

REFLECTIONS, MODELS, AND SOFTWARE FOR ITERATIVE VISUALIZATION DESIGN

by
Alex Bigelow

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

Doctor of Philosophy
in
Computing

School of Computing
The University of Utah
August 2019

Copyright © Alex Bigelow 2019

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Alex Bigelow
has been approved by the following supervisory committee members:

Miriah Meyer , Chair(s) 5 June 2019
Date Approved

Alexander Lex , Member 16 May 2019
Date Approved

Megan Monroe , Member
Date Approved

James A. Agutter , Member 16 May 2019
Date Approved

Tamara Denning , Member 16 May 2019
Date Approved

by Ross T. Whitaker , Chair/Dean of
the Department/College/School of Computing
and by David B. Kieda , Dean of The Graduate School.

ABSTRACT

Iteration is a ubiquitous need across many different visualization design workflows. For visual design, practitioners often need to transition between the automated generation of visuals and manual drawing; however, existing software makes free-form iteration between these modalities difficult or impossible. A similar challenge exists for data as well: data abstractions often need to be adjusted; however, many data reshaping operations—particularly graph wrangling operations—have support only in programming environments that may not be accessible to many visualization designers.

This dissertation begins with an effort to better understand how graphic designers, a specific subset of the broader visualization community, work with data. The lessons from that effort inspire our remaining contributions, which are directed toward the broader community of visualization practitioners: first, we present a software model and a system that make it possible to iterate between drawing and generative visualization. Second, we present a visual technique that enables inquiry into the relationship between two specific graph data wrangling operations: pivoting and filtering. Finally, we present an interface that supports a broader range of graph data wrangling operations that enable iteration between many different graph data abstractions.

CONTENTS

ABSTRACT	iii
LIST OF TABLES	vi
ACKNOWLEDGEMENTS	vii
CHAPTERS	
1. INTRODUCTION	1
1.1 Constraint on Rapid Prototyping	3
1.2 Designing and Refining Data Abstractions.....	4
1.3 Contributions	4
2. BACKGROUND AND RELATED WORK	7
2.1 Rapid Prototyping Challenges.....	9
2.2 Graph Data Wrangling Challenges	11
3. REFLECTIONS ON HOW DESIGNERS DESIGN WITH DATA	16
3.1 Motivation.....	16
3.2 Methods and Participants	17
3.3 Patterns	19
3.4 Themes	26
3.5 Opportunities and Next Steps	30
4. ITERATING BETWEEN TOOLS TO CREATE AND EDIT VISUALIZATIONS	32
4.1 Motivation.....	32
4.2 The Challenge of Iteration	33
4.3 The Bridge Model	35
4.4 Hanpuku: An Example of a Bridge	45
4.5 Other Uses of the Model	49
4.6 Discussion and Future Work	54
5. JACOB'S LADDER: THE USER IMPLICATIONS OF LEVERAGING GRAPH PIVOTS	56
5.1 Motivation.....	57
5.2 The Pivot	58
5.3 Why the Pivot?	59
5.4 Why Not the Pivot?	61
5.5 Evaluating Pivots	63
5.6 Discussion	71
5.7 Conclusions and Future Work	78

6. ORIGRAPH: INTERACTIVE NETWORK WRANGLING	80
6.1 Motivation	80
6.2 Usage Scenario	83
6.3 Terminology	84
6.4 Operations	84
6.5 Origraph Design	90
6.6 Implementation	97
6.7 Input and Output	98
6.8 Use Cases	99
6.9 Discussion	121
6.10 Conclusions and Future Work	124
7. EXTENSIONS AND FUTURE WORK	126
7.1 Software Gaps	126
7.2 Interaction Modality Gaps	127
7.3 Workflow Gaps	128
REFERENCES	130

LIST OF TABLES

2.1	Operations supported by Origraph, Orion [1], and Ploceus [2].	15
3.1	Skill levels of each designer who participated in interviews, lab studies, and the hackathon	18
3.2	Instances in which we recorded designers saying or doing something specific that supports an emergent pattern	20
3.3	Summary of relationships between observed patterns and emergent themes ..	27
5.1	Relative expertise and suggestive indicators of the 11 participants in the Jacob's Ladder study	67

ACKNOWLEDGEMENTS

This dissertation would not be possible without the contributions of many people, first and foremost, my advisor Miriah Meyer. I was largely blind to the possibility of a research career until she opened the door. She gave ample room, encouragement, and guidance to explore broad questions about people and software. She has patiently tolerated my impulsive dives into the weeds, and masterfully helped me articulate ideas that I struggled to communicate.

My coauthors have shared in many of these struggles, and I am deeply appreciative of their collective patience. In addition to Miriah: Nicola Camp, Danyel Fisher, Steven Drucker, Megan Monroe, Alex Lex, and Carolina Nobre have all patiently endured implementation detail dumps and chaotic paper deadlines. I also wish to thank Jim Agutter and Tammy Denning for helping me fill gaps outside my training, and Jeff Baumes and Roni Choudhury for significantly improving my abilities to develop software. Each of these people has inspired and guided my research, and I appreciate that none have pulled their punches when difficult questions needed to be asked.

I also owe a great deal to labmates and other friends at the University of Utah for pulling me through the disasters that resulted from applying academic rigor a little too introspectively. Much more than my research was at stake—I would not have finished without the friendship and support of Carolina Nobre, Ethan Kerzner, Lace Padilla, Sam Quinan, Jimmy Moore, Nina McCurdy, Sean McKenna, Pascal Goffin, Cameron Waller, Magdalena Schwarzl, and too many others to list.

This research also benefited from the world class directors, staff, and facilities at the SCI Institute—its culture, technical support, writing support, and the opportunity work with BruSCI all helped to keep me productive and sane while working at Utah. This dissertation also benefited from similar cultures and support at IBM Research, Kitware, and the University of Arizona.

This work is indebted to our graphic designers and study participants for their time

and insights, as well as ACM [3], IEEE [4]–[6], Grantland [7], Rachel Mercer [8], and Isabel Meirelles [9] for their permission to include and/or reproduce their material as part of this dissertation.

Finally, I wish to thank the many mentors that I have had through life—including teachers, employers, scouting and associated leaders, and family for encouraging my computing hobbies. I especially want to thank my parents for teaching me to prioritize science, education, studying, thinking, and intellectual honesty.

CHAPTER 1

INTRODUCTION

This dissertation addresses two practical challenges in data visualization authoring workflows through novel tools for rapid prototyping and for designing new data abstractions. The first challenge that we address is the inability to iterate between drawing tools and automated visualization toolkits—which constrains rapid prototyping of custom visualizations. The second challenge is that interactive software for designing and refining graph data abstractions does not yet exist. Our efforts are grounded in formative work, interviewing and observing how graphic designers work with data; reflections on our experiences show that each challenge is common to visualization practitioners with very diverse skill sets, and they also reveal new strategies that software can use to address each challenge. These strategies include a model for bridging drawing tools and automated visualization toolkits. We demonstrate its effectiveness through an Adobe Illustrator plugin. We also present novel, interactive systems that facilitate designing and refining graph data abstractions. Both approaches represent new ways to support more expressive visualization authoring.

Our work was initially motivated through informal discussions with graphic designers, as well as surprising online examples of custom visualizations. The examples that we found contained layout inconsistencies or errors that would not have been produced by a generative system, such as those in Figure 1.1 containing an out-of-order item, and Figure 1.2 containing inaccurate encodings of the data. These examples suggest that the designers followed a process for creating visualizations that was very distinct from our own workflows as visualization programmers.

To better identify and understand the differences, we conducted interviews with graphic designers and observed how they work with data in a natural context as well as controlled lab studies [3]. Reflections on the interviews and observations revealed

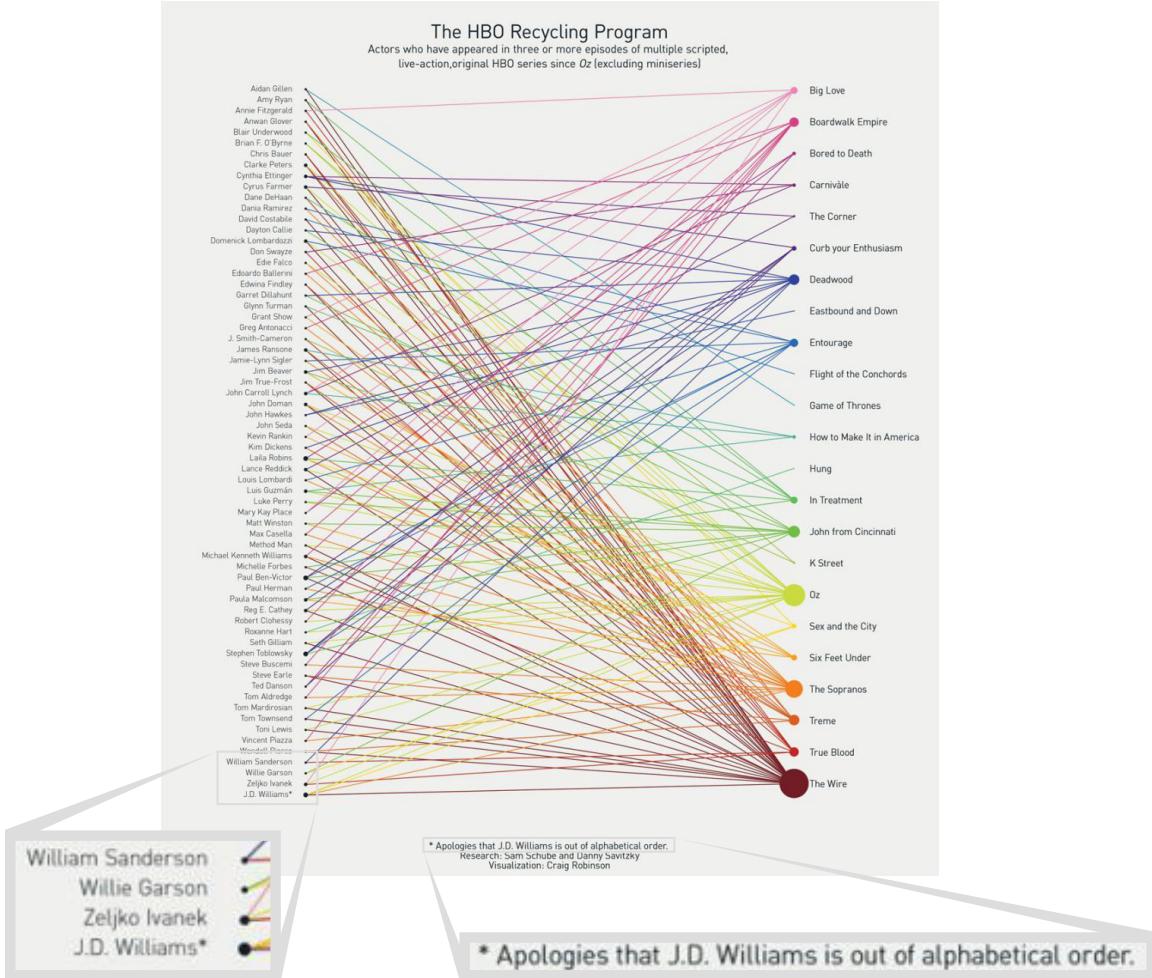


Figure 1.1. The HBO Recycling Program infographic [7] lists actors, actresses, and TV shows alphabetically, with a line indicating that a person acted in a particular show. Note that one actor, J.D. Williams, is listed out of order.



Figure 1.2. An example of an infographic where the visual representations do not explicitly match the data. The data shown in the text does not match the percentages encoded in the pie charts [8].

that visualization programmers and graphic designers share two major challenges in visualization design. The first is that it is difficult to iterate between drawing software and automated visualization toolkits, which creates a barrier to rapid prototyping. The second is that the lack of software support for designing and refining data abstractions results in reduced variation, creativity, and quality of visualization designs.

1.1 Constraint on Rapid Prototyping

One pattern that we saw was exporting the results of an automated visualization toolkit to a drawing program. Although this approach combines the advantages of each modality, it has a critical weakness: once the visualization has been exported to a drawing program, it becomes impossible to iterate on an automated aspect of the visualization. In the current software ecosystem, going from automation to drawing is a one-way journey.

This problem is common to the visualization community as a whole, especially in the context of rapid prototyping. Visualization practitioners frequently make sketches and static mock-ups with drawing tools, but they have very few ways, if any, to test the interactivity of an interface, or the behavior of real data, without starting over from scratch with code. The incompatibility of these types of tools, therefore, presents a barrier to iterating between each modality in the rapid prototyping process.

To overcome this practical challenge, we present a model for bridging drawing tools and automated visualization toolkits [4]. The key insight of the model is that it converts the serial iteration problem into a parallel merge problem—this way, each tool can keep its internal representation of a visualization, and use all of its features, without needing to account for the internal representation or features of another program. The model identifies key areas that a bridge designer should consider: how to identify changes in each tool, how to merge changes from each tool, how to capture designer intent, and how to reintegrate changes across tools. We demonstrate the kind of expressive workflows that the model enables through a proof-of-concept Adobe Illustrator extension, enabling two-way iteration between Illustrator and d3.js visualizations.

1.2 Designing and Refining Data Abstractions

Another major need that designers have in common with the larger visualization community is the ability to design and refine data abstractions. In some cases, we saw that accepting the given structure and interpretation of data—without any exploration or reinterpretation—can severely constrain the visual designs that are considered. In contrast, we learned that when designers explored or reshaped data, encountering unexpected data behavior could inspire novel, creative visual designs.

We see both effects in the visualization community: the relationship between choosing an effective data abstraction and the quality and effectiveness of the resulting visualization is well documented [10]. However, few software tools exist that assist in this process—and the few that do focus exclusively on tabular data and/or impose arbitrary interpretations that are based on the initial data structure. Consequently, alternative data abstractions are difficult to explore in practice, and data behavior that could inspire or break a visual design often remains hidden until it is too late.

We present a visual technique for extracting subgraphs as formative work [5], followed by a novel interactive system for reinterpreting graph data [6]. Most graph databases [11]–[13], and the few network modeling tools that exist [1], [2], [14] are implemented with a fixed interpretation layer on top of a relational model that defines what a node is, what an edge is, and what an attribute is. This approach results in inflexible data abstractions—users have no way to reassign what a node is, what an edge is, and what an attribute is. We present an interactive system that allows a user to flexibly specify and modify that interpretation layer, in addition to other data wrangling operations. We demonstrate the expressiveness that this approach enables, through examples employing a rich set of operations, including connection, converting between nodes and edges, edge projection, attribute promotion, faceting, direct and connectivity-based filtering, changing edge direction, and in-class and connectivity-based attribute derivation.

1.3 Contributions

The contributions of this dissertation are the following:

- Reflections on observations and interviews of designers [3] that highlight the need for tools that support iterative visualization design between automation and draw-

ing, as well as the need for tools that support designing alternative data abstractions

- A model [4] for bridging between drawing tools and automated visualization toolkits, as well as an instantiation of the model, showing how tools can support iteration between each modality
- A novel system for designing and refining graph data abstractions [6] that is informed by a formative graph exploration technique [5] and that supports exploring alternative graph data abstractions

Each contribution has a distinct focus, shown in Figure 1.3. We begin in Chapter 2 by identifying gaps in the existing literature with respect to understanding the challenges that designers face when working with data. We then identify gaps in existing software with respect to support for iteration in rapid prototyping, as well as support for iteration in graph data wrangling. In Chapter 3, we detail interviews and observations of designers and reflect on how they inform the development of future software. Here, our main objective is to learn from graphic designers.

In Chapter 4, we present a model and tool that enable iterative visual design, focusing on first steps toward filling missing software capabilities that are needed in order to

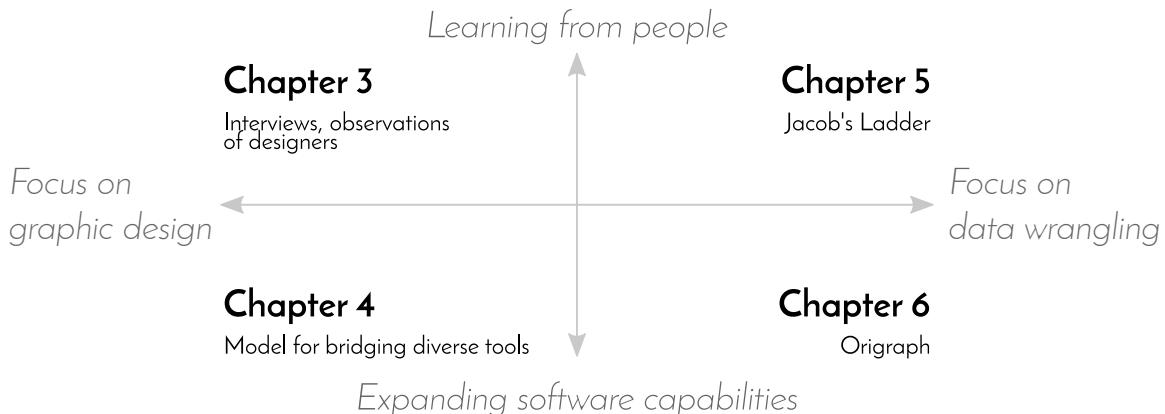


Figure 1.3. Each contribution has a different focus. In Chapter 3, the interviews and observations focus on learning from graphic designers. In Chapter 4, the model for bridging diverse tools focuses on expanding software capabilities, in a way that could eventually lead to improved tools for graphic designers creating visualizations. Jacob’s Ladder, the interface presented in Chapter 5, is evaluated with a more general set of users wrangling network data, that informs Origraph, a more powerful network wrangling system in Chapter 6.

enable iterative workflows. Chapter 5 evaluates Jacob’s Ladder, a formative technique that probes the relationship between two graph wrangling operations. The main objective in evaluating Jacob’s Ladder is to understand whether and how general users, not just graphic designers, can make use of a simple network wrangling operation. Based on these lessons, in Chapter 6, we expand network wrangling capabilities with a tool that implements a set of visual techniques that enable a broader set of operations for iterative graph data wrangling. Together, our interviews, observations, models, visual techniques, and software systems fill major gaps in the existing literature and software capabilities, and lay the groundwork for future exploration into extended practical challenges, implications for creative software design, and understanding how creative visualization practitioners think and work. We discuss these extensions and future work in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

The design and development of tools for creating visual representations of data has been a topic of interest in the computer science community for many years. In designing these tools, computer scientists have long battled the conflicting requirements of ease of use and flexibility [15]. Despite the wealth of tools that have been proposed and developed, we suspected that many people creating infographics today do not use most, if any, of these tools. Our suspicions arose from two sources. First, we found a number of infographics on the web that would not have been generated by an automated system that ties visuals closely to the data, such as the examples in Figure 1.1 and Figure 1.2. These observations match what others have reported [16]. Second, we engaged in informal conversations with design professionals and students who indicated to us that they largely plot data manually in a design tool such as Adobe Illustrator. These suspicions led us to explore how designers create visualizations and what challenges they have, as well as to probe the disconnect between how designers design with data and the tools programmers create for them to do so.

To better understand this underserved segment of the visualization practitioner community, we conduct interviews and observations of practitioners with a background in graphic design, and report our findings in Chapter 3. Our analysis methods are inspired by the long history in the human-computer interaction community for examining design practice and building tools to support a wide variety of contexts. For example, the Designer's Outpost [17] is an environment designed to support web designers; it is based on a previous study examining design practice around websites [18]. Grammel *et al.* examine how visualization novices use visualization systems and programming environments [19]. In Chapter 3, however, we explore limitations of these models in a design context. Another line of related work is that of Walny *et al.* [20], which looks at sketching on whiteboards

as a design practice. These diagrams are often representations of systems or interactions, and are not necessarily related to data; the charts tend to be highly abstract. In this work we specifically look at how designs work directly with data to create concrete charts and graphs.

Visually, one of the most fundamental goals in graphic design—whether or not data are involved—is to establish a hierarchy that matches the underlying message that a design needs to communicate [21]. What is the primary thing a viewer needs to see? Secondary? Tertiary? Similarly, the academic information visualization community tends to discuss visual hierarchy in terms of task and data abstractions [10] that map to encoding channel effectiveness [22] to show users what they most need to see. What aspect of the data should be encoded spatially? With length? With color?

Establishing an effective visual hierarchy for a data visualization can be especially challenging. Data abstractions themselves need to be designed [23], and the abstraction layer can be validated only after at least a prototype visualization is designed, implemented, and tested [10]. Stakeholders' needs and priorities are often challenging to understand and articulate, and the act of visualizing data itself can change what those needs and priorities are [24], [25].

In practice, therefore, underlying data and task abstractions are prone to frequent revisions and refinements over the course of designing a visualization—and abstraction changes can cascade to visual hierarchy changes. For example, identifying that specific biologists have a primary need to perform detailed comparisons of gene expression over time—a task abstraction refinement—can inspire a new visual hierarchy [26] that prioritizes expression levels with more salient, spatial line charts, instead of using traditional color heatmaps to encode gene expression. Similarly, recognizing that bioinformaticians intuitively understand sequencing contig overlaps as nodes, rather than edges—a data abstraction refinement—can lead to groundbreaking research into novel ways to visually encode edge attributes [27].

However, current visualization toolkits and related software provide very little, if any, support for these kinds of dramatic shifts neither at the abstraction level, nor at the visual encoding level. In both cases, practical limits on the ability to explore many varied designs [28]–[31]—with respect to either the visual hierarchy, or with respect to the un-

derlying abstraction—increase vulnerability to the threats of ineffective encoding choices, and/or abstractions that do not match user needs [10] and can result in premature fixation on a poor design [25].

Consequently, visualization practitioners need better tools that do not force them to start from scratch every time they consider an abstraction refinement or a visual encoding change. For visual encoding changes, we build on the rich history of rapid prototyping software, and focus on challenges that prevent iterating between different tool modalities, particularly in the context of graphic design. At the abstraction level, we focus on graph data wrangling challenges, as little software exists for enabling users to think about alternative network models, and no software exists that supports iterating on them.

2.1 Rapid Prototyping Challenges

Practitioners need to test whether and how a visualization design fails as quickly as possible [32], in order to focus effort on the most effective solutions. To explore the space of possible solutions, they can choose from a variety of tools, from paper [33], [34] to tangible blocks [35]. For rapid prototyping in a digital context, however, the available modalities are relatively constrained: designers can choose from generative tools that facilitate encoding many data points quickly; drawing tools that facilitate more expressive, custom designs; hybrids tools that compromise between the strengths of each modality; or multiple tools used in the same workflow.

Generative tools can help with rapid prototyping because they can efficiently render visualizations almost instantly once they have been configured. Tools like Microsoft Excel are relatively easy to configure for standard chart types; however, standard chart types in isolation may have limited use in rapid prototyping. More customization can be achieved with more complex tools. Tools such as Tableau [36] or Plotly [37] give access to more chart types and more style controls, but the amount of configuration required increases with expressive power. At the extreme end of this spectrum are programming environments such as D3 [38], Processing [39], or VTK [40] that provide extensive expressivity, at the expense of needing to spend time learning and writing code.

Alternatively, practitioners can turn to **drawing tools**, such as Adobe Illustrator, Sketch, or Inkscape. These tools give visualization designers much more flexibility to create any

visuals that they wish, with any style; however, creating a data visualization in a drawing program can be extremely time consuming, as some encodings may need to be implemented manually, one point at a time.

Efforts to make **hybrid tools** that combine the strengths of both modalities, from early efforts like SageBrush [41] and RBE [42], to more recent approaches like Lyra [43], iVisDesigner [44], SketchInsight [45], iVolVER [46], or Data Illustrator [47], could also be useful in the rapid prototyping process. However, user studies of these kinds of hybrid interfaces have consistently shown [44],[46],[47] that they suffer from the same fundamental tradeoff that purely generative toolkits have: greater expressive power comes at the cost of a longer time investment to learn and configure the interface.

A fourth alternative is to use **multiple tools** in the same workflow, and our research focuses on this approach. Some generative tools like RAWGraphs [48] are explicitly designed with this kind of workflow in mind—they are generative tools that export directly to drawing programs. Other approaches, like D3 Deconstructor [49], support extracting components of interactive visualizations for reuse in other visualization toolkits. Migrating a visualization from one tool to another has the advantage that the full capabilities of each tool can be used when needed, such as plotting many points with a generative tool, and then customizing the style or drawing unsupported encodings with a drawing tool.

A major challenge exists, however, in that there is no way to reimport a visualization from a drawing tool back into a generative environment—exporting to a drawing tool is a one-way journey. This is especially problematic for exploring alternative visual encodings, because any custom work done in a drawing tool must be discarded and redrawn if an automated aspect of the visualization needs to be changed.

To address this problem, the bridge model that we present in Chapter 4 depends on the idea of bringing together distinct sets of modifications from multiple sources, making it similar to both revision control systems [50] and collaborative editing systems. Each of these has mechanisms for merging changes made by disparate practitioners. Revision control concepts have been applied to design contexts, such as CAD tools [51] and collaborative editing of 3D models [52], [53]. One important difference is that the bridge model focuses on iteration between different classes of tools, rather than different people using the same tool.

In Chapter 4, we show how this approach overcomes practical constraints in multitool workflows that arise from the differences between generative and drawing modalities. With this model, a broad range of visual refinements become possible, including fundamental changes to the visual hierarchy of a visualization, such as converting a pie chart to a bar chart, without losing drawn annotations or styles.

2.2 Graph Data Wrangling Challenges

A similar challenge exists at the data abstraction level: here, too, significant iterative changes are difficult or impossible to accomplish with existing software, such as changing what a node is and what an edge is in a graph. Few software tools exist to assist in this part of the workflow, even though it requires a significant proportion of a practitioner’s time [54]. We explore how software can support graph reshaping operations, first with a visual technique for extracting subgraphs using pivot and filter operations in Chapter 5, and then a system that supports a wider set of operations in Chapter 6. In doing so, we build on existing tools that enable tabular data wrangling, graph editing, graph databases, pivot operations, and broader network modeling.

2.2.1 Wrangling Tabular Data

Applications for **wrangling tabular data** include tools such as Google Refine [55], Data Wrangler [56], and Microsoft Excel, which focus on data transformation and cleanup. These systems enable analysts to reformat input data to best suit their analysis tasks, but they are not designed to support network data.

Some data wrangling tools use network visualization in the process of wrangling data. D-Dupe [57] uses a network perspective to help resolve duplicate entities while cleaning a dataset, and Schema Mapper [58] uses network visualizations to explore how one hierarchical dataset maps to another. Although these approaches utilize network visualization in the wrangling process, and support wrangling tasks on nontabular data, they do not wrangle networks themselves.

NodeXL [59], an extension to Excel, allows users to import, visualize, and transform network data. However, NodeXL provides only a minimal set of network transformation features. Moreover, as pointed out by Liu *et al.* [2], conducting extensive network reshap-

ing with NodeXL would require users to be Excel experts.

2.2.2 Graph Editing

Beyond tabular data wrangling, a wide array of tools support **graph editing** that allow users to visualize and edit networks and their associated attributes. Tools such as Cytoscape [60] and Gephi [61] focus on layout of the topology of the graph, but also offer several editing features. Similarly, Graphviz [62] is a collection of graph drawing tools that can render hand-edited or programmatically generated text files.

Tools in this space allow users to modify a network, mainly by creating or deleting nodes and edges. This level of editing is useful for tasks such as finding and correcting mistakes in the data or inputting new data. However, these edits are not rule-based and are therefore limited to the instance level and do not generalize to the entire network. Additionally, graph editing tools assume a well-defined network as input. They are primarily tasked with representing network models as they exist, and do not have features aimed at creating or deriving new models.

2.2.3 Graph Databases

Unlike graph editors, **graph databases** enable generalized rule-based transformations. These systems are optimized for storing, indexing, and querying large networks, and specific languages have been developed to query and manipulate graphs in these databases. Graph databases such as Neo4j [12], OrientDB [13], and GraphDB [11] allow users to create and transform network models as needed. The potential operations that can be performed are broad—we utilize a Titan [63] graph database in Chapter 5 as the underlying technology as we explore the relationship between pivot and filter operations. In general, however, creating or changing a network model in such databases must be done through scripting, requiring users to be proficient in the database-specific query language. In contrast, the interfaces that we present in Chapter 5 and Chapter 6 are interactive; they support many wrangling operations without any programming knowledge.

2.2.4 Graph Pivots

The **pivot** operation that we explore in Chapter 5 has made both direct and indirect appearances across graph visualization literature. In using the term “pivot,” we refer to it

in the sense of traversing an existing graph, from one set of nodes to another [64]–[70], rather than toggling between node and edge interpretations [71], or aggregating node attributes [2]. Although pivots have been identified and used in the past—we do not claim the identification of pivots as a contribution—their usage is typically limited to an initial seed node set of size one. Chaining pivots together is often not supported, pivots are used to support specific tasks on specific data abstractions, and/or pivots are integrated as part of a broader system that does not present an opportunity to study them in isolation. Chapter 5 explores the power and effects of graph pivots in general, ignoring any particular abstraction.

We focus on pivots separate from other wrangling operations, as it is strictly neither a wrangling operation nor an exploration operation. In terms of the Graph Task Taxonomy [72] by Lee *et al.*, a graph pivot falls into three of the four identified categories. It is a **topology-based** operation, in that it starts by identifying the neighbors of the seed nodes. It is an **attribute-based** operation, in that it filters the neighbors to the set of target nodes. Finally, it is fundamentally a **browsing operation**, in that it traverses a set of n paths through the graph simultaneously. As pivots are essentially an aggregate form of traversal, there is no comparison to be made to traditional instance-based techniques, such as node-link diagrams or adjacency matrices. Rather, the technique that we demonstrate in Chapter 5 could be used in conjunction with standard instance-based techniques in a linked view system. Testing our technique in isolation allows us to reflect on whether and why additional views may be necessary for exploration, or for subgraph extraction.

2.2.5 Network Modeling

A broader set of wrangling operations is covered by **network modeling**, i.e., the concept of creating networks from tabular data, which has been explored by several systems. The need for these tools arises from the frequent storage of network data as tables, containing a list of items and their associated attributes. Network modeling approaches support creating graph models from tabular data, and commonly also provide visualization to better understand and explore the resulting graph structure.

Commercial systems such as TouchGraph Navigator [73] and Centrifuge [74] support network modeling by allowing users to create attribute relationship graphs from tabular

data. Attribute relationship graphs, which were introduced by Weaver [75], refer to graphs where attributes are connected based on co-occurrence. A related approach, used by both PivotGraph [65] and HoneyComb [76], generates new network models by aggregating all nodes that have a certain attribute. The Tulip framework [77] touches on network modeling by enabling users to import data and generating multiple data models for users to visualize and explore. However, users do not have control over how these data models are generated, nor can they modify them.

The two systems most related to the system that we discuss in Chapter 6 are Orion [1] and Ploceus [2]. Both systems model attribute relationship graphs from tables from relational databases. For a comparison of supported operations, refer to Table 2.1. Ploceus’s approach is based on relational algebra, whereas Orion uses relational tables as its base and represents networks as edge tables. Although Orion and Ploceus are well suited for network modeling from tabular data, they do not support reshaping existing networks or rich edge attributes beyond edge weights. The ability to iterate on chosen data abstractions is restricted to creating a new network model from the raw data. Each distinct network model needs to be specified from the ground up; users cannot create alternative network models with an existing model as a starting point. Both tools emphasize analysis of the network data within the tool, whereas our system is designed as a wrangling-first application, with visualization for analysis mostly used for validating wrangling operations.

Another related modeling tool is Graphiti [14], which uses demonstration-based interaction to create edges in networks. The tool infers possible data abstractions from instance-level operations. Although Graphiti supports modifying existing networks by adding new edge types, it does not support more extensive reshaping operations such as changing nodes to edges or deriving new classes from attributes of connected nodes or edges. Additionally, Graphiti is only suited to working with undirected, unipartite graphs. Our system offers support for unlimited types of nodes and edges as well as directed edges, allowing users to connect data from multiple sources to generate a suitable network model.

Ultimately, each software system that we present builds on a rich tradition of theory and software tools that inform and support the process of designing visualizations, including both challenges of rapid prototyping and refining graph data abstractions.

Table 2.1. Operations supported by Origraph, Orion [1], and Ploceus [2]. In some cases, operations are only partially supported, as denoted by the superscripts.

* Only for immediate neighbors

† Ploceus can identify specific node values through the search function

‡ Ploceus does allow for duplication of attributes across tables

Modeling / Reshaping	Origraph	Ploceus	Orion	Ploceus Term	Orion Term
• Connect/Disconnect Nodes	yes	yes	yes	Create Connection	Link
☒ Promote Attribute	yes	yes	yes	Create	Promote
☒ Facet Node/Edge Class	yes	yes	yes	Slice n'Dice	Split
↗ Convert Between Node/Edge	yes	no	no		
↗ Project Edges	yes	partially*	partially*		
❖ Create Supernodes	yes	no	no		
❖ Roll Up Edges	yes	yes	yes		
<hr/>					
Item Operations					
▼ Filter by Attributes	yes	no [†]	yes		
❖ Connectivity-Based Filtering	yes	no	no		
<hr/>					
Attribute Operations					
↖ Change Edge Direction	yes	no	no		
☒ Derive In-Class Attributes	yes	yes	yes		
☒ Connectivity-Based Attribute Derivation	yes	no [‡]	no		

CHAPTER 3

REFLECTIONS ON HOW DESIGNERS DESIGN WITH DATA

To better inform the development of tools intended for designers working with data, we set out to understand designers' challenges and perspectives. In this chapter, we report on an observational study with designers [3], both in the lab and *in situ*, as well as a series of interviews with design professionals. Identifying patterns in these observations and interviews exposed several emergent themes: the role of manual encoding, how and when designers make visual encoding decisions, the importance of workflow flexibility, and why data exploration and manipulation is critical to the design workflow. Finally, we discuss how these themes translate into rich opportunities for toolkit developers.

3.1 Motivation

Interest in visualization has exploded in recent years. Driven in part by the emergence of cheap, ubiquitous data, visualizations are now a common medium for exploring and explaining data produced in the sciences, medicine, the humanities, and even our day-to-day lives [15]. Among the growing number and types of visual analysis tools created for analysts, infographics [78]–[80] are a quickly emerging subclass of visualizations. These visualizations use static, visual representations of data to tell a story or communicate an idea, and they have infiltrated the public space through a wide variety of sources, including news media, blogs, and art [81]. The increasing popularity of infographics has grown a large community of practitioners. This community includes designers, artists, journalists, and bloggers whose main expertise is not in engineering or programming. These infographic designers tend to rely on drawing tools that ease the workflow while creating a visual representation of data.

3.2 Methods and Participants

The data we collected came from several sources: observations of designers in controlled, observational studies; observations of designers working on teams *in situ* at a design hackathon; and a series of unstructured and semistructured interviews. For each observation and interview, we took notes and voice recordings and collected design artifacts when appropriate. The voice recordings were later transcribed. We identified patterns through frequent discussions and several collaborative writing drafts between coauthors in a process similar to thematic analysis, but not explicitly following its methodology. These patterns are presented in Section 3.3.

In total, we observed and/or interviewed 15 designers with a diverse range of experiences and with strong design skills and training. Our participants included academics teaching in design departments; design students; and a spectrum of professional designers: at a large software company, at a large scientific research lab, and working freelance. Participants were solicited based on convenience and snowball sampling to identify individuals with appropriate expertise, and their participation was voluntary. Table 3.1 lists the designers, their design role, their experience with programming and data analysis, and in what aspect of our studies they participated.

3.2.1 Observational Studies

We designed our observational studies to better understand how designers work with data. These studies centered around two datasets that we prepared, inspired by existing infographics [7],[82]. The first was a time-travel dataset depicting years traveled in popular movies. The dataset was tabular, with the rows representing a time-travel trip in a movie, and the columns consisting of the movie title, movie release year, and the start and stop years involved with the trip. We deliberately included two outliers in the dataset in an attempt to observe how unexpected data disrupt a designer’s workflow. All the movies that we included traversed several to hundreds of years, with one exceptional trip that traversed 150 million years. A second outlier was a second trip from a movie already represented in the dataset.

The second dataset captured the reuse of actors, directors, and other staff in popular HBO TV productions. This dataset included two tables, the first where each row was

Table 3.1. Skill levels of each designer who participated. *T4 was also on the FitBit hackathon team and H1 participated in unstructured interviews.

Code	Gender	Design Role	Programming Experience	Data Experience
One 2-hour Exercise (Time Travel)				
T1	M	Student	None	None
T2	F	Student	None	None
Two 2-hour Exercises (Time Travel)				
T3	F	Industry: software	None	Basic
T4*	F	Industry: software	None	Basic
One 2-hour Exercise (HBO)				
H1	M	Industry: freelance	Basic	Basic
H2	F	Industry: software	Basic	Basic
10 hours observation at hackathon (FitBit)*				
F1	F	Industry: software	None	Basic
F2	F	Industry: software	Basic	Expert
3 hours observation at hackathon (Bug Tracking)				
B1	F	Industry: software	None	None
Unstructured Interviews*				
I1	M	Academic	None	Basic
I2	M	Industry: design firm	Expert	Expert
Semistructured Interviews				
I3	M	Academic	Expert	Expert
I4	M	Industry: freelance	Basic	Basic
I5	F	Academic	Basic	Expert
I6	M	Industry: laboratory	Basic	Expert

an individual and each column was a production title. In the cells, each individual's role was given for the productions in which he or she participated. The second table consisted of aggregate information about each production, indicating how many actors, directors, etc. were reused from another production. We designed this dataset to test how designers explore data structure. Although the data were originally inspired by the relational infographic in Figure 1.1, we instead chose to present the data in a tabular format.

3.2.2 Hackathon

We also observed two teams consisting of both designers and programmers at a three-day data visualization hackathon. The hackathon provided an opportunity to observe designers working in a real-world setting, as well as in a team environment. Each hackathon team was created organically, with participants choosing their own team, data sources, and objectives, reducing the potential bias due to lack of motivation and engagement typical in lab studies [83]. One team chose to work with FitBit activity monitor [84] data, and the other focused on software repository bug tracking data. In contrast to our observational studies, the hackathon was not strictly limited to static infographics; both teams planned interactive software visualizations. The designers discussed interactivity, but the only representations of data that they produced directly were static, and the patterns we observed were consistent with our other observations and interviews.

3.2.3 Interviews

We conducted a series of unstructured and semistructured interviews with designers both in person and over Skype. The interviews included questions about the designers' design workflow, how working with data influenced their design workflow, and the types of tools used. Semistructured interviews also specifically included two topics: whether, and to what extent, designers enjoyed repetitive tasks, as well as discussing the role of constraints in the design process, including relationships between data values and graphics elements.

3.3 Patterns

From an analysis of the data that we collected from our observations and interviews, we identified 12 emergent patterns of how designers work with data to create infographics; the evidence for each pattern is summarized in Table 3.2. Below we break these patterns into three classes: how the designers approached data, how data affected the designers and their workflow, and alternatives to manual encoding of data that the designers employed. Each specific pattern is stated in bold and numbered.

Table 3.2. Instances in which we recorded designers saying or doing something specific that supports an emergent pattern

Code	Context	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
T1	Exercise (Time Travel)	•	•		•	•	•	•	•	•	•		
T2	Exercise (Time Travel)		•					•	•	•	•		
T3	Exercise (Time Travel)	•		•	•	•	•	•	•		•	•	
T4	Exercise (Time Travel)					•	•	•	•	•	•	•	•
	Hackathon (FitBit)				•			•					
H1	Exercise (HBO)	•		•	•	•	•	•		•	•	•	
	Unstructured Interview	•			•			•					•
H2	Exercise (HBO)	•				•	•			•	•	•	
F1	Hackathon (FitBit)	•	•					•	•				
F2	Hackathon (FitBit)				•			•	•				
B1	Hackathon (Bug Tracking)		•						•				
I1	Unstructured Interview	•			•	•		•				•	
I2	Unstructured Interview		•		•	•				•			
I3	Semistructured Interview				•	•						•	
I4	Semistructured Interview						•						
I5	Semistructured Interview	•	•	•	•	•				•	•		
I6	Semistructured Interview						•		•		•	•	

3.3.1 How Designers Approached Data

In our observational studies, we observed all of the designers initially sketching visual representations of data on paper, on a whiteboard, or in Illustrator. In these sketches, **the designers would first draw high-level elements of their design such as the layout and axes, followed by a sketching in of data points based on their perceived ideas of data behavior (P1)**. An example is shown in Figure 3.1. The designers often relied on their understanding of the semantics of data to infer how the data might look, such as F1 anticipating that FitBit data about walking would occur in short spurts over time whereas sleep data would span longer stretches. However, **the designers' inferences about data behavior were often inaccurate (P2)**. This tendency was acknowledged by most of the designers; after her inference from data semantics, F1 indicated that to work effectively, she would need "a better idea of the behavior of each attribute." Similarly, B1 did not anticipate patterns in how software bugs are closed, prompting a reinterpretation and redesign of her team's visualization much later in the workflow once the data behavior was explicitly explored. In the time travel studies, T3 misinterpreted one trip that later caused a complete redesign.

