Graph Compare: Simultaneous Graph Layout and Visualization for Structural Comparison

Matthew Can
UC Berkeley Computer Science Department
matthewcan@berkeley.edu

ABSTRACT
Graphs, node-link diagrams, are frequently used to visualize structured information. But while much work has gone into methods for visualizing single graphs, it is still an open question how to visualize multiple graphs to best convey their structural and semantic similarities and differences. We present a visualization technique for comparing graphs. At its core, our method is a graph layout algorithm that computes the layout for two graphs simultaneously so that they are easy to compare when placed side by side. In particular, our algorithm favors layouts where shared graph structure appears the same way in both graph layouts. We demonstrate how our method can work for graphs with and without unique vertex labels, using inexact graph matching techniques. We show visualizations produced by our algorithm and evaluate them in an informal user study, uncovering important design principles for visualizations that compare graphs.

Author Keywords
Graph comparison, graph structure, graph layout, graph visualization.

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Algorithms, Design.

INTRODUCTION
Graphs are a common way to reason about and visualize structured objects. By structure, we mean objects that can be decomposed into smaller parts where there are relationships among the parts. For example, a social network is a structured object. We can decompose it into the set of people present in the network, with relationships between people who are socially connected. In graph terminology, we represent the people as vertices and the social connections as edges.

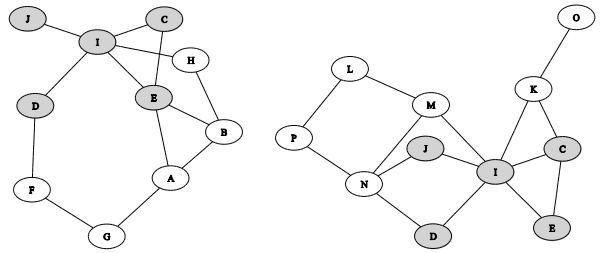

Figure 1: An example of a simultaneous layout produced by our system. We highlighted the shared nodes for the convenience of the viewer only.

For individual structured objects like social networks, there exist many visualization techniques, the most common being the graph (i.e. node-link diagram). But suppose we want to compare two or more structured objects. For example, we might want to compare the following objects:

- **Social networks**: Alice wants to compare her immediate social network with Bill's. She wants to know which groups of friends they have in common, even though the people in those groups may not correspond one-to-one.
- **Parse trees**: Our natural language parser produces a parse tree for an English sentence. We want to compare this to the true parse tree for the same sentence. This might be useful if we want to debug our parser.
- **Recipes**: A recipe is a set of ingredients and actions (e.g. bake, mix, cut). There is a directional relationship between an ingredient and an action when the ingredient is used in the action. There are also relationships between actions when the output of one action is the input to another. We might want to compare two recipes for similar dishes. We want to know not only which ingredients differ between these dishes but also how the processes of preparing the dishes differ.
- **Assembly instructions**: Assembly instructions are structured objects composed of individual steps and directional relationships that indicate the order in which the steps must be carried out. Suppose a company wants to compare two different assembly instructions for building the same desk that the company manufactures, with the goal of analyzing which instructions are most effective.

Given that we want to compare structured objects, how should we visualize the similarities and differences between such objects? As we have already said, structured objects

are commonly expressed as graphs, and in this paper, that is the approach we take. Thus, the problem we face is how to visually display graphs so that they can easily be compared. We would like our visualization to facilitate comparison between individual nodes and between larger structures such as subgraphs. An ideal visualization would help the user create a mental mapping from one graph to the other.

For simplicity, in this paper we limit ourselves to the case where we want to compare only two graphs. Our approach is to display the graphs side by side so that common graph structure has the same appearance in both graphs. Suppose we were to use a standard graph layout algorithm to lay out each graph separately and then place them next to each other. This display would have none of the properties that we desire. Instead, we design a layout algorithm that simultaneously calculates the layout for two graphs. Our algorithm attempts to maintain the same distances between corresponding nodes in the graphs. In other words, if a pair of nodes appears in both graphs, the algorithm attempts to keep the distance between the nodes the same in both graphs. This results in corresponding nodes having the same relative position to each other in both graphs, such that the shared structure between the graphs appears the same way in both.

