, we may incorrectly determine which upgrades are necessary. Thus, the provenance of the original workflow is important to define the upgrade.

At the same time, we wish to capture the provenance of the upgrades. When users either run old versions of workflows or upgrade and modify these versions, it is important to track the changes both in the execution provenance and in the workflow evolution provenance. By noting the specific module versions used in the execution provenance, we can better support reproducibility. We need to ensure that the versions recorded are exactly the versions executed, not allowing silent upgrades to happen without being noted in the provenance. Similarly, whenever a user upgrades a workflow, the changes that took place should be noted as evolution provenance so that subsequent changes are captured correctly. See Figure 6.8 for an example of captured provenance information that is relevant for upgrades.

As a workflow evolves over a number of years and is modified by a number of users, it is important to track the provenance of this evolution. Upgrades may be critical changes in workflow development and often occur when a new user starts to revise an existing result. By keeping track of these actions, we may be able to identify how, for example, inconsistencies in results may have arisen because of an upgrade. In addition, we do not lose links as workflows are refined. Without upgrades, a user may create a (duplicate) workflow rather than reuse an existing one. If that occurs,

we lose important provenance of the original workflow and related workflows.

6.2.2 Implementation

We have implemented the framework described in Section 6.2.1 in the VisTrails system. Below, we describe this implementation.

In VisTrails, when a workflow is loaded (or materialized), it is validated against the current environment: the classes defining the modules and the port types for each module. To detect whether modules have changed, we begin by checking each module and ensuring the requested version matches the registry version. Next, we check each connection to ensure that the ports they connect are also valid. Finally, we check the parameter types to ensure they match those specified by the implementation. If any mismatches are detected, we raise an exception that indicates what the problem is and which part of the workflow it affects. Note that if one problem occurs, we can immediately quit validation and inform the user, but if we wish to fix the problems, it is useful to identify all issues. Thus, we collect all exceptions during validation, and pass them to a handler.

We attempt to process all upgrades at once, with the exception of subworkflows which are processed recursively. To this end, we sort all requests by the packages that they affect, and attempt to solve all issues one package at a time. This way, a package developer can write a handler to process a group of upgrade requests instead of processing each request individually.

6.2.2.1 Replace, Remap, and Copy

Note that an upgrade that deletes an old module and adds a new version discards information about existing connections, parameters, and annotations. In order to maintain this information as well as its provenance, we extended VisTrails change-based provenance with a new change type (or action) that replaces the original module, remaps the old information, and copies it to a new version of the module. This ensure that we transfer all relevant information to the new version and maintain its provenance. The new action extracts information about connections, parameters, and annotations from the old module before replacing it, and then adds that information to the new module. Note that, because interfaces may change, we allow the user to remap parameter, port, or annotation names to match the new module's interface.

6.2.2.2 Algorithm

Formally, our algorithm for workflow upgrades takes a list of detected inconsistencies between a workflow and the module registry and produces a set of actions to revise the workflow. We categorize inconsistencies as "missing", "obsolete", or "future" modules, and this information is encoded in the exception allowing developers to tailor upgrade paths accordingly. We begin by

sorting these errors by package identifier. Then for each package, we check if the package has a handler for all types of upgrades. If it does, we call that handler. If not, we cannot hope to reconcile "missing" modules. For obsolete modules, we can attempt to automatically reconcile them by replacing them with newer versions. For future modules, we can attempt to downgrade them automatically, but usually we raise this error to the user.

Automatic upgrades work module by module, and for each module, we first check to see if an upgrade is possible before proceeding. An upgrade is possible if the module interface has not changed from the version specified by the workflow and the version that exists on the system. We check that by seeing if each connection and parameter setting can be trivially remapped. If they can, we extract all of the connections and parameters before deleting all connections to the module and the module itself. Then, we add the new version of the module and replace the connections and parameters. All of these operations are encoded as a single action.

For developer-defined upgrades, we pass all of the information about inconsistencies as well as the current state of the workflow to the package's upgrade handler. The handler can make use of several capabilities of the workflow system to minimize the amount of code. Specifically, we have a remap function that allows a developer to specify how to replace a module when interface changes are due to renaming. In addition, developers can replace entire pieces of a workflow, but doing so might require locating subworkflows that match a given template. Many workflow systems already have query capabilities, and these can be applied to facilitate these more complex upgrades. As with automatic upgrades, these operations are encoded as a single action.

If automated and developer-defined upgrades cannot achieve a compatible workflow, a user can define an upgrade path. Most of this process is manual and mirrors how a user might normally update a workflow. Until the workflow is compatible with the current environment, the workflow cannot be executed, giving users a well-defined goal. Upon achieving a valid workflow, we can save the user's actions and use workflow analogies [130] to help automate future upgrades.

6.2.2.3 Subworkflows

To handle subworkflows, both validation and upgrade handling are performed recursively. Thus, we process any workflow by first recursing on any subworkflow modules, processing the underlying workflows, and then continuing with the rest of the workflow. However, our upgrades must be handled using an extra step; if we update a subworkflow, we must also update the module tied to that subworkflow to reflect any changes. For example, a subworkflow may modify its external interface by deleting inputs or outputs. Thus, we must upgrade a module after updating its underlying subworkflow.

6.2.2.4 Preferences

While upgrades are important, we wish to add them without interfering with a user's normal work. Besides the choice between upgrading or trying to load the exact workflow with older package versions, a user may also wish to be notified of upgrades and persist their provenance in different ways. Specifically, if old versions exist, a user may wish to always try use them, automatically upgrade, or be prompted for a decision. If not, a user has a similar selection of options: never upgrade, always upgrade, or be presented with the choice to upgrade. When a user wants to upgrade, he may choose to persist the provenance of these upgrades immediately or delay saving these changes until other changes occur. If a user is browsing workflows, it may be reasonable to only persist upgrade provenance when the workflow is modified or run. This way, a user can examine a workflow as it would appear after an upgrade, but the persistence of these upgrades is delayed until something is changed or the workflow is executed. Users might also want to have immediate upgrades where the upgrade provenance is persisted exactly when any workflow is upgraded, even if the user is only viewing the workflow.