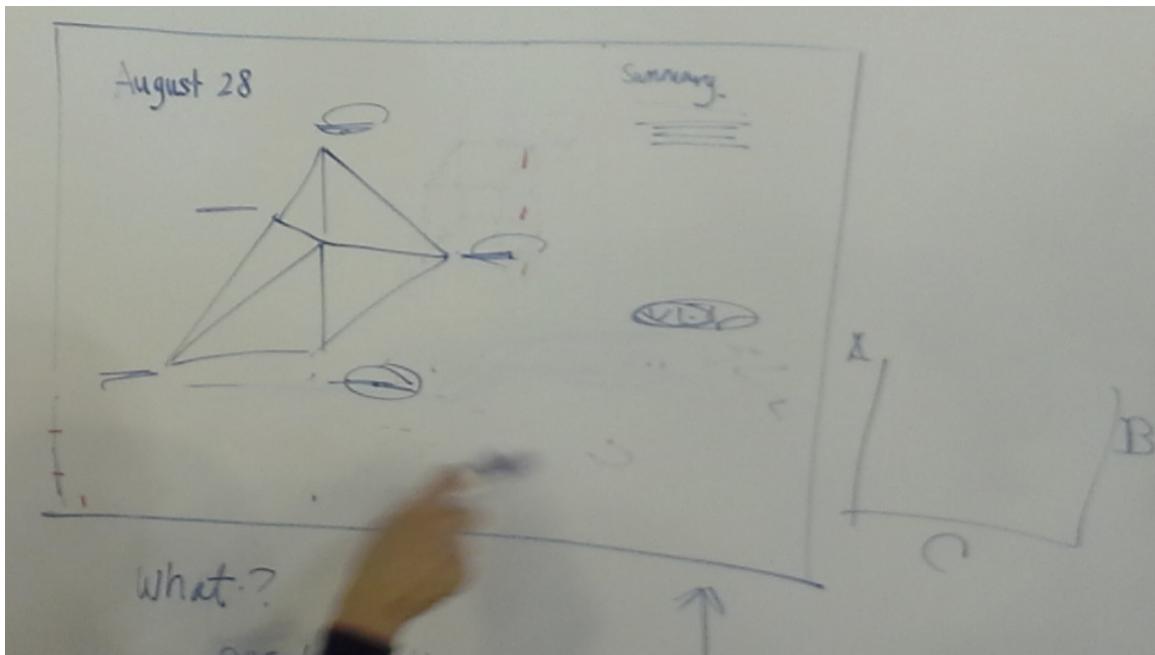


Figure 3.1. A whiteboard sketch of a repeated radar graph design from the hackathon. This sketch shows the planned visualization in the context of the rest of the design.

Furthermore, the designers' inferences about data structure were often separated from the actual data (P3). In brainstorming sessions at the hackathon, the designers described data that would be extremely difficult or impossible to gather or derive. In working with the HBO dataset, H1 experienced frustration after he spent time writing a formula in Excel only to realize that he was recreating data he had already seen in the aggregate table.

Our interviews indicated that both (P2) and (P3) are common patterns. The designers stressed that thorough data exploration is necessary to avoid misinferences, and that it is an important part of the design workflow. I5 said:

I spend most of my time with the data. That is the hard part when you are teaching because the students like to jump very quickly into solutions. It is very hard to explain that most of your time spent creating a visualization is with data.

Not surprisingly, **the amount of data exploration and manipulation was related to the level of a designer's experience working with data (P4).** In the time travel observational studies, both T3 and T4 discovered the 150-million-year outlier quickly. T3 accomplished this by asking the interviewer specific questions about the data, and T4 discovered it by creating charts in Excel. They also both discovered the repeated film. In contrast, the student designers T1 and T2 did not explore the data at all, and even resisted leading offers to have the data sorted toward the end of their exercises—they did not discover the outliers on their own. Similarly, at the hackathon, F2 was the only designer to ask the programmers on her team specific questions about the data they were working with.

Our interviews also confirmed pattern P4. I2 commented: "There's a default on the design side to go quickly to how it looks and not necessarily find the outlier." I3 similarly said:

Having a knack for the data science part often separates the good designers from the great ones. Personally I believe that the data science part is the Achilles heel of the designer. You gain insight by working directly with the data. The best designers are the ones that will open up Excel and manipulate the data before they get to the graphics part.

All but one designer in our observational studies manually encoded data; we observed T1, T3, T4, H1, and H2 looking at data points one at a time, estimating the placement of marks, and placing the marks by hand. Most often, the designers would draw an axis,

guidelines, and tick marks to assist in this process. All of our interviews confirmed that this is a common practice, but we were surprised to learn that **designers did not necessarily dislike manual encoding (P5)**. I2 said:

I am amazed at what people will sit through in terms of doing something manually with Illustrator or InDesign... not all of it is unenjoyable for them.... There's something great about just sitting there [with] my music on... [and getting] it perfect.

Furthermore, this toleration for manual work is in exchange for flexibility, which is consistent with what others have observed [85]. I1 explained:

Illustrator is the basic go-to tool that allows for enough creativity and flexibility to create and prototype what we want... [but] there's this lack of being able to connect real data to it. It just doesn't exist, so we sit there and have someone read through a spreadsheet.

We also noticed that for some of the designers, **manual encoding was a form of data exploration (P6)**. In the time travel study, both T3 and T4 discovered the outlier film with multiple trips while manually encoding the data and not during their earlier attempts to explore the data.

Manual encoding usually occurred toward the end of each designer's workflow after he or she created and refined other visual elements. This delay suggests that **data encoding was a later consideration with respect to other visual elements of the infographic (P7)**; they attempted to understand what a visualization would look like in the context of the rest of their design before real data were included. For example, in our observational studies, T3, T4, and H1 spent considerable time at the beginning of their workflow with decisions such as general layout, typefaces, color schemes, and paper sizes, along with sketching their perceptions of the data behavior. These decisions were explored long before considering where specific data points would actually fall. Another indication of this pattern was observed at the hackathon as F2 was presenting high-level ideas for a visualization to her team. When a programmer interrupted to ask what the axes were in a plot, the designers on the team emphatically responded that the level of detail would be decided later. H1 explained, "You almost see masses and blobs and shapes, and the relationship between them." I1 observed:

The way that [programmers] approach it is from the detail, back out.... [for us], the data binding stuff is really less important at first.... [Programmers might say] that's a very unstructured way to approach the problem, [but designers

say that] if you start with specifications, you've limited your design world immediately.

3.3.2 How Data Affects Designers

We observed that although the designers frequently refined the nondata visual elements of their designs, such as color and font choices, **the design of their data encodings remained unchanged until assumptions about data behavior were shown to be incorrect (P8)**. For example, in the time travel study, T3 introduced curves into her design after encountering the incompatibility of one movie in her initial design. Similarly both T1 and T2 made changes to their designs after discovering one of the outliers. T1 repeatedly drew roughly the same bar chart early in his workflow, but introduced a curled bar when shown the outlier, shown in Figure 3.2. Similarly, T2 changed to a completely different circular encoding. In contrast, the designers on the FitBit hackathon team did not have direct access to their data, and in spite of breaking out for individual brainstorming sessions, F1, F2, and T4 drew only variations on a single representation type, a radar graph as shown in Figure 3.1. Additionally, we observed that **the designers did not vary from the initial, given structure of the data (P9)**. For example, in the HBO study, both designers created a matrix-style visualization that mimicked the input file format, one of which is shown in Figure 3.3. This contrasts with the original infographic that inspired the dataset that explicitly represented the data relationally, shown in Figure 1.1. H2 even commented, “I feel like I’m stuck in this treemap world.”

Despite the benefits seen from manual encoding, we also observed that **the cost in effort and time of manual encoding kept designers from trying many varied ideas (P10)**.

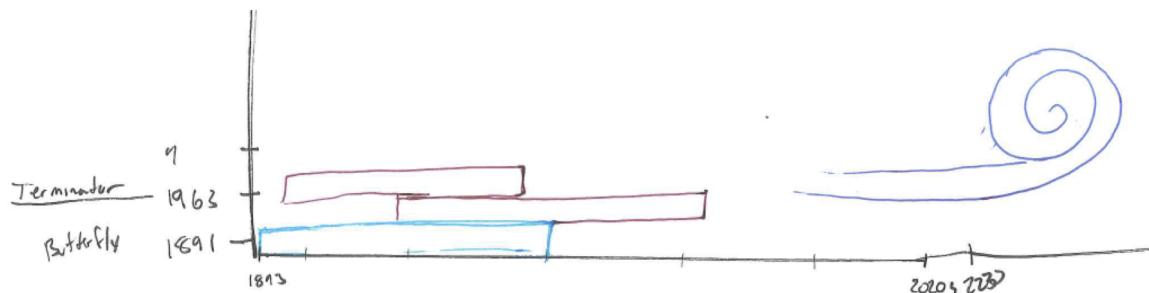


Figure 3.2. An example of how data can inspire variety: T1 introduced a curled bar when informed of the outlier.

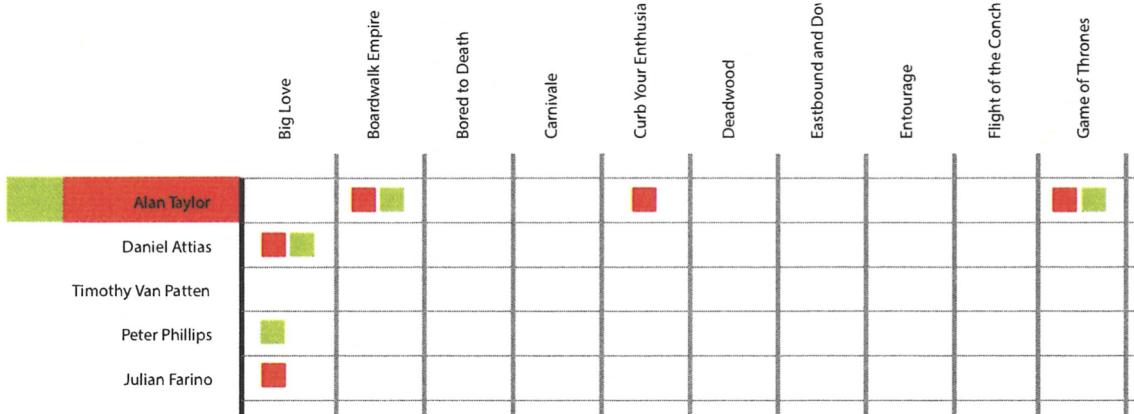


Figure 3.3. An example of how data can limit variety: H2 recreated the dataset’s tabular file format graphically.

In the time travel study, T1 plotted 21% of the movies, T3 plotted 14%, and T4 plotted 25%. T2 plotted no data points at all. For the HBO study, H1 was able to encode 26% of the actors and H2 was able to encode 32%. As the designers engaged in manual encoding, comments about inefficiency were frequent. For example, H1 said, “I don’t want to do this, but I... manually put in all the dots.” Furthermore, H1 reluctantly stuck with his original design even after making discoveries about the data that prompted dramatic changes in sketched encodings. He said:

I sketched that I was going to do [one thing]... but once I got it on the page, I saw something else.... If I twist this, then the whole grid is going to change, and I’m going to have to move everything around, and I’m going to have to manually reset all the text.

Although these effects may have been compounded by the time limits of our observations, deadlines remain critical factors in the design workflow.

3.3.3 Alternatives to Manual Encoding

We observed some designers using external tools to encode the data into a chart, and we discussed this approach in several interviews. In our observational studies, we saw two designers copy charts from Excel into Illustrator, and a third commented that she wished she could follow this pattern: “I would rather make [Excel] do this for me... [but] I don’t know how—and it’s not as pretty.” In all cases where this technique was observed or discussed, however, **the designers still performed downstream repetitive tasks to refine an imported chart to meet their design goals (P11)**. I6 described his workflow:

I created a plot in Tableau and then exported it to Illustrator to put the labels on. I had to add the specific tick-marks by hand. [For the labels], Tableau does labels, but it isn't very smart... I had to manually pull them apart when they overlapped. I did the donut chart in Excel because Tableau doesn't have them. I then changed the colors [in Illustrator] to greyscale to get rid of the Microsoft colors.

As with manual encoding, the labor-intensive nature of importing a single chart kept designers from attempting to import multiple charts.

Another technique we heard about in our interviews was to encode the data programmatically and import the resulting graphic into Illustrator. I5 described how she used programming this way:

Then I decided to use a radial layout to plot the data. I started in Illustrator, but then I realized it was going to be way too hard to do... so then I said, oh this is stupid, so I went to Processing. I looked online to see if I could learn some tutorials to help me put the data into this idea. Then I [wrote the program]. I brought all the data into my simple Processing code, and then I created a PDF, and brought it back into Illustrator.

Designers frequently expressed dissatisfaction with these approaches. I5 lamented, "I teach myself programming when I don't see any other way."

A common frustration with alternatives to manual encoding expressed in interviews was that **the designers disliked tools that enforce a design workflow (P12)**. For example, I6 indicated a strong negative reaction to Tableau's Show Me functionality [86]:

Tableau has lots of prescribed things... they say, okay, you have these two data types, then we recommend that you use this plot. I hate that thing; I always turn that thing off.

Similarly, I3 commented:

Developers always want to [control the design steps]; [they say] here is your flow, you go through these steps, here are the tools, here are the things you can do, here are things you can't do. I don't think designers want that.... [They] want to create something that you've never seen before. And to create a tool that will allow that amount of flexibility is really challenging. Designers like Illustrator because it lets them do whatever they want.

3.4 Themes

The observed patterns revealed four themes, summarized in Table 3.3. First, manual encoding is acceptable when the trade-off is flexibility, and is a form of data exploration. Second, designers prefer to place data on existing graphics instead of generating graph-

Table 3.3. Summary of relationships between observed patterns and emergent themes

	Manual encoding has its benefits	Placing data on existing graphics	Relaxing the sequence of processing	Creating an effective data abstraction
P1: Drawing high-level elements, followed by sketching in data points	•			
P2: Inferences about data behavior often inaccurate			•	
P3: Inferences about data structure often separated from actual data			•	
P4: Amount of data exploration, manipulation related to experience			•	
P5: Designers did not necessarily dislike manual encoding	•			
P6: Manual encoding was a form of data exploration	•			
P7: Data encoding a later consideration with respect to other elements		•	•	
P8: Unchanged encodings until assumptions shown to be incorrect				•
P9: No variation from the initial, given data structure				•
P10: Cost of manual encoding prevented trying many varied ideas			•	
P11: Downstream repetitive refinements of an imported chart	•	•		
P12: Designers disliked tools that enforce a design workflow	•	•		

ics directly from data. Third, to support flexibility, operations should be commutative. Finally, tool creators should be aware of designers' struggles to define and modify data abstractions. These themes have interesting implications for two known challenges in the visualization community: the challenge of creating tools that support defining and modifying the data abstraction, and the challenge of creating flexible yet efficient tools for producing visual representations of data. We denote each theme in bold.

3.4.1 Manual Encoding Has Its Benefits

Our observations and interviews point to a clear trend: **manual encoding is not only tolerated, but even embraced by designers in order to maintain flexibility and richness in the design workflow**. This finding is in juxtaposition to the motivations and goals

of many visualization creation tools that strive to ease the burden of encoding data by limiting the design space and enforcing an order of operations. Even in situations in which designers use external tools to encode data, this step was just one of many—designers still spent significant time and effort manually modifying and refining externally generated charts (**P11**). We found that designers are willing to endure repetition in order to keep their workflow and their options flexible (**P5, P12**). There are even benefits to manual encoding. It provides unique opportunities for designers to discover aspects of the data (**P6**), and it can even be enjoyable (**P5**).

There are trade-offs, however, as manual encoding consumes significant time and effort, reducing opportunities and a willingness to explore a variety of design alternatives of visual representations of data (**P10**). This limitation contrasts a central tenet of design practice to try many ideas and explore alternatives. Recent research supports this tenet by looking at how starting with multiple examples [28], exploring a variety of approaches [29], and building multiple prototypes [30], [31] can help designers approach problems more creatively and produce higher quality solutions.

This theme sheds light on a part of the visualization representation challenge that has not received much attention. Currently, users of both GUI-based and programming toolkits are vigilantly protected—and, in most cases, prevented—from engaging in manual encoding, in part to avoid the time-consuming efforts necessary in manual encoding. The benefits of manual encoding our patterns expose, however, are not currently exploited and could provide interesting new directions for future tool development.

3.4.2 Placing Data on Existing Graphics

GUI-based visualization toolkits provide ease of use, efficiency, and sometimes support user-specification of the look and feel of marks, but these features come at the expense of flexibility [15]; the underlying visual representations are all predefined or selected via a GUI. Instead, designers told us they want to be able to create something new and novel. Programming toolkits also have benefits in that they provide more flexibility, but they require the representation to be defined in a nonvisual, symbolic manner. Instead, we observed designers creating visual representations graphically (**P1**), and they were often unsatisfied by the results of both kinds of visualization toolkits (**P11,P12**).

In contrast to these two bottom-up approaches, our observations indicate that **designers create visualizations in a top-down, graphical workflow** in which they place data marks on top of other visual elements. Designers tend to try to understand the overall appearance of a visualization before plotting real data on axes that they draw (**P7**). Designers usually must consider other design elements in an infographic—data are only a part of how they tell a story—and they tend to think of influencing existing graphics that they have already created, instead of generating graphics directly from data.

The seemingly irreconcilable balance between usability and expressiveness is another part of the visual representation challenge. This theme suggests that if tools can support placing data on existing graphics instead of generating graphics from data, such tools will be both intuitive for designers and more expressive. Even though calls for this kind of approach have been articulated in websites and blogs [87], [88], little academic research has pursued this direction.

3.4.3 Relaxing the Sequence of Processing

We found that the designers we observed and interviewed avoid tools that enforce a workflow (**P12**), supporting the idea that **designers prefer a flexible design environment that does not enforce a specific order of operations**. Consistent with what others have observed [89], the order in which designers will perform various operations cannot, and perhaps even should not, be anticipated; instead, visualization design environments could support a workflow with a flexible sequence. Changes in one area of the design could, ideally and intuitively, affect changes in another.

This need is due, in part, to the fact that designers consider many aspects besides just the visual encoding itself, including annotations, labels, and embellishments (**P7**). Interestingly, designers use external tools not as closed solutions, but as just one step in their workflow (**P11**). This is a reaction to native restrictions in these tools: current tools force designers to specify a complete mapping of their data exactly once, and the result is brittle. To accommodate these tools, designers still had to engage in repetitive work similar to manual encoding (**P11**), resulting in the same drawbacks of manual encoding (**P10**) to the extent that their entire workflow had to be repeated if anything beyond a superficial change was to be made.

This theme lies at the intersection of the data abstraction and visual representation challenges. In many ways, the visualization community faces the same commutativity problem: 1) updating a visualization in response to a data abstraction change; or 2) updating a data abstraction in response to a visualization, both of which are extremely laborious. Crafting techniques to facilitate either are open areas of research.

3.4.4 Creating an Effective Data Abstraction

As designers with data experience are aware (**P4**), they need to be able to define and modify data abstractions. This is important for two reasons. First, the time spent planning or implementing ineffective or incompatible visualizations can be reduced by understanding data behavior (**P2**) and structure (**P3**) early on in the design workflow. Second, variety in design is increased when the data behavior (**P8**) and structure (**P9**) are well understood. Other research has shown that variety in design is important in that it directly affects the overall quality of the finished product [28]–[31]. Data abstraction tools that attempt to provide the freedom to explore alternative interpretations of data, including alternative data structures and deriving new data, are still in their infancy. The idea that a data abstraction is in part created and a piece of the overall design suggests that working with data abstractions may be a creative space in its own right [24], [90]. Therefore, some have suggested that the responsibility for a given data abstraction belongs to its creator [23], not necessarily the tool that facilitated its creation.

3.5 Opportunities and Next Steps

These themes expose rich opportunities for visualization creation tools. One is to separate the processes of visual encoding and data binding so that manual visual encoding is allowed but data connections are maintained. Granting designers direct control over what data bindings exist could liberate them to follow any order of operations in their workflow. Similar to existing approaches, they could create visual elements from data. Alternatively, they could first create visual elements, and later associate them with data, as our themes suggest they prefer.

Another opportunity is to allow a designer to explore data abstractions directly in the context of visualization, and *vice versa*. For example, a designer may wish to select all

visual items associated with negative data values, similar to Illustrator's functionality for selecting items of a given color. Visualization tools could support data operations internally, or connect to data abstraction tools externally. These tools would provide a flexible order of operations in a different sense, in that it would be easier for designers to iterate on data abstractions and visual representations together. Open problems in this area include how to make data abstraction tools and visualization design environments interoperable such that they are agnostic to workflow.

These two opportunities form the basis for the problems that we address in the remainder of this dissertation. Chapter 4 presents a model that enables new workflows, made possible by identifying the role of data bindings as designers iterate between drawing tools and generative visualization toolkits. Chapter 5 and Chapter 6 explore software that supports nonprogrammers in iterating on graph data abstractions.

CHAPTER 4

ITERATING BETWEEN TOOLS TO CREATE AND EDIT VISUALIZATIONS

Having discovered how and when designers make visual encoding decisions, we now discuss a software model that enables a more flexible workflow, overcoming a software-imposed hurdle that we identified in observing how designers work. In Chapter 3, we observed designers exporting graphics from generative visualization toolkits to drawing programs; however, current software limitations prevent designers from iterating on anything generative about their visualization once they have made this transition.

To support a fully flexible workflow—enabling designers to use either modality in any order—we discuss a model for bridging generative and drawing software, as well as an Illustrator plugin that implements this model [4].

4.1 Motivation

Visualization designers use a variety of tools in the practice of their craft, particularly when creating infographics and telling stories with data. Designers will often transition between tools, first making use of tools like Tableau, ggplot, and D3 to automatically encode data into a chart. Then, they make stylistic changes and add embellishments in a tool such as Illustrator [3]. This workflow, however, limits iteration: once a visualization is exported from a D3 script into Illustrator, the graphical elements are merely shapes, and are no longer linked to data. A designer cannot easily go back to modify the D3 script without losing their Illustrator work. The result of this disconnect between tools is that designers explore fewer design variations and have trouble handling changes to the underlying data [3].

We see an example of how this manifests in a visualization created by designer Craig Robinson and shown in Figure 1.1, which describes the actors employed by HBO and the TV shows in which they are cast [7]. The actors are sorted alphabetically based on their

first name, with the exception of the last actor—the asterisk next to this actor’s name is an apology for not placing him in the correct, sorted order. The likely scenario that resulted in this problem was that the designer performed significant manual work, placing nodes and connecting them with edges, in a drawing tool such as Illustrator before realizing that one actor had been left out. He would need to do much more manual work to insert this name in the right place, moving many nodes and edges manually. In a generative tool such as D3, however, adding the new name would be trivial—but it would come at the expense of losing the stylistic work done to the image in Illustrator such as color selection, font choices, and text layout.

This chapter contributes, first, a model that supports bridging between generative and drawing tools for visualization design. The model supports design iteration and reduces rework [91] by recasting the iteration problem as one of merging. This **bridge model** is general, and works across many tools; it describes a large space of design possibilities, trade-offs, and considerations. The second contribution is a single instance of the bridge model, an open-source tool called Hanpuku (Japanese: *iterate*). Hanpuku allows a designer to programmatically generate visualizations by writing or reusing D3 code, to then edit those visualizations in Illustrator, and to bring the edited visualization back into D3 for further generative modifications. This editing cycle can be repeated over and over, supporting iteration on the visual representation, the style, and the data. We illustrate the richness that Hanpuku affords to the design process in several examples. Hanpuku, along with high-level descriptions of several other possible bridges, validates the efficacy of the underlying bridge model for supporting visualization design iteration across a range of existing tools.

4.2 The Challenge of Iteration

Design iteration refers to the creative exploration of the many aspects of a design. Designers are more successful when they can explore the parts of a design in any order [89]; conversely, serial design workflows present fewer opportunities to weed out poor designs and discover creative opportunities [92]. The ability to go back and make changes at any level of the design is the meaning of **iteration** that we use throughout this work.

Visualization designers use a variety of tools because the creation of compelling, en-

gaging, and accurate infographics, based on increasingly large and complex data, implies two sets of software requirements. Many of the requirements are addressed by traditional visualization tools: handling large amounts of data, updating a visualization based on changes to the data, changing the visual representation, and algorithmic generation of certain graphical representations such as the placement of marks and creation of complex layouts. Tools such as Excel, Tableau, and ggplot, as well as programming languages and environments like D3, Processing, and VTK, are proficient at these sorts of generative tasks that a visualization creator would want to define abstractly and repeat automatically.

Conversely, designers also make use of flexible, manual controls for quick, visual iteration on stylistic elements and visual embellishments, such as color palettes, fonts, annotations, overall sizing, and placement of text. Exploring these detailed aesthetic choices can be tedious with generative controls—a designer might need to restart execution to test a different shade of blue—and instead benefit from the immediate visual feedback available in tools such as Illustrator and Inkscape. Furthermore, drawing tools focus on supporting a very rich set of manual controls as well as intuitive ways of showing design options and saving iteration provenance. The flexibility of drawing tools is important for allowing a designer to style and individualize a visualization [3].

Designers often work their way from one tool to another. An example is a visualization that designer Bang Wong created for a scientific publication [93]. He is quoted [3] as saying about his design process:

I created a plot in Tableau and then exported it to Illustrator to put the labels on. I had to add the specific tick-marks by hand. [For the labels], Tableau does labels, but it isn't very smart... I had to manually pull them apart when they overlapped. I did the donut chart in Excel because Tableau doesn't have them. I then changed the colors [in Illustrator] to greyscale to get rid of the Microsoft colors.

This workflow, which begins in generative tools to create a first cut on a visualization and then moves to a drawing tool for refinements, is common.

Unfortunately, this process can severely limit flexibility. Once a visualization has been brought over from a generative tool to a drawing one, it loses its connection to data: exporting is a one-way process. If a designer wants to modify the visualization later—perhaps to accommodate new data, or to correct algorithmic errors—he or she needs to make a difficult decision. The changes made in drawing tools cannot be transferred back

into the generative tool, and so changing a visualization entails redoing any graphical revisions.

The challenge with iteration is that the original image generated in generative visualization software embodies a mapping from underlying data into an image. The core algorithm behind many visualization programs is a simple loop: for each data item, compute the location and rendering for the corresponding mark and plot the mark. Once the marks are plotted as shapes, and the resulting visualization is produced and exported, however, the underlying mapping to the data is lost. When the designer manipulates the marks in a drawing tool, there is no longer a connection between the data and the visualization.

4.3 The Bridge Model

Our solution to this problem is the **bridge model**, so called because it maintains a bridge between the generative tool and the drawing tool. The bridge model is general across multiple platforms, generative tools, and drawing tools. In Section 4.4 we present a specific implementation of the bridge model, a tool called Hanpuku, which bridges D3 code and Adobe Illustrator. The design decisions we made in Hanpuku help illustrate the virtues and challenges of this model. We discuss high-level design decisions for several other possible bridges in Section 4.5.

The key insight in the model is to recast the iteration problem into a merge problem. We formulate the problem in terms of isolating the changes made in each tool—splitting the set of serial edits into parallel sets of edits—and then merging these changes. Under this approach, iteration becomes a process of repeated merges. This merging can be viewed as a kind of visual **join**, in the sense of the word used both by the database community and the D3 tool: each shape is associated with an identity, known as a **key**; the merger then decides how to integrate the changes to the shapes on each key, made by each side. The bridge model articulates the considerations necessary for merging changes from two different tools.

The bridge model starts from a shared representation of a visualization. Specifically, the visualization must have a representation both in the generative tool and the drawing tool. The model also requires a way to uniquely identify each graphical element within the visualization and to maintain those identities across the tools.

Each tool may imbue the visualization with **nontransferable elements**: parts of the visualization that do not translate to the other tool. For example, in a generative tool the nontransferable elements might include the source code, interactive slider bars and filter buttons, or interactive components like roll-over highlighting or user-influenced force-directed layout. Conversely, in a drawing tool there might be color palettes, layering effects, or text layout features such as kerning and drop caps that do not easily translate to standard generative tool features.

The full model is illustrated in Figure 4.1. In this figure, the left side represents work done in a generative tool and the right side represents work done in a drawing tool.

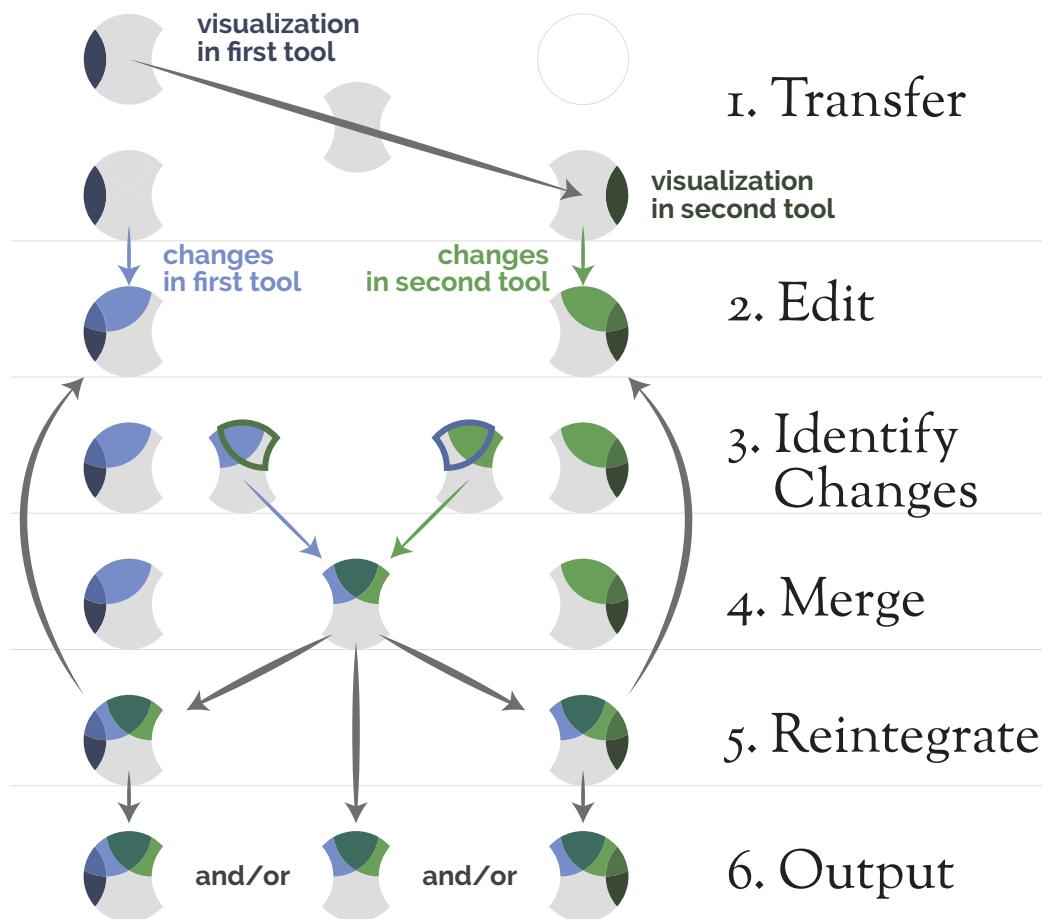


Figure 4.1. An abstract representation of each stage of the bridge model, showing how changes to the visualization in one tool can propagate to the other. Note that, although it is impossible for tool-specific, nontransferable elements to exist as part of the visualization at the same time, they can still continue to contribute to the visualization's design throughout the workflow.

1) A visualization is generated in a generative tool consisting of shared elements in gray and nontransferable elements in dark blue. The shared portion of the visualization is **transferred** to the drawing tool. 2) **Edits** to the visualization are made in either tool, shown as an intersecting light blue region for changes made in the generative tool and an intersecting light green region in the drawing tool. These changes may include nontransferable elements. 3) Changes to the visualization are **identified** by the bridge, along with an identification of changes that may be in conflict. 4) The changes are **merged** and the conflicts resolved, shown as a teal region. 5) The merged changes are **reintegrated** into the representations of the visualization in both the generative and drawing tools. At this point the process repeats as the designer iterates on the visualization. 6) The final visualization is **output**, either directly by one of the tools or as some other representation.

We believe there is no single right way to build an effective bridge; instead, bridge creators must carefully consider the design space and choose a set of strategies that balance flexibility, richness, and implementation difficulty. This design space is large. The goal of the remainder of this section is to lay out the strategies for implementing each component in order to allow bridge creators to carefully consider the multitude of options in a structured way. In Section 4.4 we will discuss our specific decisions with respect to our D3-Illustrator bridge, Hanpuku. We note that this design space is relevant for not only creating bridges between tools, but also building internal bridges within a single tool that supports both generative and manual changes to a visualization, such as IVisDesigner [44].

In this chapter, such as in Section 4.4, Figure 4.2, and Figure 4.3, we show two different paths being carried out simultaneously. Changes are seen not as simultaneous, but happening in parallel on two different tools. Even in cases where the designer simply merges the newer changes in one tool into the other tool, the edits must be consolidated together—and in many scenarios, the changes made in one tool are not reflected in the other. For example, in Hanpuku, changes that the designer makes to the visualization in Illustrator do not alter the D3 script, and so the changed D3 script’s output must be merged with the Illustrator changes.

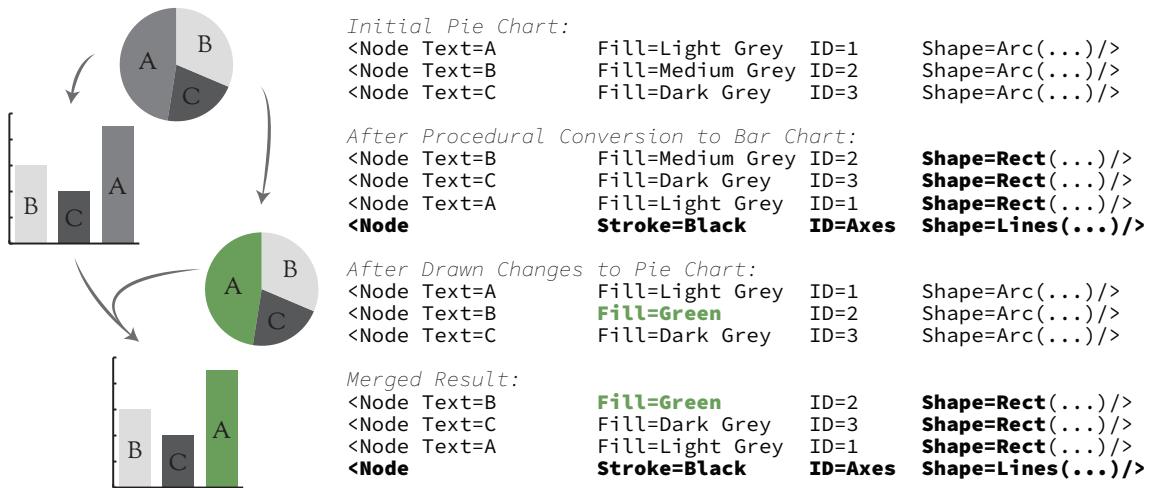


Figure 4.2. Merging changes together. On the left, the visual effect: two different changes are merged. On the right, the element IDs render the merge unambiguous.

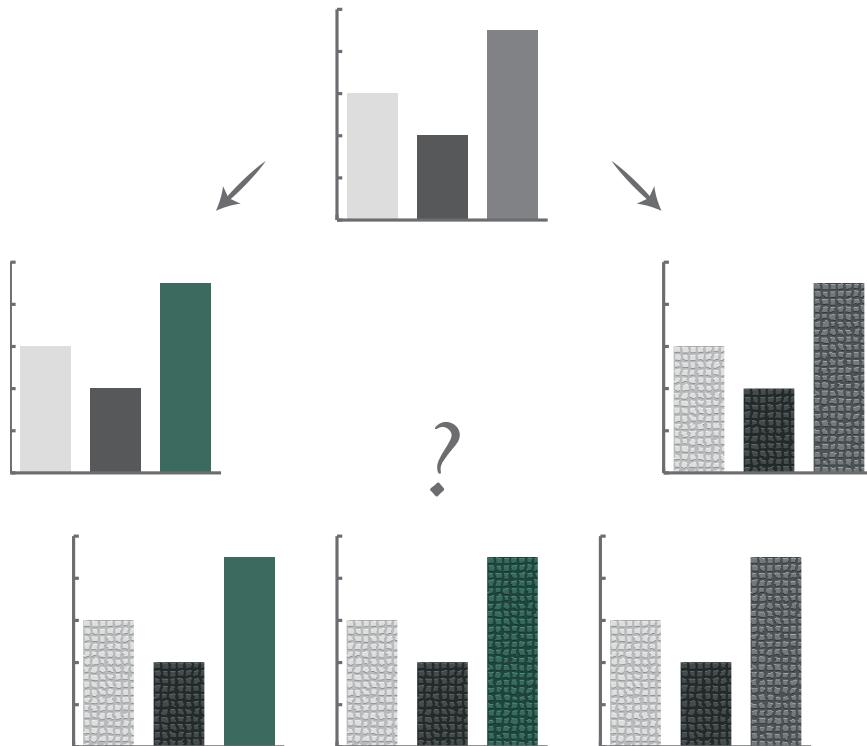


Figure 4.3. Three possible ways to handle a conflict.

4.3.1 Identifying Changes

In the bridge model, an image consists of three types of content: data-bound graphical elements, nonbound graphical elements, and nontransferable elements. Data-bound graphical elements are the visual marks that have a clear association with specific data items. The nonbound graphical elements are those elements of the image that do not correspond directly to individual data points, such as legends in a generative tool or titles and background images in a drawing tool. These elements need to be transferred between the tools as well, so the bridge must also provide them with unique identities. Finally, there are nontransferable elements that have no meaning on the other side, such as the source code in a generative tool or the paper size in a drawing tool.

An implementation of the bridge must have some mechanism both to share the original visualization between tools and to communicate what changes each tool made to the visualization. Communicating the changes requires that the bridge maintain a notion of the identity of graphic elements within the visualization. All shared elements, including data-bound and nonbound elements, must be identifiable on either side of the bridge. Additionally, data-bound elements must be linked back to the original data points to which they correspond. Nontransferable elements, however, are not shared, and therefore do not need to be identified.

A bridge must maintain a link between the data items and the representation in each tool—we call these links **data bindings**. These data bindings can exist in application-specific forms. In the simplest case, many data formats such as the HTML DOM (Document Object Model) and PDF allow arbitrary user tags to be attached to graphical elements. Those tags can be used to hold data binding identities. Tagging data items as they are transformed into shapes for a visualization is a common approach; the technique is used in VisTrails [94], [95] to track provenance, and in Weave [96] and Improvise [97] to enable brushing and linking. In D3 Deconstructor [49], data tags are used to reconstruct the original data mappings from the visualization. We take advantage of that same approach.

In Figure 4.2, we illustrate the role of the data bindings. A pie chart is created in a generative tool; each element includes an ID tag. This visualization is then modified twice: in a generative tool, the designer changes from drawing a pie chart to a bar chart; in a drawing tool, the designer highlights wedge “A” in a unique color. The merger is able

to recognize the changes made on each side: because it can detect that the new bar #2 corresponds to the wedge #2, it can apply the fill of the wedge to the bar. Maintaining data bindings in both tools allows a merger to appropriately track the changes made to the same elements.

4.3.2 Merging Changes

The goal of the merge process is to bring the edits made in each tool into a single representation of the visualization. In revision control merge algorithms, the merge attempts to choose an output that incorporates nonconflicting changes to source code. In the bridge model, a merge similarly starts with a representation of the edits made in each tool and attempts to reconcile them into one version, possibly with human assistance.

When two modifications of the visualization have no conflicts, it is easy to merge them together. For example, if there were no graphical elements changed by both tools, merging is the straightforward process of taking the newest version from each. The process becomes more challenging when there are **conflicts**. Conflicts are changes that cannot be implemented at the same time; conflict can occur when both tools make changes to the same graphical element that affect the same, or similar, encoding channels, such as both tools modifying an element's color.

The easiest way to deal with conflicts is to avoid them. One way to do this is to track and identify changes in as fine-grained a way as possible. For example, consider the refinements made to a pie chart in Figure 4.2. In this example, each tool carries out a set of changes: in the generative tool, we change the shape of the graphical elements from pie slices to bars and move their positions; in the drawing tool, we recolor one graphical element. Because we track changes at the fine-grained level of color, shape, and position, the changes are straightforward to merge as they operate on different parts of the encoding. If we looked at the graphical elements in a more coarse way—treating elements as atomic, for example—then this situation would turn out to be a conflict.

The refinements shown in Figure 4.3 are an example of conflicting changes. Starting with a bar chart, the two different tools each refine the visualization. In the drawing tool, a designer highlights one mark of interest; in the generative tool, he or she adds a texture to all of the marks. In this situation, it is not obvious what the bridge model

should do; the result is ill-defined. Valid mergers might choose from a variety of strategies: one refinement might trump another, the system might compute a combination of the refinements, or the system might ask for human input to resolve the difficulty.

Another strategy for minimizing conflict is to provide as much context as possible for each change. In the fine-grained example above, the bridge was able to look inside the graphical element to merge on its attributes. One way to help ensure that these attributes are available is to place related graphical elements into groups and label the group with the data key, which ensures that groups will move together. This strategy is useful when the system renders the label, the fill area, and the outline for a bar as three separate shapes, and it allows great flexibility: the user can use the illustration tool to remove the label or replace the bar with an image. The existence of the group helps point to what changes were made, and the shapes will be moved relative to the group.

