High-Level, User-Defined Constraints for Graph Layout

Jane Hoffswell
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA 98195, United States

Abstract

Graph layout has long been the focus of research across domains, leading to many customized layout techniques based on aesthetic and domain specific information. However, producing effective layouts often requires both domain and programming expertise. In response, we present a high-level constraint language for graph layout. The user specifies layout constraints over sets of nodes, which we convert to inter-node constraints for WebCoLa (a JavaScript library for constraint based graph layout). We show that users can compactly specify complex graph layouts that resemble those produced by customized layout engines and identify design considerations and areas for future work.

CCS Concepts **•Human-centered computing → Graph drawings;**

Keywords Visualization, Graph Layout, Network Layout, Constraint Layout

ACM Reference format:
Jane Hoffswell. 2016. High-Level, User-Defined Constraints for Graph Layout. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Graphs are common across a number of domains including social network analysis [11, 14, 24, 26, 27], biological systems [1, 13, 23, 25], and ecological networks [2, 5, 7, 15–17, 22, 28]. In order to highlight relevant trends in the data, it is important for the graph layout to utilize domain specific details. In response, various domain specific layout tools have been developed [1, 12, 18, 19, 25].

When a graph layout tool does not exist for the domain of interest, the user is required to either fit their data to one of the available tools or create a customized layout. Creating such a customized layout often requires both domain and programming expertise. When a collaboration is infeasible, the domain expert may be required to invest large amounts of time into learning the skills necessary to program the customized layout. Furthermore, these tools are hard to generalize outside their domain as they are carefully crafted by design to meet the needs of that particular domain, which means that it can be hard for domain experts to share their new programming expertise to others in their field. This barrier to creating highly customized, domain specific layouts means there is a gap between the analysis needs of some domains and the availability of tools to handle those needs.

In response, we present a high-level constraint language for graph layout. This approach allows the user to easily create a new layout customized for her needs, while deferring calculation of the layout to the underlying system. The user focuses on expressing the main layout properties of interest by specifying constraints over sets of node in the graph. Our system then converts these constraints to inter-node constraints for WebCola [8], a Javascript library for constraint based graph layout. The user can then tune properties of the layout by modifying preferences for WebCola or otherwise modifying the graph with D3 [6]. We show that users can compactly specify complex graph layouts that resemble those produced by customized layout engines. Finally, we discuss design considerations and areas for future work.

Conference'17, Washington, DC, USA
2016. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

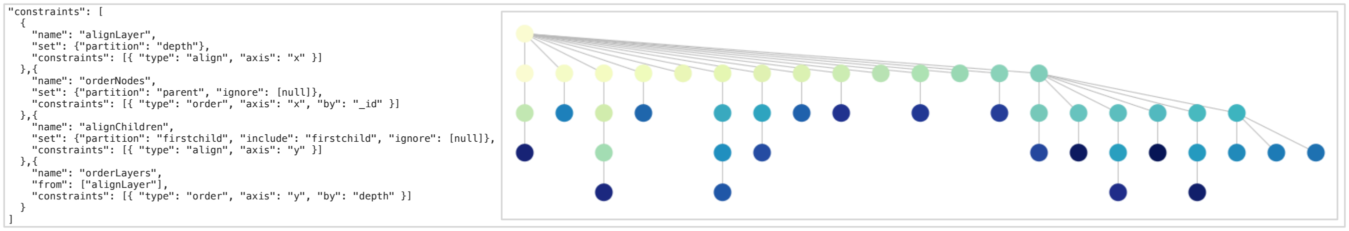
2 Related Work

There are many areas of related work surrounding the visualization and layout of graphs, and domain specific graphs in particular. Graphs are a common type of data seen across a variety of domains including social networks [11, 14, 24, 26, 27], biological systems [1, 13, 23, 25], and ecological networks [2, 5, 7, 15–17, 22, 28]. In this section, we identify some common tools for graph layout and describe some tools that have been tailored for domain specific tasks.

2.1 Graph Visualization

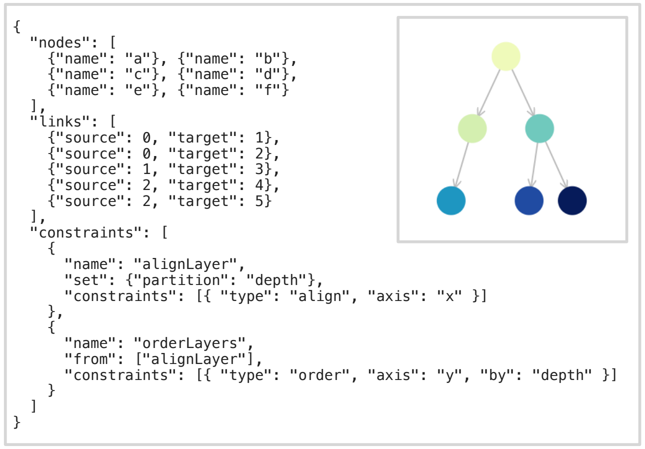
Graph layout has been a long-standing area of research and there are a number of common layout techniques for visualizing graph data including tree layouts or force-directed layouts. Graphviz [10] and Gephi [3] are two examples of visualization engines specific to graph layout. D3.js [6] is a JavaScript library that provides a number of built in layouts for graph data including force-directed and hierarchical layouts. WebCoLa [8] is a JavaScript library for constraint-based layout that can be used alongside D3 or Cytoscape [25].