RELATED WORK

There are two primary areas of work related to visualization for graph comparison: graph layout and dynamic graph drawing. We discuss each of these in turn.

Graph Layout

The work on graph layout examines ways to draw graphs with certain desirable properties. These properties can vary from one application to another, but a few are common. In general, a good layout will minimize the amount of physical space taken up by the graph. A good layout will eliminate node overlaps and minimize the number of edge crossings. For directed graphs, a good layout will maximize the number of directed edges that point downward. The goal of these heuristics is to make the graph visualization easier to comprehend.

A comprehensive review of the literature on graph layout algorithms that attempt to address these properties is beyond the scope of this paper. Instead, we describe some of the common classes of graph layout algorithms.

Undirected graph layout is typically done using force-directed placement (FDP) algorithms. In practice, these algorithms produce aesthetically pleasing layouts with nice symmetries. Their layouts generally also minimize edge crossings and keep all edge lengths the same. A well known FDP algorithm is that of Fruchterman and Reingold [5]. It treats the graph as a physical system, where the edges are springs and the vertices are charged particles. To compute the layout, their algorithm solves for the steady state of this system. Another flavor of FDP is based on the stress function, introduced by Kamada and Kawai [9]. These layouts minimize a stress function defined on the edges of the graph. Our algorithm for simultaneous graph layout is based on this approach. The most common methods for directed graph layout are based on the algorithm of Sugiyama et al. [13]. These algorithms use the y-axis to convey hierarchy. They divide the y-axis into layers and place nodes into layers so as to minimize edge length. Preprocessing is required to handle graphs with cycles. Dwyer and Koren introduced the DIG-COLA layout algorithm [3], which uses FDP to perform directed graph layout. Specifically, they minimize the stress function using majorization, a technique borrowed from multidimensional scaling. This technique iteratively bounds the stress function with a quadratic. To produce a nice directed layout, they add hierarchical constraints to the objective function and solve the resulting convex optimization problem, a quadratic program. DIG-COLA produces layouts that convey hierarchy and contain some of the nice qualities of FDP-based layouts.

Dynamic Graph Drawing

Dynamic graph drawing is the problem of drawing a graph that changes over time. The changes are visualized as a sequence of graphs, one for each time step that is depicted. The challenge is how to lay out the sequence of graphs so that the viewer can make the most sense of what nodes and edges were inserted and removed across time steps. In the literature, this is known as "preserving the mental map."

North's DynaDAG is a system for drawing dynamic directed acyclic graphs [12]. It uses a heuristic to perform incremental layout, whereas our algorithm minimizes the stress function. Diehl and Görg introduced a generic algorithm for dynamic graph drawing that can utilize different layout algorithms [2]. Their approach considers all graphs in the sequence, so it is an offline method rather than an online one. Similarly, we compute the layouts of two graphs simultaneously. In follow up work, Görg et al. explore the use of their generic algorithm in the context of orthogonal and hierarchical layouts [6]. Frishman and Tal present an algorithm for the dynamic drawing of clustered graphs [4]. It attempts to maintain the clustered structure of the graphs during incremental layout. In some sense, our algorithm also attempts to highlight clusters that are common between two graphs, but without knowing the clusters a priori. Archambault introduces the difference map, a technique for comparing the structure of two graphs in the context of dynamic graph drawing [1]. The difference map displays the union of two graphs using three colors: nodes only in the first graph are shown in one color, nodes only in the second graph are shown in a second color, and nodes common to both graphs are shown in the third color. We are also interested in visualizing structural similarities between two graphs, but our approach is to jointly compute a side-by-side graph layout.

```
initialize X;
for n from 0 to num_iter:
      initialize dX;
      for i from 0 to num_vertices:
            dX[i] = deriv(X, i);
      X = X - dX * eps;
```

Figure 2: The algorithm for stress-based layout minimizes the stress function by performing gradient descent.

In dynamic graph drawing, the assumption is that a node in one time step corresponds to a node in another if they have the same identity. That is, dynamic graph drawing operates on graphs where the nodes can be uniquely identified. But, we would like to compare graphs where the nodes do not have unique identifiers. For example, if we encode a recipe as a graph, the same ingredient or action may appear in the graph multiple times as different nodes. Our method addresses this situation.