4.3.3 Capturing Intent

So far we have talked about well-defined changes that a designer makes to a visualization. In many types of refinements, however, the designer has a broader intent for the way the visualization as a whole is being changed. These types of ill-defined changes are difficult, if not impossible, to accurately capture. In this section, we discuss two types of ill-defined, but common, changes—annotations and manually defined algorithmic rules—and explain why they pose a challenge for the bridge model.

To illustrate the challenges around intent, we can begin by looking at one problematic case, annotation. One convenience of drawing tools is that it is easy to annotate objects to call out specific aspects of the data. For example, a designer might place a textual annotation over the teal bar in Figure 4.4. In merging this change, the bridge implementation must decide what the location of this new annotation means. Is its position meant to be absolute on the page, or relative to the bar? Is it meant to be three pixels higher than the teal bar, or the highest bar? If the highest bar changes, or the order of the bars change, or the designer changes back to a pie chart, where should the annotation go?

Some of the mechanisms we outlined above can help here: putting the annotation into a group signals that if the bar is moved, the annotation moves with it; keeping the annotation disconnected from data signals that the annotation should sit at an absolute position.

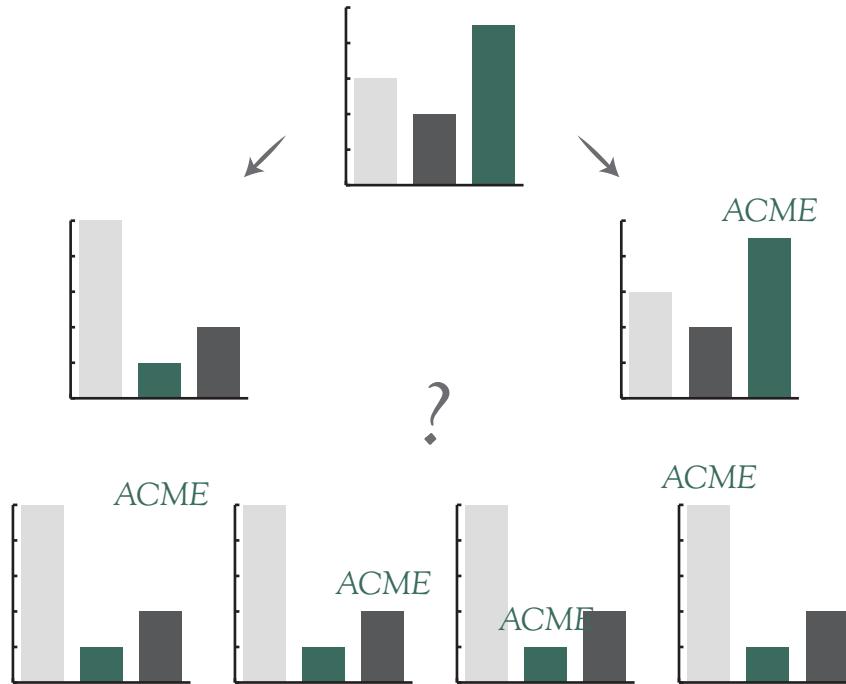


Figure 4.4. It can be difficult to automatically infer a designer's intent in placing a label.

Alternately, adding a constraint system into drawing tools could help disambiguate user intent here. The constraints would help show which items are meant to correspond to each other. Establishing constraints need not be a heavyweight process. In Microsoft's PowerPoint, for example, when a user places an object, the system shows guidelines that help align it with existing objects on the slide; it is not hard to imagine using this mechanism to store constraints.

Another form of intent appears when the designer has a sense of an algorithmic rule, but implements the rule manually in the drawing tool and not in the generative representation. For example, consider a visualization with a color map that runs from red to blue. If the designer uses a drawing tool to change one end of the color map to green, it would be difficult under the bridge model to automatically infer their intent, and recolor the rest of the color map to fit.

We encounter this uncaptured intent in the example shown in Figure 4.5. The designer manually defined a color map by explicitly setting the colors of each TV show node in an ordered fashion. These colors are then applied to edges that emanate from each show. When an additional actor node is added using a generative tool, shown in Figure 4.6, the

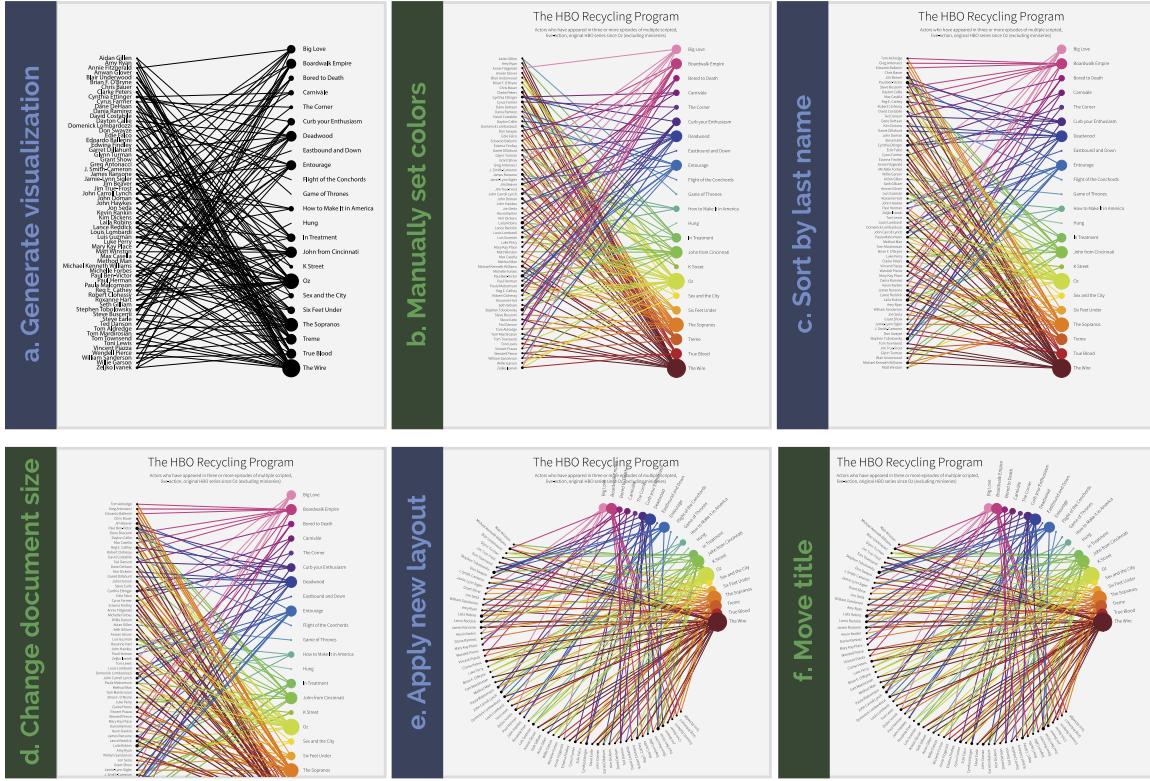


Figure 4.5. Extended recreation of Figure 1.1. Blue steps are done in a generative tool; green steps in a drawing tool. (a) An initial graphic is produced using a D3 script. (b) Illustrator’s tools are used to adjust typefaces and font sizes, add a title, a subtitle, and color to each TV show, with its corresponding edges. (c) A change is added to the D3 script to reorder the actors by last name, and then the script is re-run on the existing document. (d) The aspect ratio of the page is changed with Illustrator’s tools. (e) The layout algorithm in the D3 script is modified to accommodate the new aspect ratio. (f) The title is adjusted using Illustrator’s tools.

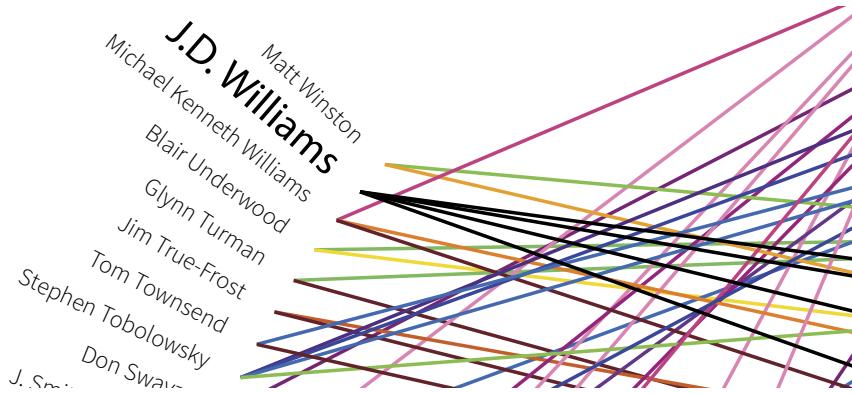


Figure 4.6. An additional actor is added to the example in Figure 4.5. Note that the text and lines do not reflect the designer’s manual color and typography assignments.

nodes are moved appropriately, but the color of the edges is not encoded in the generative representation. These colors must therefore be manually edited.

A viewer of this visualization can identify the color algorithm: edge colors should be the same as the node they are attached to. Maintaining that sort of constraint—as well as constraints like color maps—is easy to do generatively, but challenging to articulate within a drawing tool.

4.3.4 Reintegration and Output

After the merge stage of the bridge model, the process is not yet complete; the reintegration step brings the merged representation back into each tool. The previous stages of the merge process have stripped the common elements in both tools away from any nontransferable elements that may be a part of the visualization in each environment, such as scripts or interactive UI elements such as buttons and menus in the generative environment, or color swatch and layer compositing settings in the drawing environment. Reintegration brings the merged result back into the tools at each side, allowing the user to make further edits that build on the merged results without losing their previous work. Reintegration, therefore, enables the iterative loop.

Reintegration need not be a two-way process. Although in Hanpuku changes can be propagated from the design back into the generative tool, that is not strictly necessary. A one-way merger only means that integration steps are pushed forward into the drawing tool. The merger would still need to make sure that changes created on both sides be brought together. In an extreme design, it might be possible that *neither* tool sees the merged results; instead, the merged output might be in a form that neither tool can view.

One result of reintegration is output: if only a static image is the intended result, it may be appropriate to support only one-way reintegration into the drawing tool. Alternatively, if an interactive visualization is the intended result, it may be appropriate to support only one-way reintegration into the generative environment. In some cases, it may be sufficient to allow the user to make a series of changes in each tool, followed by one large merge at the end, skipping reintegration altogether. This final approach can still support an iterative workflow—changes can still be made in each respective environment, and the merged output can be regenerated.

Although nontransferable elements do not necessarily need to be identified in the bridge model, identifying and considering such elements can still be useful. As we saw in Figure 4.1, nontransferable elements from two tools can never exist together in the visualization at any point. A bridge, however, can employ various strategies regarding nontransferable elements to facilitate seamless reintegration.

The simplest strategy for handling nontransferable elements is to simply ignore any aspects of the representation that cannot be shared, leaving them in their source environments until the merge is reintegrated. The drawback to this approach is that any refinements made to these aspects of the representation will not be mergeable. This solution may be appropriate if very little of the visualization is affected, or if we expect the output to be in the tool that contains this nontransferable information.

The other strategy is to build in translations and approximations as proxies for non-transferable elements. These proxies become shared elements that can participate in the merge. For example, the merger could decide that when it sees a gradient shade applied in a drawing tool, it will fall back to a solid color instead if the generative tool does not have an equivalent feature. In more sophisticated implementations, the merger might include more sophisticated computational approximations: Adobe Illustrator cannot represent a circle directly, so any SVG elements with circles become four Bézier curves instead. The reintegration step must then decide how to apply changes that affect the proxy element to the original nontransferable element in its native environment.

4.4 Hanpuku: An Example of a Bridge

We have built an example of a bridge between a generative tool and a drawing tool in the form of Hanpuku, an Adobe Illustrator extension that merges changes made by a D3 script into an Illustrator document. Here we discuss Hanpuku, highlighting the various strategies from the bridge model.

As shown in Figure 4.7, Hanpuku provides a programming environment for editing and running D3 code. Fundamentally, Hanpuku incorporates two kinds of merges, each triggered by a button click. Clicking the “To D3” button merges changes from the Illustrator document into the programming environment, and clicking the “To Illustrator” button merges changes from the programming environment into the Illustrator document.

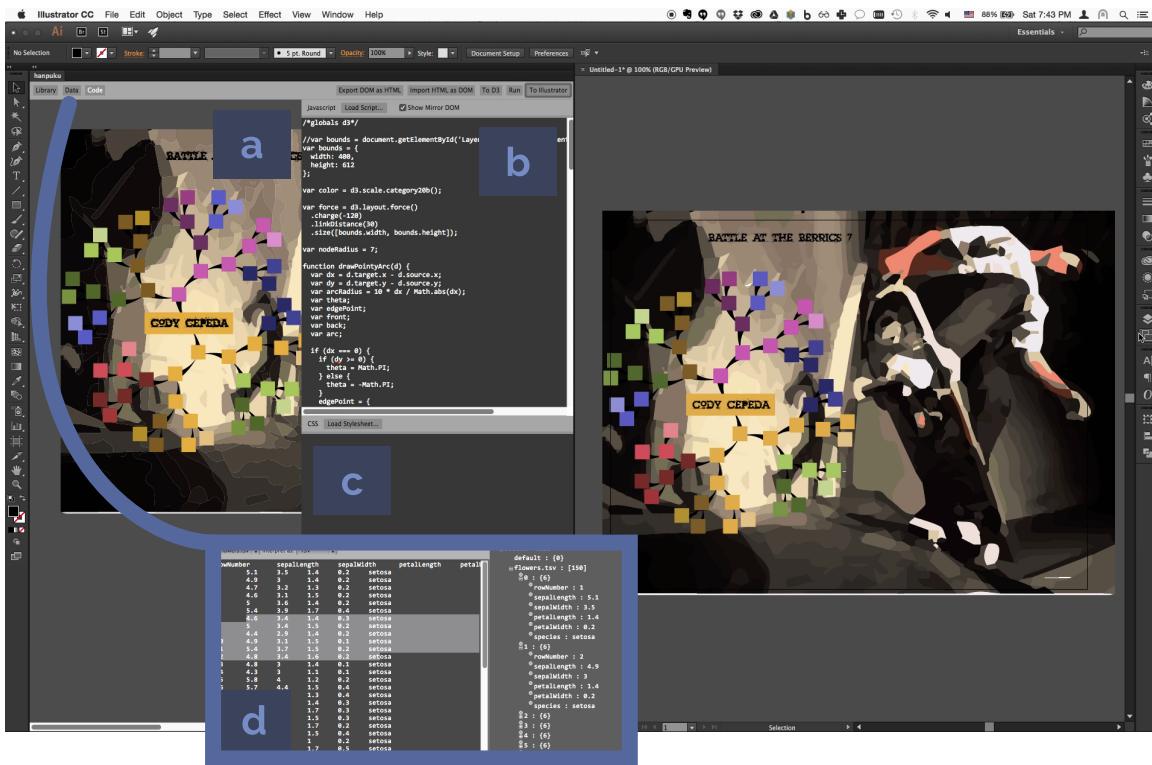


Figure 4.7. Hanpuku includes (a) a web view that mirrors the illustrator document, (b) javascript and (c) CSS code editors, and (d) a raw text data editor with an interactive preview of how D3 will parse the raw data.

4.4.1 Identify Strategies

To use Hanpuku, a designer creates an empty document in Illustrator. They drop javascript code into Hanpuku’s D3 view (Figure 4.7b), and possibly a CSS declaration (Figure 4.7c).

To match visual elements across tools and identify the corresponding changes, Hanpuku takes advantage of D3’s existing patterns: D3 scripts output an HTML DOM hierarchy; every item in the DOM that was created from data also has a data binding associated with it [38]. Illustrator uses an in-memory document hierarchy that is similar to the HTML DOM. The similarities in document structure make establishing identities between graphical elements straightforward.

D3 manages bindings between visual elements and data. D3 uses a concept of a *data join* to manage changes in data. When the main loop is executed, D3 calls a user-specified “Add” function on new data points and an “Update” function on existing data points. D3 uses a user-defined key function to identify each object to be updated, or created.

When a user runs the D3 code for the first time by pressing “To Illustrator,” the D3 code calls these “Add” functions to create a DOM hierarchy; Hanpuku then translates that into an Illustrator in-memory model and integrates the new elements into the Illustrator document. Like the D3 Deconstructor [49], Hanpuku takes advantage of the data bindings in the DOM; it translates those bindings into metadata in the Illustrator model.

After a designer has modified the visualization in Illustrator, a “To D3” merge translates the Illustrator document into an HTML DOM for D3. The data bindings from the Illustrator side are conserved back through the merge process. With the merged DOM now live in D3, the designer can modify the D3 script. When the D3 code is executed, the system executes a data join against the current dataset and the visualization. If the data have changed, new objects can be created with Add functions; Update methods are called for preexisting element. Good D3 coding style calls for Update methods not to modify anything that does not need to be edited. Therefore, any changes made in Illustrator are preserved, unless the coding has changed, and the Update function must overwrite them.

4.4.2 Merge and Reintegrate Strategies

Merge conflicts occur in Hanpuku when a user’s D3 code overwrites the same fields as were edited in Illustrator. At this time, Hanpuku employs the simplest strategy outlined in Section 4.3.2—the “To Illustrator” merge always incorporates the differences from the D3 side, and the “To D3” merge always incorporates the differences from the Illustrator side. If the Update function in the D3 code overwrites a field that was edited in Illustrator, it will be lost. In practice, this simple strategy works well. For example, a user can interact with a node-link diagram layout in the “Update” loop, and radical representational changes such as that shown in Figure 4.2 can be maintained.

Hanpuku is designed to support a workflow that produces an Illustrator document, and thus the tool is particularly careful to leave nontransferable elements in the Illustrator document intact unless the D3 script has directly changed them. A “To Illustrator” merge updates elements in place in the document hierarchy, leaving Illustrator-specific nontransferable elements untouched. In contrast, much of the nontransferable information in a D3 visualization is usually encapsulated in the D3 script itself. To avoid complications with interactive callbacks that may already exist, Hanpuku creates a fresh DOM with each “To D3” merge, containing only elements from the Illustrator document, with any data bindings attached. This way, the script can cleanly reattach its nontransferable interactive callback events.

4.4.3 Examples

In this section, we demonstrate how Hanpuku supports visualization iteration through several examples based on real-world, published infographics. In Figure 4.5, we recreate the infographic shown in Figure 1.1. Notably, the original infographic contains a name out of alphabetical order—without a bridge between a generative and a drawing tool, it is difficult to fix this problem. With Hanpuku as a bridge, however, it is straightforward to reorder the names, or to even experiment with different layouts, without discarding manual effort.

It is important to note in this example the difference between generative encodings and manual encodings. In the workflow in Figure 4.5, colors and typefaces are applied manually in Illustrator, rather than generatively using D3. Therefore, whereas additional

data can be placed appropriately in the middle of the design process—as the layout is defined generatively—any new elements will have the default color and typeface from the D3 script. Although other bridging tools, employing some of the strategies outlined above, might be able to infer the designer’s intent, Hanpuku’s simple, proof-of-concept design does not have this capability. Instead, designers can choose to continue to manually enforce color and typeface patterns for any new elements, or they can formalize those decisions in the D3 script. In either case, previous work is preserved.

As a second example, we recreated part of a diagram from the ACM SIGGRAPH 2010 Conference [9], designed by Isabel Meirelles. In this case, we were given permission to review the designer’s original process and design artifacts. Notably, she implemented much of the design using Processing code, creating new PDF files each time a generative change was made. Each generative change consequently lost any previous design work in Illustrator, requiring significant amounts of rework to reapply the manual aspects of the diagram.

We have attempted to follow her process as closely as possible in Figure 4.8 using Hanpuku (and D3 instead of Processing). Hanpuku made the iterative process far less difficult by removing the rework incurred in the original design process—generative adjustments, such as tweaking the scale in Figure 4.8(e), were straightforward to perform without having to repeat previous manual effort. Consequently, our version took very little time to create, and we were able to skip many of the steps that she was forced to take.

4.5 Other Uses of the Model

The bridge model is meant to generalize beyond the specific technical considerations of bridging D3 and Illustrator. We made some strategic decisions in implementing the bridge model in Hanpuku; bridges between other environments will similarly need other considerations. In this section, we briefly outline a hypothetical Processing-Illustrator bridge and a Processing-Photoshop bridge to discuss different instantiations of the bridge model, as well as to emphasize its generality.



Figure 4.8. Partial recreation of an ACM SIGGRAPH 2010 Conference Diagram [9]. Blue steps are done in the generative tool; green steps in the drawing tool. (a) A graphic is produced using a D3 script, encoding conference participant numbers for each group (student volunteers, conference chair, etc) with bar width, and months of participant involvement as the curved bar length. (b) Illustrator’s tools are used to adjust the colors. (c) The direction of the bars is reversed, and the width of each bar is changed to encode total hours spent by the group. (d) Colors are again adjusted in Illustrator, this time identifying appropriate PANTONE colors, and labels are added. (e) As most of the bars happen to nearly achieve right angles, the scale is increased in D3 to achieve that effect. (f) Scale labels are drawn using Illustrator’s tools.

4.5.1 Bridging Processing and Illustrator

Processing is a programming language that has gained currency among designers to create visualizations. Some designers employ a one-way workflow between Processing and Illustrator, such as the original workflow used to create the SIGGRAPH infographic in Figure 4.8. Here we describe a possible bridge between the two tools, illustrated in Figure 4.9.

The first issue to be resolved is maintaining data bindings. Although Processing can emit PDFs, it does not ordinarily maintain ID tags with the graphical objects. One

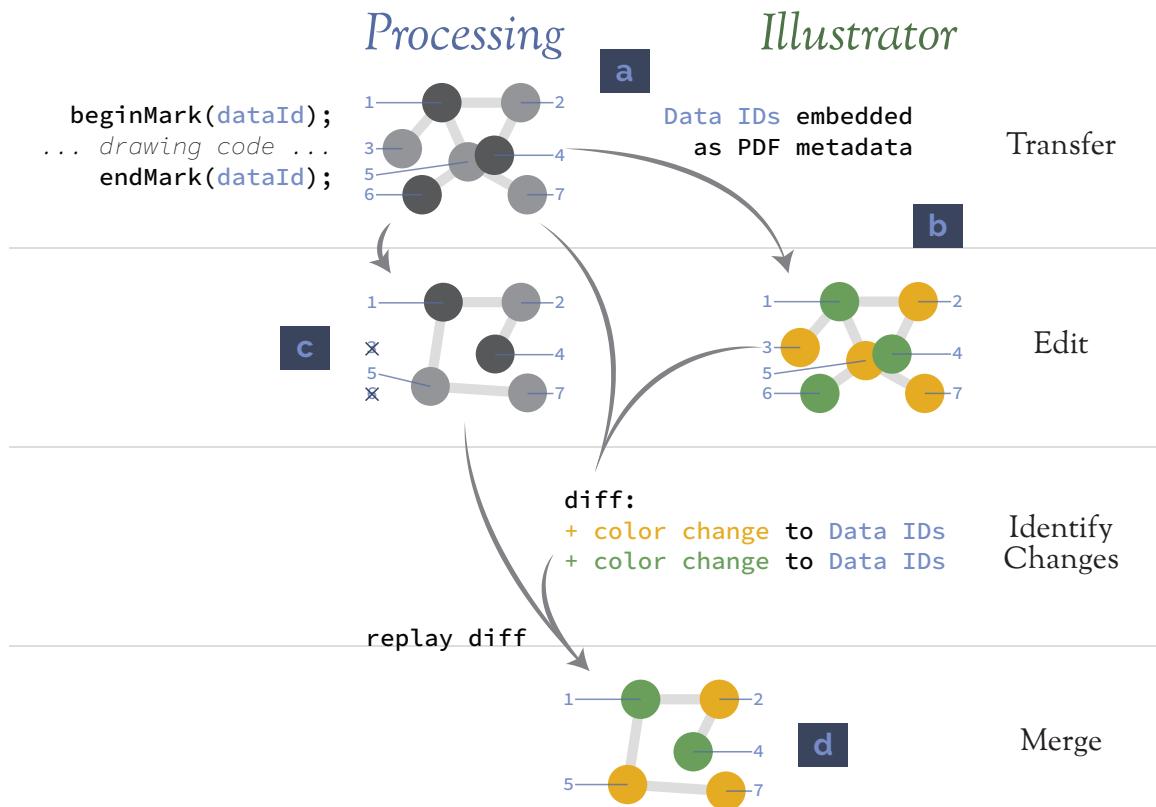


Figure 4.9. A hypothetical bridge between Processing and Illustrator. Additional syntax is added to Processing's language to associate drawing commands with data IDs (1) that are preserved in an exported PDF file. The exported file is edited in a drawing program (2). Two nodes are then deleted in the Processing sketch (3). An external merge program compares (1) and (2), identifying any changes made by a drawing program, and replays these changes on (3), producing a PDF file with both the drawing and generative changes.

workaround would be to insert ID tags into the metadata fields in the PDF output through an extension of Processing’s PDF export library. Illustrator would then edit the PDF.

For Hanpuku, we were able to use D3’s update function to use parts of existing graphics. Processing’s drawing commands do not provide this functionality. Therefore, we would use one-way integration: when a designer modified the Processing code, the bridge would propagate the changes forward to Illustrator, merging their changes on each side.

The bridge software for this scenario would then merge based on the following files: 1) an initial visualization in PDF format produced by Processing; 2) a modified PDF of the original visualization, edited in Illustrator; and 3) a modified PDF produced by Processing with additional generative edits. The bridge would then identify differences between the PDFs in 1 and 2, and merge those differences on 3, producing a PDF containing changes from both the drawing and generative tools. Rather than comparing artifact differences directly, this approach attempts to reconstruct and replay the user’s actions, similar to the way provenance mechanisms work in VisTrails [94], [95].

4.5.2 Bridging Processing and Photoshop

Another potential instantiation of the bridge model is between Processing and Photoshop, shown in Figure 4.10. A challenge with this bridge is that the common representation of the visualization between the two programs is a bitmap—there is no way to attach metadata directly to graphical elements as there is with PDF. A basic approach takes two files from Processing: a bitmap file of the visualization and an additional log file containing a many-to-many mapping between graphical element identities and related pixels. The bridge also requires a Photoshop extension that augments the tool’s existing ability to export a text history log with a one-to-many mapping between Photoshop actions and the affected pixels in a given bitmap image. Given both log files and both images, the merge program—as either a Photoshop extension or an independent program—would extend or adjust the relevant Photoshop actions so that the effects are still applied to data-bound pixels, even if those locations have changed.

Bridging tools through a bitmap presents a different set of challenges similar to those that Savva *et al.* address in the Revision system [98]. Although unambiguous operations could be supported, such as a drawing operation affecting only pixels associated with

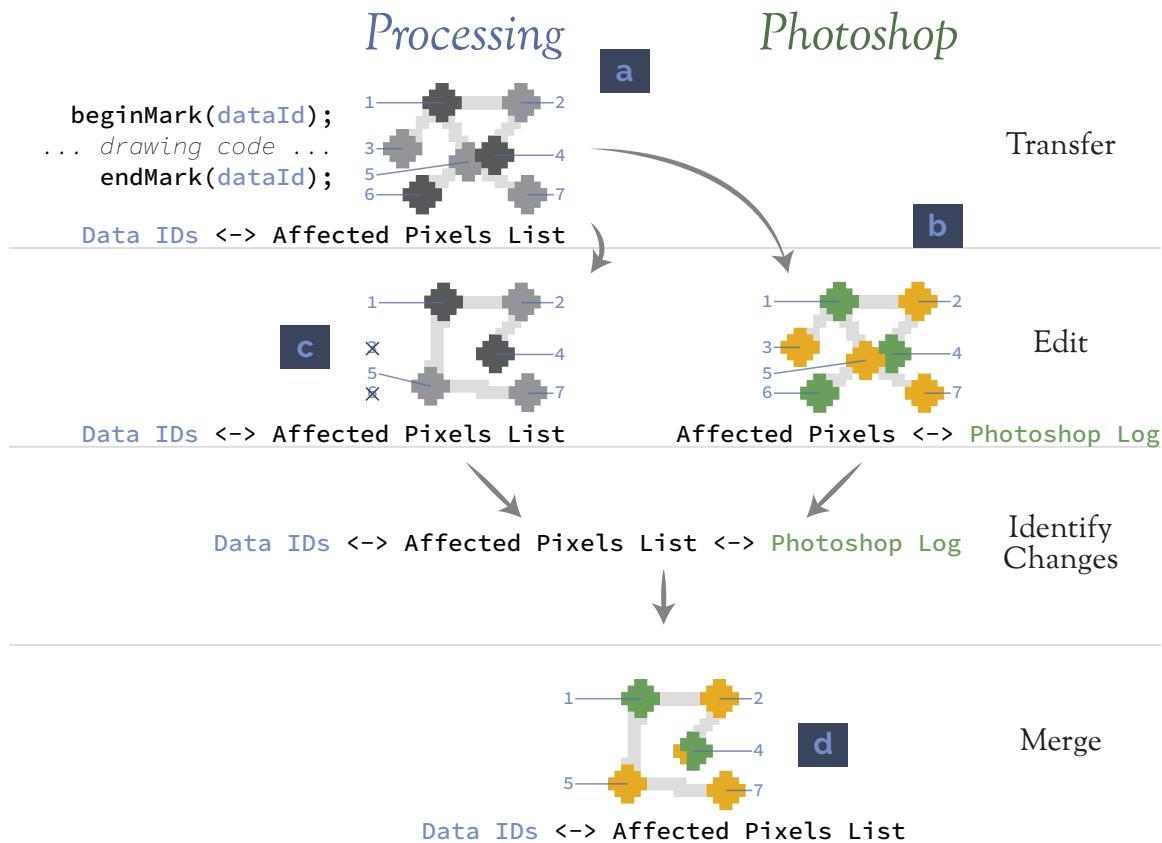


Figure 4.10. A hypothetical bridge between Processing and Photoshop. As before, additional syntax is added to Processing's language to associate drawing commands with data IDs (1) that are preserved in an external file that maps data IDs to their affected pixels. The exported bitmap is edited in a drawing program (2), and a log is created, matching the drawing program's actions to the affected pixels. Two nodes are then deleted in the Processing sketch (3). A merge program replays the drawing program's log on the updated bitmap, adjusting the effects where data pixels are missing or have moved (4). Note that, in this naive bridge, an artifact has been left on node D because a painting action meant for node E also affected it.

data, operations on pixels that correspond to multiple occluding marks present identification and conflict resolution problems, as shown in Figure 4.10. The bridge model makes the problem in this particular design clear—this bridge does not adequately provide for maintaining graphical element identities, or, more specifically, data bindings.

This is not to say that bridges cannot be constructed between bitmap-based tools, or that such bridges are not useful. For example, the bridge model describes how it may be possible to perform data-aware merges between 3D volume renderings and postprocessing effects performed on 2D rasterized graphics. In the case of this basic model, improvements might include more intelligent drawing tools that use the associated data IDs to constrain painting effects, limiting their influence to only the relevant pixels. A different approach would be to utilize layers instead of pixel maps to keep element identities distinct.

4.6 Discussion and Future Work

The bridge model describes how different modes of work can be used in the same workflow, fitting the best tool to the best task. For example, it is completely possible to perform typography and label placement adjustments with D3, or to manually draw data-driven marks in a chart with Illustrator. We commonly see examples of each of these in practice. Generative visualization tools and drawing tools overlap in some of their technical capabilities, but, as we have shown, using both modes of work in the same workflow can constrain the ability to iterate. The bridge model helps identify what is common between each way of working, what is distinct, and how these differences can be managed through the design workflow. The model is not a panacea—it does not make all tools come together. Rather, we propose the bridge model as a different way to think about how we create visualization tools that are rich, flexible, and efficient across the broad range of tasks performed throughout the design process.

Although we have mainly discussed bridges between distinct software tools, the concepts are still important for all-in-one tools that attempt to encompass both generative and drawing modes of working such as iVisDesigner [44]. Internally, such tools must still consider the effects of generative operations on existing drawings, as well as the effects of drawing operations on existing generative specifications.

The model does not, however, encompass approaches like CSS that seek to separate design tasks, albeit it in a generative way. Although stylesheets separate the stylistic and functional aspects of a visualization, a stylesheet does not contain an independent representation—there is nothing to merge. The model instead describes workflows that include a variety of tools, each of which operates independently on the visualization.

With any bridge, some sacrifices are likely unavoidable. As we discussed in Section 4.5.2, a hypothetical Processing-Photoshop bridge is likely to be constrained by the technical limitations of each tool, including what aspects of the visualization can be shared and what kinds of changes can be merged. Both hypothetical bridges, as well as Hanpuku itself, enable a more iterative workflow, but limitations remain. In all the bridges we have discussed, users likely need to understand each tools' underlying document structure in order to effectively prepare for merging.

The bridge model exposes ways that visualization toolkits and drawing programs can make themselves more interoperable. Our choice of specific tools is not an accident: we repeatedly use Illustrator as an example throughout this paper because it is a widely used drawing program that preserves arbitrary metadata attached to elements in its documents. Data bindings can be preserved when visualizations are edited. We also focus heavily on D3 because it can perform data joins with existing graphical elements such that manual changes done in a drawing program can survive generative updates. For now, the pass-through relationship between these two tools makes interoperability straightforward.

Additional benefits can come from their flexibility. For example, Illustrator currently allows users to select elements by color and other visual properties. Because our system now adds metadata and identify information to visual objects, it would be straightforward to support user selections based on data, similar to the GUESS system [99].

Going forward, we are interested in instantiating other bridges, such as those we describe in Section 4.5. We believe these, and other variations, will become easier to design and implement as the inputs and outputs of proprietary tools become more easily accessible. Another interesting line of future work is to consider a format standard for visualizations that explicitly supports metadata attached to graphical elements. This standard would support writing these files, reading these files, and editing the files, as well as explicitly making use of the metadata in visualization tools.

CHAPTER 5

JACOB'S LADDER: THE USER IMPLICATIONS OF LEVERAGING GRAPH PIVOTS

Unlike the rapidly expanding set of tools for creating visual representations of data, there is far less software support for reimagining the shape of the data that will be visualized. Data wrangling tools are particularly undersupported for nontabular data types, such as graphs. Consequently, we present two works that inform how software can support reshaping graph data. As there is less existing software, our intended user population is broader than our earlier focus on graphic designers. Instead, our objective is to explore the space of what is possible with respect to general network wrangling interfaces, that can inform future interfaces that are more targeted to specific users with specific needs and expertise.

The work in this chapter explores in detail the relationship between two specific wrangling operations that are relevant for graphs. This work is formative for the broader set of graph wrangling operations presented in Chapter 6.

This chapter reports on an aggregate visual technique that simplifies extracting a subgraph down to two operations—pivots and filters—that is agnostic to both the data abstraction and its visual complexity scales independent of the size of the graph. The system’s design, as well as its qualitative evaluation with users, clarifies when and how the user’s intent in a series of pivots is ambiguous, and, more usefully, when it is not. Reflections on our results show how, in the event of an ambiguous case, this practical operation could be further extended into “smart pivots” that anticipate the user’s intent beyond the current step. Our reflections also reveal ways that a series of graph pivots can expose the semantics of the data from the user’s perspective, and how this information could be leveraged to create adaptive data abstractions that do not rely as heavily on a system designer to create a comprehensive abstraction that anticipates all the user’s tasks.

5.1 Motivation

Graph-based data systems are everywhere. Once thought of as a fallback option for data that could not be finagled into a relational database, graphs are now emerging as the data format of choice, not only for overtly networked systems, such as social networks and citation networks, but also for biological systems, traffic patterns, and all of human knowledge [100], [101].

For the domain experts who will ultimately be using these data, however, graph databases offer only a new spin on a classic conundrum: how to answer new and evolving questions. There are countless ways to explore graph data programmatically, but command-based queries often exceed the technical capabilities of end users. Conversely, reporting tools can provide answers to a predetermined set of frequently asked questions, but this approach relies on a technical expert to foresee and interpret the users' needs.

This latter influence, in fact, is nearly impossible to erase since it is a technical expert who must impose the initial data abstraction that will dictate how all subsequent queries will be executed. This choice of abstraction, which can be highly subjective, crucially determines how the data can be used. An ill-informed choice can dramatically reduce the efficiency and accessibility of the data for the users' most high-value tasks. It can preclude certain visualization and exploration tools from being used at all.

The ultimate goal of this work is to identify first steps toward severing the dependence of a graph's utility on its initial data abstraction. To do this, we focus on a graph-based operation known as a "pivot." The pivot allows users to evaluate one set of nodes in the context of some subset of its neighbors. It offers the unique advantages of its visual complexity being agnostic to the graph's size, and its simplicity making it compatible with.

We present an overview visual technique, dubbed Jacob's Ladder, that allows users to traverse, query, and extract subsections of a graph using only chained sequences of pivots and filters. We report on how the strengths and weaknesses of this technique's design influence users' ability to grok the underlying data abstraction. Using this tool, we are able to observe where and how ambiguity can arise in a series of pivots. We propose "smart pivot" heuristics as a means of overcoming these natural ambiguities. Finally, we discuss the potential of graph pivots in exposing inconsistencies between the data abstraction and the users' needs. We outline examples for how these pivots could inform an **adaptive**

abstraction, in which a system reshapes its schema on the fly to become more semantically relevant and efficient as questions are asked, rather than rely on a technician’s *a priori* intuition about what future users’ questions might be.

5.2 The Pivot

As shown in Figure 5.1, we define a **pivot** as an operation in which a user navigates from a set of **seed** nodes S to another set of **target** nodes T , in which every target node $t \in T$ has a connection to at least one seed node $s \in S$. Note that the sets of seed and target nodes need not have any internal structural relationship; S and T may be arbitrarily large, and the members of each set may be entirely disconnected from members of the same set.

This operation can be chained together, with the target nodes T_0 from the previous step serving as the seed nodes in the current step: $T_0 = S_1$. For example, in a simple social network of friends, the “friends of friends” for any node can be found by performing two pivots. Although the pivot, by itself, creates a fairly simplistic fan-out effect, it is considerably more expressive with these common extensions:

5.2.1 Categorical Pivoting

A pivot does not necessarily need to swing out to all of the connected neighbors of the seed set. When a graph is heterogeneous, consisting of multiple types of nodes and edges, a pivot can swing out to only nodes of a certain type or along edges of a certain type or both. For example, in the data system for a large hospital, a doctor, Alice, might want to find out which other doctors her patients are seeing. By first finding herself in the data

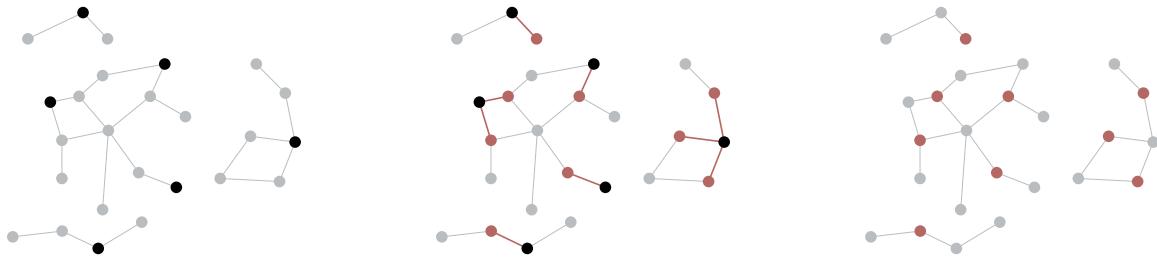


Figure 5.1. The basic pivot: On the left, the dark set of seed nodes are selected. The selection then swings out to a subset of neighboring target nodes (middle, red), resulting in a new set of seed nodes (right).

system (for example, $D_0 = \{Alice\}$), she can pivot out to all of her patients (for example, $P = \{Bob, Carol\}$), and then pivot back to all of the doctors associated with those patients (for example, $D_1 = \{Alice, Dave, Eve\}$).

5.2.2 Filtering

After any pivot, if a graph is multivariate, users may want to filter the subgraph of seed and neighbor nodes before the next pivot is performed. For example, instead of finding the other doctors that all of her patients are seeing, maybe Alice only needs to find the other doctors of her female patients. In this case, the set of patient nodes can be filtered down to just the female patients (for example, $P' = \{Carol\}$) before performing the second pivot back to doctors (for example, $D_1 = \{Alice, Eve\}$). This filtering can be based on node attributes, edge attributes, the number of incoming or outgoing edges, or any other metric that can be computed against the subgraph of seed and neighbor nodes. Filtering, as well as categorical pivoting, makes it possible to perform multiple consecutive pivots without continually increasing the number of nodes involved in each pivot, achieving a fan-in effect.

For our purposes, we will describe **direct** filters as those performed directly on a set of nodes, such as filtering patients nodes by their sex attribute. We will describe **connective** filters as those that indirectly filter a different set of nodes, such as the second set of doctors (D_1) having been being filtered indirectly by their patients' sex.