In addition to these standard layout techniques, there is a large field of related work surrounding graph layout techniques that emphasize particular aesthetic or structural



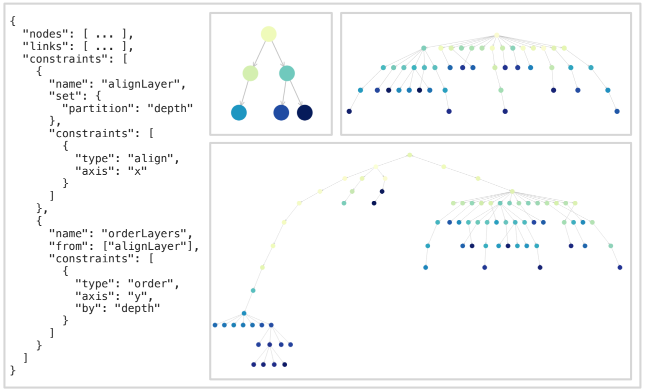
```
"constraints": [
  {
    "name": "alignLayer",
    "set": {"partition": "depth"},
    "constraints": [{ "type": "align", "axis": "x" }]
  },{
    "name": "orderNodes",
    "set": {"partition": "parent", "ignore": [null]},
    "constraints": [{ "type": "order", "axis": "x", "by": "_id" }]
  },{
    "name": "alignChildren",
    "set": {"partition": "firstchild", "include": "firstchild", "ignore": [null]},
    "constraints": [{ "type": "align", "axis": "y" }]
  },{
    "name": "orderLayers",
    "from": ["alignLayer"],
    "constraints": [{ "type": "order", "axis": "y", "by": "depth" }]
  }
]
```

Figure 1. An example constraint specification showing how to specify a left-aligned tree. Alignment and order constraints arrange the layers and nodes within the layers. Nodes are colored based on their "_id".



```
{
  "nodes": [
    {"name": "a"}, {"name": "b"},
    {"name": "c"}, {"name": "d"},
    {"name": "e"}, {"name": "f"}
  ],
  "links": [
    {"source": 0, "target": 1},
    {"source": 0, "target": 2},
    {"source": 1, "target": 3},
    {"source": 2, "target": 4},
    {"source": 2, "target": 5}
  ],
  "constraints": [
    {
      "name": "alignLayer",
      "set": {"partition": "depth"},
      "constraints": [{ "type": "align", "axis": "x" }]
    },
    {
      "name": "orderLayers",
      "from": ["alignLayer"],
      "constraints": [{ "type": "order", "axis": "y", "by": "depth" }]
    }
  ]
}
```

Figure 2. A simple constraint specification for a tree layout and the resulting layout on a six node tree.



```
{
  "nodes": [ ... ],
  "links": [ ... ],
  "constraints": [
    {
      "name": "alignLayer",
      "set": {
        "partition": "depth"
      },
      "constraints": [
        {
          "type": "align",
          "axis": "x"
        }
      ]
    },
    {
      "name": "orderLayers",
      "from": ["alignLayer"],
      "constraints": [
        {
          "type": "order",
          "axis": "y",
          "by": "depth"
        }
      ]
    }
  ]
}
```

Figure 3. The constraint specification for a tree and the result on three trees with six, forty-seven, and eighty nodes.

properties in the graph. HOLA [21] presents a layout engine to produce layouts similar to those produced by hand by identifying common aesthetic properties of the graph used in manual layouts and using the results to drive the design of a new layout algorithm. Kieffer et al. [20] present a constraint-based layout for creating graphs with node and edge alignment, and demonstrate the feasibility of such a technique in an interactive system, Dunnart [9].

While these are some examples of customized layouts, the customization reflects general properties of the graph as opposed to domain specific details. Our work focuses on providing a general language for supporting customized graph layouts with minimal programming expertise.

2.2 Domain Specific Graph Visualization

To address the domain specific needs for graph analysis, a number of tools have been developed to utilize domain specific information in the layout. Cerebral [1] is a tool designed to visualize biological systems and support interactive exploration of different experimental conditions. Cytoscape [25] is a visualization system designed to explore biomolecular interaction networks and provides a framework for accepting customized plugins to extend the system. Becker and Rojas [4] provide a customized algorithm for visualizing metabolic pathways. Kearney has developed D3 [18] and Ecopath [19] plugins to aid in the visualization of food webs.

Each of these tools was designed with a particular domain in mind to address the needs of domain experts who were unable to find the necessary support within existing tools. Our work aims to reduce the barrier of creating these customized systems by providing a compact way to specify domain specific graph layouts.