METHOD

We now describe our algorithm for simultaneous graph layout and visualization. Since our layout algorithm is based on minimizing the stress function in [9], we begin by introducing stress-based layout for a single graph. Then we explain how we use this to lay out two graphs simultaneously. Furthermore, we describe how we extend our method to graphs with non-unique node labels by computing an inexact graph matching.

Stress-based Graph Layout

Formally speaking, we are given a graph G = (V,E) of vertices and edges. Computing a layout for G means computing x,y positions for each vertex in G. We assume the vertices have unique labels from a finite label set L. Let X_i be the coordinates of vertex i and X be the coordinates of all vertices. The stress function is defined as follows:

$$\text{stress}(X) = \sum_{i<j} w_{ij} (\|X_i - X_j\| - d_{ij})^2$$

Here, d_{ij} is the graph theoretic distance between vertices i and j (shortest path from vertex i to vertex j) and w_{ij} equals d_{ij}^{-2}, a normalization constant.

In stress-based graph layout, the layout is calculated by minimizing the stress function. Why should we expect this to result in a good layout? Let us look at the stress function closely. By minimizing the stress function, we try to bring the Euclidean distance between two vertices close to their graph theoretic distance. In terms of visual variables, we can see that this attempts to use length to encode graph theoretic distance. For simplicity, let us assume that the edges of our graph have no weights associated with them. Then, minimizing stress also makes the length of all edges as close to each other as possible.

We minimize the stress using gradient descent. The partial derivative of the stress with respect to the i^{th} node is:

$$\partial s / \partial X_i = \sum_{j \neq i} 2 * w_{ij} (\|X_i - X_j\| - d_{ij}) (X_i - X_j) / \|X_i - X_j\|$$

Gradient descent iteratively computes the gradient of an objective function and takes a step in the negative direction of the gradient. The algorithm is in Figure 2.

Simultaneous Graph Layout

Given two graphs G_1 and G_2, we compute a simultaneous layout by constructing a new graph G, composed of both graphs, and laying that out via stress minimization. Graph G contains all vertices and edges in G_1 and G_2. Additionally, G has zero-weight edges connecting vertices of G_1 and G_2 whose labels match. Again, we assume unique labels, so a match on the labels implies that the vertices have the same identity. This illustration shows how we compute G from two graphs:

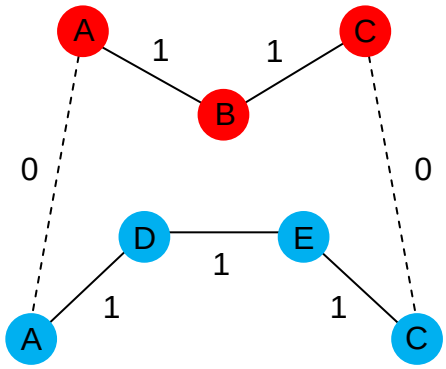

Figure 3: To lay out two graphs (in red and blue), we first create a new graph that contains them along with additional zero-weight edges between matching nodes.

By computing a layout for G, we also compute layouts for G_1 and G_2. To produce our visualizations, we simply render the two graphs using the layout information and place them side by side. This method will try to preserve the distances between corresponding nodes in the two graph layouts, while also considering a layout suitable to the structure of each graph. This is true because stress minimization will prefer layouts where corresponding nodes have the same position. The hypothesis is that it will help people notice shared graph structure if that structure appears the same way in both graphs. Before we demonstrate the results of our algorithm, we extend it to work with graphs whose vertices take on non-unique labels.

Non-unique Vertex Labels

With unique vertex labels, we can construct the joint graph described above because a vertex in one graph can only correspond to one vertex in the other. In the absence of unique vertex labels, we need some other way of computing a matching between the vertices in the two graphs. This is known as the inexact graph matching problem.