5.3 Why the Pivot?

There are three reasons why the pivot stands out as a potential lynchpin of usable graph exploration: they enable extracting manageable subgraphs, they can provide deep coverage of a graph, and they can operate agnostic to any graph schema.

5.3.1 Manageable Subgraphs

As graph data stores become larger and increasingly complex, the assumption that the graph can be loaded into memory and visualized in its entirety will eventually stumble. Thus, in isolation, systems such as Gephi [61], g-Miner [102], Tulip [103], and a host of others [60], [104], [105] that rely on a holistic display of the graph, will fail to scale.

In contrast, the pivot provides a consistent and easy-to-interpret means of displaying a

partial view of the underlying graph. In order to perform consecutive pivots, users need to see only their current subgraph of seed and target nodes, and the options for where they can pivot next. As we show in this chapter, novice users can extract and understand meaningful subsets of a graph by employing only pivots and filters, even though the topology of individual nodes and edges remains hidden. The pivot can be executed and visualized without having to account for the size and complexity of the entire graph.

5.3.2 Coverage

Before users can analyze data, they must first be able to isolate the data that are relevant to their questions. This challenge can be steep when users do not have flexible access to the underlying data system. This work was motivated, in part, by a series of interviews with a group of bank employees who regularly interacted with a large reporting tool system. Users expressed consistent frustration with not being able to investigate connections between elements that were, in fact, connected in the underlying data. The phrase we heard over and over again was “I can’t get from _____ to _____.”

The pivot operation addresses this difficulty by allowing movement to take place across any existing connections in the underlying graph. So long as the graph is connected, pivoting allows users to navigate between any two nodes in the system. This navigation might not precisely represent the intent of the user’s ultimate objective, but as we will discuss in subsequent sections, it can significantly narrow down the space of what that objective might be.

5.3.3 Abstraction Agnostic

It is easy to underestimate the subjectivity of a graph’s data abstraction [10]. Decisions must be made about what will be a node, what will be an edge, and what will be the attributes of those nodes and edges. A city, for example, could be easily viewed as an attribute of a university node (i.e., the city in which that university is located). However, it might make more sense for each city to be its own node in the graph, and for the location of a university to be represented by an edge to that city node. Flipping the notion of nodes and edges entirely can also produce a more intuitive graph [27]. The overall utility of a graph can depend heavily on how well the data abstraction matches the queries that will ultimately be run against it. We refer to these arbitrary abstraction decisions as the **schema**

of the graph, including what a node is; what an edge is; what node or edge types exist; whether a graph is undirected, directed, or mixed; whether parallel edges are allowed; whether structures such as supernodes or hyperedges are supported; and whether nodes and/or edges are multivariate.

The advantage of the pivot is that it is simple enough to work on any graph schema, as long as one has been identified. However a graph is represented internally—whether an adjacency matrix, a node-link list, or modeled from a relational database [2], [106]—the concept of a pivot is still valid. This universal applicability can be contrasted with techniques that require restrictive assumptions about what the underlying data will be, and how it will be organized [64], [66], [69]. Unlike pivots, techniques that require schema definitions beyond simply having nodes and edges are immediately ruled out when a dataset does not match the needed abstraction.

Pivots also have the potential to reveal the semantics of the user’s tasks. Consider the hospital example: if our doctors need to find the other doctors their patients are seeing, they can find themselves, pivot to patient nodes, and then pivot back to doctor nodes. The series of pivots explicitly encodes the semantics of the doctor’s intent in a very simple way that could be collected to better understand users’ needs and improve the underlying data abstraction. We discuss two specific ways this could happen in more detail in Section 5.6.4.

5.4 Why Not the Pivot?

The obvious drawback of formulating meaningful queries or explorations by chaining together a series of pivots is that each pivot operation is atomic. A single pivot sees only the seed nodes and their immediate neighbors, not the series of pivots that led up to that point. The ramifications of this limit can be illustrated by the following scenario: consider a doctor (D') who would like to know what kinds of treatments he or she has prescribed to patients (P_0) with a particular insurance provider (I'). The resulting sequence of pivots, shown in Figure 5.2, might start with doctors locating themselves in the data system, pivoting out to their patients, then pivoting out to the insurance providers of those patients, and filtering that list down to the provider of interest. But now our doctor has a problem. Pivoting back to patients (P_1) will yield a list of all the patients who have that insurance provider, not necessarily *that* doctor’s patients who have that insurance

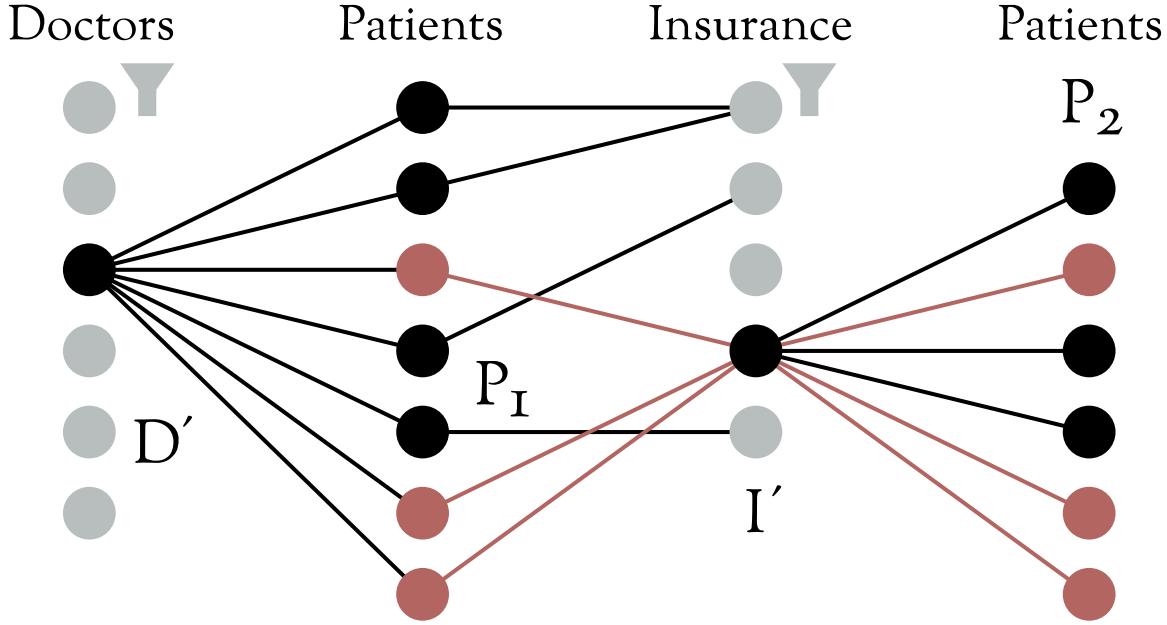


Figure 5.2. To find patients of a given doctor who are covered by a certain insurance provider, the user starts by filtering the doctor nodes down to a single doctor (D'). The user then pivots to patients (P_1), then to insurance providers (I' , where another filter is applied), and then back to patients (P_2). However, when the user pivots back to patients, the pivot returns all of the patients with the specified insurance provider, but not necessarily patients of the original doctor (in red).

provider. That next pivot does not inherently understand that the pool of patients was already narrowed down and, as a result, the pivot sequence starts to diverge from the intent of the query. This difficulty can be reproduced in a system like GraphTrail [68], which looks only at a single pivot at time.

This speaks to one of our main contributions: is it possible to make these pivots smarter, so that their meaning is always unambiguous? The benefits of such an improvement are twofold. Users would, of course, have a more expressive, powerful way to query and navigate a database. Additionally, clearing up this ambiguity supports our final major contribution: the simple, unambiguous nature of a series of pivots will allow researchers and systems to collect data that directly exposes what users are actually looking for.

5.5 Evaluating Pivots

The critical weakness of pivots, as we have discussed, lies in the ambiguity that arises as the user traverses deeper into the graph with a series of pivots. Where does this ambiguity come from, and what could a user do to help clarify it?

To better understand the translation between real-world questions and sequences of graph pivots, we implemented the pivot operation as a web-based front-end to a Titan graph database, using the Gremlin query language. Although traversal languages such as Gremlin are well suited to computing pivots in our case, we attempt to focus on how users understand pivots, rather than how to compute pivots efficiently—these technology choices may not be appropriate or efficient for every graph data abstraction.

5.5.1 Interface Design

The resulting application, dubbed Jacob’s Ladder, is shown in Figures 5.3 and 5.4. It allows users to select an initial set of seed nodes, apply filters to the set, and then pivot to a new set of connected nodes. This process can be repeated as many times as needed, with a summary of previous pivots and filters represented as lines across the top of the screen for reference. At any point, users can undo a pivot, an associated filter, or clear their history of pivots and start from scratch. As this work focuses on understanding the role of previous filters in the context of subsequent pivots, Figure 5.4 shows how filters can be inspected and edited individually using filter lines, or toggled across the board using a global scope button in the search bar.

Because Jacob’s Ladder operates at such a simple, aggregate level, it completely bypasses the scale problems of traditional graph visualization systems. The required screen real estate is a function of the various types of nodes and edges in the schema of the graph, not the actual number of nodes and edges. Consequently, there is no visual limitation with respect to the actual size of the graph.

The interface is designed primarily for subgraph extraction. As a user pivots through the graph, the consecutive sets of seed nodes form a smaller, more manageable subgraph that can be downloaded as common graph formats that include the edges used in the traversal. Ideally, Jacob’s Ladder should be used to extract a meaningful, manageable subgraph from a large database for closer analysis in other tools, such as Gephi [61]—

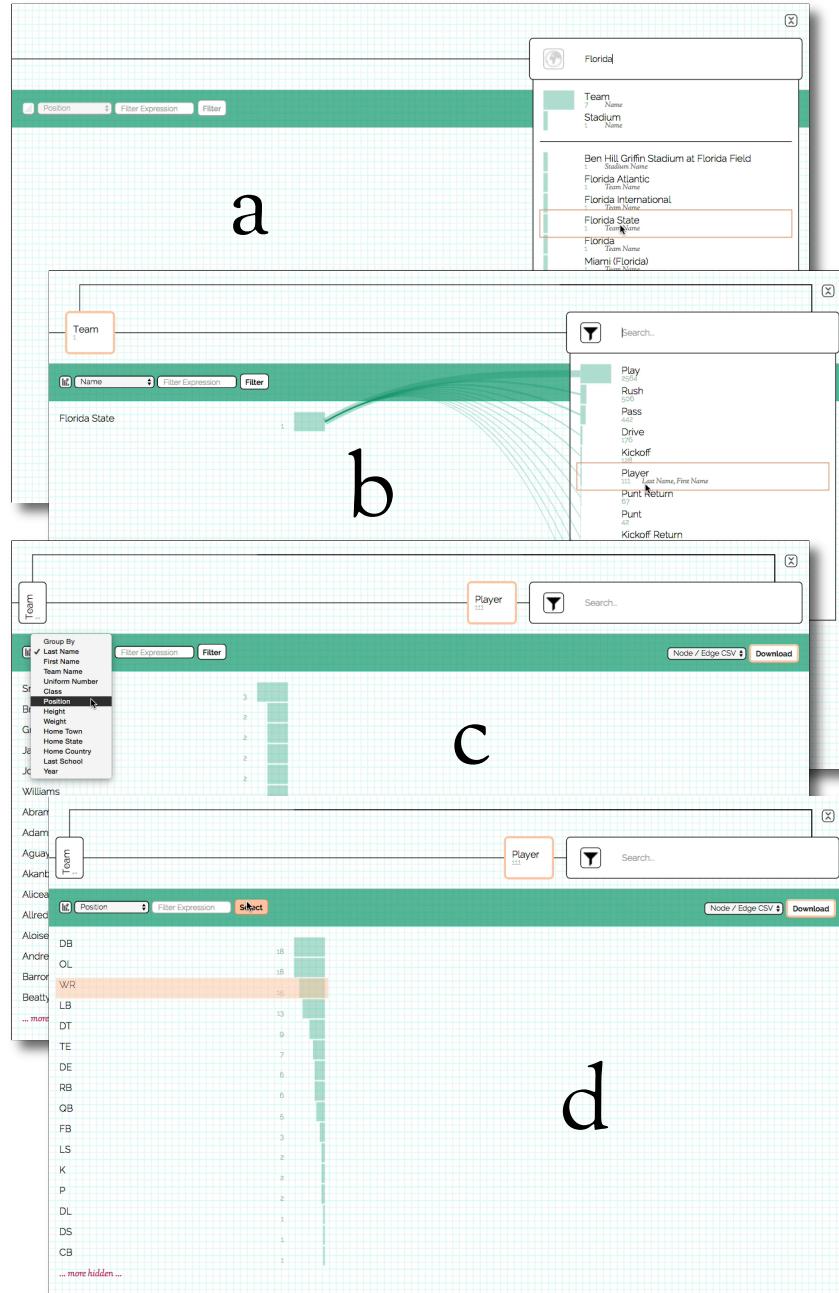


Figure 5.3. Jacob's Ladder allows users to pivot from one category of nodes to another. A search box (a) shows search matches in the menu below. Matching nodes can be selected in aggregate, based on node type ("Team" or "Stadium" above the line), or individually based on value (below the line). Once a set of nodes has been selected, it is displayed as a histogram on the left of the search field (b). Subsequent searches are limited to the set of nodes that are connected to the previous selection, with line thickness encoding potential connections. The histogram supports regrouping and sorting (c), as well as selecting and filtering nodes (d) based on node attributes. The series of actions depicted are as follows: a) Florida State is selected, b) the user pivots to Florida State's players, c) players are grouped by position, and d) the wide receivers ("WR") are selected.

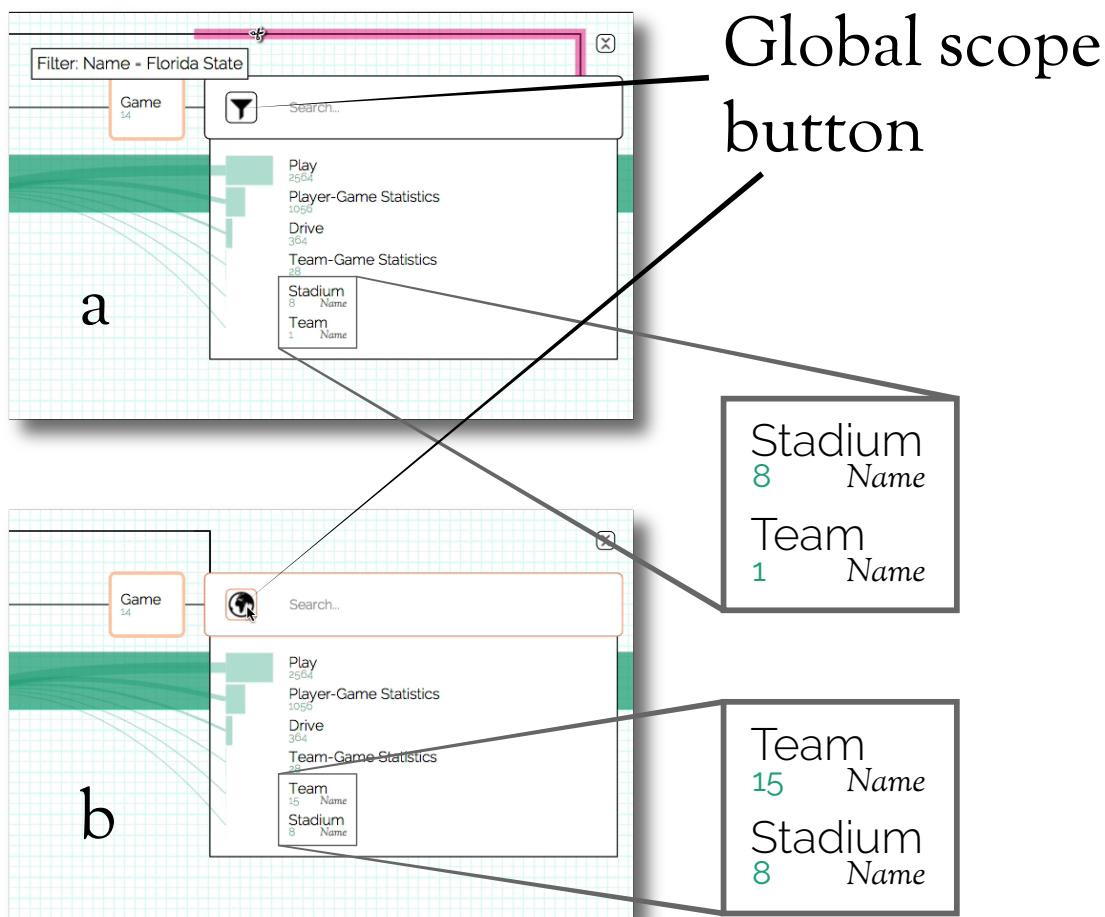


Figure 5.4. When filters are applied to a selection of nodes, a line is placed at the top of the interface to indicate that the filter is active (a). Because the difference between fanning in and fanning out is so critical, it can be toggled in two ways: a global scope button inside the search field removes or restores all filters, or individual filters can be removed by “snipping” the line. Note how, in (a), only one Team node can be selected, because the filter is still in place. Clicking “Team” will fan in. In (b), because the global scope button has been clicked, the set of available Team nodes is larger; the filter has been removed. Clicking “Team” will fan out.

Jacob’s Ladder is not designed to support low-level, per-node analysis. Visualizations of individual nodes and edges are deliberately omitted from its interface.

5.5.2 Lab Tests and Design Adjustments

The limited scope of Jacob’s Ladder presents an opportunity to study graph pivots in relative isolation. Over the course of three months, we loaded Jacob’s Ladder with a wide range of graph datasets, from IMDB’s movie graph to financial and medical data, and tested where and how ambiguities arise in the pivoting process.

We can further simplify our discussion of pivots if we treat edges as distinct entities—our experience designing Jacob’s Ladder itself yielded this insight. Where relevant, to allow a simpler interface, the tool reinterprets any edges as interleaving nodes. For example, if a relationship edge in a social network has attributes, it would be replaced with an edge, a node containing those attributes, and another edge. Early prototypes of the system maintained a distinction between the two; however, the redundancy became obvious very quickly. For the sake of simplicity, we chose to avoid additional UI elements that differentiate between data on nodes and edges.

As we designed Jacob’s Ladder and used it to explore these datasets in the lab, we came to develop a prediction that **ambiguity arises in a series of pivots only when the series includes both filters and cycles.**

5.5.3 Qualitative Evaluation

To learn how users understand graph pivots, whether they are useful, where ambiguity arises, and how pivots reveal a user’s semantic understanding of a graph, we conducted an informal, qualitative study of users. Because we had developed some initial predictions, we were careful to design our experiment to evaluate those predictions explicitly. We were also careful to watch for trends that we did not anticipate, including unexpected or surprising behavior.

5.5.3.1 Participants

Initially, this system was developed for internal use within a large financial institution, and its use in a hospital database was also anticipated. Unfortunately, due to confidential data and legal complexities, we were not able to gain access to real users in or out of their

native work environment.

Consequently, we selected a publicly available NCAA American College Football dataset. This dataset was interpreted as a graph with many node types, such as Players, Teams, Games, Stadiums, Conferences, etc. As shown in Table 5.1, a diverse range of participants was selected, from graduate students who have experience with graph data but minimal knowledge of football, to passionate football fans with little to no graph exposure. All were asked to self-report their understanding or expertise with regard to graph data and American football.

Table 5.1. Details about each of the 11 participants, including their relative expertise and suggestive indicators that emerged as the study progressed. Participants are classified as “Novices” when they self-reported little to no understanding or prior experience; “Intermediate” when they reported or demonstrated some familiarity, but no strong interest or experience; and “Expert” when they reported or demonstrated strong interest or experience. * This participant briefly clicked the button at an inappropriate point, but quickly reverted the decision. † This participant specifically asked about the filter lines, so they were given an explanation. ‡ Technically, this participant found the “fumble return” attribute—a different attribute of the Player-Game Statistics node type. Structurally, this is equivalent.

Participant	Participant Details		Filter Scope		Topology		Other	
	College football interest / exposure	Interest / exposure to graph data	Correctly clicked the global scope button	Incorrectly clicked the global scope button	Reported / demonstrated confusion about the meaning of filter lines	Was able to find the fumble attribute	Reported / demonstrated confusion about the graph topology	Typed a search for a node class (e.g. “Team”) instead of a value (e.g. “Smith”)
1	Novice	Expert	Yes	No	No	No	Yes	Yes
2	Intermediate	Expert	Yes	No	No	No	Yes	Yes
3	Novice	Intermediate	Yes	No	Yes	No	Yes	No
4	Novice	Expert	Yes	No	Yes	No	Yes	Yes
5	Expert	Expert	No	No	No	No	Yes	Yes
6	Novice	Intermediate	No	No	Yes	Yes‡	Yes	Yes
7	Novice	Expert	No	No*	No†	No	Yes	No
8	Intermediate	Expert	No	Yes	No	No	Yes	Yes
9	Novice	Expert	Yes	Yes	No	No	Yes	Yes
10	Expert	Novice	Yes	No	Yes	No	Yes	No
11	Expert	Novice	No	No	No	No	Yes	No

5.5.3.2 Hypotheses and Tasks

The interface of Jacob’s Ladder provided an opportunity to assess both the power and limitations of graph pivots. Specifically, we designed tasks that address the following research questions.

- 1) Can the graph pivot enable technical and domain novices to extract meaningful subsets of a large graph, even when traditional instance-level visualizations are not included?
- 2) How does the technique obscure the topology of the database?
- 3) Does the user understand the scope of the next pivot? Is the interface sufficient to resolve ambiguous cases?

For each question, we assigned the following corresponding tasks.

- 1) With minimal introduction, observe whether users can select all the quarterbacks on a specific team (users must filter, then pivot, then filter).
- 2) Observe whether users can anticipate where to find the “fumble” attribute without help (filter, pivot, connective filter, pivot back).
- 3) Given a specific team, observe whether users can select the set of teams that the seed team beat. This task requires the user to either remove the initial filter, or toggle the global scope button (filter, pivot, filter, toggle scope, pivot back).

It is important to note that these tasks were designed to aggressively discover the limitations of our technique, rather than merely serve as existence proofs of where it succeeds [107]. Consequently, we focus on these limitations in discussing our observations, as they form the seeds for reflection in Section 5.6.

5.5.3.3 Experiment

Each 30-minute session involved the participant and the researcher seated at mirrored displays, each with a mouse and keyboard. In addition to the researcher’s notes, screen capture software was used to record the user’s actions and voice. Where necessary, participants were first given a brief introduction to the dataset, including explanations about

college football and/or graph data. Participants were then given a 5-minute introduction to the tool, including two brief demonstrations of the system, similar to Tasks 2 and 3 that the participants would later be given. As we were particularly interested in understanding whether users could decipher the scope of their applied filters on their own, only the function of the global scope button was explained and demonstrated; the filter lines were ignored. Next, participants were given the three tasks in order. Finally, users were given time to explore the data freely, and comments, questions, and discussion were encouraged. As we were particularly interested in understanding whether users could decipher the scope of their applied filters on their own, only the function of the global scope button was explained and demonstrated; the filter lines were ignored.

5.5.3.4 Task 1 Observations

All participants were able to accomplish the initial filter and pivot in Task 1 with ease. Interestingly, although most users were able to perform the final filter without difficulty, many users were not aware that they had already successfully completed Task 1.

Participants navigated from Team nodes to Player nodes, and the interface initially displayed the principally descriptive attribute of the nodes they had selected: in this case, player names. Users would switch the histogram to group players by their position attribute, and then filter players by selecting the “QB,” or quarterback position. At this point, users had technically succeeded in selecting the quarterback player nodes, but because the histogram displayed only one “QB” bin, they often were not aware that they were finished until they switched back to the player name attribute.

5.5.3.5 Task 2 Observations

In Task 2, participants were asked to find the set of players on a team of their choice that had fumbled the ball at some point in the season. This question was difficult for all participants to perform because it required traversing from Team nodes to Player nodes, and then to Player-Game Statistics nodes. No “fumble” attribute was directly visible from the Player nodes. Therefore, only one participant was able to come close to successfully navigating to this set.

5.5.3.6 Task 3 Observations

The third task was to identify the set of teams that a team of their choice beat. This task was an opportunity to observe whether users understood the scope of the filters. We specifically tracked whether participants clicked the global scope button at the correct point in the task.

Although a very similar demonstration was shown to all participants at the beginning of the study, they displayed mixed results in their success. The fact that filters were still in place as they pivoted back to a previous node type (Team → Game → Team) appeared to be somewhat unintuitive.

5.5.3.7 Incidental Observations

Although the study was somewhat controlled by the tasks issued to the participants, we were careful to observe whether additional patterns surfaced.

We observed some confusion between the filter functionality of the tool and the pivot functionality. Participants would sometimes go to the search box when they meant to filter, or to the filter controls when they meant to pivot. This behavior reveals a design flaw in the Jacob's Ladder interface: the search field technically applies a filter, as do the more traditional filter controls. Applying filters in multiple locations in the interface caused some confusion.

Another unexpected pattern that we observed was that participants would often enter a node class name, such as “Team,” in the search box instead of attribute values. Because the system expects attribute queries only in the search field, it would try to find node attributes that match “Team” instead of finding nodes by class name. “Team” nodes would subsequently disappear from the menu, resulting in confusion.

Finally, we were surprised by how well the participants were able to interpret the meaning of their current selection. Particularly during Task 2, participants were observed performing long chains of pivots in search of the “fumble” attribute. Almost all participants were cognizant of the fact that they needed to be somewhere else in the graph. Impressively, almost all participants were able to articulate the meaning of their current selection when asked, even if many pivots were involved. For example, during Task 3, Participant 5 navigated from Team (Ohio State) → Team-Game Stats (WIN) → Team (failed

to remove the Ohio State filter) → Game → Team-Game Stats (WIN filter still applied, grouped by team name). When asked, he correctly interpreted the visible set as any team that had won a game that Ohio State was involved in.

5.6 Discussion

Overall, our results support, with some qualifications, our hypothesis that the graph pivot is a powerful tool that can enable novice or disinterested users to extract meaningful subsets of the graph without visualizing low-level graph topology. The tests also confirmed our predictions about where ambiguity arises in a series of pivots. Finally, the tests showed that a series of pivots can expose the user’s understanding of the semantics of the data in a way that could easily allow for a system to reshape its data abstraction based on user behavior.

5.6.1 When Are Other Views Needed?

Our tests confirmed Hypothesis 1, that the simple pivot operation can empower novice users to extract meaningful subsets from large graphs. Our aggregate visual technique that lists each pivot at the top of the interface circumvents the scalability issues of traditional graph visualizations by avoiding local topology altogether—we demonstrate that, for many graph data tasks, it is not necessary to render detailed node-link diagrams. Working at the aggregate level that pivots enable is often sufficient and intuitive for many graph visualization tasks.

Although visualization of local topology is not necessary for many tasks, we also learned from participants’ performance in Task 2 that our particular implementation of the graph pivot in Jacob’s Ladder obscures the global topology of the graph—we were perhaps too minimal in its design. An even higher level overview of the schema of the graph, such as the technique demonstrated by Van den Elzen *et al.* [108], is still likely necessary to help the user plan how to pivot and filter toward node types and attributes of interest, especially for unfamiliar datasets.

5.6.2 Delineating Where Ambiguity Occurs

Task 3 confirmed our initial predictions about where ambiguity arises. As shown in Figure 5.5, the meaning of a user’s pivot is always clear unless a cycle and a filter are

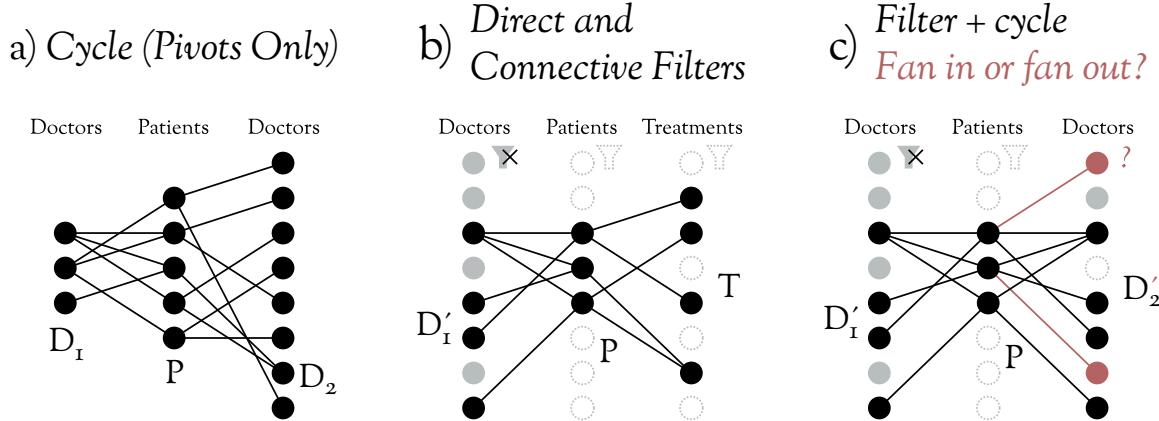


Figure 5.5. We can see that ambiguity in a series of pivots arises only when filters and cycles occur in the same traversal; when cycles are present without filters (a), the only logical action is to fan out. When filters are present, without cycles (b), the only logical action is to keep the filter in place and fan in. However, when both are present (c), it is not clear whether to fan in or fan out: should the initial filter on the Doctor nodes be reapplied?

encountered together.

5.6.2.1 Pivots Only

In our initial explorations of the data before the user study, we first discovered was that it was impossible to create ambiguity by performing pivots without filters (Figure 5.5a). If no filters are enacted during a series of pivots, then the only logical outcome of the next pivot is to return all of the connected nodes of the specified category; however, pure pivots without filters may not be very useful.

5.6.2.2 Pivots and Filters

When a filter is enacted at a certain point in the pivot sequence, it manifests in two ways (Figure 5.5b). The first is as a direct filter against the category to which it has been applied. For example, if users want to see all of the doctors who are women in a medical database, they can group doctors D by a “gender” attribute, and select only the group of women, resulting in a subset D' . This filter is applied directly to the doctor category, and depends only on an attribute of the doctor nodes.

However, when users pivot from doctors to their patients, that gender filter against the doctor category serves a dual function as a connective filter against the patient category;

the resulting set of patients P would likely be larger, had the filter not been applied to D' . This connective filter has an increasingly indirect effect on each category of nodes that is visited after the filter is applied.

Our study showed that these indirect effects were not difficult to understand. Even though users sometimes became “stuck” in their exploration of the football dataset after a long series of pivots and filters, they could still generally articulate the meaning of the nodes that they had arrived at, including the effects of upstream filters. Additionally, so long as no category is visited more than once after the filter has been applied, the only logical outcome is still to pivot out across all of the available connections. There is no previous interaction with that next category to suggest otherwise, and thus, no ambiguity.

5.6.2.3 Pivots and Filters and Cycles

As illustrated in Figure 5.5c, ambiguity arises only when a given category of nodes is visited more than once after a filter has been applied. When this occurs, the revisited category is carrying with it a set of direct filters that the user might or might not want to restrict the current pivot operation. Continuing with our previous example, where we filtered the list of doctors D_0 to only see women (D'_0), and pivoted out to patients (P), if we then pivot back to doctors, which doctors does the user want to see? We know that the user wants to see the doctors (D_1) associated with those patients; however, should the original direct filter on the “gender” attribute remain for this second set (D'_1)?

More generally, these options can be described as follows.

- 1) Perform the pivot operation normally, swinging out to all of the connected neighbors that match the specified category, keeping connective filter effects, but without re-applying previous direct filters (Fan-out pivot).
- 2) Further restrict the nodes returned by a normal pivot—retaining both connective filter effects from other categories, as well as re-applying previous direct filters on that category (Fan-in pivot).

5.6.3 Implications for Smart Pivots

The question, then, is how to determine which of these options the user intends and, if it is the latter option, whether the user intends to retain all of the direct and connective

filters that have been applied, or only a subset of them. Although it is not possible to guess the exact intent of the user at every turn, we can better narrow down this problem space to isolate the exact source of the ambiguity. Our experience with Jacob’s Ladder and its user tests are suggestive of heuristics to follow for intuitive behavior in ambiguous cases.

As described above, we encounter ambiguity only in the case where users are pivoting back to a category to which they had already applied a direct filter—for example, consider the series of pivots from a filtered set of actors A'_0 , to movies M_0 , to directors D , to movies M_1 , to actors A_1 (or A'_1 , the question being whether to keep the direct filter on A_1). We can assume that the meaning of the first set, A'_0 , was unambiguous when users applied the filter to it. The interim pivots (M_0, D, M_1) between that point and the returning pivot to A_1 are therefore the source of ambiguity that we must decipher.

Jacob’s Ladder itself does not implement any “smart pivot” heuristics—we include these heuristics as insight based on what we saw when we deliberately challenged users with questions about the data that led to both fan-out and fan-in scenarios. Users often failed to remove filters when they needed to. However, they almost never reenacted filters incorrectly. Therefore, we propose the following heuristics, and advocate for testing them formally in future work.

5.6.3.1 Returning After Intermediate Filters

The ability to enact connective filters is powerful. In the above example, we could apply a filter to directors D' , such as $age > 40$, whereupon the resulting traversal results in a connective-filtered set of actors of *the original set A'_0* that worked in films whose directors were over the age of 40. We suspect that erring on the side of fan-in—leaving the direct filter in place—will do the right thing most of the time. Our rationale is that an intermediate, connective filter is a strong potential reason for a user to have performed interim pivots that lead back to the same category, and its presence is very suggestive that it may indeed be what the user was thinking. In the event that leaving the filter is an error, systems should always have a mechanism for users to understand and correct where this heuristic fails.

5.6.3.2 Returning Without Intermediate Filters

In contrast, where no filters were enacted during interim pivots, we assume that the user intends to fan-out. Our rationale here is that, were the direct filter to be retained, the user will almost always arrive at exactly the same set that they started with—in our example, $A'_0 = A'_1$, rendering the interim pivots meaningless.

It is possible for a subtle difference to exist without intermediate filters—for example, if an actor in A'_0 acted only in one movie in M_0 that did not have any connected directors in D in the database, then that actor would be missing in the resulting set of actors A'_1 . However, we expect that corner cases such are rare. Furthermore, there is a straightforward interpretation of a series of unfiltered pivots that implies ever-widening sets of nodes. In the above example, without an intermediate filter on directors D , A_1 is the full set of actors who also worked with directors that worked with the original set A_0 .

Consequently, the heuristic for intermediate pivots without filters is to remove the original direct filter upon return. Although we suspect the likelihood of errors in this case to be lower, systems that automatically remove filters should make their actions clear, and easy to revert.

5.6.4 Implications for Learning From Pivots

In addition to their potential in helping users more freely navigate graphs, pivots also present opportunities for system designers to develop adaptive data abstractions. As we have mentioned, a critical difficulty in visualization design is the inability to validate the accuracy of data and task abstractions before implementing a system [10]. A visualization designer must arbitrarily decide the structure of the data before implementing a visualization—all too frequently, system designers choose an abstraction that does not correctly anticipate users' tasks or data, only to discover this error after significant work has been put into implementing a system.

Exposing users to purely structural operations like the graph pivot can make these misunderstandings more apparent; we saw examples of this in our study. Users were often not aware that they had successfully completed Task 1, they would often go to the “wrong” part of the interface to filter a set of nodes, and they would often type node classes in the search box, such as “Team,” instead of querying node attributes. Although this behavior

may have been in part due to their unfamiliarity with the interface, it makes sense that users would not immediately know whether to think of a value as an attribute of a node, a distinct node entity, an edge, or even a node class. These are arbitrary decisions that may or may not correspond to the user's expectations.

Pivots do not merely expose the arbitrary nature of certain data abstraction decisions. They can also work the other way, in that they expose what a user expects the data abstraction to be. The abstraction-agnostic nature of pivots presents an opportunity to learn about and adapt to the semantics of the data on the fly, rather than having to anticipate it completely from the start. A series of pivots is a very simple—yet explicit—indication of the data semantics from the user's perspective. When a series of pivots is unambiguous, it creates an unprecedented theoretical possibility: a system could observe user behavior, and reshape the data on the fly to more appropriately match the users' tasks and data.

5.6.4.1 Adaptive Connections

For example, suppose, in the hospital database scenario in Figure 5.6, that doctors must frequently determine which treatments can be prescribed based on patients' insurance providers. However, let us assume that in the initial graph abstraction, insurance providers and treatments are connected only through patients. Although pivoting and filtering make it possible to identify which treatments specific insurance companies have allowed, this traversal is a very roundabout way of answering that question, and it encounters the somewhat complex semantics of connective filtering that we discuss above.

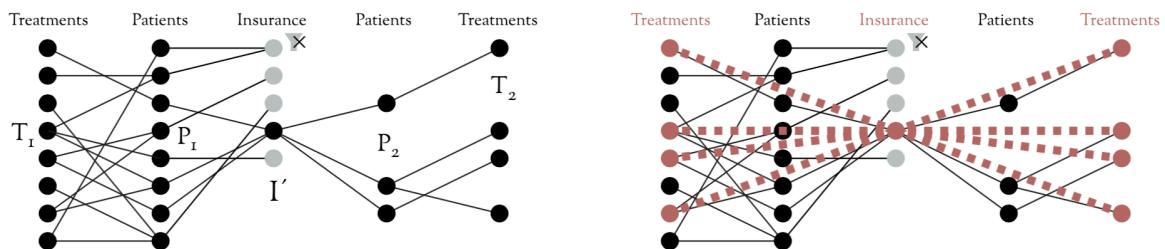


Figure 5.6. In this scenario, doctors frequently perform connective filtering on potential treatments by the insurance companies that have covered those treatments for patients in the past. The system observes this behavior, and adapts the underlying data abstraction in response, connecting treatments and insurance companies directly.

In this example, the system could observe users performing frequent pivots from treatments T_0 , to patients P_0 , to insurance providers I , applying a filter I' , and pivoting back (P_1, T_1). When this pattern reaches a certain threshold of usage, the system could automatically add a set of edges that directly connect the insurance providers with the prescribed treatments, bypassing the need to pivot through patients. From usage patterns alone, a machine could automatically “invent” a new category of semantically meaningful edges; in this case, edges that indicate that a specific insurance company has covered a specific treatment in the past. These new edges would allow users to move directly between these elements and make correlations without having to pivot through the patient nodes—enhancing both the semantic relevance of the underlying data abstraction, as well as database efficiency.

5.6.4.2 Adaptive Attributes

Edge topology is not the only arbitrary schema decision a technical expert may make with regard to a graph data abstraction. For example, as we have discussed above, the decision whether something is a node or an attribute of a node is arbitrary, and may or may not be amenable to a user’s task. These decisions, too, can benefit from observing user behavior in the context of a series of pivots.

Suppose that administrators at a university are frequently trying to pair students with professors from their home country. Let us assume that in the initial data system, the home countries of both students and professors are stored as an attribute of those nodes.

The system could observe users frequently using this attribute to correlate these two types of nodes. In response, the system can push the country attribute of student and professor nodes out into the graph as independent country nodes, allowing users to make direct pivots between students and professors from the same country.

5.6.4.3 Advantages and Limitations

The result of these alterations to the underlying data structure is that the graph can adapt to better support current and new questions. The system learns which connections hold the most valuable, real-world knowledge and exposes those connections as directly as possible. These updates can be performed automatically, either as the relevant patterns are detected, or as the processing and storage resources become available to support the added

complexity. Overall, this kind of system would allow the underlying data abstraction to be improved *in situ*, without constant collaboration between the technical experts and the domain experts. Using this method of back-filling the database structure, the graph automatically adapts to be able to efficiently deliver what users need from it.

Of course, the broad decision to interpret the data as a graph is still an arbitrary, *a priori* assumption that a technical expert makes that they cannot validate without implementing and evaluating a system with user testing. Learning from graph pivots provides some wiggle room only within that broad decision—the pitfall of choosing the wrong broad data abstraction remains.

Furthermore, a visualization that relies on an adaptive data structure must be somewhat general, like Jacob’s Ladder, employing general techniques such as graph pivots. Specialized visualizations that rely on dataset and domain-specific semantics, such as anticipating certain entities as nodes, and others as node attributes, will not be able to make use of this kind of approach.

Although our experience with Jacob’s Ladder has exposed these two examples—adaptive connections and adaptive attributes—as ways that graphs could self-adapt to changing user needs, we cannot enumerate all the possibilities for self-adapting data abstractions. Instead, by introducing the theoretical possibility of adaptive graph data abstractions, we advocate for future work into similar approaches for graphs and other data abstraction types. It may be possible, for example, for a system to automatically derive new set definitions as users interact with general-purpose set visualization systems such as UpSet [109], or to automatically pre-compute frequent weighted attribute combinations in general-purpose ranking systems such as LineUp [110].

5.7 Conclusions and Future Work

Our purpose in this chapter has been to articulate how users understand pivots, how they can be useful, and to explore a visually scalable technique for representing pivots; however, in our efforts to describe pivots in a general task sense, agnostic to any particular graph’s schema, size, or complexity, we do not discuss how to compute pivots efficiently. In a computational sense, however, pivots are not agnostic to schema, size, or complexity, and we leave computational scalability challenges for future work.