6.2.3 Discussion

While perfect reproducibility cannot be guaranteed without maintaining the exact system configuration and libraries, we believe that workflow upgrades offer a sensible approach to manage the migration from older workflows to new environments. Note that provenance allows us to always revisit the original workflow, and we can run this version if we can reconstruct the same environment. By storing the original implementations along with workflows, we may be able to reproduce the original run, although changes in the system configuration may limit such runs. Thus, coupling provenance with version control systems could ensure that we users can access previous package implementations. However, when extending prior work in new environments, upgrades also serve to convert older work to more efficient and extensive environments. In addition, managing multiple software versions is a nontrivial task, and even with a modern OS package management system, installing a given package in the presence of conflicts is actually known to be NP-Complete [41]. Thus, we cannot expect in general to easily run arbitrarily old library versions. Because workflows abstract the implementation from the computational structure, the results of upgrades are more likely to be valid.

Some workflow systems use Web services or other computational modules that are managed externally. In these cases, we may not know if the interface or implementation may have changed, so it is harder to know when upgrades are necessary. However, the services may make version information available or the workflow system may be able to detect a change in the interface [21]. In this case, we are not able to leverage developer-specified upgrade routines, but we should be able

to accomplish automatic or user-specified upgrades.

When using change-based provenance to track upgrades, a user can see both the original evolution as well as the upgrades and progress after the upgrades. See Figure 6.11 for an example. It may be useful to upgrade an entire collection of related workflows while retaining the original provenance of exploration, but adding the upgraded versions may lead to a complex interface. We believe that restructuring the tree to display the original history but with links to the upgrades might be useful. Finally, we emphasize that the change-based model for the workflows provenance in VisTrails is an attractive medium in which to incorporate the upgrading data. Since the upgrades are represented as actions, they are treated as first-class data in the system, and so the extensive process provenance capabilities of VisTrails can be directly used. For example, upgrade actions can then be used in queries or incorporated into statistical analyses [93, 129, 130].

6.2.4 Related Work

Workflow systems have recently emerged as an attractive alternative for representing and managing complex computational tasks. The goal behind these systems is to provide the utility of the shell script in a more user-friendly, structured manner. Workflow systems incorporate comprehensive metadata which, among other advantages, facilitates programming and distribution of results [95], reproducibility [48], allows better execution monitoring [104], and provides potential efficiency gains [17].

As the auditability and cost of generating results has increased, managing the provenance of data products [137] and computational processes [48] has become very important. Together, these ideas allow users to obtain a fairly comprehensive picture of the programs and data that were used to generate final results. However, these descriptions are, in a sense, static. In general, the processes are assumed to stay the same for the lifetime of the workflow, and, as we have argued before, longevity necessarily introduces changing requirements and interfaces. Our approach serves to detect and manage these changes to underlying implementations while still keeping the attractive features of workflow systems described above.

It is well known that longevity introduces novel challenges for maintainability of software systems, in particular in the presence of complicated dependencies [115]. There have been a number of approaches to the problem of managing software upgrades, in particular, in understanding and ensuring safety properties of dynamic updates in, for example, running code or persistent stores [20, 43]. In small-scale environments, the solutions tend to involve the description (or prediction) of desired properties to be maintained [98]. For deployments at the scale of entire institutions or large computer clusters, they tend to involve careful scheduling, and staged deployment of upgrades [4, 32].

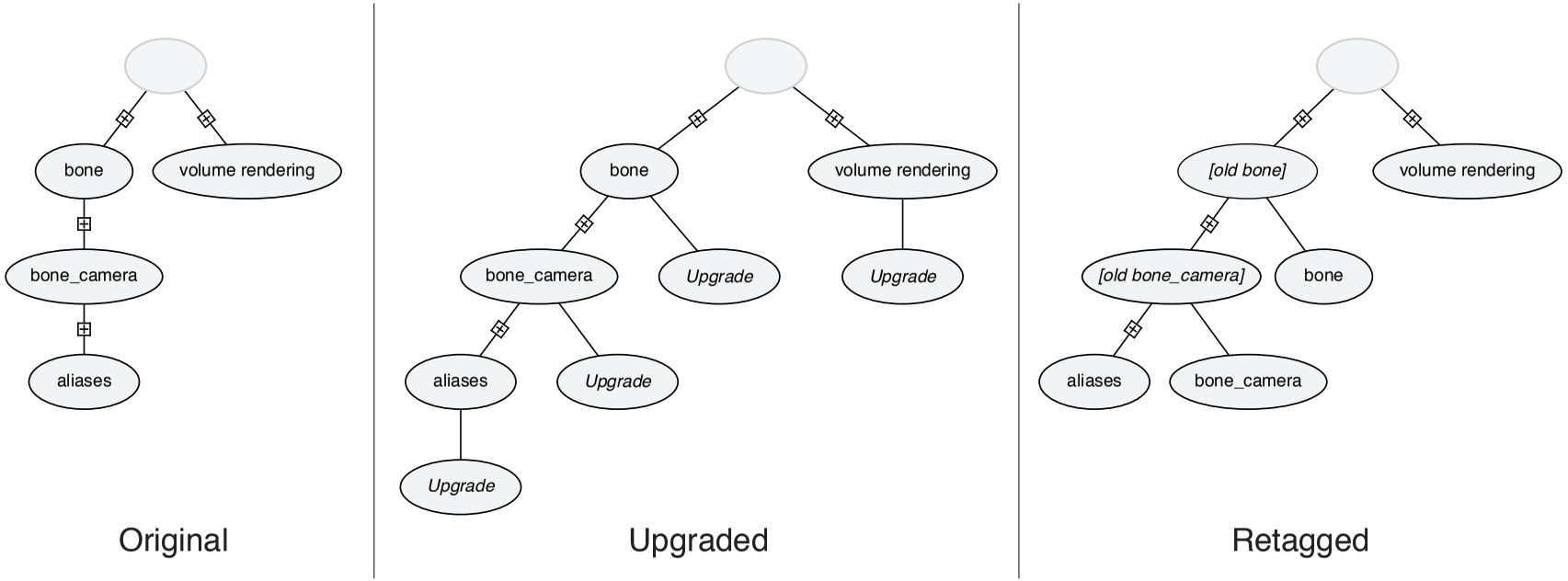


Figure 6.11: Workflow Evolution before and after upgrades as well as after retagging the nodes.