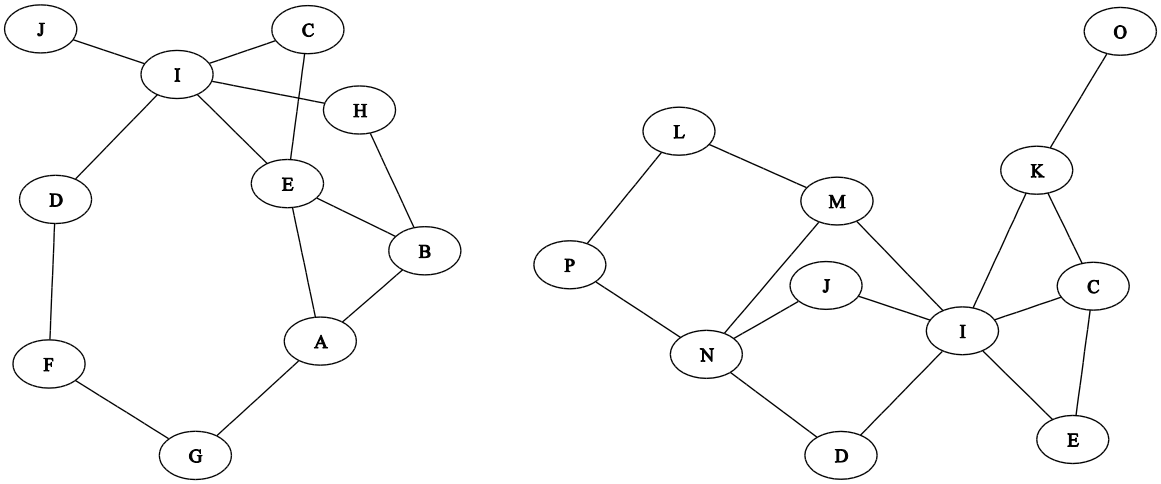

Figure 4: A side-by-side graph layout produced by our system. Note that the common structure, J-I-C-D-E, has the same appearance in both graphs because distances and relative positions between common nodes are the same.

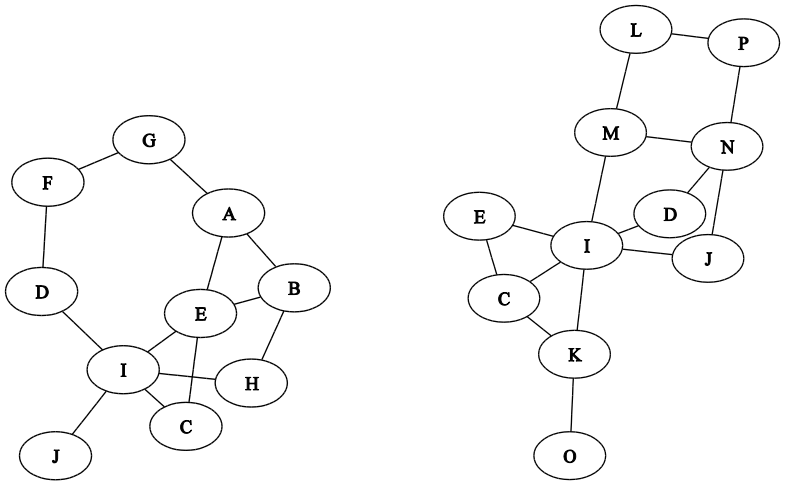

Figure 5: These are the same graphs as above, with the layouts computed by Graphviz's neato algorithm. The common structure does not have the same appearance in both graphs.

Inexact graph matching is related to the concept of graph edit distance. Let us assume the following edit operations on a graph: vertex insertion and removal, edge insertion and removal, and vertex relabeling. Each of these operations has an associated cost. The graph edit distance between two graphs is the least cost sequence of edit operations that transforms one graph into the other. By computing the graph edit distance, we also compute an inexact matching between the graphs. A vertex in the first graph matches a vertex in the second if its label is changed to match it.

The problem of computing graph edit distance is NP-complete. We frame it as a search problem and use uniform-cost search to search through the space of edit sequences for the least-cost sequence. Our method is based on the A* search algorithm presented by Neuhaus et al. [11], except that we do not use a heuristic.