Across our lab tests and user tests, Jacob’s Ladder helped us to examine the expressive abilities and ambiguities that arise when constructing queries using sequences of pivot operations. The graph pivot is a very simple and intuitive, yet powerful, operation that shows promise for the future of graph data analysis, especially as it does not suffer from visual scalability with respect to the size of a graph. When coupled with filtering, users with a diverse range of expertise were able to discover and extract data subsets of interest at this aggregate, categorical level.

Although we have demonstrated that visualizing local topology is not necessary for many analysis tasks, our observations suggested that an even higher level overview of the global schema would be beneficial to help users plan where to filter or pivot. In continuing this work, we plan to more thoroughly test smart pivoting heuristics; build and test systems that adapt their abstractions; and further explore computational scalability issues.

Finally, our tests have exposed, but not fully answered, two important questions relating to pivots: whether smart pivots can accurately predict user intent with respect to filters, and how the simple nature of the graph pivot could make it possible to learn semantic information from user behavior, potentially granting visualization designers some flexibility in their initial data abstractions. Future systems that adapt their underlying data structure to user queries should become more semantically relevant.

Many of the lessons from Jacob’s Ladder have informed the creation of Origraph, a new network wrangling tool, that we detail in Chapter 6. With its broader set of operations, Origraph implements manual, user-driven precursors to what could become the automatic adaptations that Jacob’s Ladder envisions. For example, edges that bypass a series of intermediate nodes—our concept of adaptive connections—are implemented in Origraph as an edge projection operation. Similarly, Origraph has the ability to derive connected attributes that could form the basis for automatic, adaptive attributes in the future. Most importantly, Origraph implements a connective filter operation that is directly inspired by Jacob’s Ladder; although some of its exploratory power is minimized in Origraph’s wrangling-focused interface, the combined expressive power of pivots and filters to specify meaningful subsets is preserved.

CHAPTER 6

ORIGRAPH: INTERACTIVE NETWORK WRANGLING

Having studied the interplay between pivot and filter operations while a user is exploring and extracting subgraphs, we now focus on how a broader set of graph reshaping operations can be supported in an interactive system. In this chapter, that is currently in the process of peer review [6], we focus strictly on data wrangling operations for networks.

Networks are a natural way of thinking about many datasets. The data on which a network is based, however, are rarely collected in a form that suits the analysis process, making it necessary to create and reshape networks. Data wrangling is widely acknowledged to be a critical part of the data analysis pipeline; yet interactive network wrangling has received little attention in the visualization research community. In this chapter, we discuss a set of operations that are important for wrangling network datasets and introduce a visual data wrangling tool, Origraph, that enables analysts to apply these operations to their datasets. Key operations include creating a network from source data such as tables, reshaping a network by introducing new node or edge classes, filtering nodes or edges, and deriving new node or edge attributes. Our tool, Origraph, enables analysts to execute these operations with little to no programming and to immediately visualize the results. Origraph provides views to investigate the network model, a sample of the network, and node and edge attributes. In addition, we introduce interfaces designed to aid analysts in specifying arguments for sensible network wrangling operations. We demonstrate the usefulness of Origraph in two use cases: first, we investigate gender bias in the film industry, and then the influence of money on the political support for the war in Yemen.

6.1 Motivation

Data wrangling—which includes cleaning data, merging datasets, and transforming representations—is well known to be a tedious and time-consuming part of data analy-

sis [54]. Historically, wrangling was done with scripting languages such as Python, Perl, and R, or manipulation in spreadsheet tools, requiring significant computational skills. More recently, a new generation of interactive data wrangling tools instead uses visualization, interactive specification of rules, and machine learning to improve the efficiency and scale of data manipulation tasks while also providing accessibility to a broader set of analysts [56], [111], [112].

These powerful, interactive data wrangling tools, however, ignore a data type that is increasingly important [113]: networks. Although some datasets inherently represent a network that exists in the physical world, such as the connections between neurons in a brain or roads between cities, many other datasets also benefit from a network representation during analysis. The influence of social connections on obesity rates [114], the spread of information via a digital media platform [115], or the evolution of sticky feet of geckos [116] are but a few examples.

In such cases, an analyst has one, or possibly many, mental models of the data as a network. However, source data rarely conform to the way an analyst thinks about it. To model data as a network, analysts must wrangle the dataset, often starting with tabular or key-value data. The transformation process itself can lead to new hypotheses, and thus a new network representation of the data. Also, new tasks often necessitate new data abstractions [10]. It stands to reason that the ability to rapidly and easily transform network data can foster creative visualization solutions and simplify both exploration and communication of the key aspects of a dataset.

Existing network wrangling tools, most notably Ploceus and Orion [1], [2], focus on creating an initial network model, but no tools yet exist to iteratively and interactively reshape the network model itself. Several wrangling operations, especially nontabular reshaping operations, can still be performed only with code. Consequently, the steep learning curve of programming languages creates an unnecessary barrier that prevents many analysts from wrangling their own networks.

In this chapter, we introduce Origraph, our primary contribution (see Figure 6.1). Origraph is a visual, interactive network wrangling tool that allows analysts to model and reshape networks from input data in various forms. The goal of Origraph is to allow analysts to translate their data into a network representation that is most suited to answer

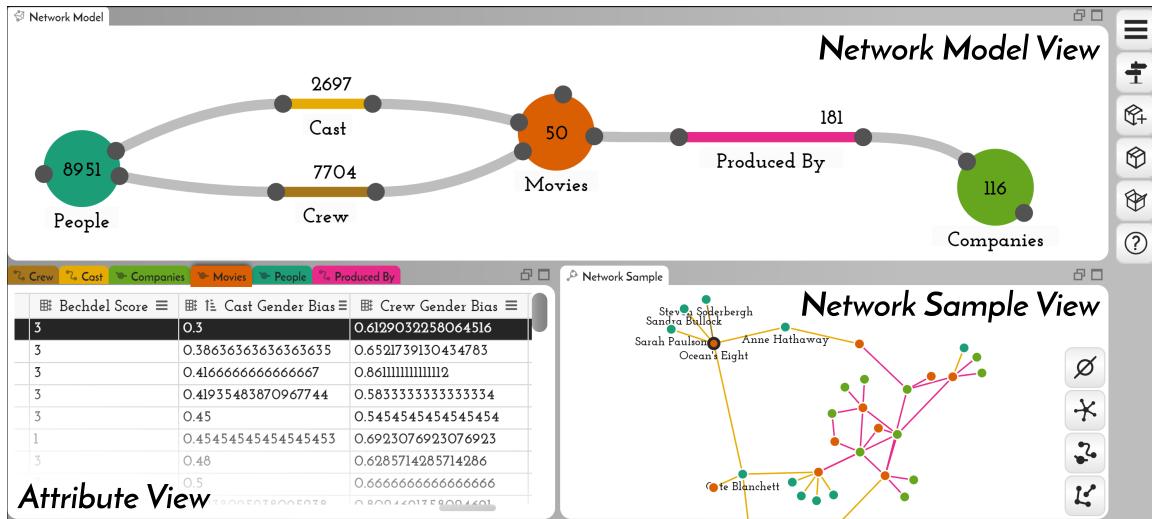


Figure 6.1. Overview of the Origraph UI. The **network model view** shows relationships between node and edge classes and is the primary interface for operations related to connectivity. The **attribute view** shows node and edge attributes in a table and is the primary interface for attribute-related operations. The **network sample view** visualizes a preview of the current state of the network.

their analysis questions. The design of Origraph is grounded in an analysis of operations for wrangling networks. Operations that are unique to Origraph are concerned with introducing new nodes, edges, or attributes based on leveraging network structures and multivariate attributes simultaneously. For example, Origraph supports an operation to introduce edges between two nodes if these nodes are connected by a path with specific properties.

Origraph includes visualizations of the data that support analysts in their reshaping and analysis process. Dedicated views communicate the state of the network, the attributes of the nodes and edges, and a sample of the network as currently modeled. Specialized views and algorithms support analysts in making network wrangling decisions. Origraph is web based and open source. A prototype is available at <https://origraph.github.io>.

We designed Origraph with a diverse audience in mind: from data journalists who analyze bot networks on social media, to social scientists who investigate the spread of specific terms in political circles, to biologists who study protein interaction. We do not expect users of Origraph to be able to program. Some advanced functionality is made available for more skilled analysts, such that they can write expressions slightly above the

level of formulas in spreadsheet software to perform sophisticated filtering and aggregation. At the same time, we believe that Origraph can also speed up the wrangling process of skilled programmers.

We validate Origraph in two complex network modeling use cases. First, we reshape a movie dataset so that we can investigate gender biases in recently popular movies. In the second use case, we integrate data from various sources and build a network that allows us to investigate how money from donors could influence votes in the US Senate on issues related to the war in Yemen.

6.2 Usage Scenario

Here we introduce a scenario of analyzing a movies dataset to illustrate how analysts might want to reshape a network to answer specific analysis questions. We will also use this hypothetical movies dataset to introduce network wrangling operations, and describe a concrete analysis of movies data with a real dataset in Section 6.8.1.

Specifically, we would like to analyze gender bias in recently popular movies. We retrieve a movies dataset from an API and load a *movies* table, which contains movie titles and other attributes; a *cast* table, which contains the name and a description of the characters in the movies; and a *people* table, which has attributes about the actors. The cast table contains identifiers allowing us to establish relationships between movies and actors. We start by assigning movies and actors to be nodes, roles to be edges, and connecting the movies to the actors via the nodes. Now, as a first metric to analyze gender bias, we want to compute the ratio of female to male actors in the movie. To do that, we derive a new movie attribute, gender bias, based on the gender of the connected actors. We can analyze movies with a low gender-bias and the actors that appear in them using the network sample and attribute views. We have recreated this state in Figure 6.1.

Next, we want to see whether gender bias is more prevalent in certain genres. We first promote the attributes of the genre class to create separate nodes that connect all the movies of the same genre to a new node representing the genre. Now we derive a new attribute for the genre nodes: the average gender bias for each genre.

Finally, we want to find out whether there is a group of female actresses who frequently costar in movies with at least a balanced gender ratio. To do so, we filter actors to contain

only women; filter movies to only include those with at least a balanced gender ratio; and then project new edges through the existing movie nodes. The resulting network connects actresses who have costarred in movies with at least as many women as men. We can explore that network in the built-in network viewer, or export it for rendering and layout in an external tool.

6.3 Terminology

Here we define terminology that we use to describe the operations, as well as the functionality of Origraph. In this work, we focus on modeling **networks** using **nodes** and **edges**. Both nodes and edges can have **attributes** associated with them. A **class** defines a common set of attributes for either nodes or edges: a **node class** defines a class of nodes, and an **edge class** defines a class of edges. We treat nodes and edges as fluid concepts that can change, and use the term **class** to generically refer to node and edge classes and **items** to refer to generic instances of nodes or edges. **Supernodes** are nodes representing a set of other nodes.

We use the term **network model** for the set of classes and their relationships. The network model describes relationships between types of nodes and edges on an abstract level and is independent from concrete instances of the classes, similar to how a database schema is independent from the specific data in the database. Concrete nodes and links make up the **network topology**. It is noteworthy that networks with trivial models (e.g., a single node class and a single edge class) can be arbitrarily large and complex.

Edges can be directed or undirected. Node and edge attributes can be numerical, ordinal, nominal, sets, or labels/identifiers. Attributes may also contain complex data structures such as nested hierarchies, lists, and objects.

6.4 Operations

We elicited a set of operations for wrangling multivariate graphs based on a literature review and our own experience with designing network visualization tools and preparing data for visualization [5], [26], [117]–[123]. Although we do not claim that this list is exhaustive, we demonstrate that the combination of these operations extends the space of transformations currently possible with Orion and Ploceus [1], [2].

We classify these operations into three categories: **modeling operations** that modify the network model and network topology by introducing new node and/or edge classes in a network; **item operations** that modify the network topology by removing or introducing items (instances of links or nodes); and **attribute operations** that manipulate the attributes of, or add new attributes to, existing classes.

6.4.1 Modeling Operations

Modeling operations affect the network model by introducing or removing node or edge classes. Modeling operations indirectly also affect the network topology because they implicitly create new node and edge instances.

 **Connecting or disconnecting** items is the most fundamental modeling operation. Connections can be established by leveraging the primary key / foreign key approach well known in the database literature, where each item of a tabular dataset has a unique ID, and another column has a foreign key pointing to the primary key of the same or another table. This is also a common way to store graphs in nonvolatile memory: many network file formats store lists of nodes and lists of links between these nodes. We also consider cases where an item in a table stores a list of connection targets using foreign keys. More generally, connections can be derived from arbitrary attributes, for example by (partially) matching strings, or by evaluating arbitrary functions on attributes. In the movie dataset, for example, we could introduce edges between movies that have a significant number of female actors, to form a clique of gender-balanced movies.

 **Promoting attributes** allows users to promote the unique values of an attribute column into a new class (Figure 6.2). In the movies example, we could take a column containing film genres and promote these to a separate “genres” class, while at the same time introducing edges between the new genre nodes and the movie nodes.

 **Faceting** slices a class based on the value of an attribute and creates new classes for each slice, as illustrated in Figure 6.3. An example is to facet the movie class on the genre attribute, creating a new class for each unique genre containing only that genre’s movies.

 **Converting between nodes and edges** transforms connected nodes into edges, or *vice versa*, retaining the connectivity and semantics of the network, as illustrated in Figure 6.4. For example, given actors connected to movies, we convert the movie nodes to edges,

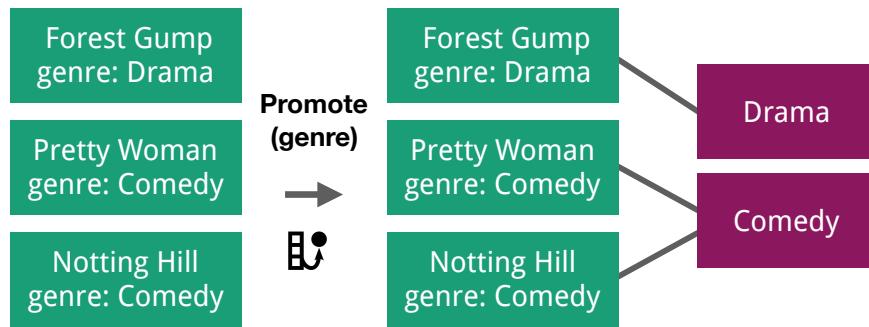


Figure 6.2. Promoting the genre attribute of the three movies. A new genre node class with two instances contains two nodes with the unique genres. The new nodes are connected to their source nodes.

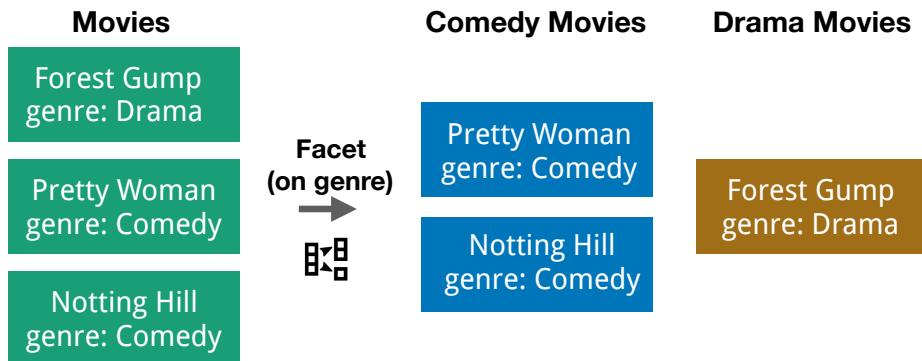


Figure 6.3. Faceting a movies node class based on genres. The facet operation results in two new node classes, one for “Comedy Movies,” the other for “Drama Movies.”

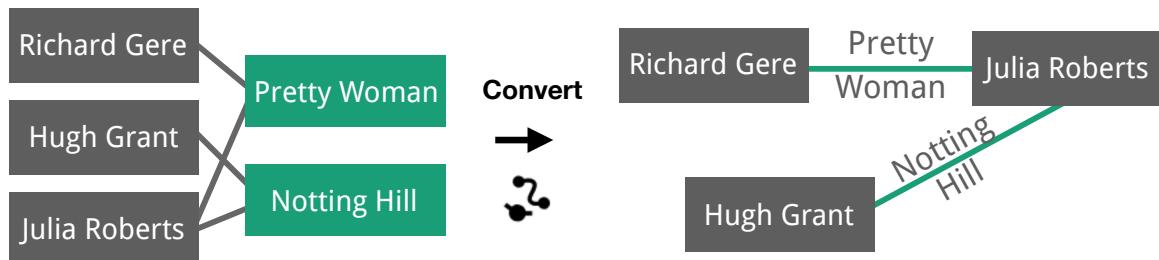


Figure 6.4. Converting nodes to edges. Here, movie nodes are converted to edges, resulting in a network linking actors that appeared together.

which results in a collaboration network between actors, where edges connect actors that have acted together in a movie.

⌚ **Edge projection** introduces an edge based on a path in the network model between nodes, as illustrated in Figure 6.5. The path can be specified with a set of rules leveraging the classes and the attributes of the network. For example, in an actor–role–movie–production-company network, edge projection can be used to connect an actor with the production-company. Another example is to project edges from an actor–movie–actor relationship to an actor–actor relationship, limiting edges to financially successful collaborations by only considering edges through movies that had a box office return above a specified number.

⌚ **Creating supernodes** aggregates the information of multiple nodes into a single supernode that represents all the constituting nodes. For example, we could create a supernode that represents all comedy movies (Figure 6.6). There are different ways to realize a “create supernode” operation: one can retain all aggregated nodes and edges, or replacing the aggregated nodes with the supernode. In Origraph, we chose to retain all nodes and edges, which can be filtered out in a subsequent step.

✎ **Rolling up edges** combines parallel edges, or multiple edges that connect the same two nodes, into a single edge of a new edge type. For example, in a coappearance network of actors, some actors might have appeared together in multiple movies, represented by multiple edges. The rollup operation results in a single new edge connecting these actors.

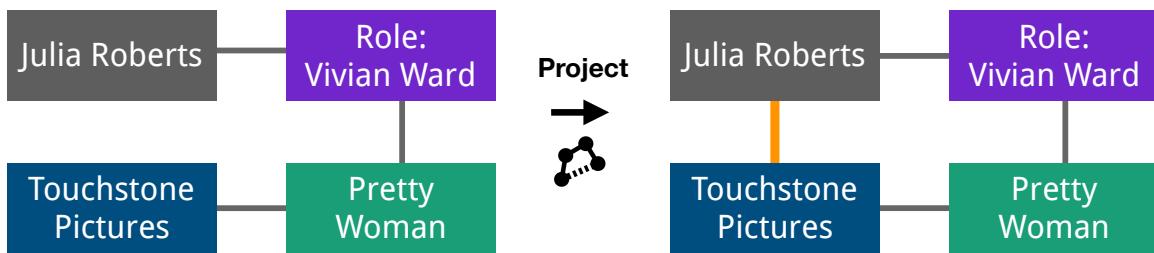


Figure 6.5. Projecting an edge. Paths connecting an actor to a role to a movie to a production company are projected to an actor–production-company edge (orange).

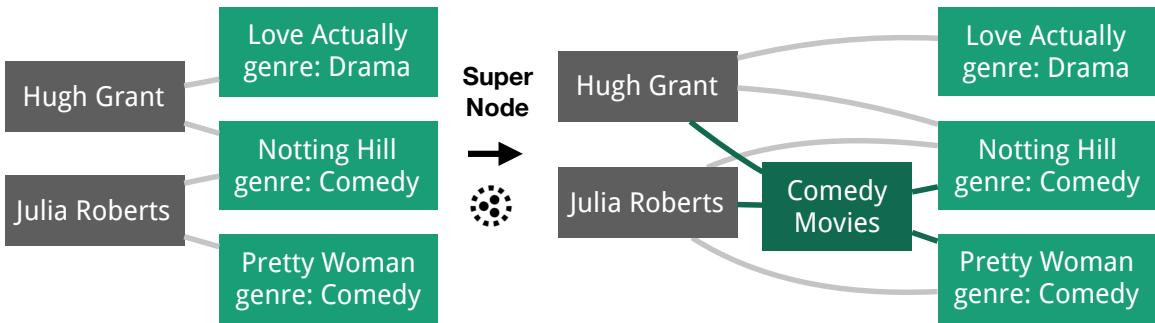


Figure 6.6. Creating a supernode. Notting Hill and Pretty Woman are combined into the supernode Comedy Movies. This supernode inherits all edges and also has edges to the aggregated movies.

6.4.2 Item Operations

Item operations change the number of items in existing classes. They may leverage the network model, but they do not modify it. Item operations affect the topology of the network, as they manipulate which nodes and edges exist.

Filtering by attributes removes nodes or edges based on the values of an attribute. For example, we could filter by removing all movie nodes that grossed less than \$10 million, or remove all actor-movie edges where the role was not a speaking role.

Connectivity-based filtering describes filter operations on items (nodes or edges) that leverage the connectivity of a network. In the movie network, where actors are connected to movies they acted in, an example of connective filtering is to remove all actors who have never acted in a movie that grossed more than \$100 million. Connectivity-based filtering can also leverage complex, multihop operations [5].

6.4.3 Attribute Operations

Attribute operations modify class attributes or create new ones, but do not impact the network model or the network topology. They are, however, an important prior or subsequent step in many modeling operations.

Deriving in-class attributes leverages existing attributes in a node or edge class to derive new ones. For example, for actor nodes with birth and death year, we could derive the attribute “age,” representing either the actors’ current age, if they are alive, or their age at death.

☞ **Connectivity-based attribute derivation** is concerned with deriving attributes for a node or edge class based on attributes in a possibly indirectly connected class (Figure 6.7). As an example, in an actor-movie network, we can compute a new attribute “gender bias” on the movie class by iterating through all the actors connected to each movie and dividing the number of men by the total number of actors in that movie. Connectivity-based attribute derivation can be very useful as a follow-up step to many modeling operations. For example, when ☞ creating supernodes, it is often relevant to also aggregate some of the attributes of the original nodes into the supernode. Combined with modeling operations such as ☛ promoting attributes, ↗ edge projection, ☞ creating supernodes, or ↘ rolling up edges, this is the network version of the split-apply-combine (or map-reduce) strategy common in tabular data analysis [124]. As with tabular data, there is an immense diversity of potentially relevant apply functions.

↳ **Changing edge directionality** introduces or removes directionality into previously undirected edges, or changes the direction of edges. Like all other operations, changing edge direction is rule based. For example, in a network of movies and actors, edges between movies and actors can be made directional to represent an “acted in” relationship.

6.4.4 Housekeeping Operations

In addition to these data wrangling operations, a system implementing these operations will also need to enable a series of basic operations, such as importing data from various file formats, extracting nested file structures, exporting data for consumption by

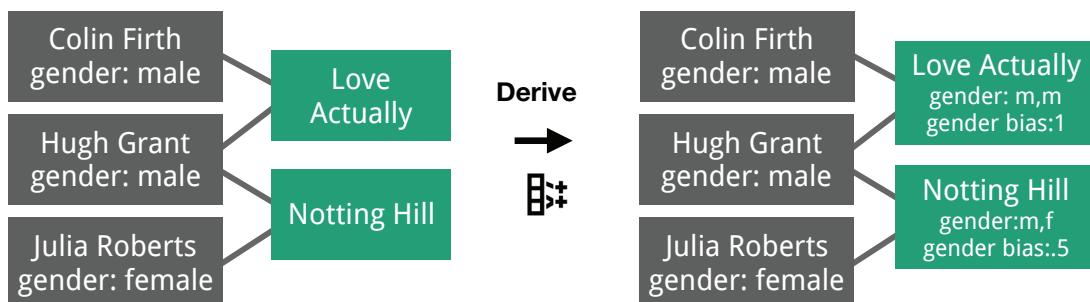


Figure 6.7. Deriving attributes across connected classes. Nodes of the movies class derive attributes from the adjacent actors class. In this case, the gender of the actors is used to calculate a gender bias score.

graph visualization tools, renaming classes and attributes, removing or hiding classes or attributes, etc. Since these are not specific to network wrangling, we do not discuss them in detail.

6.4.5 Discussion

Some of the described operations can be achieved by sequentially executing other operations. For example, ~~the~~ **connectivity-based filtering** can be achieved by first ~~the~~ **deriving** attributes based on connectivity and then ~~the~~ **filtering** based on attributes. Similarly, ~~the~~ converting nodes to edges could be achieved by first ~~the~~ **projecting edges** and then deleting the intermediate nodes and edges. We have chosen to include these operations nevertheless, because we believe that they are closer to an analyst’s mental model and require less indirection.

6.5 Origraph Design

We implemented the network wrangling operations we identified in Origraph, complementing them with visualizations supporting the operations and visualizing the state of the network in various views. A key design goal of Origraph is to immediately visualize the effect of an operation on the network model, the underlying topology, and the attributes, so that analysts can easily understand their actions.

The interface contains three main views, shown in Figure 6.1: a network model view, an attribute view, and a network sample view, in addition to various operation-specific views. The network model view is one of several ways users can modify the state of the network by generating new node and edge classes, and connect existing classes. The attribute view shows a table for each node and edge class, where attributes are columns and rows are instances. Table headers enable users to filter, sort, control instance labeling, and generate new node classes based on attributes. The network sample view displays a sample of the network with a concrete instance of each class in the network model. Views can be flexibly placed and scaled.

Origraph aims to make interactions for operations as close to their visual representation as possible. For example, operations such as ~~the~~ **connecting** node classes are supported through direct manipulation of the network model; similarly, operations that utilize at-

tributes are accessible from the attribute headers.

6.5.1 Network Model View

The network model view serves to represent all classes and their relationships, and to initiate modeling operations. The network model is represented as a node-link diagram, as shown in Figure 6.8. Node classes are represented as circles and edge classes as lines. Generic classes (before they are assigned to be nodes or edges) are represented as diamonds. We chose these encodings because they are consistent with the mental model most analysts have about nodes and edges. Recall that edges can have arbitrary many and complex attributes, just as nodes, and that they can exist independent of nodes. That implies that we need to interact with edges in the same ways as with nodes. Our final edge encoding enforces that a segment of the edge is always horizontal and allows a connection to a node on each side (notice that a segment of the cast edge is horizontal in Figure 6.8). This ensures readable labels (no tilted text) and a simplified way of interacting with edges (no rotation). A downside of these restrictions is that effective layouts can be challenging with off-the-shelf algorithms, which is why we rely mostly on manual layouts. We also experimented with other glyphs for edges that are more flexible in terms of where we can draw connection lines. We, however, found it more important to use the edge metaphor than to enable better automatic layouts.

The number of items in each class is displayed with labels. We also experimented with graphical encodings for the number of items, but found that significantly different class sizes are common, resulting in unbalanced visuals and difficulties in interacting with the

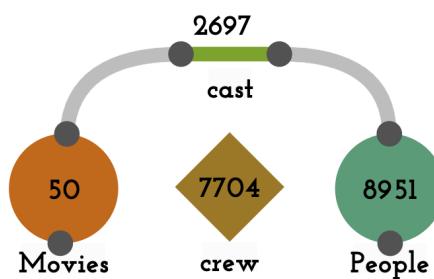


Figure 6.8. A simple network model for the movies dataset. Node classes (movies, people) are represented as circles, edge classes as lines (cast), and generic classes are shown as diamonds (crew). The cast edge class connects between movies and people.

classes. Each class is assigned a unique color and a class label. Both are used consistently across all views for elements associated with that class. Although qualitative color scales are limited in the number of distinguishable colors, we observed that most cases do not exceed six or seven classes. For more than eight classes, we use gray as the color and fall back to labels and interactive highlighting.

Nodes and edges have handles that can be used to initiate **connection** and **edge projection** operations by dragging the handle of one class to the other. **Conversion** and **direction changes** are available in-place in the network model view.

6.5.2 Attribute View

The **attribute view** uses a tabular layout with multiple tabs to represent classes, attributes, and items. Each class is displayed in a separate table/tab; with a column for each class attribute and a row for each item. In contrast to other tools [1], [2], we chose to show not only the attributes and data types, but the whole table, because we believe that it is important to be able to quickly assess the full dataset when making wrangling decisions. Our attribute view shows the data in cells as numbers or strings, but we consider visual encodings for future versions [125]. The attribute table can also handle nested data structures, such as arrays and objects, which we encode using curly or square brackets for objects and arrays respectively, enclosing a number indicating the size of the data structure.

Column headers serve as the interface for initiating operations that are based on attributes, including the modeling operations **promotion** and **faceting**, the item operations **direct** and **connectivity-based** filtering, as well as the attribute operations for **in-class** and **across-class** attribute derivation.

6.5.3 Network Sample View

The network sample view renders a force-directed node-link diagram, containing a sample of the network in its current state. The nodes are colored according to their class; node labels are shown on hover, and are rendered persistently for selected nodes and their neighbors. The attribute view and network sample view are linked through highlighting. Which attribute to use as a label can be changed in the attribute view.

By default, we sample from node and edge classes while ensuring that the sampled items are connected. We achieve this by using the depth-first sampling approach described

by Hu and Lau [126], but slightly modify it to balance sampling across classes. Alternatively, users can seed nodes or edges of interest using controls in the network sample view, or add specific items from an attribute table. To see the relationships of a specific node, its neighbors can be added on demand.

Our strategy of working with network samples and selected items is based on the assumption that most networks wrangled with Origraph exceed what can sensibly be drawn using a simple node link layout. Sampling is appropriate when the goal of the analyst is to quickly judge the effect of wrangling operations, although the approach of selectively querying and expanding a subgraph allows analysts to address focused network tasks, where the details of specific node and edge neighborhoods and their attributes matter [123].

6.5.4 Operation Views

In addition to the views that represent the state of the network and the underlying data, Origraph provides several views designed to aid analysts in executing operations.

6.5.4.1 Connection Support Interface

The `“::connect` operation matches items in two classes. Origraph supports node-edge or node-node connections. Node-edge connections only connect the nodes to one “side” of an edge and are commonly followed by a similar operation on the other side. For example, when treating movie roles as edges, we connect that edge first to actors and then to movies using two connect operations. Node-node operations implicitly introduce a new edge class.

Connecting two classes can be difficult, especially if the attribute tables are large, and inconsistent column labels are used. Origraph supports this process through a visual interface, shown in Figure 6.9. The interface calculates all pairwise matches between the source and target classes for all attribute combinations and the index of the items. Based on that information, we calculate a heuristic score for every attribute combination between the items of two classes (*src* and *trg*) as follows:

$$score_{n,m} = \sum_{class \in \{src, trg\}} \frac{\sum_{item \in class} \begin{cases} \frac{1}{deg_{n,m}(item)}, & \text{if } deg_{n,m}(item) > 0 \\ -1, & \text{otherwise} \end{cases}}{|class|}$$

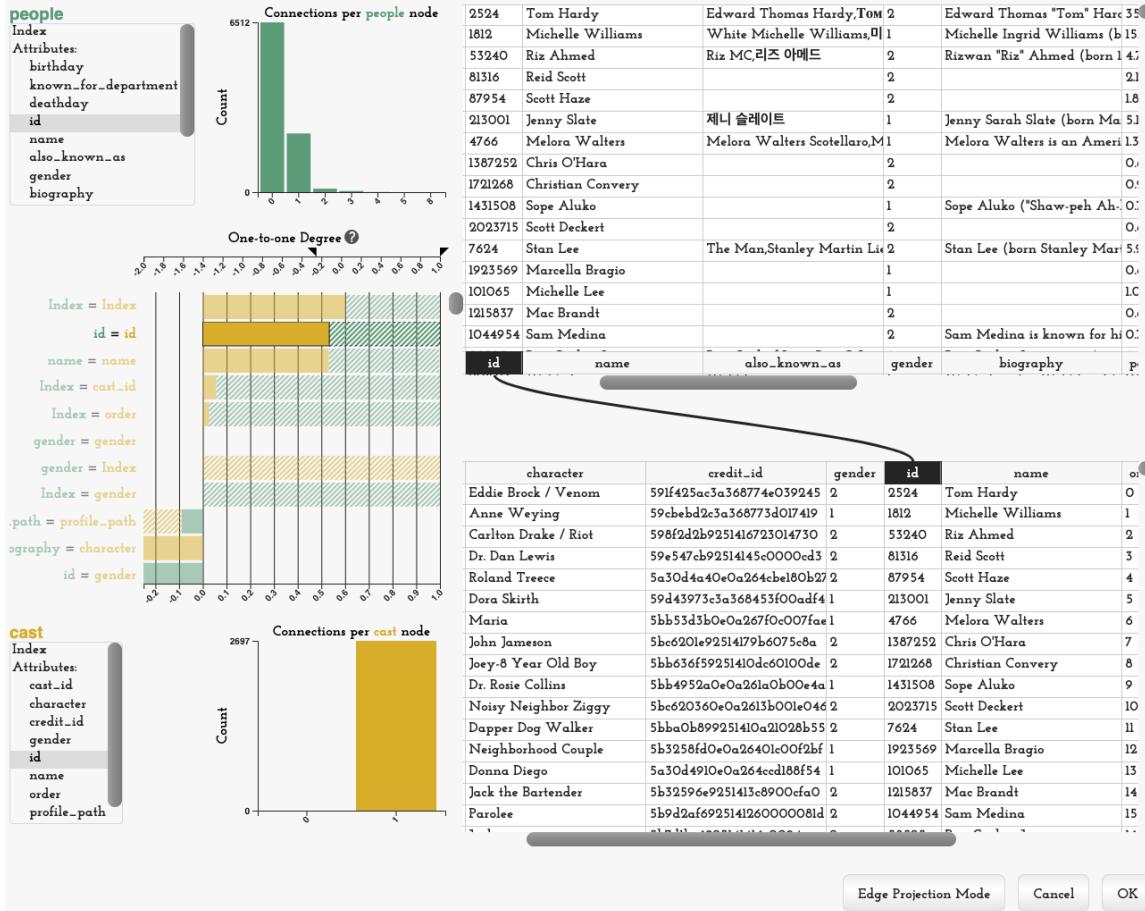


Figure 6.9. The connection support interface aids in finding the right connections between attributes of two different classes. The raw data of each class is shown on the right. The left side shows a score for different attribute combinations; here the column named ‘id’ is selected in both classes (people and cast). The histograms at the top and the bottom show the degree distribution for the selected attribute.

Here, n is the attribute in the src class, m in the trg class, and $deg_{n,m}(item)$ is the degree of the item when connecting between the classes based on these attributes. The heuristic is designed to give a high weight to 1-1 matches and slightly discounts matches to nodes with higher degrees. Discounting nodes with higher degrees avoids issues with cases where a small set of categorical labels that are identical between the two classes produce many matches. The heuristic penalizes missing connections and normalizes the final scores to the class size. The heuristic results in a score of 2 if each item in the source class matches exactly one item in the target class and no items are unmatched, and -2 if no matches are found.

The contribution of each class to this heuristic is shown in the interface as a stacked bar chart. Solid bars encode the final score, and shaded bars encode one-sided scores where the overall score is smaller than the score of one class. Additionally, histograms show the degree distribution of the items. Figure 6.9 shows connections for people to a list of cast members. The people class contains many individuals who are crew but not cast members, but each cast member has a match in the people table. The people degree distribution shows that most people acted in no movies (the crew), followed by many in a single movie, and much fewer in more. The heuristic score shows that the highest score matches the indices of the two classes, which is not correct. The score between the two id attributes, which is selected, is slightly lower. The relationship between these attributes is also shown as a link between the tables.

6.5.4.2 Path-Based Operations

Several operations, including **edge projection**, **deriving attributes based on connectivity**, and **connectivity-based filtering**, require analysts to specify conditional paths in a network. For example, to connect an actor with a direct edge to a production company, as illustrated in Figure 6.5, analysts have to specify the path from actor to role to movie to production company.

To simplify selecting these paths, Origraph includes an interface that displays all paths across node and edge classes starting from a selected node, as shown in Figure 6.10. Analysts can select specific paths (including loops), which are shown in a breadcrumb interface at the top. These paths are then used as input to the aforementioned operations.

6.5.4.3 Specifying Functions

The two operations concerned with attribute derivation (**in-class** and **connectivity-based** attribute derivation) and the filter operations (**direct** and **connectivity-based** filtering) are based on evaluating functions over one or multiple values. Here, the tension is most acute between supporting operations that do not strictly require programming, yet could benefit significantly from its expressiveness. To address this balance, each of these operations has two modes: a standard, nonprogramming mode provides sets of common functions such as count, sum, or median for attribute derivation; or simple less-than, equality, or greater-than comparisons for filtering (see Figure 6.10).

Choose a path

Name the new attribute

Choose a value

Index

Attributes:

- birthday
- known_for_department
- deathday
- id
- name
- also_known_as
- gender
- biography

Choose a function

Custom

In-class:

- Duplicate

Across Classes:

- Count
- Sum
- Mean
- Median
- Mode
- Concatenate

	#actors associated with company
0	18
1	32
2	17
3	17
4	2
5	2
6	2
7	2
8	2
9	0
10	0
11	0
12	15
13	130
14	0
15	3
16	0
17	0
18	57
19	2
20	3

[Advanced Mode](#) [Cancel](#) [OK](#)

Figure 6.10. Path selection (left) and apply function (right) interface showing a **derive connected attribute** operation counting the number of actors associated with a production company. A preview of the results is shown on the right.

An advanced mode drops analysts into a prepopulated code template that implements the standard function they had previously chosen. They can then adapt this function as needed. For example, Figure 6.11 shows the computation of a gender bias ratio, which was minimally adapted from autogenerated code for computing the median.

6.6 Implementation

Origraph is a client-only web application. Its source code is available under the MIT open-source license at <https://github.com/origraph/origraph.github.io>.

Origraph is designed to help analysts create a set of rules for reshaping a graph in a lightweight, flexible interface. In its current form, it scales to medium-sized networks (up to tens of thousands of nodes) that can fit in memory. However, Origraph's interface is built on top of an independent graph processing library `origraph.js`, available at <https://github.com/origraph/origraph.js>. Ultimately, our goal is to support interactive operations with an in-memory sample of a large graph, and then export a script capable of applying users' rules to much larger datasets, similar to the strategy pioneered

The screenshot shows the Origraph interface. On the left is a code editor with the following JavaScript code:

```

1  async function (node_Movies) {
2    let nWomen = 0;
3    let nMen = 0;
4    const class1 = origraph.currentModel.findClass('cast');
5    for await (const edge_Cast of node_Movies.edges({ classes: [class1] })) {
6      const class2 = origraph.currentModel.findClass('People');
7      for await (const node_People of edge_Cast.nodes({ classes: [class2] })) {
8        let value = await node_People.row['gender'];
9        if (value === 1) {
10          nWomen++;
11        } else if (value === 2) {
12          nMen++;
13        }
14      }
15    }
16    return nMen / (nWomen + nMen);
17 }

```

To the right of the code editor is a modal dialog titled "Name the new attribute". It contains a text input field with the value "Cast Gender Bias" and a "Preview" section. The "Preview" section displays a table with the following data:

title	Cast Gender Bias
Venom	0.52941176470
Mission: Impossible - Fallout	0.68
Bohemian Rhapsody	0.88888888888
Avengers: Infinity War	0.6792452830
The Predator	0.9090909090
Incredibles 2	0.63636363636
A Star Is Born	0.75
First Man	0.89285714285
The Girl in the Spider's Web	0.571428571428
The House with a Clock in Its Walls	0.666666666666
The Meg	0.777777777777

At the bottom of the dialog are three buttons: "Template Mode", "Cancel", and "OK".

Figure 6.11. Interface for deriving a custom attribute using code. In this case, the user adapts an autogenerated template for computing the mean value, changing eight lines of code: the counter initialization lines (2,3); the *if* conditionals and increments (9-13); and the return statement (16).

by existing tabular data wrangling tools [56], [111], [112].

The underlying library interacts with raw data using two different layers of abstraction: a **table network** layer, underneath an **interpreted network model** layer. Similar to the schema of a relational database, the table network is a lazily evaluated specification of raw and derived tables, and any connections between them. The interpreted network layer maps tables in the table network to classes in the network model: both node and edge classes must map to a specific target table. This way, edges in the graph are always guaranteed to support features, such as edge attributes, that existing network modeling tools [1], [2] cannot represent. Furthermore, edge classes reference up to two paths through the table network, from the target edge table to its connected node tables, which implies that edge classes can be fully or partially disconnected, and these states are reflected in Origraph’s interface. Keeping these two layers separate allows for free-form reinterpretation of the underlying tables as node or edge classes.

6.7 Input and Output

As for any data wrangling tool, input and output are important considerations. Our design goal for input is to ingest data formats from “in the wild” data sources and avoid the need for preprocessing. To this end, Origraph supports tabular data that can contain explicit node and link lists; alternatively, links can be inferred based on attributes.

Another input data type is hierarchical, specifically JSON, which is a common format for API responses from numerous online services. The hierarchy commonly contains multiple levels of items that can be represented as nodes and edges individually. A movie object, for example, can contain an array of all cast members, themselves represented as complex objects. Origraph implements special **unroll** and **expand** operations to convert these nested structures into separate classes.

Origraph currently exports d3.js-style JSON, zipped CSV files, and GEXF for analyzing the wrangled graphs in off-the-shelf graph visualization tools such as Gephi or Cytoscape. The prototype comes with multiple datasets that can be loaded in raw or preassembled format.