Component-based software has evolved to separate different concerns in order to provide wide-ranging functionality. While usually at a lower level than workflow systems, component objects use well-defined interfaces and are substitutable [142]. For this reason, it is also important to track the component evolution and versioning [139]. The term "dependency hell" was coined to describe problems with compatibility when replacing components with new versions. McCamant and Ernst describe methods to identify such incompatibilities [99] while Stuckenholz proposes "intelligent component swapping" to update multiple components at once [139].

Web services are another kind of component-based architecture. Since the standards do not address the evolution of Web services, developers must rely on design patterns and best practices [21]. Specifically, adding to an interface is possible, but changing or removing from that interface is not. Andrikopoulos *et al.* formalize the concepts of service evolution [11]. There are a variety of approaches that seek to develop mechanisms to version Web services including using a chain of adapters [79] and hierarchical abstraction [148]. In order to publish such versions, services are distinguished via namespaces or URLs. In contrast to much of the work for component upgrades, our approach seeks to add capability by updating older workflows rather than only maintaining backward compatibility.

In this chapter, we focus on the problem of providing a means of describing upgrade paths so that a workflow can be automatically updated, its upgrade history appropriately recorded, and its execution made sufficiently similar to the one before the upgrade. Such problems exist even when lower-level upgrades are successfully deployed. In that sense, our mechanisms for coping with upgrades are closer in spirit to mechanisms for automatically updating database queries after relational schemas have changed [35].

6.2.5 Summary

We have proposed a framework for workflow upgrades and described its implementation in the VisTrails system. Our framework handles three types of upgrades—automated, developer-specified, and user-defined, and we have discussed how these can be supported in a systematic fashion. We have also shown how the framework leverages provenance information to accomplish upgrades and produces updated provenance detailing the changes introduced by the upgrades. Our implementation is currently available in nightly releases of VisTrails, and we are planning to incorporate it into the next major release of VisTrails.

One area that we would like to explore further is the interface for involving the user in upgrades. The "replace module" action allows users to specify how an upgrade is accomplished, and we believe a user might drag a new module onto the incompatible module to replace it. At the same time, if the routine specifications do not exactly match, the user should be able to specify the

remapping, similar to the method available to developers in their code. We might extend this functionality to allow the user to specify the connections visually.

In addition to capturing the provenance of upgrades and using this information to guide future user-driven, manual upgrades, we believe we might also use this provenance for further analysis. For example, we might be able to examine the actions used in upgrades to mine rules for packages whose developers have not defined upgrade paths. It may also be interesting to try to analyze performance or accuracy changes in workflow execution after upgrades.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This dissertation has presented a set of techniques for both managing and using provenance information. From linking data with provenance to suggesting possible workflow completions, the methods blend infrastructure with applications. In doing so, this work looks beyond normal capture and archival of provenance information to using those data to drive new ideas and to improve the access to details of algorithms, implementations, and experiments.

Chapter 3 introduced a method for reusing the provenance from a collection of workflows to derive suggestions as users built workflow structures. This solution allows novice users to browse the set of possible completions for ideas while other users can benefit from a reduced number of operations to complete a pipeline. One of the remaining questions is how this technique can be extended to support parameter selection. Because parameters often depend on input data (or other parameters), it may be best to present possible ranges or compute the workflows with a variety of different parameters so users can evaluate the results. Another opportunity is changing the algorithm to be more query-driven, utilizing the framework from Chapter 4, to show full results rather than generalizing skeletons.

The next chapter, Chapter 4, presents a technique for indexing provenance and workflow graphs. Extending feature-based graph indexing techniques, it seeks to address more exploratory queries where the number of results might be large as well as queries that are not exactly specified. Using summary graphs, it reduces the number of costly verification steps which require a solution to subgraph isomorphism. In order to efficiently evaluate queries where the connectivity may not be fully specified, transitive closure for verticies must be checked, and a solution where connectivity can be indexed may aid in speeding up those queries further.

In Chapter 5, summary graphs are introduced as a method for browsing and comparing collections of graphs, including sets of provenance graphs. Here, the summary graph encapsulates similarities and differences using a heurisitic graph matching algorithm in a hierarchical agglomeration. The graph matching algorithm allows users to control the level to which nodes are merged, meaning the display can be tweaked in an intuitive manner. The algorithm works well for similar graphs with some differences, but for larger, more heterogenous collections, the pieces will share less overlap. This might be addressed by using clustering to develop groups of similar graphs and

generate a set of summary graphs instead of a single one. The scaling both of the number of graphs and size of graphs are something which deserves more investigation.

Chapter 6 addresses longevity concerns related to using provenance in publications. These include preserving and linking data with provenance information as well as providing upgrade paths so older results can be used and compared using updated hardware or algorithms. Referring to data with filenames is problematic since both the location and data might change. The persistence framework records information about data content and computational signatures while transparently managing and versioning the data so they can be later retrieved. Workflow upgrades use information about versions of workflow modules to determine when they may be out of date. Given an older version, there are three types of upgrades that can be used: automatic, developer-defined, and user-defined. Provenance is maintained throughout so the original and new versions are maintained. These solutions are important for scientific publication because reviewers and readers can better understand and validate experiments and computations when full provenance is available for reproduction and reuse. In the future, sharing both managed data and upgraded algorithms is an important concern, especially given the growing size of the data involved in scientific calculations

The presented techniques are designed to allow for more efficient knowledge discovery, and each recognizes the changing nature of scientific exploration from ideas through publications. Provenance captures the scientific process, the exploration and ideas that lead to published results. For this reason, it is valuable information that can be used to further scientific discovery, but efficient algorithms to process large collections of provenance are needed. This dissertation outlines several solutions for dealing with collections of provenance, ranging from query support to completion infrastructure. Scaling these approaches for the growing amount of data and provenance will be an important direction as this work continues.