IMPLEMENTATION

We implemented our algorithm in the Java programming language. This includes the stress-based layout and the graph edit distance computation. Both were our own implementation. While our algorithm computes the node positions in the graph layout, it does not render the visual display. Instead, we output the layout to the DOT language and use Graphviz [7] to render the graph. Graphviz uses our calculated positions as the final node positions.

RESULTS

We now provide the results of our simultaneous graph layout algorithm for graph comparison. First, we show example layouts produced by our system and discuss their strengths and weaknesses. We then evaluate our visualizations in an informal user study.

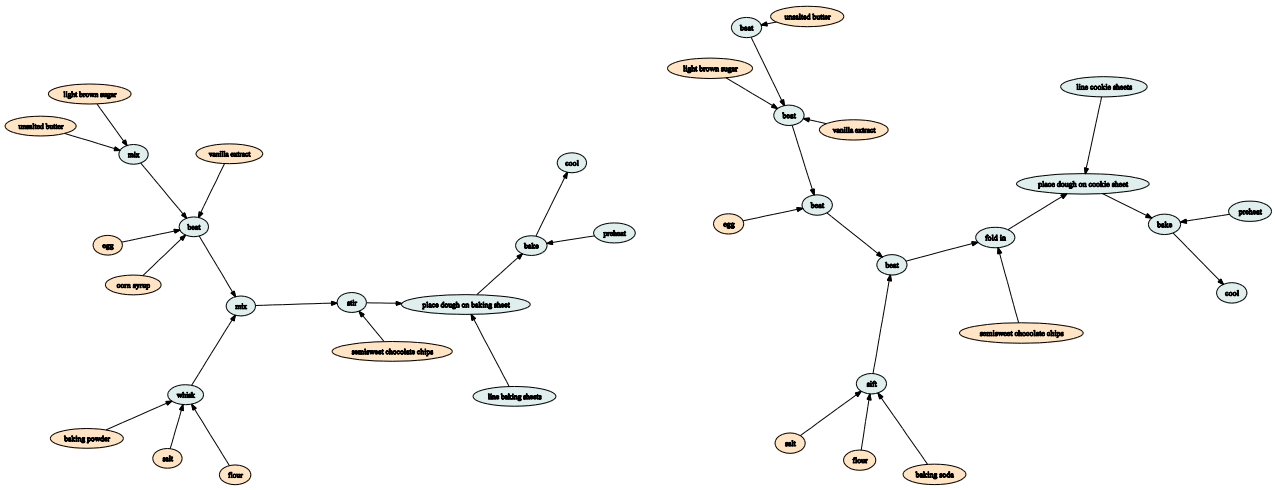

Figure 6: Comparison of two chocolate chip cookie recipes, represented as directed graphs, using our algorithm. Our layout makes it easy to notice the common, general structure of mixing wet and dry ingredients, followed by baking.

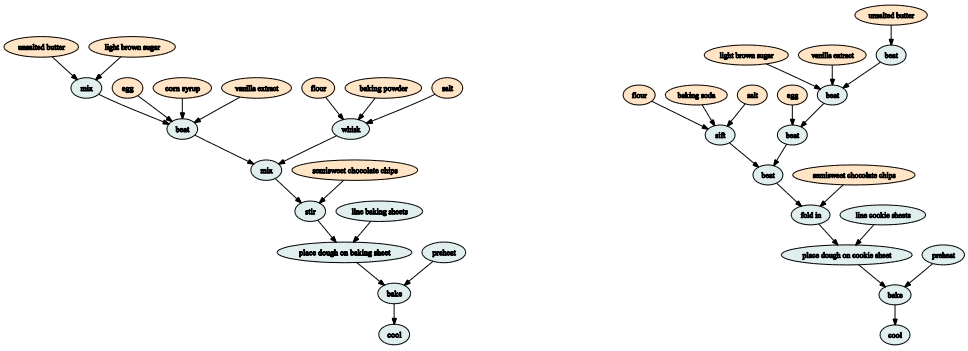

Figure 7: The same graphs as above, with the layouts computed by Graphviz's dot algorithm. This Sugiyama-style layout does a better job than our algorithm at conveying the process flow of baking chocolate chip cookies.