6.8 Use Cases

Here we demonstrate how Origraph can be used to wrangle two network datasets. The data for both cases were retrieved from several different APIs, and no data wrangling was performed outside of Origraph. The scripts used to access the API endpoints and the resulting data can be found at <https://github.com/origraph/data>; each dataset is also provided as a sample in Origraph. Figures 6.12–6.28 show the full state of the interface at each major step.

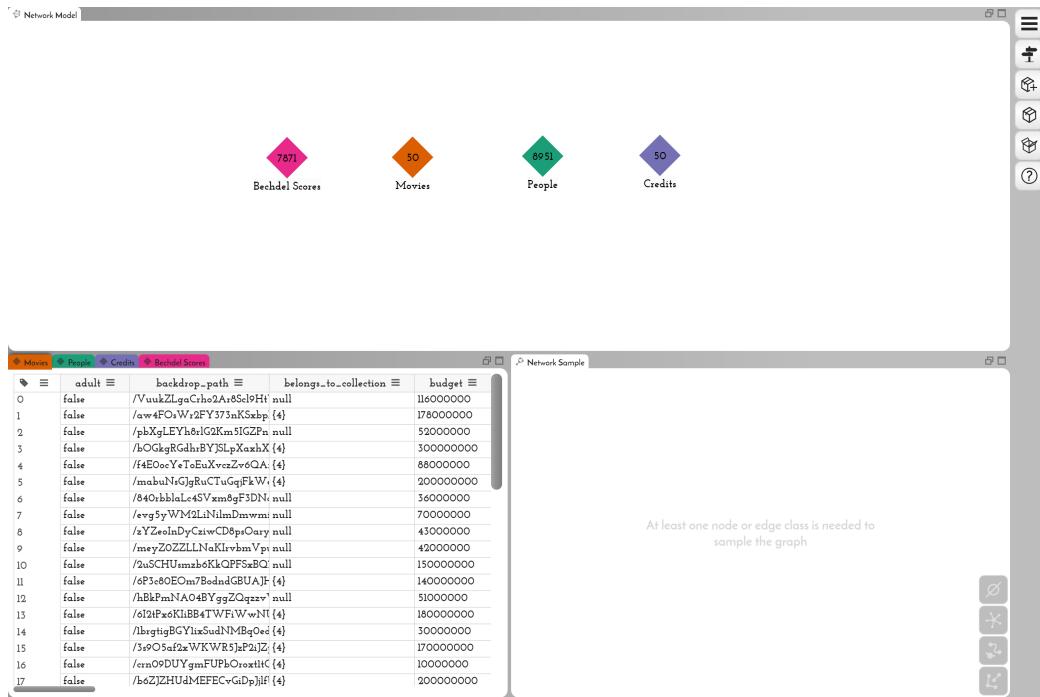


Figure 6.12. In this use case, we investigate gender bias in movies. Here we loaded raw movies datasets, consisting of TMDB Movies, Credits, and People [127], as well as Bechdel Scores [128].

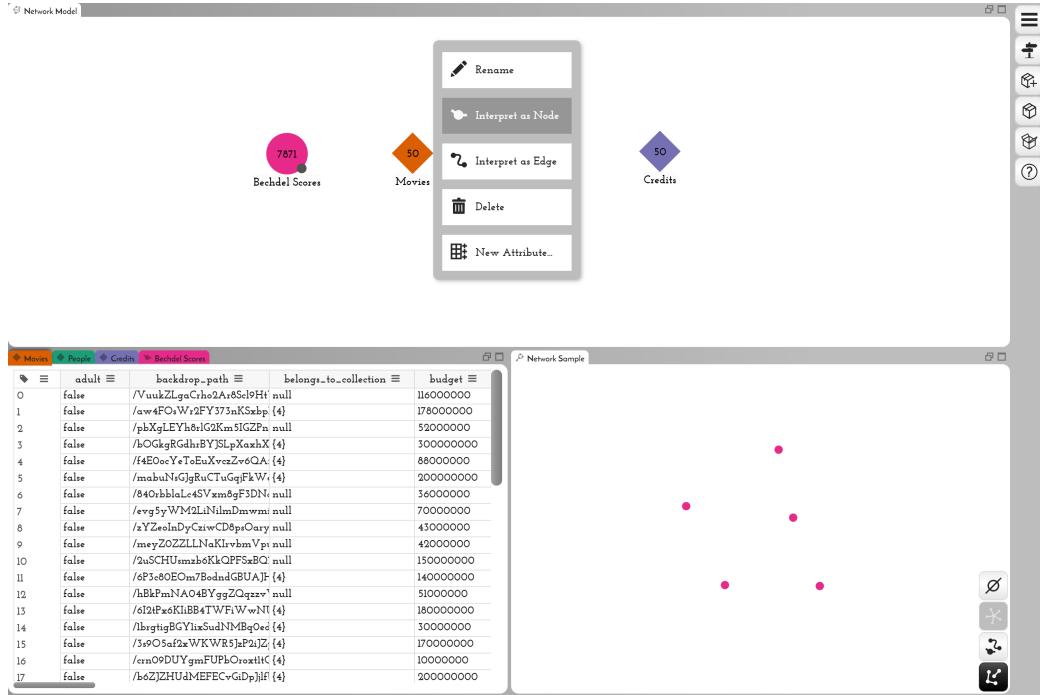


Figure 6.13. The Bechdel Scores and Movies tables are from different sources, but both refer to movies—in order to combine them, we first convert each to node classes.

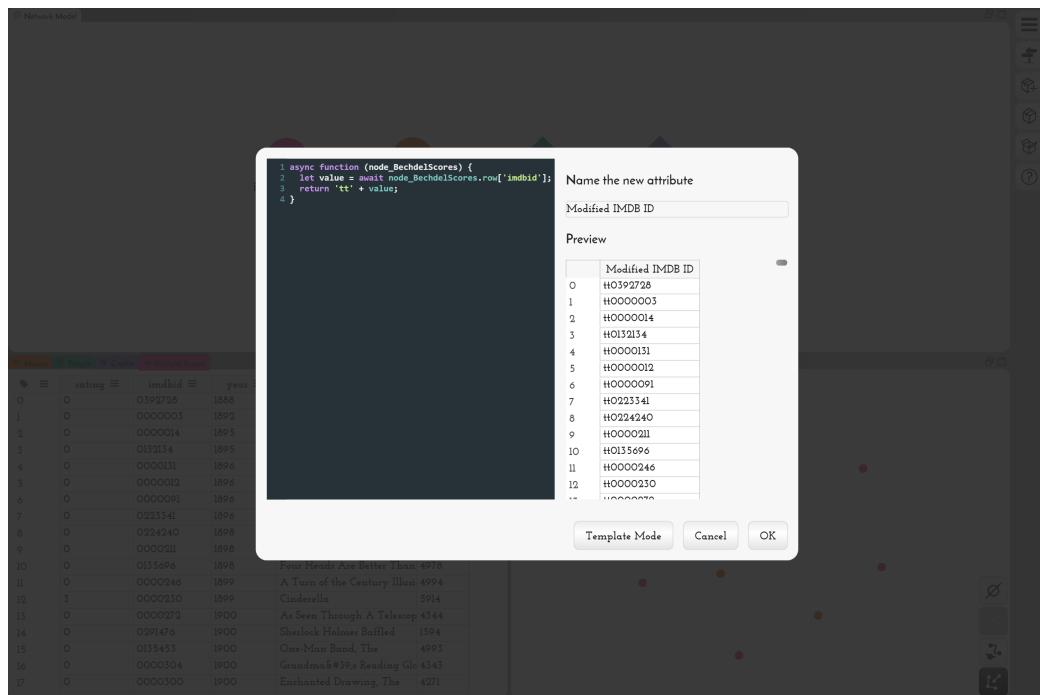


Figure 6.14. We want to connect both classes through IMDB IDs; however, the Movies table has a “tt” string prepended to its `imdb_id` attribute. Here, we derive a new attribute based on the Bechdel Scores’ IMDB IDs, to make connection possible.

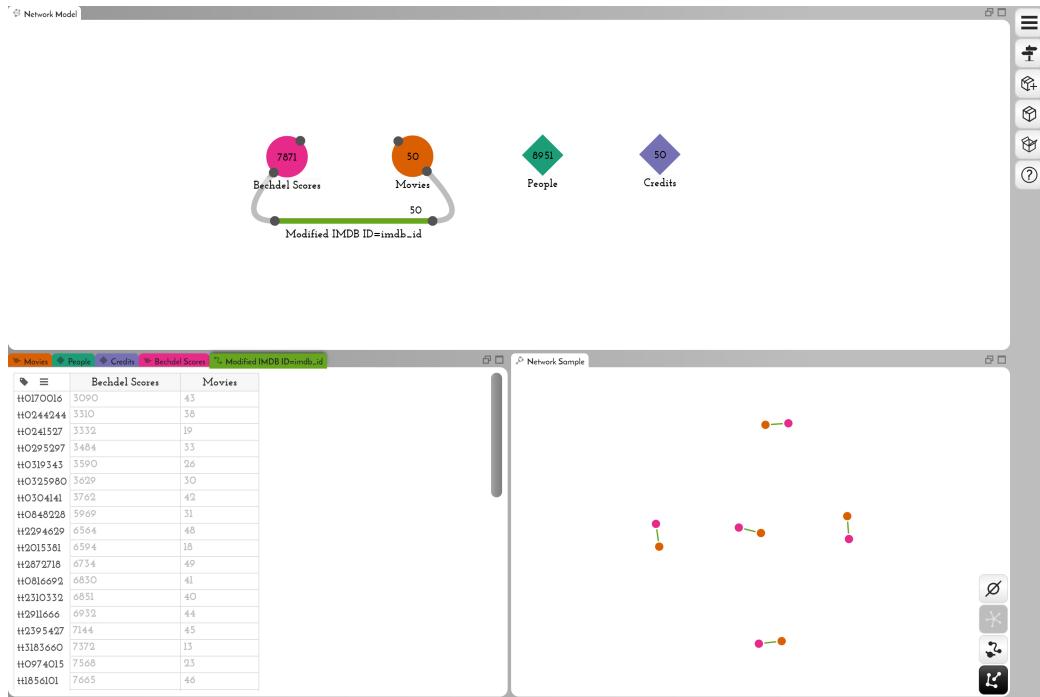


Figure 6.15. The state of the network model after connecting Bechdel Scores and Movies.

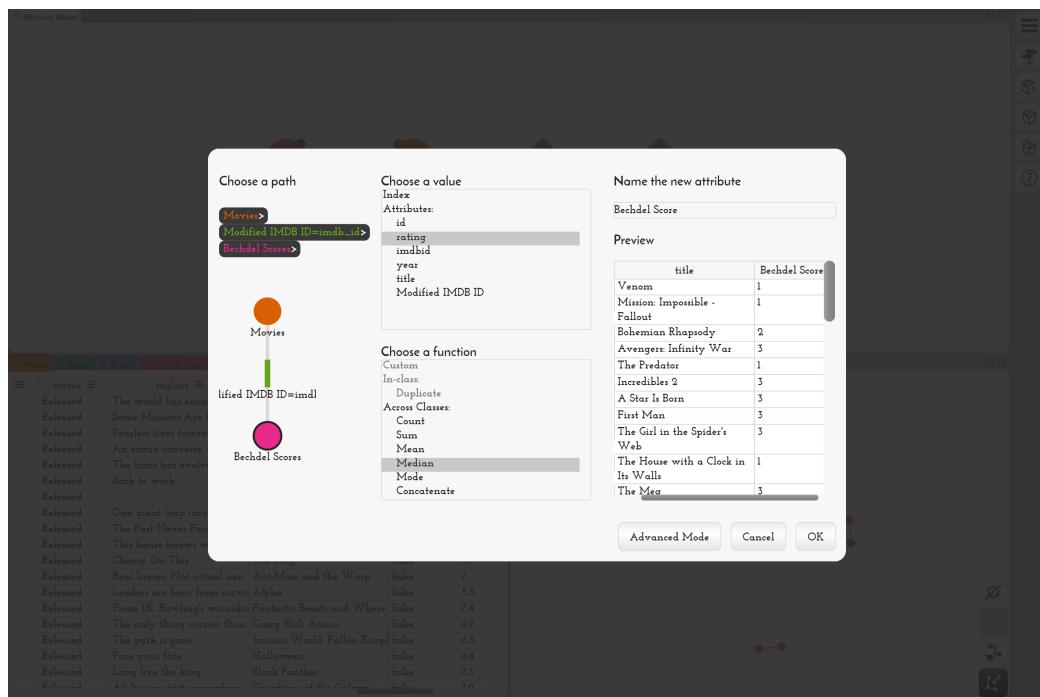


Figure 6.16. The standard attribute derivation interface, showing how attributes can be derived from connected nodes and edges. Here, we are computing the median rating for a Movie, based on its connected Bechdel Score. Because Movies and Bechdel Scores have a 1-1 mapping, this *defacto* simply copies the rating.

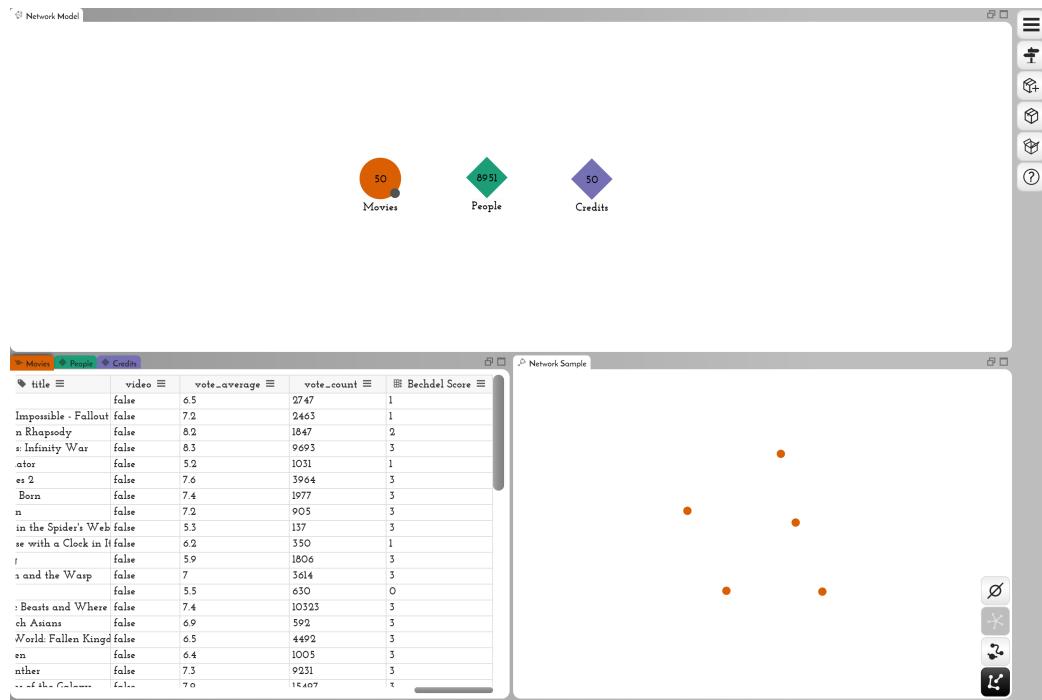


Figure 6.17. The remaining classes after the Bechdel Scores class has been deleted, as it is no longer needed.

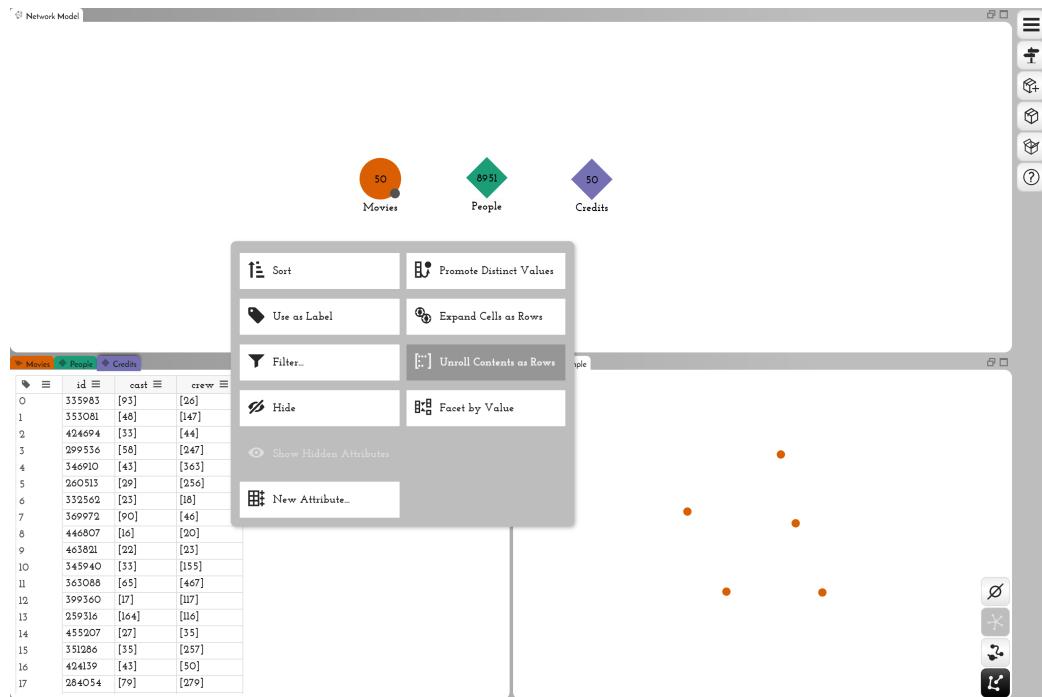


Figure 6.18. Here, we unroll nested crew structures (displayed as bracketed counts in the table) as a distinct crew class.



Figure 6.19. The state of the network model after crew and cast have been unrolled.



Figure 6.20. The connection interface, where users select a pair of attributes to join, in order to establish a connection between classes. Histograms at the top and bottom indicate how many connections per item will be created. The stacked bar chart in the center shows how each class contributes to an overall heuristic that is suggestive of which pairs of attributes may be a sensible match. The tables on the right are linked to provide context, giving a preview of the values that will be matched with each other.

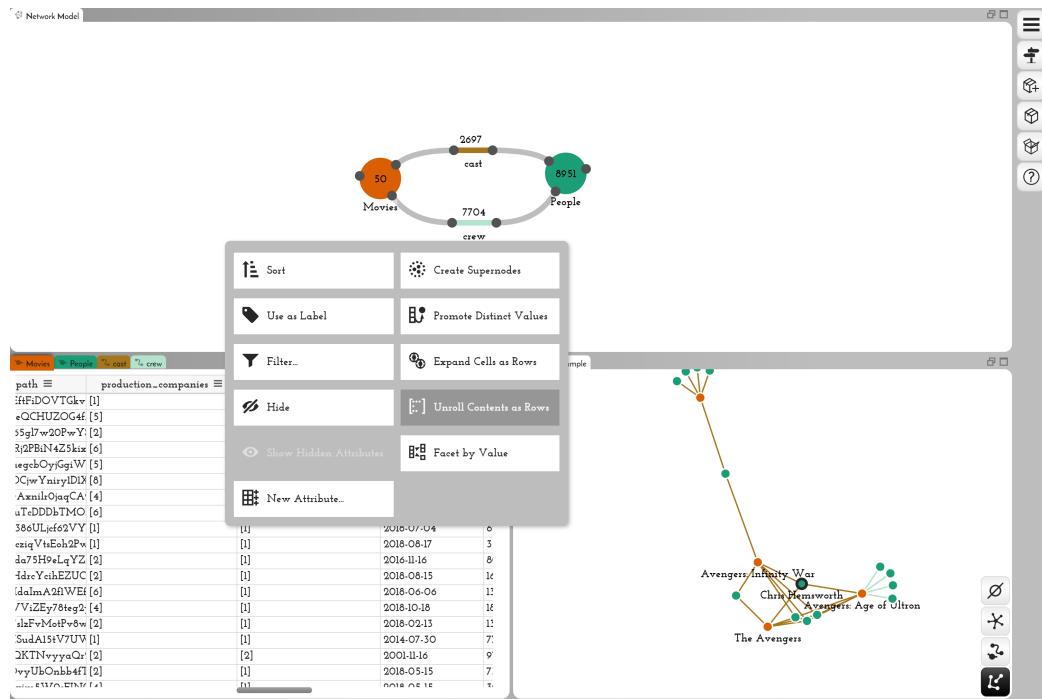


Figure 6.21. Here we unroll nested production company structures; as before, the count of nested objects within each list is displayed in brackets in the table.

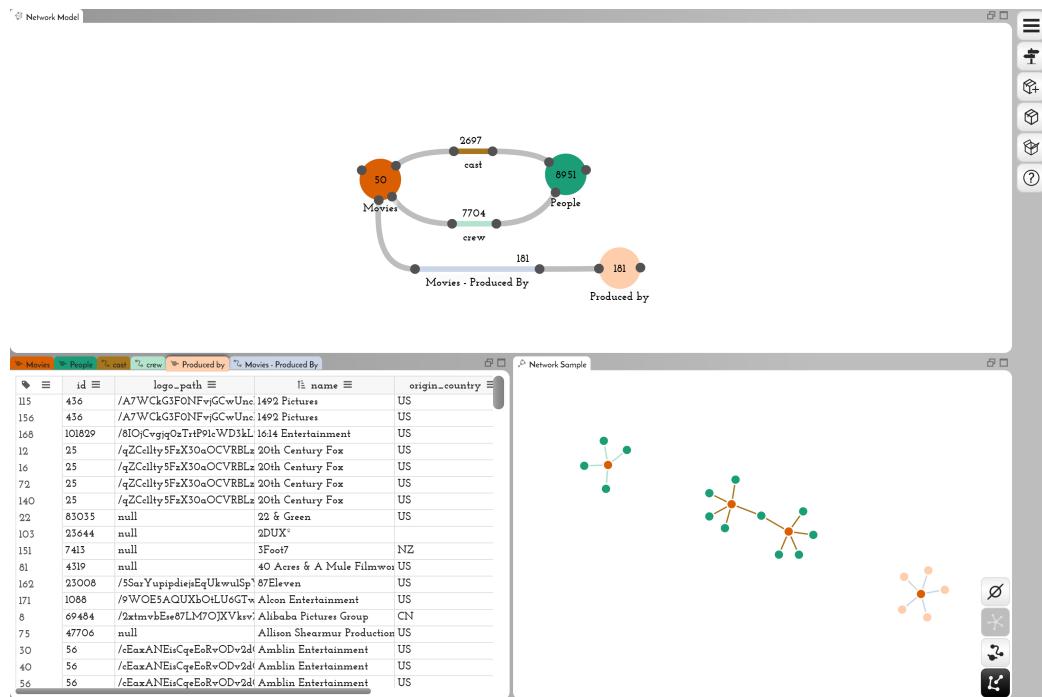


Figure 6.22. With production company objects unrolled, we can see that there are many duplicates in the name column—these are redundant items from the original dataset that need to be combined.

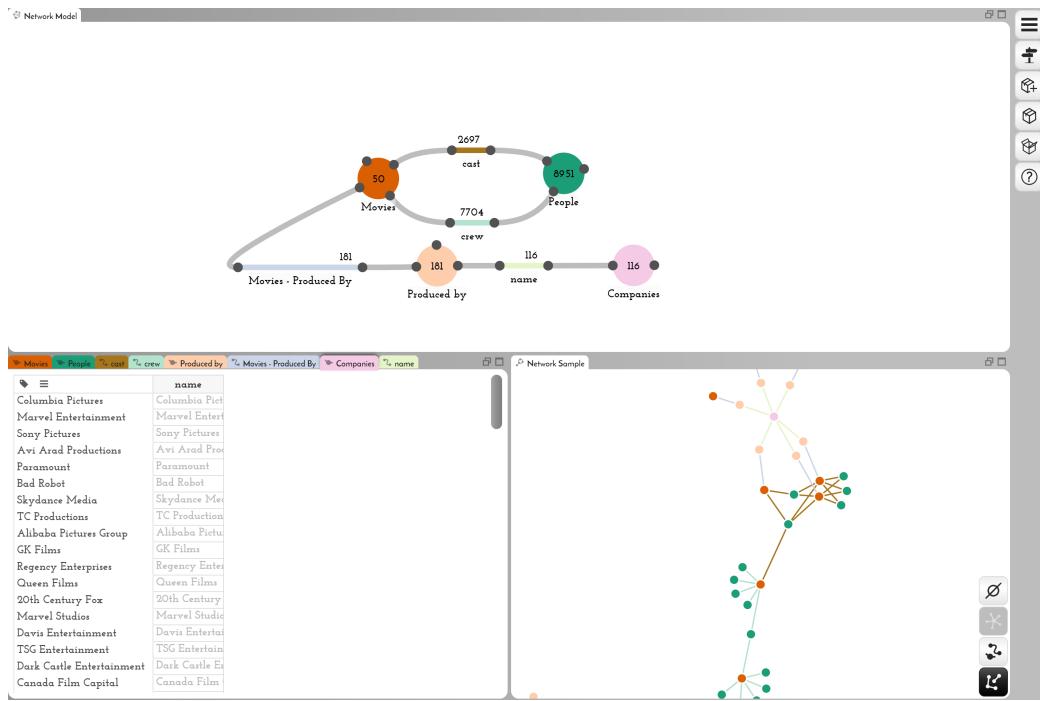


Figure 6.23. The state of the network after promoting the name attribute of the duplicate items.

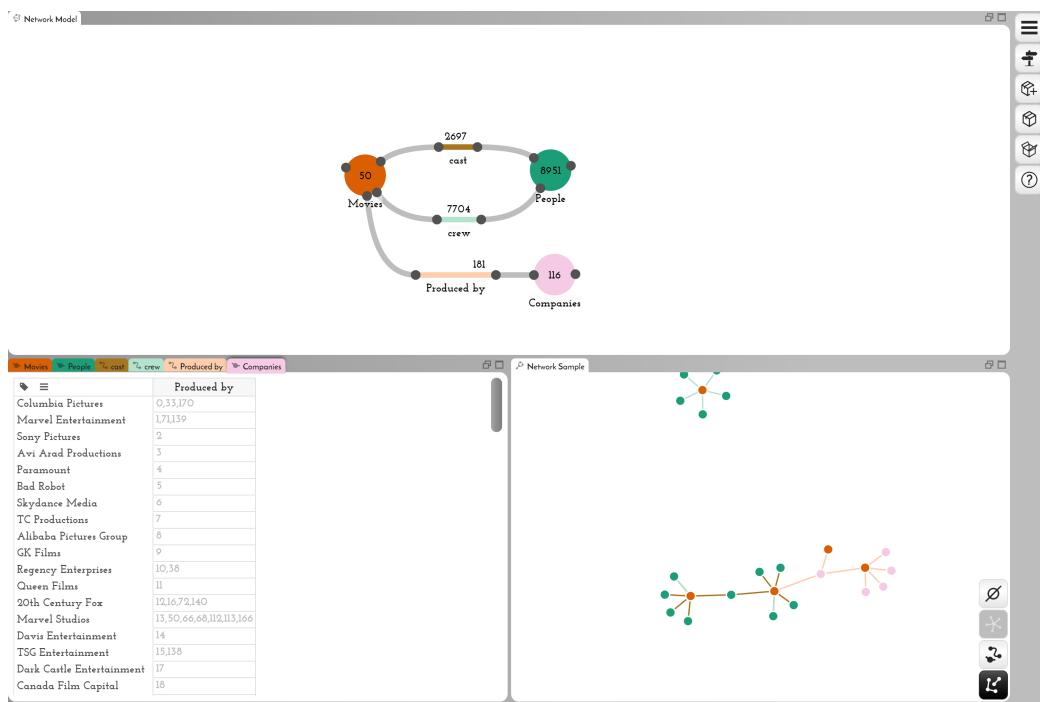


Figure 6.24. Here, we have converted the intermediate, redundant items into edges, for a cleaner network model.

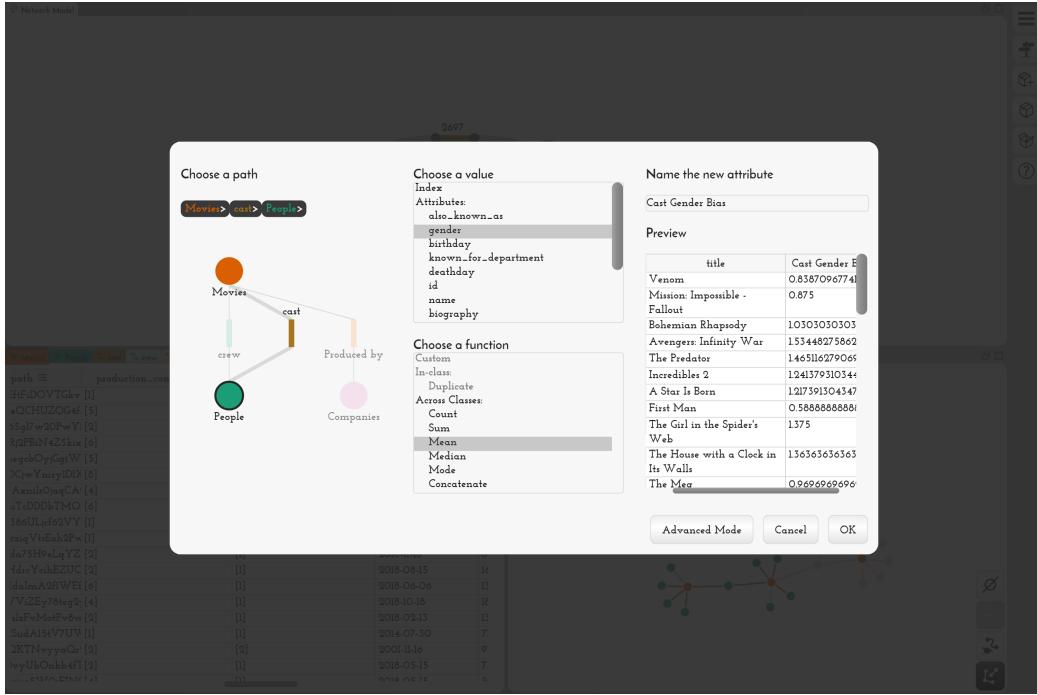


Figure 6.25. We wish to better understand gender bias in Movies; however, the gender attribute only exists on People. Here we begin to derive a Cast Gender Bias metric through cast edges, first by choosing the mean calculation as a template that we will adapt.

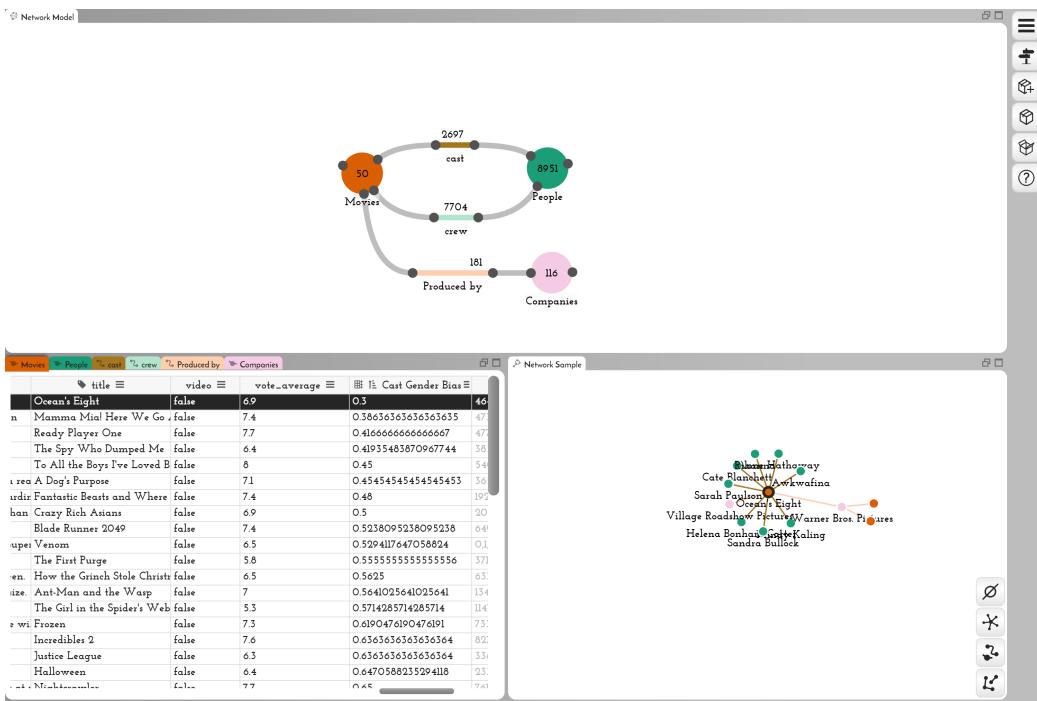


Figure 6.26. If we sort by our custom Cast Gender Bias, we see that Ocean's Eight has the lowest gender bias of the Movies in this dataset.

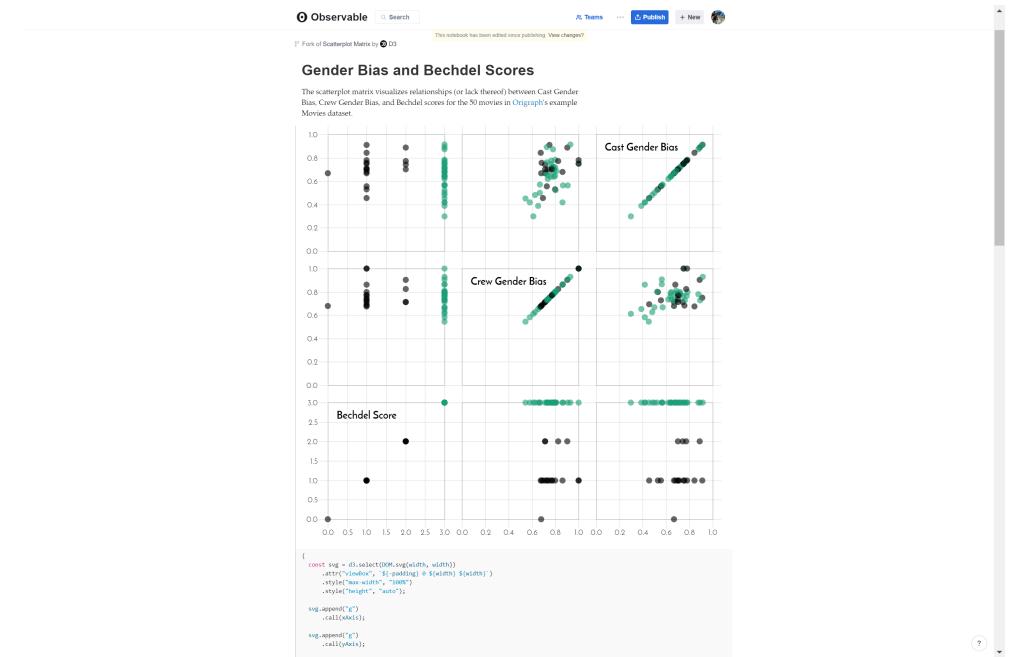


Figure 6.27. To better understand the relationships between Bechdel Scores, Cast Gender Bias, and Crew Gender Bias, we can export the Movies class as a CSV file, that we can visualize in an Observable Notebook scatterplot matrix. Green dots indicate movies that pass the Bechdel test.

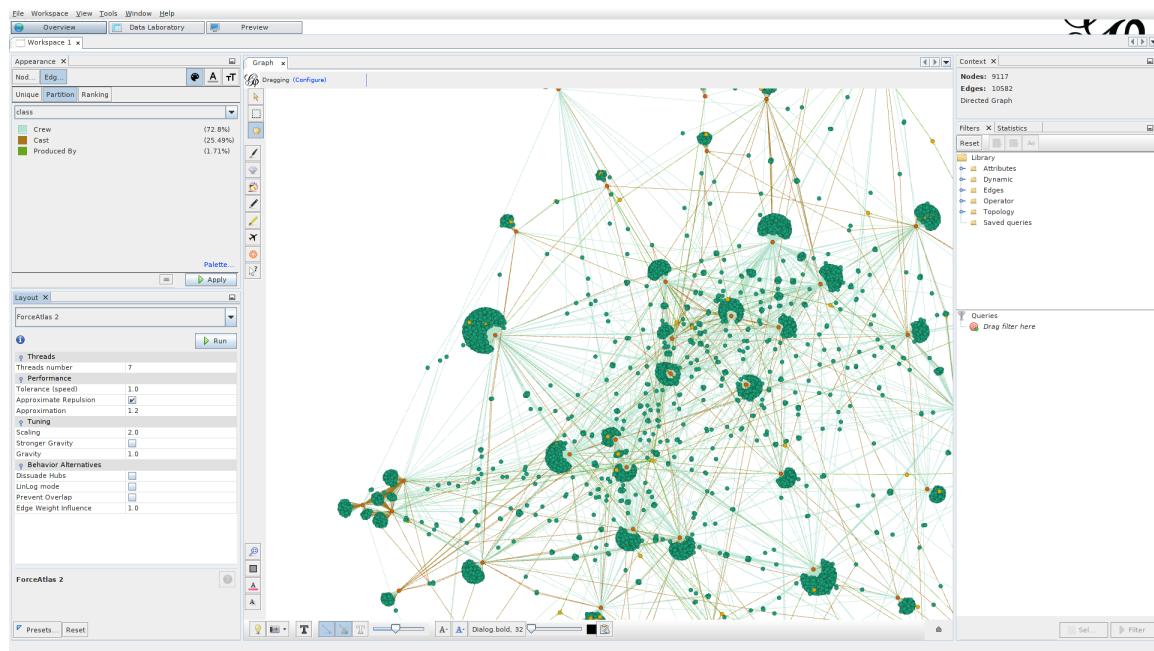


Figure 6.28. We can also export the network that we have modeled to other general-purpose visualization systems; in this case, we loaded the exported graph in Gephi.

6.8.1 Gender Bias in Movies

The network used in this example is a network of movies, actors, crew members, and production companies, which we already introduced in Section 6.2. The data were retrieved from “The Movie Database” [127], a community-built movie and TV database. The analysis question we want to investigate concerns issues related to gender bias in the movie industry.

We select the 50 most popular movies (according to TMDB’s internal ranking of popularity) and retrieve data related to these movies and store them in three JSON files. The movies file contains information on the 50 most popular movies, including attributes such as movie id, title and year of release, budget, genre, spoken languages, popularity, run time, revenue, and nested objects describing the production companies involved. The people file contains information on all people in these movies and a unique id for each person in addition to attributes such as gender, popularity, place of birth, and day of death. The credits file contains two nested objects, one for cast and one for crew. These objects contain attributes such as roles, departments, as well as the ids for movies and people. In total, the datasets contains 2689 cast items, 7704 crew items, 116 production companies, 181 relationships between production companies and movies, and 8951 people.

We also retrieved a dataset containing Bechdel ratings for 7871 movies [128], including the 50 that we retrieved from TMDB. The Bechdel test assigns a rating from 0-3 to each movie based on three criteria: 1) the movie must have at least two women in it, who 2) talk to each other, about 3) something besides a man. A movie that fails all three criteria is assigned a score of 0; a movie that passes the test is assigned a score of 3. Supplementing the TMDB data with the Bechdel rating allows us to answer some interesting gender-related questions about the movies.

We start the process of modeling a network by importing the raw data described above. Each of the imported data files is immediately available as separate, generic classes, represented by a table in the attribute view and a diamond in the network model view (Figure 6.12). The first step in modeling the network is choosing which entities to interpret as nodes and which as edges. We start by  converting generic movies to nodes.

The raw data, however, have movie information from two different sources, but we need to combine the Bechdel movies and the TMDB movies into a single node class. To

do this, we also  **convert** the Bechdel movies to nodes (Figure 6.13). Each class has an IMDB identifier as an attribute; however, in the TMDB dataset, those IDs have a “tt” prefix before each number. In order to  **connect** the two node classes, we first need to  **derive an attribute** for the Bechdel movies, appending a “tt” string to its IMDB identifier (Figure 6.14). Now, we can  **connect** each movie class (Figure 6.15).

In combining the movie classes, our main objective is to copy the Bechdel rating into the richer TMDB data for each movie. To do this, we  **derive a connected attribute** for the TMDB movies, duplicating the original Bechdel score—in this case, we select the “rating” attribute from the Bechdel movie class, and compute the median rating (Figure 6.16). Note that the general case for deriving connected attributes is a many-to-one relationship. Because the data here happens to have a one-to-one relationship, the median computation trivially copies the only connected value. Once we have copied the value, the Bechdel movie class and its connection to the TMDB movies are no longer needed, and both classes can be deleted (Figure 6.17).

We then select the credits table, which contains the cast and crew information as nested objects. We use the unroll operation on both cast and crew, which creates a new class for each one (Figure 6.18). Since we no longer have a need for the credits table, we delete it from our model. We are interested in using the cast and crew information to connect people to the movies they were involved in. To do that, we first  **convert** people to nodes, cast and crew to edges (Figure 6.19), and then  **connect** both cast and crew to people and movies (Figure 6.20).