REFERENCES

[1] AALST, W., AND HEE, K. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[2] ADAMS, III, E. N. Consensus techniques and the comparison of taxonomic trees. *Systematic Zoology 21*, 4 (December 1972), 390–397.

[3] AGRAWAL, R., IMIELIŃSKI, T., AND SWAMI, A. Mining association rules between sets of items in large databases. *SIGMOD Rec. 22*, 2 (1993), 207–216.

[4] AJMANI, S., LISKOV, B., AND SHRIRA, L. Scheduling and simulation: how to upgrade distributed systems. In *HOTOS* (2003), pp. 8–8.

[5] ALBUQUERQUE, A., ALET, F., CORBOZ, P., DAYAL, P., FEIGUIN, A., FUCHS, S., GAMPER, L., GULL, E., GÜRTLER, S., HONECKER, A., IGARASHI, R., KÖRNER, M., KOZHEVNIKOV, M., LÄUCHLI, A., MANMANA, S., MATSUMOTO, M., MCCULLOCH, I., MICHEL, F., NOACK, R., PAWLOWSKI, G., POLLET, L., PRUSCHKE, T., SCHOLLWÖCK, U., TODO, S., TREBST, S., TROYER, M., WERNER, P., AND WESSEL, S. The alps project release 1.3: open source software for strongly correlated systems. *J. Mag. Mag. Mat. 310* (2007), 1187.

[6] ALLCOCK, W., BESTER, J., BRESNAHAN, J., CHERVENAK, A., LIMING, L., AND TUECKE, S. Gridftp: Protocol extensions to ftp for the grid. *Global Grid Forum* (2001), 3.

[7] The ALPS project. `http://alps.comp-phys.org`.

[8] ALTINTAS, I., BARNEY, O., AND JAEGER-FRANK, E. Provenance collection support in the kepler scientific workflow system. In *Proceedings of IPAW* (2006), pp. 118–132.

[9] ANDERSON, E. W., AHRENS, J. P., HEITMANN, K., HABIB, S., AND SILVA, C. T. Provenance in comparative analysis: A study in cosmology. *Computing in Science and Engineering 10*, 3 (2008), 30–37.

[10] ANDERSON, E. W., PRESTON, G. A., AND SILVA, C. T. Towards development of a circuit based treatment for impaired memory: A multidisciplinary approach. In *IEEE EMBS Neural Engineering* (2007).

[11] ANDRIKOPOULOS, V., BENBERNOU, S., AND PAPAZOGLOU, M. P. Managing the evolution of service specifications. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 359–374.

[12] ARCHAMBAULT, D., MUNZNER, T., AND AUBER, D. Topolayout: Multilevel graph layout by topological features. *Visualization and Computer Graphics, IEEE Transactions on 13*, 2 (2007), 305–317.

[13] BALZER, M., AND DEUSSEN, O. Level-of-detail visualization of clustered graph layouts. In *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on* (2007), pp. 133 –140.

[14] BAO, Z., COHEN-BOULAKIA, S., DAVIDSON, S., AND GIRARD, P. Pdiffview: Viewing the difference in provenance of workflow results. *PVLDB, Proc. of the 35th Int. Conf. on Very Large Data Bases 2*, 2 (2009), 1638–1641.

[15] BAO, Z., COHEN-BOULAKIA, S., DAVIDSON, S. B., EYAL, A., AND KHANNA, S. Differencing provenance in scientific workflows. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 808–819.

[16] BAPTISTA, A., HOWE, B., FREIRE, J., MAIER, D., AND SILVA, C. T. Scientific exploration in the era of ocean observatories. *Computing in Science and Engineering 10*, 3 (2008), 53–58.

[17] BAVOIL, L., CALLAHAN, S., CROSSNO, P., FREIRE, J., SCHEIDEGGER, C., SILVA, C., AND VO, H. VisTrails: Enabling interactive, multiple-view visualizations. In *Proceedings of IEEE Visualization* (2005), pp. 135–142.

[18] BEERI, C., EYAL, A., KAMENKOVICH, S., AND MILO, T. Querying business processes. In *VLDB* (2006), pp. 343–354.

[19] BETHEL, W., SIEGERIST, C., SHALF, J., SHETTY, P., JANKUN-KELLY, T. J., KREYLOS, O., AND MA, K.-L. VisPortal: Deploying grid-enabled visualization tools through a web-portal interface. In *Third Annual Workshop on Advanced Collaborative Environments* (2003).

[20] BOYAPATI, C., LISKOV, B., SHRIRA, L., MOH, C.-H., AND RICHMAN, S. Lazy modular upgrades in persistent object stores. *SIGPLAN Not. 38*, 11 (2003), 403–417.

[21] BROWN, K., AND ELLIS, M. Best practices for Web services versioning. *IBM developer-Works* (2004). `http://www.ibm.com/developerworks/webservices/library/ws-version/`.

[22] BURKARD, R. E., DELL'AMICO, M., AND MARTELLO, S. *Assignment Problems*. SIAM, 2009.

[23] CARD, S. K., MORAN, T. P., AND NEWELL, A. The keystroke-level model for user performance time with interactive systems. *Commun. ACM 23*, 7 (1980), 396–410.

[24] CARD, S. K., NEWELL, A., AND MORAN, T. P. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1983.

[25] CHEN, C., LIN, C. X., FREDRIKSON, M., CHRISTODORESCU, M., YAN, X., AND HAN, J. Mining graph patterns efficiently via randomized summaries. *Proc. VLDB Endow. 2*, 1 (2009), 742–753.