Examples

Undirected Graph

Figure 4 shows the output of our system for a pair of undirected graphs that share a 5-node subgraph. In our simultaneous layout, the shared subgraph has the same appearance in both graphs. That is, if you look only at the layout of that subgraph in both graphs, it is nearly identical. In this case, our algorithm works as we had hoped it would. But, what is not yet clear is whether this helps convey the shared structure to the viewer. We come back to this point in our evaluation.

We also lay out the same pair of graphs in Figure 5 using a standard undirected layout procedure in the Graphviz graph drawing tool. Unlike our output, the shared graph structure does not have the same appearance in both graphs. Nodes common to both graphs are not in the same relative positions to each other.

Directed Graph

So far in our system description, we have not made any distinction between directed and undirected graphs. Since our simultaneous layout algorithm is based on stress minimization, we designed it with undirected graphs in mind. However, nothing prevents us from using it to lay out graphs with directed edges. Aside from rendering directed edges in the display, the only change we make to the algorithm is to compute the graph theoretic distances on an undirected copy of the graph. But because our algorithm does nothing to visualize hierarchy in graphs, it is likely to yield poor layouts for directed graphs. In any case, we show the results.

The visualization in Figure 6 is a comparison of two chocolate chip cookie recipes. The recipes are encoded as directed graphs. Ingredients are pink nodes and cooking actions are blue nodes. A directed edge from an ingredient to an action indicates that the ingredient was an input to that action. A directed edge from one action to another indicates that the output of the first action was an input to the second.

One good property of our simultaneous layout is that it highlights the general structure shared by both cookie recipes; wet ingredients are combined with dry ingredients, and the result goes through a baking process. Even without

reading the labels on the nodes, a viewer can glean that this general structure exists and is common to both graphs. This is less salient in the layouts produced by Graphviz's standard layout for directed graphs. However, Graphviz's Sugiyama-style layout does a better job than ours at showing the hierarchical information in the graphs. That is, it is easier to follow the flow of ingredients and intermediate outputs through the actions. Note that in our layouts, many directed edges point upwards, making this harder.

Evaluation

To get some understanding of the efficacy of our side-by-side layout for graph comparison, we conducted an informal user study. Our goal was not only to judge how well our visualization technique performs, but also to learn some general principles for creating visualizations that make it easy for people to compare graphs.

We conducted our study with six participants, all undergraduates at UC Berkeley. Each participant was presented with the visualizations in Figure 4 and Figure 6. For Figure 4, we told them that the graphs were social networks where edges indicated friendship. For Figure 6, we explained to them how we encoded the recipes as graphs.

We asked participants to look at the side-by-side graphs and tell us what similarities and differences they noticed between the graphs. They were instructed to compare specific graph elements such as nodes and more general graph elements like subgraphs. We asked participants to talk aloud as they did this, and in particular, we asked that they attempt to describe what helps them locate and visualize the structure that they compare.

For the undirected graphs in Figure 4, participants had a difficult time making a comparison. Only a couple participants noticed the entirety of the shared structure. Others noticed that the I-C-E "triangle" was common to both graphs. One participant did not notice any shared components at all. When explaining how they made comparisons, participants described that they began by scanning the nodes in the graphs to find a single one that the two had in common. Often, node I was the first comparison made. From that first node, they branched out and looked for other neighbors that were shared. One participant described that this comparison was difficult because the shape of the two graphs did not give off any patterns, so he had to start scanning the nodes. In general, participants noticed "lonely" nodes like O and J, which extended from the graphs and stood out alone.

Participants had a much easier time comparing the cookie recipes in Figure 6. All participants noticed that the two graphs shared the same macro structure made of three clusters: dry ingredients, wet ingredients, and the baking process. They noticed this structure before reading the labels on the nodes, and read the labels afterward to verify the similarity. One person said that while the clusters were not exactly the same, the two graphs appeared to have the same pattern. Participants described that they made the mapping between clusters of nodes based on the fact that clusters looked the same way in both graphs. We probed further, asking participants to be as explicit as possible about how they made a mapping between the graphs. The general process is that participants first noticed a structure in one graph, and then checked to see if it was present in the other. The first place they checked was the same relative location on the other graph. Three participants also described that they started looking at the graphs at the top left. They also mentioned that the color coding helped them make comparisons.