We then scroll through the movie attributes to “production_companies.” The values in this column are arrays of objects, so we unroll them into their own class (Figure 6.21). The unroll operation also connects the new items to the movie nodes they originated from. We rename the newly created class to “produced by” to better reflect the meaning of this class in our model.

At this stage, because the same production company is associated with multiple movies, we notice the presence of duplicate rows, showing information for the same company (Figure 6.22). We leverage the  **promote** operation on the company “name” attribute to create a new class with unique company names. We then rename the newly created class to “Companies” (Figure 6.23). However, companies are now connected to

movies only indirectly via the “produced by” nodes. Semantically, it would be more meaningful to connect companies directly to movies via an edge, which we can achieve with the `convert` operation, applied to the “produced by” node class (Figure 6.24). Since Origraph supports rich edge attributes, these edges preserve all the attributes of the original data.

With the initial network model built, we are ready to reshape the network to answer the analysis questions regarding gender bias for each movie. To do this, we compute a new attribute in the movie table. Similar to the earlier task of copying Bechdel scores, this operation requires information from connected classes, so we use the `connectivity-based attribute derivation` operation. In order to calculate the gender bias for actors in a movie, we must access the gender of all the people who are connected to the movie via a cast edge. In this example, we are interested in the ratio of men to women, so we select gender from the list of attributes, choose the mean computation that approximates what we want to compute (Figure 6.25), and then select the advanced mode, which allows us to modify the mean code to compute the ratio of men to women (Figure 6.11). A preview table on the right shows sample values to ensure we are computing the attribute correctly.

Once the gender bias has been computed, we sort on this attribute in the table, which reveals that of the movies in this dataset, “Ocean’s Eight” is the one with the highest ratio of women to men (Figure 6.26). Interestingly, the movie with the highest gender bias, “BlacKkKlansman,” passes the Bechdel test, with a score of three—this suggests that one (or both) metrics possibly oversimplify the concept of gender bias. To explore the relationship between each metric, we can export the Movies class as a CSV file, and visualize Cast Gender Bias, Crew Gender Bias, and Bechdel scores in a scatterplot matrix (Figure 6.27).

A follow-up question is to find out which actors tend to be cast in movies with a lower gender bias. We can explore the local connections around movies of interest using Origraph’s sample interface (Figure 6.1), or export finished network dataset to a more dedicated network visualization tool such as Gephi (Figure 6.28).

6.8.2 Money and Political Support for the War in Yemen

Our second use case focuses on a network of current US senators, voting behavior on the recent bill regarding US support of the Yemen war, donors of these senators, and their social media statements related to this issue. This topic has received considerable attention in the media [129], [130] and serves as an interesting use case to demonstrate the ability of Origraph to connect data from disparate sources to tell a compelling story. Figures 6.29–6.44 show the full state of the interface at each major step.

The bill in question (session 2, roll call 250) was to determine whether the US should remove military support from the war in Yemen. The final vote count was in support of removing support. Yet it is still interesting to model a network that can address questions related to the votes of senators and their connections to donors. One such question is whether senators who voted against the Yemen bill were financially supported by a specific subset of donors with an interest in the conflict. A related question is whether senators were more or less vocal in their support or opposition, which we can estimate based on press releases and tweets.

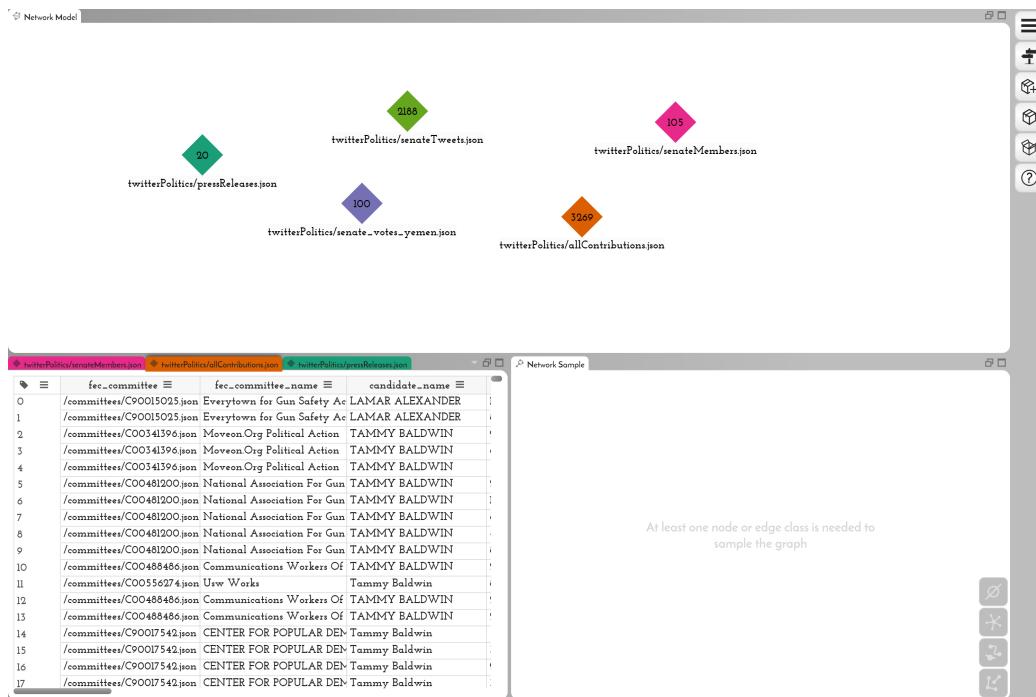


Figure 6.29. In this use case, we combine data from ProPublica [131] and Twitter [132] to investigate the relationship of donors and US senators, specifically how senators voted about the war in Yemen.

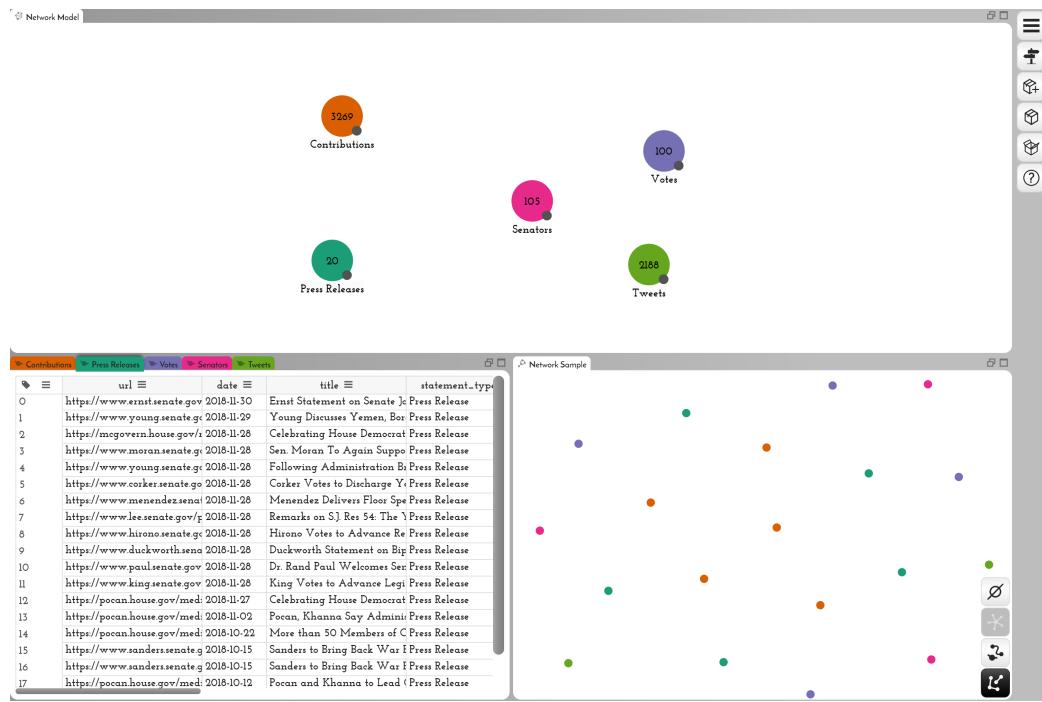


Figure 6.30. We begin by converting each raw table to a node class.

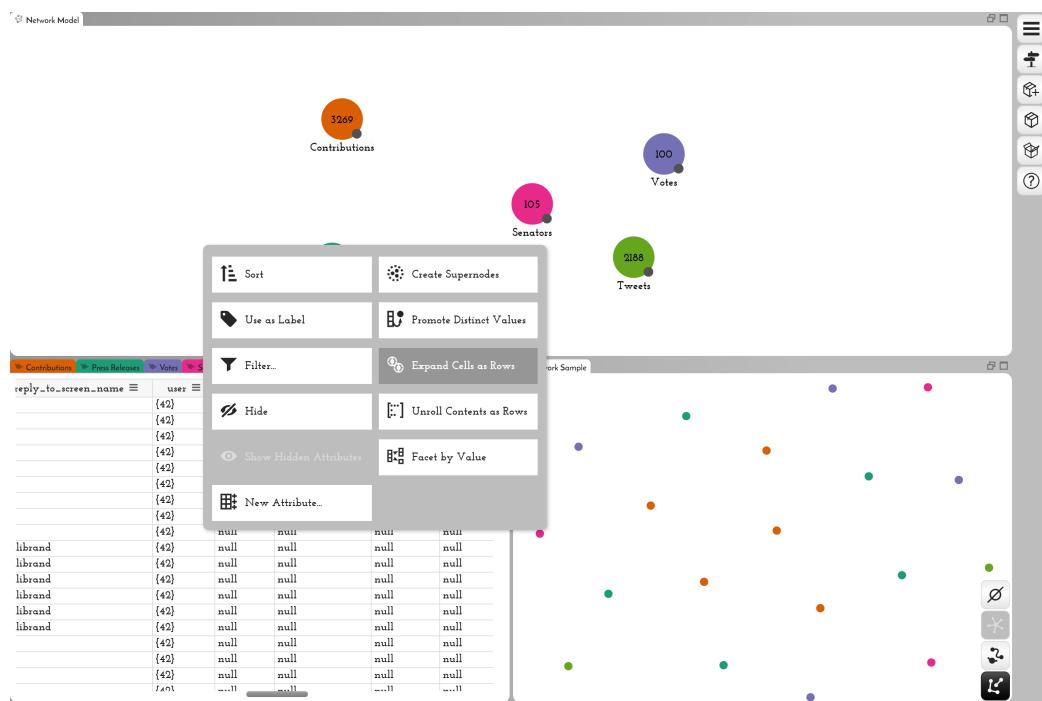


Figure 6.31. To connect individual tweets to senators, we first expand nested user information. The curly braces in the cells indicate that a single object is nested in each row, with 42 attributes of its own.



Figure 6.32. With user objects expanded, we can see there are many duplicates—there is a distinct user object for each tweet. At this stage, we could promote repeated values; however, our objective is only to connect tweets to senators.



Figure 6.33. To connect senators to tweets, we match their `screen_name` attribute from the Twitter user nodes to the `twitter_account` attribute of ProPublica Senator nodes.

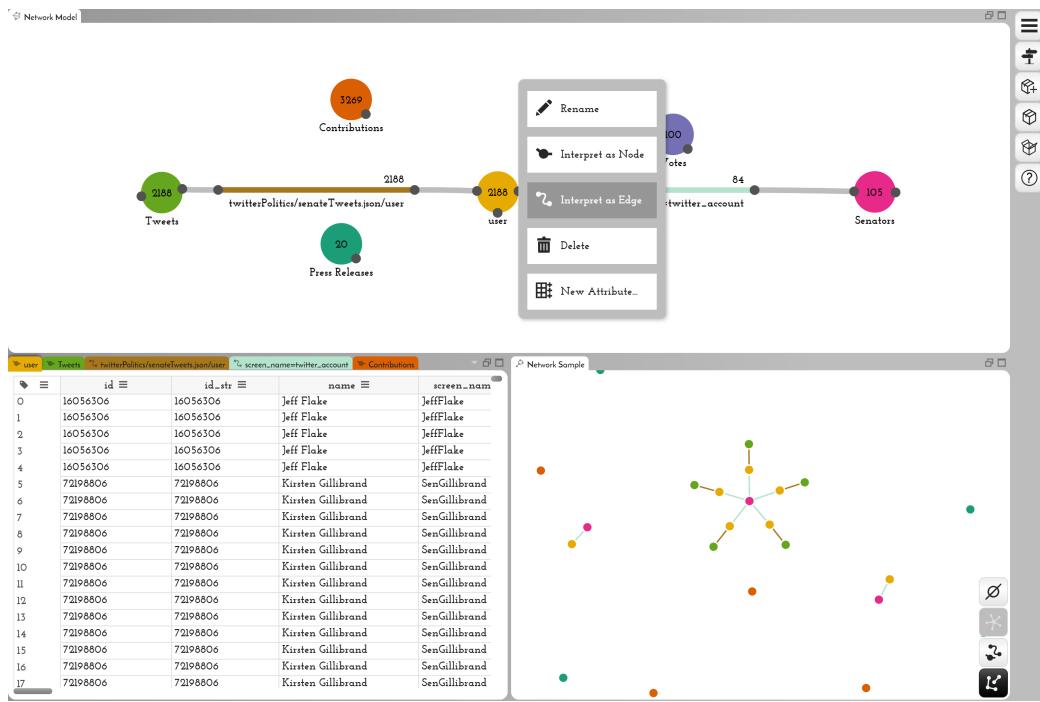


Figure 6.34. We clean up the network by converting the intermediate user nodes to edges.

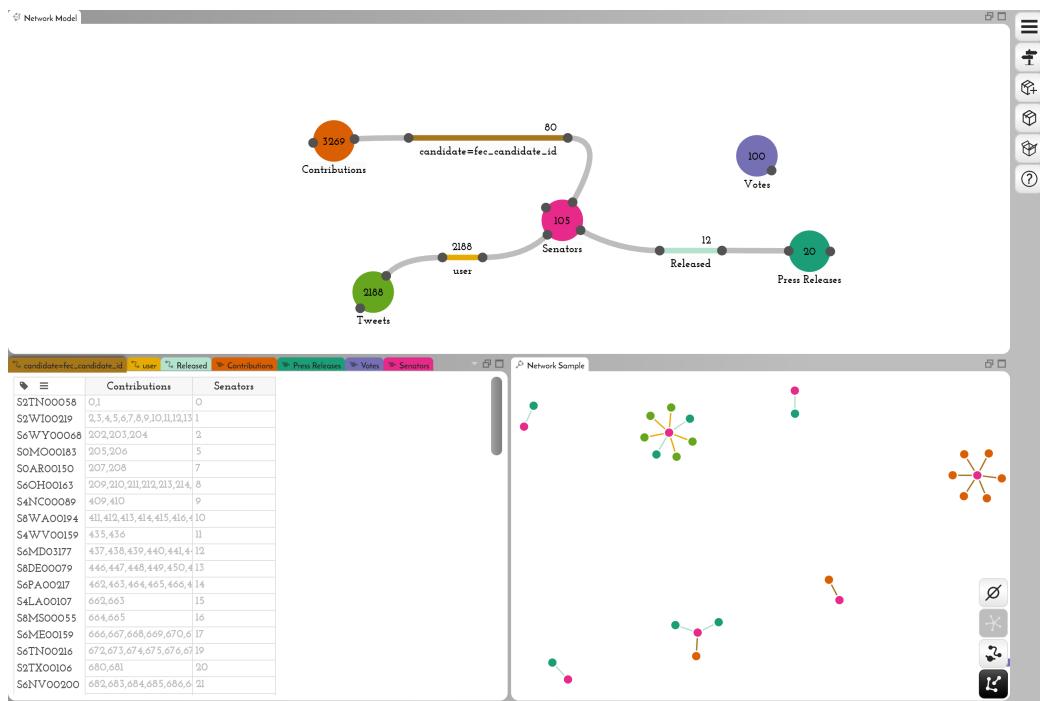


Figure 6.35. With tweets connected, we add also connect the Contributions and Press Releases classes based on their native IDs.

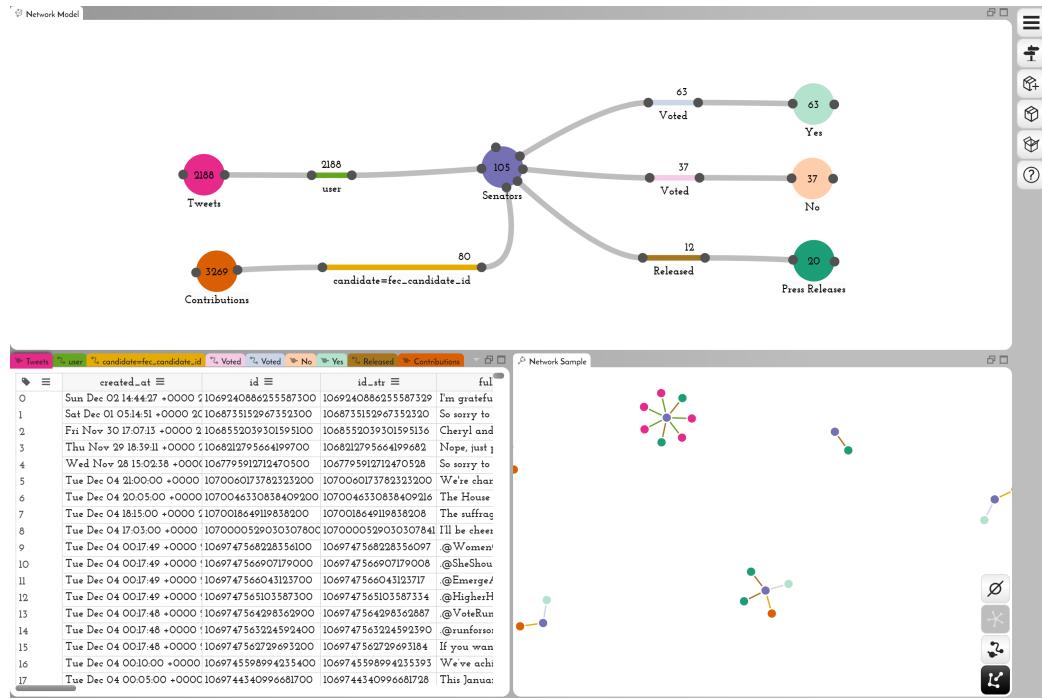


Figure 6.36. As we are interested in how Senators voted, we facet the Votes class before connecting them to Senators.

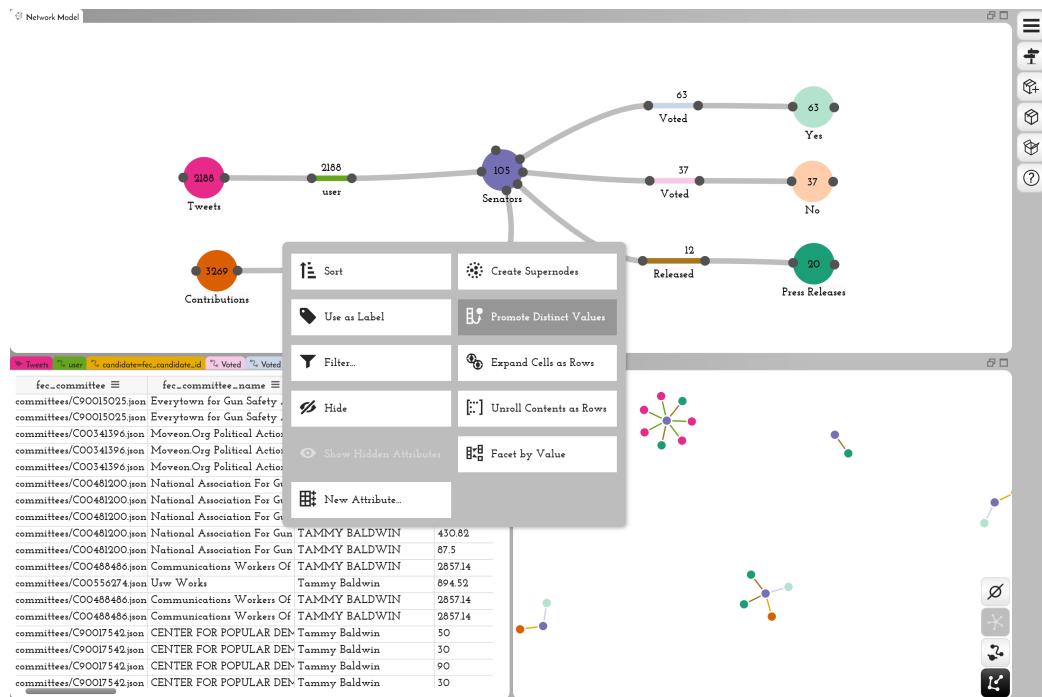


Figure 6.37. We are also interested in donors, and their relationships to Senators—however, the Contributions table contains many rows from the same donor. To identify individual donors, we promote the fec_committee_name attribute.

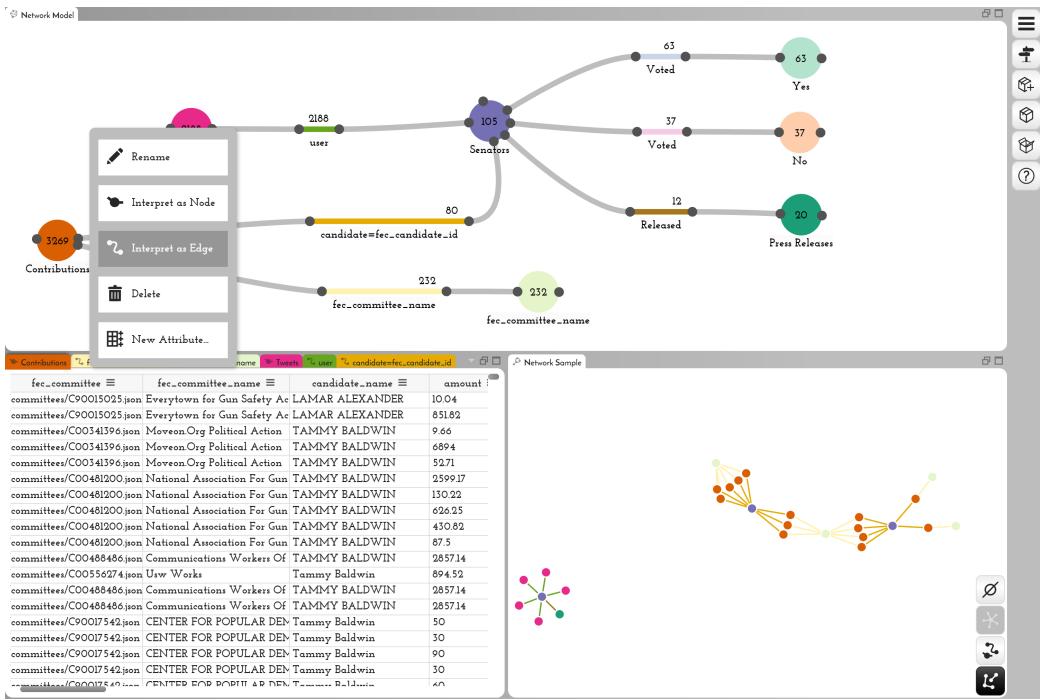


Figure 6.38. To clean up the network model, we now convert the Contributions class to an edge class.

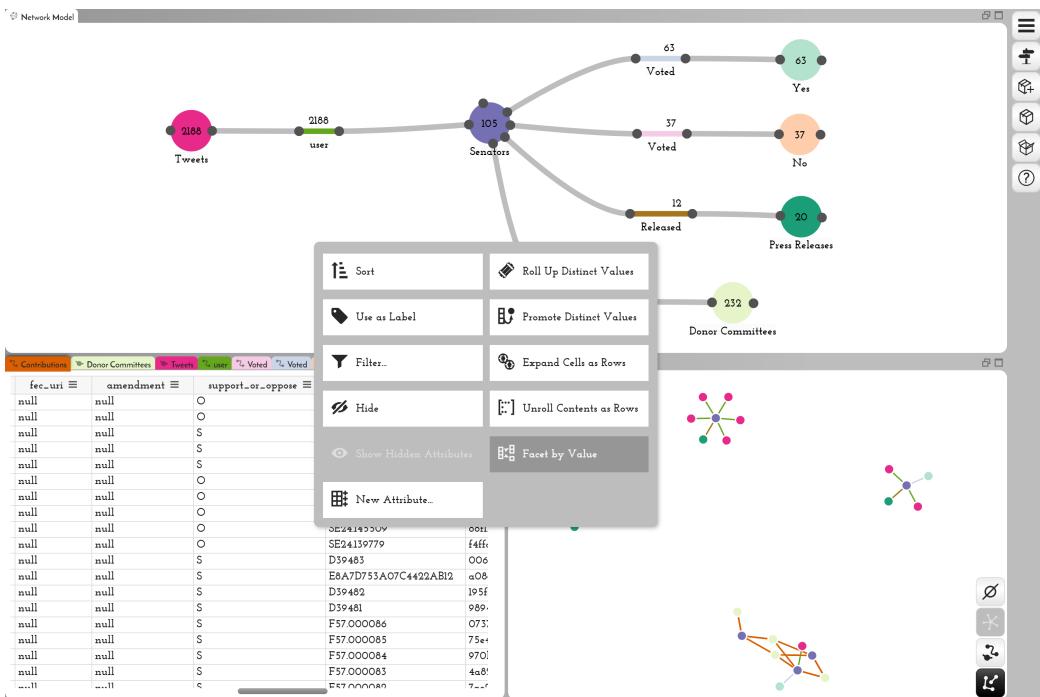


Figure 6.39. Political contributions can be made both in favor, or in opposition to, a Senator. To distinguish between these different relationships, we facet the Contributions edge class.

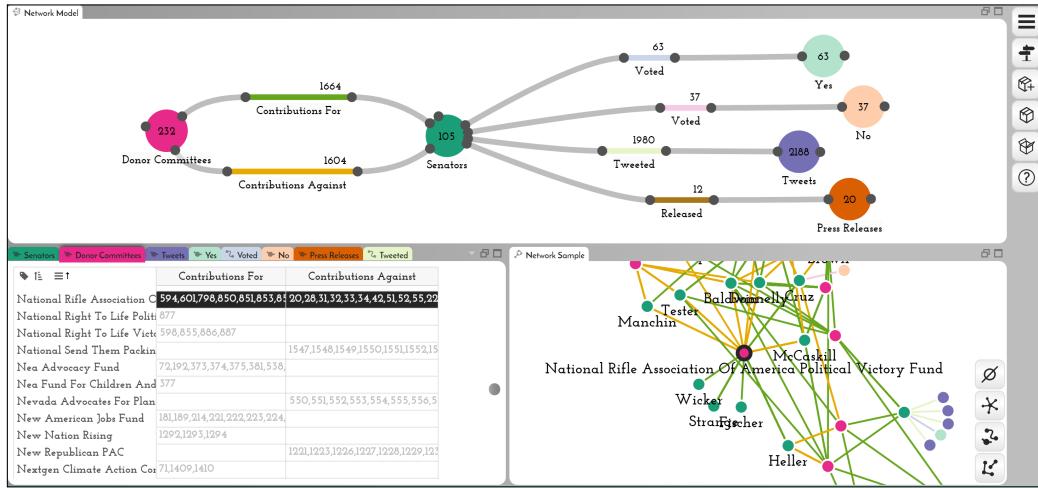


Figure 6.40. A political donor, vote, tweet, and press release network. Senators are connected to committees that donated to either support or oppose them. They also have edges connecting them to how they voted on the Yemen bill, their tweets, and press releases.

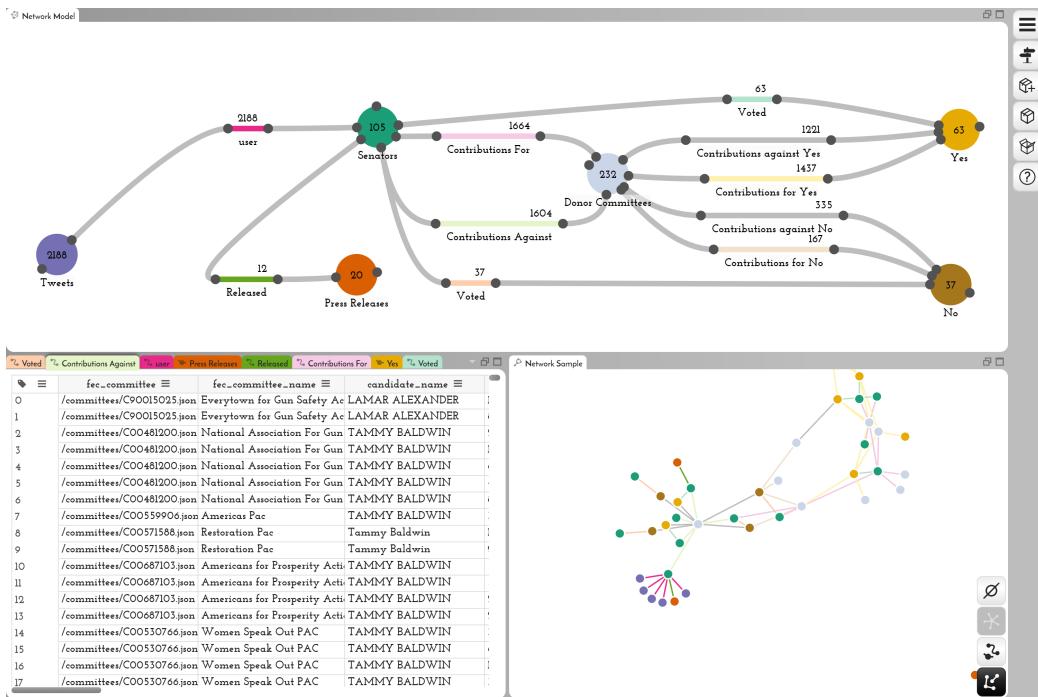


Figure 6.41. If we wish to model relationships between donors and the votes on the Yemen bill, we can project edges that bypass Senators, connecting donors and votes directly. Here, we produce four new edge classes: links between donors that financially supported, or opposed, votes in favor, or against, the bill.

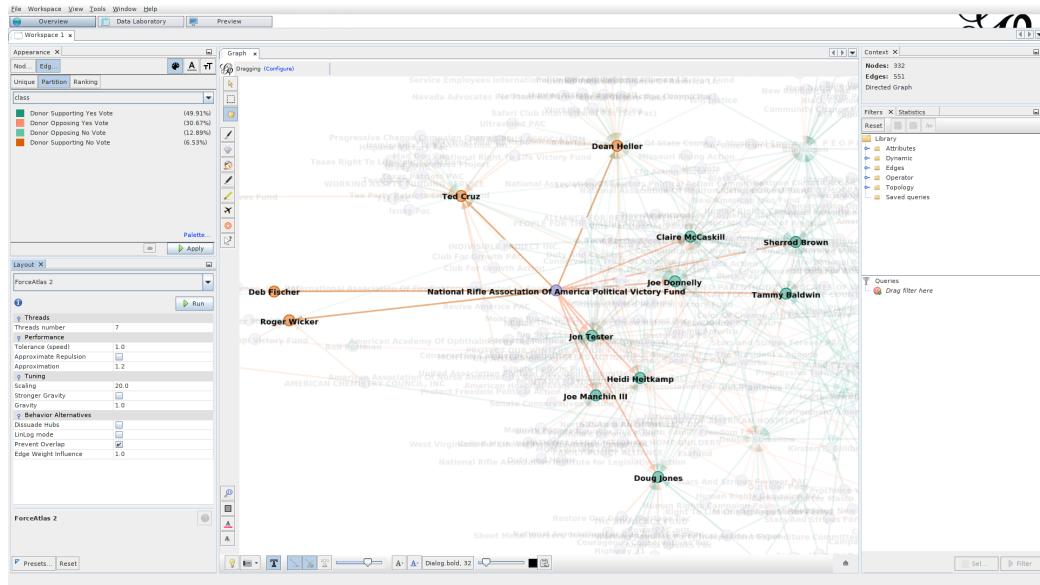


Figure 6.42. Projected edges, donors, and votes are exported to Gephi. We highlight the NRA, as it did not donate to any senators who voted Yes—only to those who voted No, and donated in opposition to other senators who voted Yes.

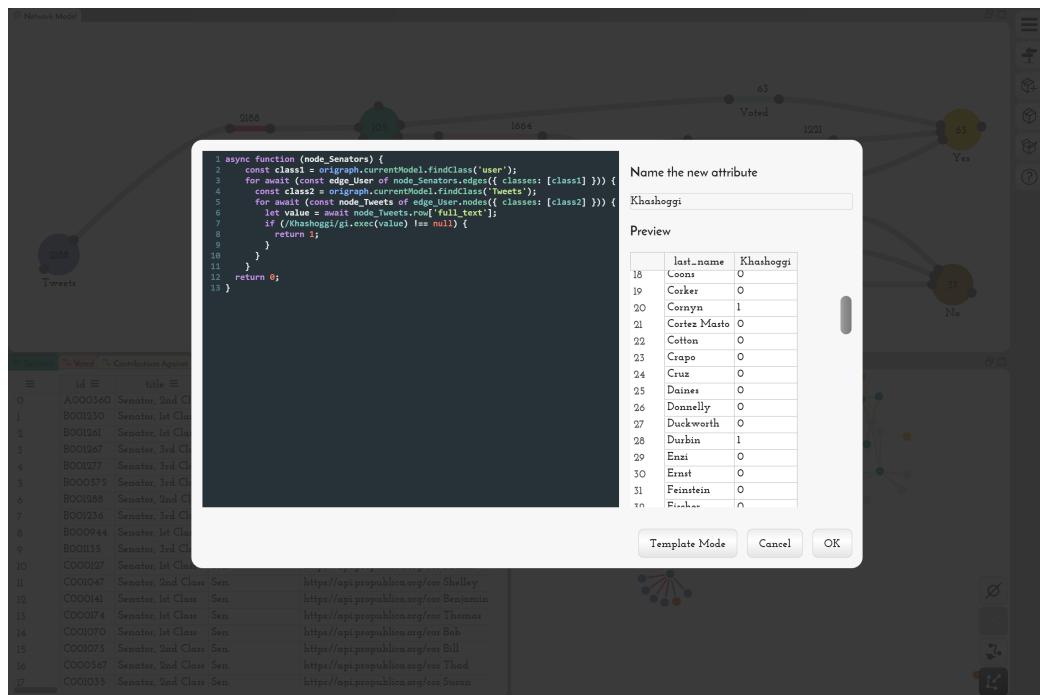


Figure 6.43. To ascertain which Senators were more or less vocal about the issue, we can derive attributes that indicate whether or not strings of interest appeared in their tweets or press releases. In this instance, we compute whether the string “Khashoggi” appears in the text of a Senator’s tweet.

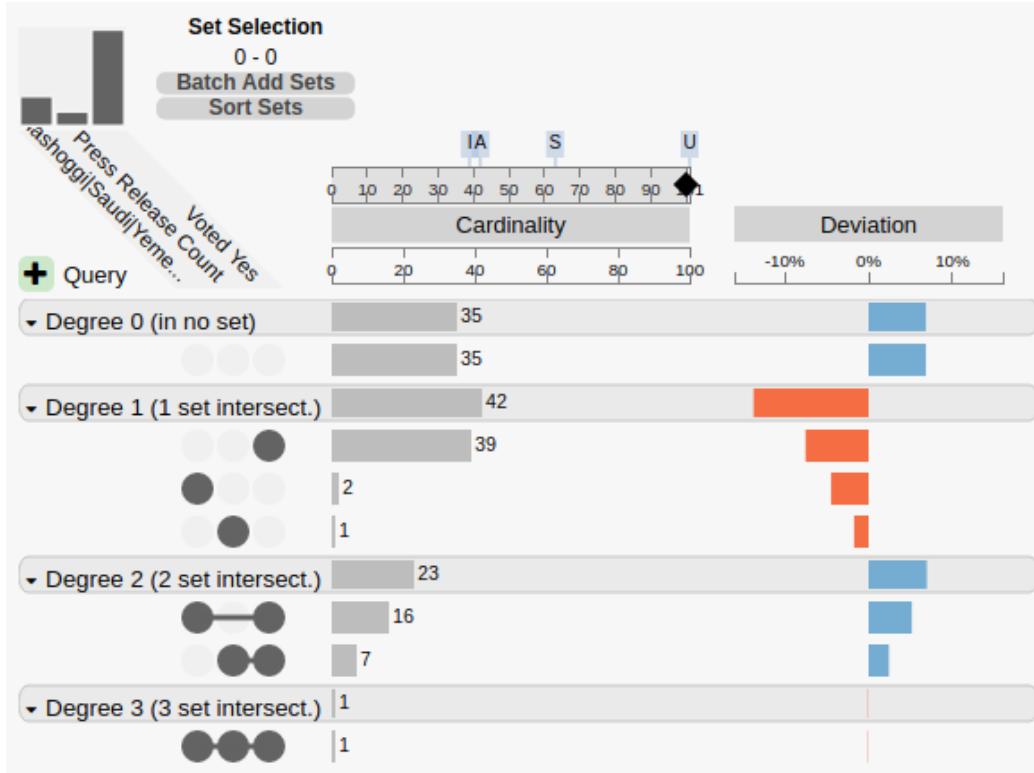


Figure 6.44. Whether senators tweeted about, issued press releases, and/or voted Yes, visualized in UpSet [109]. We can see that, with the exception of only three senators, all senators who voted No were relatively silent about their votes.

We obtained the data from the ProPublica Congress API [131] and through the Twitter API [132]. Again, we did not perform any preprocessing on the datasets retrieved from these APIs. The raw data files imported in this example include information for all current senators including gender and political party, the way each senator voted for the bill on Yemen, all press releases made by members of the senate in relation to the Yemen bill, all FEC reports about donations made in 2018 that either support or oppose a member of the senate, and all Tweets made by senators from Nov. 28 to Dec. 4, 2018.

Once the raw files have been loaded (Figure 6.29), we **convert** senators, votes, campaign contributions, press releases, and tweets to nodes (Figure 6.30). In order to **connect** senators to their tweets, we must first extract the Twitter user information from each tweet. User information is stored as nested objects within each tweet; we generate a new class with all users by expanding the user attribute in the tweet class (Figure 6.31), which automatically connects the instances of the newly created user class to the original tweets

(Figure 6.32). We can now **connect** senators to their Twitter accounts based on their screen name, an attribute of the senators class (Figure 6.33). Because we ultimately care about connecting senators to their tweets, we can abstract away the Twitter account class by **converting** it to an edge. We now have a model of senator nodes, connected directly to their tweets (Figure 6.34). The next step involves **connecting** senators to their votes, their press releases, and any donations made towards or against their campaign. This can be done directly with the connect interface, as Press Releases, Votes, and Donations all have an attribute that directly references the senator IDs (Figure 6.35). We are particularly interested in distinguishing between senators who voted for or against the bill, so we **facet** the votes node into separate “Yes” and “No” classes before connecting them (Figure 6.36).

The last step in modeling this network involves extracting out the individual donation committees. Since several donations are made by the same committee, we first want to extract all unique committees from the “Donations” class. We leverage the **promote** operation on the “committee name” attribute which generates a new “Donor Committees” class, with edges connecting each committee to their donation in the “Donations” class (Figure 6.37).

Because we are interested in which committees support or oppose certain senators, we **convert** the donations node into an edge (Figure 6.38), and then **facet** the edge into contributions that support and those that oppose a senator (Figure 6.39).

With this network modeled, as shown in Figure 6.40, we can turn to the initial questions regarding relationships between donors and vote outcomes on the Yemen bill. Our current network connects donor committees to Yes and No votes through specific senators. If we are interested in understanding the direct connection between financial contributions and votes, we can **project new edges** that connect donor committees directly to vote classes (Figure 6.41). This gives us a reasonable proxy for donor interest in this issue; exporting the new projected edges, with donor and vote nodes, into Gephi allows us to explore the donor interest network (Figure 6.42). We discover that, even though many republicans voted Yes on this bill, the NRA only supported republican senators who voted No, and opposed senators who voted Yes. Although this does not imply a causal relationship for support of that particular conflict, this could suggest a point for an analyst to investigate

further.

Having incorporated Twitter and press release data, we may also be interested in whether specific senators were vocal about this issue. We can derive attributes on senators to ascertain whether their tweets contain relevant “Khashoggi,” “Saudi,” or “Yemen” strings, as well as whether or not a senator issued a press release about their vote (Figure 6.43). As we are interested in the set relationships between these new attributes, we can export the senators class to UpSet [109], where we can see that, with the exception of only three senators, all senators who voted No were relatively silent about their votes (Figure 6.44).

6.9 Discussion

Data abstractions often change in response to evolving exploration and analysis tasks. The shape of the data is a proxy for an analyst’s real-world problem, and must be continuously assessed and refined [133]. We argue that transforming a dataset into the form best suited to answer analysis questions is essential, yet the tools currently available for wrangling multivariate network datasets are not powerful enough. The alternative of using scripting or database query languages is time consuming, requires skill sets many analysts do not have, and does not allow for rapid iterative exploration of the changes made.