[26] CHEN, Z., GEHRKE, J., KORN, F., KOUDAS, N., SHANMUGASUNDARAM, J., AND SRIVASTAVA, D. Index structures for matching xml twigs using relational query processors. *Data Knowl. Eng. 60*, 2 (2007), 283–302.

[27] CHENEY, J., CHITICARIU, L., AND TAN, W. C. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases 1*, 4 (2009), 379–474.

[28] CHENG, J., KE, Y., AND NG, W. Efficient query processing on graph databases. *ACM TODS 34*, 1 (2009), 1–48.

[29] CHENG, J., KE, Y., NG, W., AND LU, A. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD* (2007), pp. 857–872.

[30] CHILDS, H., BRUGGER, E. S., BONNELL, K. S., MEREDITH, J. S., MILLER, M., WHITLOCK, B. J., AND MAX, N. A contract-based system for large data visualization. In *IEEE Visualization* (2005), pp. 190–198.

[31] COOK, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), STOC '71, ACM, pp. 151–158.

[32] CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev. 41*, 6 (2007), 221–236.

[33] Crowdlabs. `http://www.crowdlabs.org`. Accessed 07/22/2010.

[34] CUI, W., ZHOU, H., QU, H., WONG, P. C., AND LI, X. Geometry-based edge clustering for graph visualization. *Visualization and Computer Graphics, IEEE Transactions on 14*, 6 (2008), 1277–1284.

[35] CURINO, C., MOON, H. J., AND ZANIOLO, C. Automating database schema evolution in information system upgrades. In *HotSWUp* (2009), pp. 1–5.

[36] Concurrent Versions System. `http://www.nongnu.org/cvs`.

[37] DAGOTTO, E., AND RICE, T. M. Surprises on the Way from One- to Two-Dimensional Quantum Magnets: The Ladder Materials. *Science 271*, 5249 (1996), 618–623.

[38] DAVIDSON, S. B., BOULAKIA, S. C., EYAL, A., LUDÄSCHER, B., MCPHILLIPS, T. M., BOWERS, S., ANAND, M. K., AND FREIRE, J. Provenance in scientific workflow systems. *IEEE Data Eng. Bull. 30*, 4 (2007), 44–50.

[39] DAVIDSON, S. B., AND FREIRE, J. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of SIGMOD* (2008), pp. 1345–1350.

[40] DEELMAN, E., SINGH, G., SU, M.-H., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BERRIMAN, G. B., GOOD, J., LAITY, A., JACOB, J. C., AND KATZ, D. S. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal 13*, 3 (2005), 219–237.

[41] DI COSMO, R. Report on formal management of software dependencies. Tech. rep., INRIA, Sep 2005. EDOS Project Deliverable WP2-D2.1.

[42] DOLGERT, A., GIBBONS, L., JONES, C. D., KUZNETSOV, V., RIEDEWALD, M., RILEY, D., SHARP, G. J., AND WITTICH, P. Provenance in high-energy physics workflows. *Computing in Science and Engineering 10*, 3 (2008), 22–29.

[43] DUMITRACS, T., AND NARASIMHAN, P. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Middleware* (2009), pp. 1–20.

[44] EADES, P., AND FENG, Q.-W. Multilevel visualization of clustered graphs. In *Graph Drawing*, S. North, Ed., vol. 1190 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997, pp. 101–112.

[45] FOMEL, S., AND CLAERBOUT, J. F. Guest editors' introduction: Reproducible research. *Computing in Science and Engineering 11* (2009), 5–7.

[46] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. Chimera: A virtual data system for representing, querying and automating data derivation. In *SSDBM* (2002), pp. 37–46.

[47] FREIRE, J., KOOP, D., SANTOS, E., AND SILVA, C. T. Provenance for computational tasks: A survey. *Computing in Science and Engineering 10*, 3 (2008), 11–21.

[48] FREIRE, J., SILVA, C. T., CALLAHAN, S. P., SANTOS, E., SCHEIDEGGER, C. E., AND VO, H. T. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop (IPAW)* (2006), LNCS 4145, pp. 10–18. Invited paper.

[49] FREW, J., METZGER, D., AND SLAUGHTER, P. Automatic capture and reconstruction of computational provenance. *Concurr. Comput. : Pract. Exper. 20* (April 2008), 485–496.

[50] FU, X., BUDZIK, J., AND HAMMOND, K. J. Mining navigation history for recommendation. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces* (2000), pp. 106–112.

[51] FURNAS, G. W., AND ZACKS, J. Multitrees: enriching and reusing hierarchical structure. In *CHI'94* (1994), pp. 330–336.

[52] GANSNER, E., KOREN, Y., AND NORTH, S. Topological fisheye views for visualizing large graphs. *Visualization and Computer Graphics, IEEE Transactions on 11*, 4 (2005), 457 –468.

[53] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper. 30* (September 2000), 1203–1233.

[54] GenePattern: A platform for integrative genomics. `http://www.broad.mit.edu/cancer/software/genepattern`.

[55] GILSON, O., SILVA, N., GRANT, P., CHEN, M., AND ROCHA, J. VizThis: Rule-based semantically assisted information visualization. Poster, in Proceedings of SWUI 2006, 2006.

[56] git. `http://git-scm.com`.

[57] Google Suggest. `http://www.google.com/support/web-search/bin/answer.py?hl=en\&answer=106230`.

[58] GRAHAM, M., AND KENNEDY, J. Exploring multiple trees through dag representations. *IEEE Transactions on Visualization and Computer Graphics 13* (November 2007), 1294–1301.

[59] GRAHAM, M., AND KENNEDY, J. B. A survey of multiple tree visualisation. *Information Visualization 9*, 4 (2010), 235–252.

[60] HASAN, R., SION, R., AND WINSLETT, M. The case of the fake picasso: preventing history forgery with secure provenance. In *FAST '09: Proccedings of the 7th conference on File and storage technologies* (2009), pp. 1–14.