DISCUSSION

Based on our informal evaluation of our simultaneous graph layout algorithm, we have come up with a few design implications for visualizations that compare graphs.

Use color to improve comparison

In the chocolate chip cookie comparison, one of the things that helped our participants make comparisons was the color coding on the nodes. There are two ways that color can facilitate comparison. In the case of the recipes, color *adds structure*. For example, consider the dry ingredients in the bottom left of the graphs in Figure 6. Without the color, all the viewer knows is that the commonality is three nodes pointing to another node. With the color, she knows that it is three ingredients being used in an action. In the latter case, she has a stronger basis for the similarity. The second way in which color can help is if it explicitly *highlights common structure*. In Figure 1, the nodes common to both graphs are colored gray. This makes it immediately apparent to the viewer which nodes the graphs have in common.

Same structure looks the same

As we had hypothesized, our evaluation revealed that making shared graph structure appear visually similar in both graphs improves the viewer's ability to see it. For our cookie recipe graphs, participants made mappings between the graphs based on visually similar groups of nodes (comparison based on appearance of graph structure), and then checked the labels on the nodes to further verify similarity.

Same structure in same relative location

We also learned from our evaluation that it is not enough for common structure to look the same in both graphs. What we also need for ease of comparison is that the common structure appear in the same relative location in both graphs. Consider Figure 4 as an example. Although the common subgraph looks the same in both graphs, it is in a different location. In the graph on the left, it is in the upper left. In the graph on the right, it is in the lower right. This is what made it difficult for participants of our study to locate

the common structure. In contrast, in the cookie recipe graphs, most of the common structures were in the same relative location in both graphs. And when they were not, participants took notice and commented that while the structures were the same, they were not placed in the same location. They remarked upon this as something that was "wrong" with the layout.

Convey hierarchical information

One of the complaints with our recipe graph visualization is that participants had a preference for reading it from top to bottom, but our directed edges had a general flow from left to right. In fact, when commenting on hierarchical relationships, participants saw the top-to-bottom relationships better. For directed graphs, it is important to have as many directed edges point downwards as possible. When edges must point in another direction, we recommend that corresponding edges between the two graphs point in the same direction.

FUTURE WORK

Our algorithm for simultaneous graph layout is just a first step toward creating visualizations that facilitate graph comparison. There is much yet to be done in this area and there are many possible extensions to our current method.

Directed graphs

Our existing algorithm does not faithfully support directed graphs in the sense that it does not compute layouts that convey the hierarchical information in directed graphs. We plan to extend our algorithm to directed graphs by modifying the stress-based objective function that our algorithm minimizes. We will incorporate the hierarchy constraints used by the DIG-COLA algorithm [3].

Evaluation

Our evaluation merely evaluated our visualizations in isolation. In the future, we would like to compare our simultaneous side-by-side layout to other visualizations for graph comparison, such as the difference map [1].

In addition, we would like to have a better understanding of what mappings people make when they compare graphs. And perhaps a way to evaluate visualizations for graph comparison is to examine how the visualizations differ based on what mappings they elicit from users.

We plan to use Mechanical Turk to gather mappings for pairs of graphs by asking workers to draw marks around what they perceive to be common structure. Kittur et al. have shown that with careful experimental design, Mechanical Turk can be viable way to perform user studies quickly and cheaply [10]. In the field of visualization, Heer and Bostock replicated known results in graphical perception with workers on Mechanical Turk, providing a basis for conducting graphical perception experiments on the system [8].

Better use of inexact graph matching

Our method used inexact graph matching to create a mapping between the nodes of two graphs when the nodes did not have unique labels. However, we might want to create such a mapping even when the nodes do have unique labels. Here is the justification:

Remember that our algorithm favors layouts where corresponding nodes in the two graphs are the same distances from other corresponding nodes. This tends to result in layouts where shared structure looks the same in both graphs. But, as there are more correspondences between nodes (i.e. as the matching becomes more inexact), we have more soft constraints on the layout our algorithm can produce. This will tend to give us layouts where the common structure not only has the same appearance, but also appears in the same relative location. We believe the use of inexact graph matching is the reason why the graphs in Figure 6 preserved relative location in many cases.