To address this, we introduce novel operations and visual analysis interfaces to support sophisticated interactive network wrangling. By giving users the freedom to explore alternative data abstractions, tools like Origraph have the potential to become powerful thinking tools that could begin to address the open challenge of how to support visualization practitioners in exploring more diverse data abstractions [3], [10]. Although the design space of data wrangling tools is still in its infancy, it is worth considering its breadth: identifying data wrangling as a distinct category from data analysis [54] may need to be extended further to distinguish between data analysis, classical “wrangling,” and abstraction transformation. The standard notion of data wrangling [54] is often viewed through a lens of data integration, diagnosing, and cleaning data problems. In contrast, abstraction transformation is motivated by users’ mental models and hence requires a distinct set of methods that support different operations. Nevertheless, each category benefits from tight

integration with the others. For example, in Origraph, we integrate data analysis through visualization with wrangling, which is also imperative for checking whether wrangling operations have their intended effects.

6.9.1 Comparing Origraph to a Computational Workflow

All wrangling operations that can be executed with Origraph can also be implemented using different programming languages. However, using a programming language has multiple disadvantages: first, the skill level required to implement certain operations limits the number of individuals that can execute them significantly, making it prohibitive for data-literate nonprogrammers (e.g., individuals that use Excel, Tableau, or Statistics software such as SPSS) to do network wrangling.

All modeling operations in Origraph can be executed without programming. This is noteworthy because operations that change the database schema are relatively complicated in scripting languages such as SQL. Only nonstandard attribute operations require programming in Origraph: the “apply” functions following the split-apply-combine paradigm and advanced conditionals for filtering items. Origraph provides multiple defaults for these functions, such as sums, means, and concatenation. However, the design space of useful functions is considerable, hence Origraph allows analysts to write custom expressions. We argue, however, that these expressions operating on lists of attributes are much easier to write compared to, for example, operations that change the network model. Nevertheless, we plan on simplifying our scripting interface, for example, by making it more like Excel, with basic formulas, easy references to data sources, and accessible documentation.

6.9.2 Comparison to Other Tools

Origraph shares parts of the vision and functionality of Orion [1] and Ploceus [2]. Origraph, however, is unique and novel with regards to three aspects: (1) it supports important operations that are missing in Orion and Ploceus, (2) it treats edges as first-class objects that can be associated with complex attributes, and (3) it contains sophisticated, algorithm-supported visual interfaces that make executing complex operations easier. Orion and Ploceus also do not support nested data structures (lists and hierarchies) as attributes of the tables.

With regards to operations, Origraph supports all operations listed in Section 6.4. Ploceus and Orion do not support \heartsuit **converting between nodes and edges**, \heartsuit **creating supernodes**, \heartsuit **connectivity based filtering**, \heartsuit **changing edge direction**, and **connectivity based attribute derivation**. Although Origraph and Ploceus support \clubsuit **projecting edges**, they do so only for immediate neighbors, whereas Origraph can project edges based on arbitrary complex paths. Notably, several of the operations that are unique to Origraph are about leveraging network structures, e.g., aggregating attributes at a certain distance from source nodes. Table 2.1 contains a comparison of these operations. Finally, in contrast to Orion and Ploceus, the code of Origraph is open source and the tool is available for anyone to upload datasets.

6.9.3 Limitations

6.9.3.1 Visualization

The current focus of Origraph is on making sophisticated data wrangling operations as easy as possible to execute. Although Origraph supports analysis through visualizations, there are many ways in which the visualizations of the network and the attributes can be improved. We are planning to integrate Origraph with a general purpose multivariate network visualization system that is based on existing tools developed at our lab [120], [121], [123].

6.9.3.2 Cleanup and Missing Data.

Origraph does not support operations on tabular data that analysts have come to expect from a wrangling tool. For example, the support for cleanup and dealing with missing data is limited. We believe that such operations are better addressed in separate tools before importing tables into Origraph.

6.9.3.3 Scalability

Although Origraph scales to thousands of nodes and edges, it does not scale to arbitrarily large networks. We plan on improving scalability using an approach common to visual data wrangling tools: loading a sampled dataset. Operations can then be applied and refined on the sample using visual inspection. After the transformation is completed, a script is generated that can be used to process a larger dataset offline, potentially lever-

aging cloud infrastructure.

6.9.3.4 Connectivity Heuristic

In our experiments with various datasets, we found that our heuristic always gives high scores to the right combination of attributes, but also observed that index to index scores are commonly highly ranked. If, for example, two datasets have the same number of rows but have nothing else in common, our heuristic would give a perfect score to the index to index connection. Since some datasets rely on the index of items to establish connections between datasets, we can not generally exclude it from our computation. However, we are considering to treat connectivity combinations involving an index separately from attribute to attribute connections. With regards to scalability, we observed that our heuristic can be slow with large classes. It has a runtime complexity of $\mathcal{O}(n * m * i * j)$, where n and m are the number of unique items in the classes (which are frequently very large), and i and j are the number of attributes plus the index in the classes (which are typically small). A sampling based heuristic that reduces the runtime complexity to $\mathcal{O}(k * m * i * j)$, where k is a user-chosen value that is much smaller than n and should result in a significant speed-up with a limited loss of accuracy.

6.10 Conclusions and Future Work

By implementing the set of operations that we have identified, Origraph is a first step toward allowing data-literate nonprogrammers to wrangle networks to answer important questions in their area of inquiry and to produce visualizations they can use to communicate their findings. Origraph is designed to ingest raw data, e.g., in the form of JSON retrieved from an API, and then enable analysts to wrangle the data into a network that corresponds to their mental model. Build-in visualization capabilities enable analysts to quickly iterate between analysis and modeling. Once a network is in the desired form, it can either be investigated within Origraph or exported for analysis in more sophisticated multivariate network visualization tools. Through two use cases, we have demonstrated Origraph’s expressive potential.

We hope to explore and augment Origraph through ongoing deployment, testing, and redesign. This will include more exhaustive evaluations with users testing and improving

the tool's usability; exploring the usefulness of each operation and what operations may be useful beyond the ones that we have identified; and identifying gaps in its expressiveness. Additionally, we plan to integrate more representative sampling and improve scalability; to give users access to more sophisticated algorithms, such as seriated instance ordering, or clustering-based group assignments; to add provenance features to make it possible to edit, audit, share, document, and replicate users' transformation processes; and to integrate more closely with data cleaning and multivariate graph visualization to more fully support end-to-end analysis workflows.

CHAPTER 7

EXTENSIONS AND FUTURE WORK

Interviews and observations of graphic designers working with data [3] have provided unique insight for initial solutions to two major challenges in visualization design: a bridge model that enables software to bridge different modalities such that it can better support rapid prototyping workflows [4]; and a visual technique [5] and software system [6] that enable reshaping graph data abstractions. Although we have focused on expanding missing software capabilities, reflections [134] on our observations, models, and software design process reveal many gaps in existing knowledge, and are suggestive of ways that gaps can be filled. These include gaps in existing software functionality, as well as knowledge gaps with respect to interaction modalities and understanding real-world workflows.

7.1 Software Gaps

We discuss four major themes in Chapter 3—the practice of manual encoding; relaxing the sequence of operations; placing data on existing graphics; and designing effective data abstractions—however, our practical solutions only represent initial explorations of two. The bridge model in Chapter 4 shows one way to relax the sequence of operations in a visual design workflow, and the tools in Chapters 5 and 6 support exploring alternative data abstractions. Others have begun to explore the remaining two themes: of supporting designers’ established practice of manual encoding [35], [135], as well as tools that fit the mental model of placing data on existing graphics [46], [47], [136], [137].

There is ample opportunity for software to improve with respect to each of these four themes, in the form of new interface designs, as well as refinements on existing designs—especially our own. In discussing each tool, we have identified possible features that we have not implemented, or tested, including the following: additional, hypothetical bridges in Chapter 4; adaptive edges and attributes in Chapter 5; and exporting scripts that can wrangle network models at scale in Chapter 6.

7.2 Interaction Modality Gaps

In exploring various interface designs, we have repeatedly encountered the fundamental tension between usability and expressiveness [138] in both visual design and data wrangling contexts. Although we have principally focused on demonstrating *integration* and *expressiveness* through the use of example galleries [139], future redesigns for *creativity support*, and *usability* evaluations of our tools may provide useful test environments for exploring how to balance the trade-offs between different interaction modalities. For example, direct manipulation and programming are both very expressive ways of working, yet direct manipulation has clear advantages for tasks like drawing [140],[141], and textual programming has advantages for things such as specifying custom logic [142]. More traditional GUI elements, such as menus and forms, tend to afford less expressiveness than direct manipulation or textual programming—yet their familiarity to most users suggests that they may have *guidance*, *learnability*, and *usability* advantages, in spite of their increased semantic and articulatory distances [143]. Future bridges between visual tools may represent an opportunity to examine more closely which interaction modalities represent the most effective balance for which design tasks.

Similarly, in Origraph’s design, we use all three interaction modalities for different tasks, including direct manipulation, traditional GUI elements, and textual programming. There is also some overlap:  deriving and  filtering attributes can be accomplished with traditional GUI controls, as well as with textual programming. The hand-off between each interaction modality—in which the system provides GUI-informed, auto-generated code templates as starting points for more expressive user needs—may not only add *flexibility* [139] to the system: it may also represent an opportunity to better understand how interfaces can assist nonprogrammers in learning to code on an *ad hoc*, as-needed basis, without needing to read extensive documentation or enroll in programming courses, further enhancing *learnability* and *guidance*. Focused tests of Origraph’s interface may yield insights into which modalities are most appropriate for specific tasks and contexts, and ways to transition between them as the context shifts.

7.3 Workflow Gaps

Our early efforts to understand how graphic designers work with data [3] grounded our later efforts to build tools, as well as tools that others have built [35], [46], [47], [135]–[137], yet these tools have yet to demonstrate usability or usefulness in graphic design workflows. Collectively, our efforts have only scratched the surface of what the broader visualization community needs to learn from graphic designers, as well as other data and visualization practitioners with diverse skill sets that are less common in the research community.

In focusing on the fact that data abstractions are *designed*, rather than natural [10], [23], it may be worth considering whether new, data-specific design activities [144] can be adapted from established theory about how designers think and work [25], [145], [146]. For example, designers can identify a visual hierarchy [21] in an existing visualization, in terms of its meaning, message, perceptual rules, and / or visual hierarchy principles. With a hierarchy identified, deliberately exchanging primary, secondary, and tertiary foci can be an effective way to consider a more diverse set of visual designs. This pattern could be adapted to a visual encoding hierarchy, deliberately exchanging channels with different effectiveness ranks [22] as an example to provoke reflection about the most important attributes in the data. With tools such as Origraph, this pattern could even be adapted at the data abstraction level, deliberately exchanging what is considered a node, what is considered an edge, and what is considered an attribute, to provide a fresh perspective on the data [27].

There are more gaps in understanding real-world workflows and perspectives that are relevant to data and visualization. In mathematics and theoretical computer science, concepts such as supernodes and hyperedges are commonplace abstractions; yet there is little to no support for them in existing file formats and software. This calls into question whether and how they are—or could be—used in practice. Interviews and observations of mathematicians and data scientists may yield additional operations for tools like Origraph; insights into appropriate ways to represent and transform them; and increase their *integration* by translating or approximating them for file formats and software libraries where they are nontransferable [4].

With respect to data workflows in general, data provenance can increase workflow

reproducibility [147]. Here, too, is a gap that tools like Origraph can fill, by bridging with literate programming systems [148] such as Jupyter [149] or Observable [150]. Each operation in Origraph corresponds to a small number of library commands that can be automatically generated—not only could these be used to generate a more scalable offline script; they could also be used to document the data wrangling process.

In conclusion, this dissertation provides qualitative insight from interviews and observations of graphic designers that inform systems that address practical challenges in iterating on visual designs, and in iterating between graph data abstractions. We anticipate that it will inform continuing studies of creative software design, interaction modalities, and the workflows of diverse practitioners.

REFERENCES

- [1] J. Heer and A. Perer, "Orion: A System for Modeling, Transformation and Visualization of Multidimensional Heterogeneous Networks," *Inf. Vis.*, vol. 13, no. 2, pp. 111–133, Apr. 2014.
- [2] Z. Liu, S. B. Navathe, and J. T. Stasko, "Ploceus: Modeling, Visualizing, and Analyzing Tabular Data as Networks," *Inf. Vis.*, vol. 13, no. 1, pp. 59–89, Jan. 2014.
- [3] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer, "Reflections on How Designers Design with Data," in *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*, ser. AVI '14. ACM, 2014, pp. 17–24.
- [4] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer, "Iterating between Tools to Create and Edit Visualizations," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 481–490, Jan. 2017.
- [5] A. Bigelow and M. Monroe, "Jacob's Ladder: The User Implications of Leveraging Graph Pivots," *IEEE Trans. Vis. Comput. Graph.*, Jun. 2019.
- [6] A. Bigelow, C. Nobre, M. D. Meyer, and A. Lex, "Origraph: Interactive Network Wrangling," *Forthcoming*, 2019.
- [7] A. Greenwald, C. Robinson, S. Schube, and D. Savitzky. (2011, Jun.) The HBO Recycling Program. [Online]. Available: <http://grantland.com/features/the-hbo-recycling-program/>
- [8] R. Mercer. (2011) Daily Social Interactions. [Online]. Available: <http://portfolio.rachelmercer.org/projects/2634975>
- [9] I. Meirelles. (2010) ACM SIGGRAPH 2010 Conference. [Online]. Available: <http://isabelmeirelles.com/acm-siggraph-2010/>
- [10] T. Munzner, "A Nested Model for Visualization Design and Validation," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 6, pp. 921–928, Nov. 2009.
- [11] Ontotext. (2000) GraphDB. [Online]. Available: <http://graphdb.ontotext.com/>
- [12] Neo4j, Inc. (2010) Neo4j. [Online]. Available: <https://neo4j.com/>
- [13] Callidus Software Inc. (2010) OrientDB. [Online]. Available: <https://orientdb.com/>
- [14] A. Srinivasan, H. Park, A. Endert, and R. C. Basole, "Graphiti: Interactive Specification of Attribute-Based Edges for Network Modeling and Visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 226–235, Jan. 2018.
- [15] J. Heer, F. Ham, S. Carpendale, C. Weaver, and P. Isenberg, "Creation and Collaboration: Engaging New Audiences for Information Visualization," *Inf. Vis.*, pp. 92–133, 2008.

- [16] S. Myers, "A Quantitative Content Analysis of Errors and Inaccuracies in Missouri Newspaper Information Graphics," MA Thesis, University of Missouri-Columbia, May 2009.
- [17] S. R. Klemmer, M. W. Newman, R. Farrell, M. Bilezikjian, and J. A. Landay, "The Designers' Outpost: A Tangible Interface for Collaborative Web Site," in *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '01. ACM, 2001, pp. 1–10.
- [18] M. W. Newman and J. A. Landay, "Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice," in *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, ser. DIS '00. ACM, 2000, pp. 263–274.
- [19] L. Grammel, M. Tory, and M.-A. Storey, "How Information Visualization Novices Construct Visualizations," *IEEE Trans. Vis. Comput. Graph.*, vol. 16, no. 6, pp. 943–952, 2010 Nov-Dec.
- [20] J. Walny, S. Carpendale, N. H. Riche, G. Venolia, and P. Fawcett, "Visual Thinking in Action: Visualizations as Used on Whiteboards," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2508–2517, Dec. 2011.
- [21] R. Arnheim, "Chapter 4: Two and Two Together," in *Visual Thinking*. University of California Press, 2004, pp. 54–79.
- [22] T. Munzner, "Marks and Channels," in *Visualization Analysis and Design*. Boca Raton, FL, USA: CRC Press, Dec. 2014.
- [23] M. Da Gandra and M. Van Neck, *InformForm: Information Design: In Theory, an Informed Practice*. Mwmcreative Limited, Jul. 2012.
- [24] M. Sedlmair, M. Meyer, and T. Munzner, "Design Study Methodology: Reflections from the Trenches and the Stacks," *IEEE Trans. Vis. Comput. Graph.*, vol. 18, no. 12, pp. 2431–2440, Dec. 2012.
- [25] N. Cross, *Designerly Ways of Knowing*. London: Springer-Verlag, 2006.
- [26] M. Meyer, B. Wong, M. Styczynski, T. Munzner, and H. Pfister, "Pathline: A Tool For Comparative Functional Genomics," *Comput. Graph. Forum*, vol. 29, no. 3, pp. 1043–1052, Jun. 2010.
- [27] C. Nielsen, S. Jackman, I. Birol, and S. Jones, "ABySS-Explorer: Visualizing Genome Sequence Assemblies," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 6, pp. 881–888, Nov. 2009.
- [28] B. Lee, S. Srivastava, R. Kumar, R. Brafman, and S. R. Klemmer, "Designing with Interactive Example Galleries," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. ACM, 2010, pp. 2257–2266.
- [29] Y.-K. Lim, E. Stolterman, and J. Tenenberg, "The Anatomy of Prototypes: Prototypes as Filters, Prototypes as Manifestations of Design Ideas," *ACM Trans. Comput.-Hum. Interact.*, vol. 15, no. 2, pp. 1–27, Jul. 2008.

- [30] S. P. Dow, A. Glassco, J. Kass, M. Schwarz, D. L. Schwartz, and S. R. Klemmer, "Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-Efficacy," *ACM Trans Comput-Hum Interact*, vol. 17, no. 4, pp. 18:1–18:24, Dec. 2010.
- [31] S. P. Dow, J. Fortuna, D. Schwartz, B. Altringer, D. L. Schwartz, and S. R. Klemmer, "Prototyping Dynamics: Sharing Multiple Designs Improves Exploration, Group Rapport, and Results," in *Design Thinking Research: Measuring Performance in Context*, ser. Understanding Innovation, H. Plattner, C. Meinel, and L. Leifer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 47–70.
- [32] J. Gothelf, "Chapter 2: Principles," in *Lean UX: Applying Lean Principles to Improve User Experience*. "O'Reilly Media, Inc.", Feb. 2013, pp. 5–13.
- [33] J. Walny, J. Haber, M. Dork, J. Sillito, and S. Carpendale, "Follow That Sketch: Lifecycles of Diagrams and Sketches in Software Development," in *6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Sep. 2011, pp. 1–8.
- [34] J. Walny, S. Huron, and S. Carpendale, "An Exploratory Study of Data Sketching for Visual Representation," *Comput Graph Forum*, vol. 34, no. 3, pp. 231–240, Jun. 2015.
- [35] S. Huron, Y. Jansen, and S. Carpendale, "Constructing Visual Representations: Investigating the Use of Tangible Tokens," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2102–2111, Dec. 2014.
- [36] C. Stolte and P. Hanrahan, "Polaris: A System for Query, Analysis and Visualization of Multi-Dimensional Relational Databases," in *Proceedings of the IEEE Symposium on Information Vizualization 2000*, ser. INFOVIS '00. IEEE Computer Society, 2000, pp. 5–.
- [37] Plotly. (2013) Plotly. [Online]. Available: <https://plot.ly/>
- [38] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.
- [39] C. Reas and B. Fry, "Processing.Org: A Networked Context for Learning Computer Programming," in *ACM SIGGRAPH 2005 Web Program*, ser. SIGGRAPH '05. ACM, 2005.
- [40] W. Schroeder and B. Lorensen, *Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1996.
- [41] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein, "Interactive Graphic Design Using Automatic Presentation Knowledge," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '94. ACM, 1994, pp. 112–117.
- [42] R. Krishnamurthy and M. M. Zloof, "RBE: Rendering By Example," in *Proceedings of the Eleventh International Conference on Data Engineering*, ser. ICDE '95. IEEE Computer Society, 1995, pp. 288–297.

- [43] A. Satyanarayan and J. Heer, "Lyra: An Interactive Visualization Design Environment," in *Proceedings of the 16th Eurographics Conference on Visualization*, ser. EuroVis '14. Eurographics Association, 2014, pp. 351–360.
- [44] D. Ren, T. Höllerer, and X. Yuan, "iVisDesigner: Expressive Interactive Design of Information Visualizations," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2092–2101, Dec. 2014.
- [45] B. Lee, G. Smith, N. H. Riche, A. Karlson, and S. Carpendale, "SketchInsight: Natural Data Exploration on Interactive Whiteboards Leveraging Pen and Touch Interaction," in *2015 IEEE Pacific Visualization Symposium (PacificVis)*, Apr. 2015, pp. 199–206.
- [46] G. G. Méndez, M. A. Nacenta, and S. Vandenheste, "iVoLVER: Interactive Visual Language for Visualization Extraction and Reconstruction," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. ACM, 2016, pp. 4073–4085.
- [47] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko, "Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM Press, 2018, pp. 1–13.
- [48] M. Mauri, T. Elli, G. Caviglia, G. Ubaldi, and M. Azzi, "RAWGraphs: A Visualisation Platform to Create Open Outputs," in *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*, ser. CHItaly '17. ACM, 2017, pp. 28:1–28:5.
- [49] J. Harper and M. Agrawala, "Deconstructing and Restyling D3 Visualizations," in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '14. ACM, 2014, pp. 253–262.
- [50] W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, ser. ICSE '82. IEEE Computer Society Press, 1982, pp. 58–67.
- [51] H. Fa-Zhi, W. Shao-Mei, and S. Guo-Zheng, "From WYSIWYG to WYSIWIS: Research on CSCW Based CAD," in *Fifth Asia-Pacific Conference on ... and Fourth Optoelectronics and Communications Conference on Communications*, vol. 2, Oct. 1999, pp. 1095–1096 vol.2.
- [52] J. D. Denning and F. Pellacini, "MeshGit: Differing and Merging Meshes for Polygonal Modeling," *ACM Trans. Graph.*, vol. 32, no. 4, p. 1, Jul. 2013.
- [53] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai, "Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration," *ACM Trans Comput-Hum Interact*, vol. 13, no. 4, pp. 531–582, Dec. 2006.
- [54] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono, "Research Directions in Data Wrangling: Visualizations and Transformations for Usable and Credible Data," *Inf. Vis.*, vol. 10, no. 4, pp. 271–288, Oct. 2011.

- [55] D. Huynh and S. Mazzocchi. (2011) Google Refine. [Online]. Available: <https://code.google.com/archive/p/google-refine/>
- [56] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: Interactive Visual Specification of Data Transformation Scripts," *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, pp. 3363–3372, 2011.
- [57] M. Bilgic, L. Licamele, L. Getoor, and B. Shneiderman, "D-Dupe: An Interactive Tool for Entity Resolution in Social Networks," in *2006 IEEE Symposium On Visual Analytics Science And Technology*, Oct. 2006, pp. 43–50.
- [58] G. G. Robertson, M. P. Czerwinski, and J. E. Churchill, "Visualization of Mappings Between Schemas," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '05. ACM, 2005, pp. 431–439.
- [59] M. A. Smith, B. Shneiderman, N. Milic-Frayling, E. Mendes Rodrigues, V. Barash, C. Dunne, T. Capone, A. Perer, and E. Gleave, "Analyzing (Social Media) Networks with NodeXL," in *Proceedings of the Fourth International Conference on Communities and Technologies*. ACM, Jun. 2009, pp. 255–264.
- [60] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks," *Genome Res.*, vol. 13, no. 11, pp. 2498–2504, Jan. 2003.
- [61] M. Bastian, S. Heymann, and M. Jacomy, "Gephi : An Open Source Software for Explorating and Manipulating Networks," *Proc. Third Int. ICWSM Conf.*, p. 2, 2009.
- [62] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz: Open Source Graph Drawing Tools," in *Graph Drawing*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sep. 2001, pp. 483–484.
- [63] C. Chialin, B. Moon, A. Anurag, C. Shock, A. Sussman, and J. Saltz, "Titan: A High-Performance Remote-Sensing Database," in *Proceedings 13th International Conference on Data Engineering*, Apr. 1997, pp. 375–384.
- [64] H. Kang, C. Plaisant, B. Lee, and B. B. Bederson, "Exploring Content-Actor Paired Network Data Using Iterative Query Refinement with NetLens," in *Proceedings of the 6th ACM/IEEE-CS Joint Conference on Digital Libraries*, ser. JCDL '06. ACM, 2006, pp. 372–372.
- [65] M. Wattenberg, "Visual Exploration of Multivariate Graphs," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. ACM, 2006, pp. 811–819.
- [66] J. Stasko, C. Gorg, Z. Liu, and K. Singhal, "Jigsaw: Supporting Investigative Analysis through Interactive Visualization," in *2007 IEEE Symposium on Visual Analytics Science and Technology*. IEEE, Oct. 2007, pp. 131–138.
- [67] F. van Ham and A. Perer, ""Search, Show Context, Expand on Demand": Supporting Large Graph Exploration with Degree-of-Interest," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 6, pp. 953–960, Nov. 2009.

- [68] C. Dunne, N. Henry Riche, B. Lee, R. Metoyer, and G. Robertson, "GraphTrail: Analyzing Large Multivariate, Heterogeneous Networks While Supporting Exploration History," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12. ACM, 2012, pp. 1663–1672.
- [69] M. Dork, N. H. Riche, G. Ramos, and S. Dumais, "PivotPaths: Strolling through Faceted Information Spaces," *IEEE Trans. Vis. Comput. Graph.*, vol. 18, no. 12, pp. 2709–2718, Dec. 2012.
- [70] S. Ghani, N. Elmquist, and D. S. Ebert, "MultiNode-Explorer: A Visual Analytics Framework for Generating Web-based Multimodal Graph Visualizations," in *EuroVA 2012: International Workshop on Visual Analytics*, K. Matkovic and G. Santucci, Eds. The Eurographics Association, 2012.
- [71] B. Renoust, G. Melançon, and T. Munzner, "Detangler: Visual Analytics for Multiplex Networks," *Comput. Graph. Forum*, vol. 34, no. 3, pp. 321–330, 2015.
- [72] B. Lee, C. Plaisant, C. S. Parr, J.-D. Fekete, and N. Henry, "Task Taxonomy for Graph Visualization," in *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization*, ser. BELIV '06. ACM, 2006, pp. 1–5.
- [73] TouchGraph. (2018) Graph Visualization and Social Network Analysis Software — Navigator - TouchGraph.Com. [Online]. Available: <https://www.touchgraph.com/navigator>
- [74] Centrifuge Systems. (2018) Centrifuge. [Online]. Available: <https://centrifugesystems.com/>
- [75] C. Weaver, "Multidimensional Data Dissection Using Attribute Relationship Graphs," in *2010 IEEE Symposium on Visual Analytics Science and Technology*. IEEE, Oct. 2010, pp. 75–82.
- [76] F. van Ham, H.-J. Schulz, and J. M. Dimicco, "Honeycomb: Visual Analysis of Large Scale Social Networks," in *Human-Computer Interaction – INTERACT 2009*, ser. Lecture Notes in Computer Science, T. Gross, J. Gulliksen, P. Kotzé, L. Oestreicher, P. Palanque, R. O. Prates, and M. Winckler, Eds. Springer Berlin Heidelberg, 2009, pp. 429–442.
- [77] D. Auber, D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Mélançon, B. Pinaud, B. Renoust, and J. Vallet, "TULIP 5," in *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj and J. Rokne, Eds. Springer New York, Aug. 2017, pp. 1–28.
- [78] S. Bateman, R. L. Mandryk, C. Gutwin, A. Genest, D. McDine, and C. Brooks, "Useful Junk?: The Effects of Visual Embellishment on Comprehension and Memorability of Charts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. ACM, 2010, pp. 2573–2582.
- [79] M. A. Borkin, A. A. Vo, Z. Bylinskii, P. Isola, S. Sunkavalli, A. Oliva, and H. Pfister, "What Makes a Visualization Memorable?" *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 12, pp. 2306–2315, Dec. 2013.

- [80] J. Hullman, E. Adar, and P. Shah, "Benefitting InfoVis with Visual Difficulties," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2213–2222, Dec. 2011.
- [81] E. Segel and J. Heer, "Narrative Visualization: Telling Stories with Data," *IEEE Trans. Vis. Comput. Graph.*, vol. 16, no. 6, pp. 1139–1148, Nov. 2010.
- [82] D. McCandless, D. Busby, and A. Cho. (2009, Aug.) Timelines: Time Travel in Popular Film and TV. [Online]. Available: <https://informationisbeautiful.net/visualizations/timelines-time-travel-in-popular-film-and-tv/>
- [83] C. Plaisant, "The Challenge of Information Visualization Evaluation," in *Proceedings of the Working Conference on Advanced Visual Interfaces*, ser. AVI '04. ACM, 2004, pp. 109–116.
- [84] Fitbit, Inc. (2009) Fitbit Tracker. [Online]. Available: <https://www.fitbit.com>
- [85] A. Volda, E. Harmon, and B. Al-Ani, "Homebrew Databases: Complexities of Everyday Information Management in Nonprofit Organizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. ACM, 2011, pp. 915–924.
- [86] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show Me: Automatic Presentation for Visual Analysis," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1137–1144, Nov. 2007.
- [87] C. Viau. Scripting Inkscape with D3.js. [Online]. Available: https://christopheviau.com/d3_tutorial/d3_inkscape/
- [88] B. Victor. (2011, Mar.) Dynamic Pictures. [Online]. Available: <http://worrydream.com/DynamicPicturesMotivation/>
- [89] R. Teal, "Developing a (Non-Linear) Practice of Design Thinking," *Int. J. Art Des. Educ.*, vol. 29, no. 3, pp. 294–302, 2010.
- [90] M. Meyer, M. Sedlmair, P. S. Quinan, and T. Munzner, "The Nested Blocks and Guidelines Model," *Inf. Vis.*, vol. 14, no. 3, pp. 234–249, Jul. 2015.
- [91] R. Costa and I. Sobek, Durward K., "Iteration in Engineering Design: Inherent and Unavoidable or Product of Choices Made?" *15th Int. Conf. Des. Theory Methodol.*, vol. 3b, no. 37017b, pp. 669–674, 2003.
- [92] D. Lloyd and J. Dykes, "Human-Centered Approaches in Geovisualization Design: Investigating Multiple Methods Through a Long-Term Case Study," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2498–2507, Dec. 2011.
- [93] S. Santagata, M. L. Mendillo, Y.-c. Tang, A. Subramanian, C. C. Perley, S. P. Roche, B. Wong, R. Narayan, H. Kwon, M. Koeva, A. Amon, T. R. Golub, J. A. Porco, L. Whitesell, and S. Lindquist, "Tight Coordination of Protein Translation and HSF1 Activation Supports the Anabolic Malignant State," *Science*, vol. 341, no. 6143, p. 1238303, Jul. 2013, (Figure 2).

- [94] C. Scheidegger, H. Vo, D. Koop, J. Freire, and C. Silva, "Querying and Creating Visualizations by Analogy," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1560–1567, Nov. 2007.
- [95] S. P. Callahan, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Towards Provenance-Enabling ParaView," in *Provenance and Annotation of Data and Processes*, ser. Lecture Notes in Computer Science, J. Freire, D. Koop, and L. Moreau, Eds. Springer Berlin Heidelberg, 2008, pp. 120–127.
- [96] A. Baumann, "The Design and Implementation of Weave: A Session State Driven, Web-Based Visualization Framework," PhD Thesis, University of Massachusetts Lowell, 2011.
- [97] C. Weaver, "Building Highly-Coordinated Visualizations in Improvise," in *IEEE Symposium on Information Visualization*, Oct. 2004, pp. 159–166.
- [98] M. Savva, N. Kong, A. Chhajta, L. Fei-Fei, M. Agrawala, and J. Heer, "ReVision: Automated Classification, Analysis and Redesign of Chart Images," in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11. ACM, 2011, pp. 393–402.
- [99] E. Adar, "GUESS: A Language and Interface for Graph Exploration," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Apr. 2006, pp. 791–800.
- [100] A. Singhal, "Introducing the Knowledge Graph: Things, Not Strings," *Off. Google Blog*, May 2012.
- [101] D. A. Ferrucci, "IBM's Watson/DeepQA," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. ACM, 2011.
- [102] N. Cao, Y.-R. Lin, L. Li, and H. Tong, "G-Miner: Interactive Visual Group Mining on Multivariate Graphs," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. ACM, 2015, pp. 279–288.
- [103] D. Auber, "Tulip — A Huge Graph Visualization Framework," in *Graph Drawing Software*, ser. Mathematics and Visualization, M. Jünger and P. Mutzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 105–126.
- [104] E. M. Bonsignore, C. Dunne, D. Rotman, M. Smith, T. Capone, D. L. Hansen, and B. Shneiderman, "First Steps to Netviz Nirvana: Evaluating Social Network Analysis with NodeXL," in *2009 International Conference on Computational Science and Engineering*. IEEE, 2009, pp. 332–339.
- [105] A. Bezerianos, F. Chevalier, P. Dragicevic, N. Elmqvist, and J. D. Fekete, "Graphdice: A System for Exploring Multivariate Social Networks," in *Proceedings of the 12th Eurographics / IEEE - VGTC Conference on Visualization*, ser. EuroVis'10. The Eurographics Association & John Wiley & Sons, Ltd., 2010, pp. 863–872.
- [106] J. Heer and A. Perer, "Orion: A System for Modeling, Transformation and Visualization of Multidimensional Heterogeneous Networks," in *IEEE Visual Analytics Science \& Technology (VAST)*, 2011, p. 10.

- [107] S. Greenberg and B. Buxton, "Usability Evaluation Considered Harmful (Some of the Time)," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '08. ACM, 2008, pp. 111–120.
- [108] S. van den Elzen and J. J. van Wijk, "Multivariate Network Exploration and Presentation: From Detail to Overview via Selections and Aggregations," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2310–2319, Dec. 2014.
- [109] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "UpSet: Visualization of Intersecting Sets," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 1983–1992, Dec. 2014.
- [110] S. Gratzl, A. Lex, N. Gehlenborg, H. Pfister, and M. Streit, "LineUp: Visual Analysis of Multi-Attribute Rankings," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 12, pp. 2277–2286, Dec. 2013.
- [111] R. Verborgh and M. D. Wilde, *Using OpenRefine*. Birmingham, UK: Packt Publishing Ltd, Sep. 2013.
- [112] Trifacta. (2012) Trifacta Wrangler. [Online]. Available: <https://www.trifacta.com/>
- [113] M. Behrisch, D. Streeb, F. Stoffel, D. Seebacher, B. Matejek, S. H. Weber, S. Mittelstaedt, H. Pfister, and D. Keim, "Commercial Visual Analytics Systems—Advances in the Big Data Analytics Field," *IEEE Trans. Vis. Comput. Graph.*, pp. 1–24, Jul. 2018.
- [114] N. A. Christakis and J. H. Fowler, "The Spread of Obesity in a Large Social Network over 32 Years," *N. Engl. J. Med.*, vol. 357, no. 4, pp. 370–379, Jul. 2007.
- [115] C. A. Bail, "Combining Natural Language Processing and Network Analysis to Examine How Advocacy Organizations Stimulate Conversation on Social Media," *Proc. Natl. Acad. Sci. USA*, vol. 113, no. 42, pp. 11 823–11 828, Oct. 2016.
- [116] T. J. Hagey, J. C. Uyeda, K. E. Crandell, J. A. Cheney, K. Autumn, and L. J. Harmon, "Tempo and Mode of Performance Evolution across Multiple Independent Origins of Adhesive Toe Pads in Lizards," *Evolution*, vol. 71, no. 10, pp. 2344–2358, Oct. 2017.
- [117] C. Partl, A. Lex, M. Streit, D. Kalkofen, K. Kashofer, and D. Schmalstieg, "enRoute: Dynamic Path Extraction from Biological Pathway Maps for in-Depth Experimental Data Analysis," in *2012 IEEE Symposium on Biological Data Visualization (BioVis)*, Oct. 2012, pp. 107–114.
- [118] C. Partl, A. Lex, M. Streit, D. Kalkofen, K. Kashofer, and D. Schmalstieg, "enRoute: Dynamic Path Extraction from Biological Pathway Maps for Exploring Heterogeneous Experimental Datasets," *BMC Bioinformatics*, vol. 14, no. 19, p. S3, Nov. 2013.
- [119] A. Lex, C. Partl, D. Kalkofen, M. Streit, S. Gratzl, A. M. Wassermann, D. Schmalstieg, and H. Pfister, "Entourage: Visualizing Relationships between Biological Pathways Using Contextual Subsets," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 12, pp. 2536–2545, Dec. 2013.
- [120] C. Partl, S. Gratzl, M. Streit, A. M. Wassermann, H. Pfister, D. Schmalstieg, and A. Lex, "Pathfinder: Visual Analysis of Paths in Graphs," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 71–80, Jun. 2016.

- [121] E. Kerzner, A. Lex, C. Sigulinsky, T. Urness, B. Jones, R. Marc, and M. Meyer, "Graffinity: Visualizing Connectivity in Large Graphs," *Comput Graph Forum*, vol. 36, no. 3, pp. 251–260, Jun. 2017.
- [122] C. Nobre, N. Gehlenborg, H. Coon, and A. Lex, "Lineage: Visualizing Multivariate Clinical Data in Genealogy Graphs," *IEEE Trans. Vis. Comput. Graph.*, pp. 1–1, 2018.
- [123] C. Nobre, M. Streit, and A. Lex, "Juniper: A Tree+Table Approach to Multivariate Graph Visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 25, no. 1, pp. 544–554, Jan. 2019.
- [124] H. Wickham, "The Split-Apply-Combine Strategy for Data Analysis," *J. Stat. Softw.*, vol. 40, no. 1, pp. 1–29, Apr. 2011.
- [125] K. Furmanova, S. Gratzl, H. Stitz, T. Zichner, M. Jaresova, A. Lex, and M. Streit, "Taggle: Combining Overview and Details in Tabular Data Visualizations," *ArXiv171205944 Cs*, Dec. 2017.
- [126] P. Hu and W. C. Lau, "A Survey and Taxonomy of Graph Sampling," *ArXiv13085865 Cs Math Stat*, Aug. 2013.
- [127] The Movie Database Community. (2018) The Movie Database (TMDb). [Online]. Available: <https://www.themoviedb.org>
- [128] (2018) The Bechdel Test Movie List. [Online]. Available: <https://bechdeltest.com>
- [129] C. Maza, "Republican Senators Who Tried to Kill Yemen War Resolution Were Paid by Saudi Lobbyists," *Newsweek*, Nov. 2018.
- [130] B. Freeman, "Meet the Senators Who Took Saudi Money — The American Conservative," *Am. Conserv.*, Dec. 2018.
- [131] ProPublica, "The ProPublica Congress API," *ProPublica Data Store*, 2018.
- [132] Twitter, Inc. (2018) Twitter Developer Platform. [Online]. Available: <https://developer.twitter.com/>
- [133] D. Fisher and M. Meyer, *Making Data Visual*. Sebastopol, CA, USA: O'Reilly Media, Jan. 2018.
- [134] M. Meyer and J. Dykes, "Reflection on Reflection in Applied Visualization Research," *IEEE Comput. Graph. Appl.*, vol. 38, no. 6, pp. 9–16, 2018.
- [135] D. Ren, M. Brehmer, T. Höllerer, and E. K. Choe, "ChartAccent: Annotation for Data-Driven Storytelling," in *2017 IEEE Pacific Visualization Symposium (PacificVis)*, Apr. 2017, pp. 230–239.
- [136] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister, "Data-Driven Guides: Supporting Expressive Design for Information Graphics," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 491–500, Jan. 2017.
- [137] H. Xia, N. Henry Riche, F. Chevalier, B. De Araujo, and D. Wigdor, "DataInk: Direct and Creative Data-Oriented Drawing," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. ACM, 2018, pp. 223:1–223:13.

- [138] M. Resnick, B. Myers, K. Nakakoji, B. Shneiderman, R. Pausch, T. Selker, and M. Eisenberg, "Design Principles for Tools to Support Creative Thinking," *Carnegie Mellon Univ. Res. Showc.*, p. 16, Oct. 2005.
- [139] D. Ren, B. Lee, M. Brehmer, and N. H. Riche, "Reflecting on the Evaluation of Visualization Authoring Systems : Position Paper," in *2018 IEEE Evaluation and Beyond - Methodological Approaches for Visualization (BELIV)*, Oct. 2018, pp. 86–92.
- [140] I. E. Sutherland, "Sketch Pad a Man-Machine Graphical Communication System," in *Proceedings of the SHARE Design Automation Workshop*, ser. DAC '64. ACM, 1964, pp. 6.329–6.346.
- [141] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, vol. 16, no. 8, pp. 57–69, Aug. 1983.
- [142] T. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, Jun. 1996.
- [143] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct Manipulation Interfaces," *Hum-Comput Interact*, vol. 1, no. 4, pp. 311–338, Dec. 1985.
- [144] S. McKenna, D. Mazur, J. Agutter, and M. Meyer, "Design Activity Framework for Visualization Design," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2191–2200, Dec. 2014.
- [145] B. Lawson, *How Designers Think: The Design Process Demystified*, ser. How Designers Think: The Design Process Demystified. Elsevier/Architectural, 2006.
- [146] K. Krippendorff, *The Semantic Turn: A New Foundation for Design*. Boca Raton, FL, USA: CRC Press, Dec. 2005.
- [147] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Visualization Meets Data Management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006, pp. 745–747.
- [148] D. E. Knuth, "Literate Programming," *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.
- [149] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows." in *ELPUB*, 2016, pp. 87–90.
- [150] M. Bostock. (2018) Observable. [Online]. Available: <https://observablehq.com/>