[61] HAYS, J., AND EFROS, A. A. Scene completion using millions of photographs. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 26*, 3 (2007).

[62] HE, H., AND SINGH, A. K. Closure-tree: An index structure for graph queries. In *ICDE* (2006), p. 38.

[63] HEER, J., VIÉGAS, F. B., AND WATTENBERG, M. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)* (2007), pp. 1029–1038.

[64] HEILAND, R., SWAT, M., ZAITLEN, B., GLAZIER, J., AND LUMSDALE, A. Workflows for parameter studies of multi-cell modeling. In *Proceedings of the ACM High Performance Computing Symposium* (2010), pp. 140–145.

[65] HEINIS, T., AND ALONSO, G. Efficient lineage tracking for scientific workflows. In *SIGMOD* (2008), pp. 1007–1018.

[66] HERMAN, I., MELANCON, G., AND MARSHALL, M. Graph visualization and navigation in information visualization: A survey. *Visualization and Computer Graphics, IEEE Transactions on 6*, 1 (2000), 24–43.

[67] HEYMANS, M., AND SINGH, A. K. Deriving phylogenetic trees from the similarity analysis of metabolic pathways. In *ISMB (Supplement of Bioinformatics)* (2003), pp. 138–146.

[68] HIRSH, H., BASU, C., AND DAVISON, B. D. Learning to personalize. *Communications of ACM 43*, 8 (2000), 102–106.

[69] HOLTEN, D. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics 12* (September 2006), 741–748.

[70] HOWE, B., LAWSON, P., BELLINGER, R., ANDERSON, E., SANTOS, E., FREIRE, J., SCHEIDEGGER, C., BAPTISTA, A., AND SILVA, C. End-to-end escience: Integrating workflow, query, visualization, and provenance at an ocean observatory. In *IEEE International Conference on eScience* (2008), pp. 127–134.

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

[72] IGARASHI, T., AND HUGHES, J. F. A suggestive interface for 3d drawing. In *ACM symposium on User interface software and technology (UIST)* (2001), pp. 173–181.

[73] ISENBERG, P., AND CARPENDALE, S. Interactive tree comparison for co-located collaborative information visualization. *IEEE Transactions on Visualization and Computer Graphics 13* (November 2007), 1232–1239.

[74] JAHN, K., AND KRAMER, S. Optimizing gspan for molecular datasets. In *Proc. Third Int. Workshop on Mining Graphs, Trees and Sequences* (2005).

[75] JÄNICKE, S., HEINE, C., HELLMUTH, M., STADLER, P., AND SCHEUERMANN, G. Visualization of graph products. *Visualization and Computer Graphics, IEEE Transactions on 16*, 6 (nov.-dec. 2010), 1082–1089.

[76] JANKUN-KELLY, T. J., KREYLOS, O., MA, K.-L., HAMANN, B., JOY, K. I., SHALF, J. M., AND BETHEL, E. W. Deploying web-based visual exploration tools on the grid. *IEEE Computer Graphics and Applications 23*, 2 (2003), 40–50.

[77] JANKUN-KELLY, T. J., AND MA, K.-L. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics 7*, 3 (July/September 2001), 275–287.

[78] JANKUN-KELLY, T. J., MA, K.-L., AND GERTZ, M. A model and framework for visualization exploration. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (March/April 2007), 357–369.

[79] KAMINSKI, P., LITOIU, M., AND MÜLLER, H. A design technique for evolving web services. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (New York, NY, USA, 2006), ACM, p. 23.

[80] KANEHISA, M., AND GOTO, S. KEGG: Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Research 28*, 1 (Jan. 2000), 27–30.

[81] The Kepler Project. `http://kepler-project.org`.

[82] KITWARE. Paraview. http://www.paraview.org.

[83] KITWARE. VTK. `http://www.vtk.org`.

[84] KOOP, D., SANTOS, E., BELA BAUER, MATTHIAS TROYER, J. F., AND SILVA, C. T. Bridging workflow and data provenance using strong links. In *Scientific and Statistical Database Management*, M. Gertz and B. Ludäscher, Eds., vol. 6187 of *LNCS*. Springer Berlin / Heidelberg, Jul 1, 2010, pp. 397–415.

[85] KOOP, D., SCHEIDEGGER, C., CALLAHAN, S., FREIRE, J., AND SILVA, C. VisComplete: Automating suggestions for visualization pipelines. *IEEE TVCG 14*, 6 (2008), 1691–1698.

[86] KOOP, D., SCHEIDEGGER, C., FREIRE, J., AND SILVA, C. T. The provenance of workflow upgrades. In *Provenance and Annotation of Data and Processes*, D. McGuinness, J. Michaelis, and L. Moreau, Eds., vol. 6378 of *LNCS*. Springer Berlin / Heidelberg, Jun 15, 2010, pp. 2–16.

[87] KORVEMAKER, B., AND GREINER, R. 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* (2000), pp. 230–235.

[88] KUHN, H. W. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly 2*, 1-2 (1955), 83–97.

[89] KURAMOCHI, M., AND KARYPIS, G. Frequent subgraph discovery. In *ICDM* (2001), pp. 313–320.

[90] LALONDE, J.-F., HOIEM, D., EFROS, A. A., ROTHER, C., WINN, J., AND CRIMINISI, A. Photo clip art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 26*, 3 (2007).

[91] LAWRENCE LIVERMORE NATIONAL LABORATORY. VisIt: Visualize It in Parallel Visualization Application. `https://wci.llnl.gov/codes/visit` [29 March 2008].

[92] LEE, E. A., AND PARKS, T. M. Dataflow Process Networks. *Proceedings of the IEEE 83*, 5 (1995), 773–801.

[93] LINS, L., KOOP, D., ANDERSON, E., CALLAHAN, S., SANTOS, E., SCHEIDEGGER, C., FREIRE, J., AND SILVA, C. Examining statistics of workflow evolution provenance: a first study. In *Proceedings of SSDBM* (2008).