Furthermore, we would like to experiment with different cost functions in the graph edit distance computation to see how they affect the simultaneous layout. In particular, we would like to know how robust the layout is with respect to the cost function. Our function gave a cost of 1 to all insertion and deletion operations, a cost of 0 to relabeling operations where the labels were identical, and infinity to relabeling operations where the labels were different. We anticipate that different applications will require different cost functions.

We have also thought of using the results of the graph edit distance to modulate the stress function of the combined graph that we actually lay out. Currently, we place an edge of zero weight between corresponding nodes of the two input graphs. Instead, we might let that weight vary based on the strength of the correspondence, as computed by the graph edit distance.

Finally, we plan to implement some form of approximate graph edit distance rather than our exact algorithm because the computational complexity of graph edit distance prevents us from using it on large graphs.

Miscellaneous

We used gradient descent to minimize the stress function. In the future, we plan to use the majorization technique used by DIG-COLA [3]. It is a global optimization with the nice property that it guarantees a monotonic decrease of the stress function.

Currently, our method only supports comparing two graphs. We would like to explore ways of comparing several graphs, perhaps in a small multiples type of visualization. A naïve way to do this is by using our method on pairs of adjacent graphs, fixing the layout of the first graph. But, there might be a more sophisticated method that considers all of the graphs simultaneously.

Lastly, we would like to explore the use of other visual variables like color and node size in our visualizations. We have already discussed the potential for color to improve the viewer's ability to make mappings between the graphs.

CONCLUSION

We have presented a visualization technique for comparing two graphs. Our method is to place the graphs side-by-side with a layout that maintains the same visual appearance of structure that is common between the two graphs. We have described an algorithm based on stress minimization that produces layouts with this property. Examples show that our algorithm produces the kinds of layouts we desire. An informal evaluation of our visualizations reveals several design principles for graph comparison visualizations. While it is the case that common structure should have the same visual appearance, that alone is not enough. It is also important that common structure appear in the same relative location in both graphs. Most importantly, we have outlined the work yet to be done in this area of research.

ACKNOWLEDGMENTS

We thank Maneesh Agrawala and Björn Hartmann for the discussions that have led to this work and their invaluable suggestions that have helped guide it.

REFERENCES

1. Archambault, D. Structural differences between two graphs through hierarchies. In *Proc. of Graphics Interface*, 2009.
2. Diehl, S. and Görg, C. Graphs, they are changing – dynamic graph drawing for a sequence of graphs. In *Proc. of Graph Drawing*, 2002.
3. Dwyer, T. and Koren, T. DIG-COLA: Directed Graph Layout through Constrained Energy Minimization. In *Proc. of IEEE Symposium on Information Visualization*, 2005.
4. Frishman, Y. and Tal, A. Dynamic drawing of clustered graphs. In *Proc. of IEEE Symposium on Information Visualization*, 2004.
5. Fruchterman, T. and Reingold, E. Graph Drawing by Force-directed Placement. In *Software – Practice & Experience*, 1991.
6. Görg, C., Birke, P., Pohl, M., and Diehl, S. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proc. of Graph Drawing*, 2004.
7. Graphviz. http://www.graphviz.org/.
8. Heer, J. and Bostock, M. Crowdsourcing Graphical Perception: Using Mechanical Turk to Assess Visualization Design. In *Proc. CHI '10*, 2010.
9. Kamada, T. and Kawai, S. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 1989.
10. Kittur, A., Chi, E., and Suh, B. Crowdsourcing User Studies With Mechanical Turk. In *Proc. CHI '08*, 2008.
11. Neuhaus, M., Riesen, K., and Bunke, H. Fast Suboptimal Algorithms for the Computation of Graph Edit Distance. In *Structural, Syntactic, and Statistical Pattern Recognition*, Springer 2006.
12. North, S. Incremental layout in dynaDAG. In *Proc. of Graph Drawing*, 1995.
13. Sugiyama, K., Tagawa, S., and Toda, M. Methods for Visual Understanding of Hierarchical Systems. In *IEEE Transactions on Systems, Man, and Cybernetics*, 1981.