[94] LU, J., LING, T. W., CHAN, C.-Y., AND CHEN, T. From region encoding to extended dewey: on efficient processing of xml twig pattern matching. In *VLDB* (2005), pp. 193–204.

[95] LUDASCHER, B., ALTINTAS, I., BERKLEY, C., HIGGINS, D., JAEGER, E., JONES, M., LEE, E. A., TAO, J., AND ZHAO, Y. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper. 18*, 10 (2006), 1039–1065.

[96] MA, K.-L. Visualizing visualizations: User interfaces for managing and exploring scientific visualization data. *IEEE Comput. Graph. Appl. 20*, 5 (2000), 16–19.

[97] MARKS, J., ANDALMAN, B., BEARDSLEY, P. A., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., RYALL, W., SEIMS, J., AND SHIEBER, S. Design galleries: A general approach to setting parameters for computer graphics and animation. In *ACM SIGGRAPH* (1997), pp. 389–400.

[98] MCCAMANT, S., AND ERNST, M. D. Predicting problems caused by component upgrades. In *ESEC* (2003), pp. 287–296.

[99] MCCAMANT, S., AND ERNST, M. D. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference* (Oslo, Norway, June 16–18, 2004), pp. 440–464.

[100] MELNIK, S., GARCIA-MOLINA, H., AND RAHM, E. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering* (2002), pp. 117–128.

[101] MERCURY COMPUTER SYSTEMS. Amira. `http://www.amiravis.com`.

[102] MEVIS RESEARCH. MeVisLab. `http://www.mevislab.de`.

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

[104] Microsoft Workflow Foundation. `http://msdn2.microsoft.com/en-us/netframework/aa663328.aspx`.

[105] MOREAU, L., FREIRE, J., FUTRELLE, J., MCGRATH, R. E., MYERS, J., AND PAULSON, P. The open provenance model: An overview. In *IPAW* (2008), pp. 323–326.

[106] MOUALLEM, P., BARRETO, R., KLASKY, S., PODHORSZKI, N., AND VOUK, M. Tracking files in the kepler provenance framework. In *SSDBM 2009: Proceedings of the 21st International Conference on Scientific and Statistical Database Management* (2009), pp. 273–282.

[107] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), USENIX Association, pp. 4–4.

[108] MUNZNER, T., GUIMBRETIÈRE, F., TASIRAN, S., ZHANG, L., AND ZHOU, Y. Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility. *ACM Trans. Graph. 22* (July 2003), 453–462.

[109] MUNZNER, T., JOHNSON, C., MOORHEAD, R., PFISTER, H., RHEINGANS, P., AND YOO, T. S. NIH-NSF visualization research challenges report summary. *IEEE Computer Graphics and Applications 26*, 2 (2006), 20–24.

[110] myexperiment. `http://www.myexperiment.org`.

[111] NAG. Iris Explorer. `http://www.nag.co.uk/welcome\_iec.asp`.

[112] NANOPOULOS, A., KATSAROS, D., AND MANOLOPOULOS, Y. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering 15*, 5 (Sept.-Oct. 2003), 1155–1169.

[113] NORMAN, D. A. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Addison Wesley, 1994.

[114] PARKER, S. G., AND JOHNSON, C. R. SCIRun: a scientific programming environment for computational steering. In *Supercomputing* (1995).

[115] PARNAS, D. L. Software aging. In *ICSE* (1994), pp. 279–287.

[116] PASKIN, N. Digital object identifiers for scientific data. *Data Science Journal 4* (2005), 12–20.

[117] The Pegasus Project. `http://pegasus.isi.edu/`.

[118] PENG, R. S., AND ECKEL, S. P. Distributed reproducible research using cached computations. *Computing in Science & Engineering 11*, 1 (2009), 28–34.

[119] PLAISANT, C., FEKETE, J.-D., AND GRINSTEIN, G. Promoting insight-based evaluation of visualizations: From contest to benchmark repository. *Visualization and Computer Graphics, IEEE Transactions on 14*, 1 (2008), 120 –134.

[120] PLALE, B., ALAMEDA, J., WILHELMSON, B., GANNON, D., HAMPTON, S., ROSSI, A., AND DROEGEMEIER, K. Active management of scientific data. *IEEE Internet Computing 9*, 1 (2005), 27–34.

[121] First provenance challenge. `http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge`, 2006.

[122] Second provenance challenge. `http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge`, 2007.

[123] Third provenance challenge. `http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge`, 2008.

[124] RIESEN, K., AND BUNKE, H. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput. 27* (June 2009), 950–959.

[125] SALAMONE, S. Lsid: An informatics lifesaver. *Bio-ITWorld* (2004).

[126] SANTOS, E., FREIRE, J., AND SILVA, C. Information Sharing in Science 2.0: Challenges and Opportunities. CHI Workshop on The Changing Face of Digital Science: New Practices in Scientific Collaborations, 2009.

[127] SANTOS, E., KOOP, D., VO, H. T., ANDERSON, E. W., FREIRE, J., AND SILVA, C. T. Using workflow medleys to streamline exploratory tasks. In *SSDBM* (2009), pp. 292–301.

[128] SANTOS, E., LINS, L., AHRENS, J., FREIRE, J., AND SILVA, C. T. Vismashup: Streamlining the creation of custom visualization applications. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1539–1546.

[129] SCHEIDEGGER, C. E., KOOP, D., SANTOS, E., VO, H. T., CALLAHAN, S. P., FREIRE, J., AND SILVA, C. T. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience 20*, 5 (2008), 473–483.

[130] SCHEIDEGGER, C. E., VO, H. T., KOOP, D., FREIRE, J., AND SILVA, C. T. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of Visualization) 13*, 6 (2007), 1560–1567.

[131] SCHEIDEGGER, C. E., VO, H. T., KOOP, D., FREIRE, J., AND SILVA, C. T. Querying and re-using workflows with vstrails. In *SIGMOD* (2008), pp. 1251–1254.

[132] SENAY, H., AND IGNATIUS, E. A knowledge-based system for visualization design. *IEEE Comp. Graph. and Appl. 14*, 6 (1994).

[133] SHASHA, D., WANG, J. T. L., AND GIUGNO, R. Algorithmics and applications of tree and graph searching. In *PODS* (2002), pp. 39–52.

[134] SHOSHANI, A., SIM, A., AND GU, J. *Storage resource managers: essential components for the Grid*. Kluwer Academic Publishers, 2004, pp. 321–340.

[135] SILVA, C., FREIRE, J., AND CALLAHAN, S. P. Provenance for visualizations: Reproducibility and beyond. *IEEE Computing in Science & Engineering* (2007). To appear.

[136] SIMMHAN, Y., BARGA, R., VAN INGEN, C., LAZOWSKA, E., AND SZALAY, A. Building the trident scientific workflow workbench for data management in the cloud. In *International Conference on Advanced Engineering Computing and Applications in Sciences* (2009), pp. 41–50.

[137] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *SIGMOD Rec. 34*, 3 (2005), 31–36.

[138] SIMMHAN, Y. L., PLALE, B., GANNON, D., AND MARRU, S. Performance evaluation of the karma provenance framework for scientific workflows. In *International Provenance and Annotation Workshop (IPAW), Chicago, IL* (2006), vol. 4145 of *Lecture Notes in Computer Science*, pp. 222–236.

[139] STUCKENHOLZ, A. Component evolution and versioning state of the art. *SIGSOFT Softw. Eng. Notes 30*, 1 (2005), 7.

[140] Subversion. `http://subversion.tigris.org`.

[141] Swivel. `http://www.swivel.com`.

[142] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[143] TAKAHASHI, S., FIJISHIRO, I., TAKESHIMA, Y., AND NISHITA, T. A feature-driven approach to locating optimal viewpoints for volume visualization. In *IEEE Visualization* (2005), pp. 495–502.

[144] TAN, W. C. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull. 30*, 4 (2007), 3–12.

[145] The Taverna Project. `http://www.taverna.org.uk`.

[146] TODO, S., AND KATO, K. Cluster algorithms for general- s quantum spin systems. *Phys. Rev. Lett. 87*, 4 (Jul 2001), 047203.

[147] TOHLINE, J. E., GE, J., EVEN, W., AND ANDERSON, E. A customized python module for cfd flow analysis within vistrails. *Computing in Science and Engineering 11*, 3 (2009), 68–73.

[148] TREIBER, M., JUSZCZYK, L., SCHALL, D., AND DUSTDAR, S. Programming evolvable web services. In *PESOS '10: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems* (New York, NY, USA, 2010), ACM, pp. 43–49.

[149] The Triana Project. `http://www.trianacode.org`.

[150] TROYER, M., TSUNETSUGU, H., AND WÜRTZ, D. Thermodynamics and spin gap of the heisenberg ladder calculated by the look-ahead lanczos algorithm. *Phys. Rev. B 50*, 18 (Nov 1994), 13515–13527.

[151] TSANG, S., BALAKRISHNAN, R., SINGH, K., AND RANJAN, A. A suggestive interface for image guided 3d sketching. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI)* (2004), pp. 591–598.

[152] TU, Y., AND SHEN, H.-W. Visualizing changes of hierarchical data using treemaps. *IEEE Transactions on Visualization and Computer Graphics 13* (November 2007), 1286–1293.

[153] UNIVERSITY OF UTAH. VisTrails. `http://www.vistrails.org`.

[154] UPSON ET AL, C. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications 9*, 4 (1989), 30–42.

[155] VAN HAM, F., AND VAN WIJK, J. Interactive visualization of small world graphs. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on* (2004), pp. 199–206.

[156] VDS - The GriPhyN Virtual Data System. `http://www.ci.uchicago.edu/wiki/bin/view/VDS/VDSWeb/WebMain`.

[157] VIEGAS, F. B., WATTENBERG, M., VAN HAM, F., KRISS, J., AND MCKEON, M. ManyEyes: a site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of InfoVis) 13*, 6 (2007), 1121–1128.

[158] VORDOLOI, U. D., AND SHEN, H.-W. View selection for volume rendering. In *IEEE Visualization* (2005), pp. 487–494.

[159] WANG, H., PARK, S., FAN, W., AND YU, P. S. Vist: a dynamic index method for querying xml data by tree structures. In *SIGMOD* (2003), pp. 110–121.

[160] WATTENBERG, M. Visual exploration of multivariate graphs. In *Proceedings of the SIGCHI conference on Human Factors in computing systems* (New York, NY, USA, 2006), CHI '06, ACM, pp. 811–819.

[161] WEBB, E. C., AND INTERNATIONAL UNION OF BIOCHEMISTRY AND MOLECULAR BIOLOGY. *Enzyme nomenclature 1992. Recommendations of the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology on the Nomenclature and Classification of Enzymes*. Academic Press, 1992.

[162] WILLIAMS, D., HUAN, J., AND WANG, W. Graph database indexing using structured graph decomposition. *ICDE* (April 2007), 976–985.

[163] Yahoo! Pipes. `http://pipes.yahoo.com`.

[164] YAN, X., AND HAN, J. gspan: Graph-based substructure pattern mining. In *ICDM* (2002), p. 721.

[165] YAN, X., AND HAN, J. Closegraph: mining closed frequent graph patterns. In *KDD* (2003), pp. 286–295.

[166] YAN, X., YU, P. S., AND HAN, J. Graph indexing: a frequent structure-based approach. In *SIGMOD* (2004), pp. 335–346.

[167] ZENG, Z., TUNG, A. K. H., WANG, J., FENG, J., AND ZHOU, L. Comparing stars: on approximating graph edit distance. *Proc. VLDB Endow. 2* (August 2009), 25–36.

[168] ZHAO, P., YU, J. X., AND YU, P. S. Graph indexing: Tree + delta $\geqslant$ graph. In *VLDB* (2007), pp. 938–949.