3 High-Level, User-Defined Constraints

In this section we describe our constraint language for high-level, user-defined constraints. This languages includes support for specifying sets of nodes over which to run the constraints and three types of constraints over the nodes: *alignment*, *order*, and *position*. Figure 2 and Figure 3 show an example specification for a tree layout and the resulting tree.

We refer to two types of constraints throughout this paper: *high-level* constraints and *layout* constrains. *High-level* constraints identify sets over which node-specific *layout* constraints are applied. Every *high-level* constraint must include a "name" property and specify sets over which to apply the *layout* constraints. However, *layout* constraints for a *high-level* constraint are optional; in cases where *layout* constraints are not provided, the user may simply want to define sets of nodes to refer to later in her constraint specification.

3.1 Specifying Sets

Every high-level constraint must specify a group of sets over which the layout constraints are defined. There are two types of high-level constraints that can be specified: *within set* constraints and *between set* constraints. *Within set* constraints describe constraints that are applied to all of the nodes within a given set whereas *between set* constraints describe constraints that are between larger sets. We describe how to define each type of constraint and show examples of how these constraints are applied.

3.1.1 Within Set Constraints

The user specifies within set constraints with the `"set"` property as shown for `alignLayer` in Figure 2. There are two ways to specify sets for within set constraints: the user can either partition all the nodes into disjoint sets or the user can specify expressions to create specific sets of nodes.

The simplest set definition is to `"partition"` nodes into sets based on a property of the node. When generating the sets, the system extracts the property of the node to use as the key for the set. Using `"partition"`, nodes can only occur in one set specification for the given constraint.

There are two additional properties that users can apply to set partitioning: `"include"` and `"ignore"`. `"include"` allows the user to specify a property of the accepted node to *include* in the set. This property must also be a node, for example the `"parent"` or `"firstchild"` of the node (e.g. the `alignChildren` constraint of Figure 1). `"ignore"` represents a list of keys to *ignore* when creating the sets. This property can be useful for preventing nodes with specific properties from being included in the layout (e.g. `orderNodes` in Figure 1 and `layer` in Figure 11).

Alternatively, the user can specify a concrete list of sets to compute. In this representation, the user defines an expression that returns a boolean value of whether or not the node should be included in the set. The user may also specify an optional `"name"` property that is used by the debugging tools. Using this syntax, users may specify sets such that a node can appear in multiple sets. However, we currently throw an error when such a specification occurs. Future work is required to determine whether or not this behavior would be useful for complex layouts. The `"expr"` definition currently supports value comparisons (e.g. ==, <=, >), and conjunctions (&&) and disjunctions (||) of value comparisons and boolean values. The user may use `datum.property` to refer to properties of the node.

Within set constraints define constraints that should be applied to all nodes in a set separate from nodes outside the set. In Figure 2, the user partitions nodes into sets based on their depth and creates an *alignment* constraint to align the nodes in each set along the x-axis.

3.1.2 Between Set Constraints

The user specifies between set constraints using the `"from"` property as shown by the `orderLayers` constraint in Figure 2. The user specifies between set constraints as a list of previously named constraints. Between set constraints define constraints that should be applied between sets but not between nodes in a given set. In Figure 2, the user selects all the sets defined by `alignLayer` and creates an *order* constraint to sort the sets by their depth along the y-axis.

3.2 Specifying Constraints

Once the user has defined sets of nodes over which to apply the constraints, she may optionally define layout constraints for that specification. Every constraint must have a `"type"` property that defines the type of constraint. We support three types of layout constraints: *alignment* (`"align"`), *order* (`"order"`), and *position* (`"position"`).

3.2.1 Alignment Constraints

Alignment constraints are the easiest constraint to specify in our constraint language and define using WebCoLa. Alignment constraints must have two properties, `"type"` and `"axis"`. The property `"axis"` can be defined as either `"x"` or `"y"`. Figure 2 shows an example of *alignment* for the high-level constraint `alignLayer`.

3.2.2 Order Constraints

Order constraints have three required properties: `"type"`, `"axis"` (either `"x"` or `"y"`), and `"by"`. The property `"by"` defines the property with which to order the nodes. If the `"by"` property does not have an obvious ordering (e.g. numeric, alphabetical), the user may optionally define an `"order"` property that is an ordered list of the expected inputs to the ordering function. The user may also optionally define a `"reverse"` property to reverse the behavior of the ordering. Figure 2 shows an example of an *order* constraint for the high-level constraint `orderLayers`, which forces the layers to be positioned along the y-axis based on their depth.

3.2.3 Position Constraints

Position constraints have three required properties: `"type"`, `"position"`, and `"of"`. The property `"position"` accepts the values `"right"`, `"left"`, `"above"`, or `"below"`. The `"of"` property can be defined as a node, for example the `"parent"` or `"firstchild"` of the current node, or as a temporary point. The temporary point definition can include any combination of the properties `"name"`, `"x"`, and `"y"`. If `"x"` or `"y"` is not defined, it is initialized to zero by default. Specifying a `"name"` property allows the new node to be reused in different parts of the specification. The temporary nodes are optionally visualized in the layout and can be used to modify the final layout by moving the nodes. An example of this behavior is shown in Figure 10b,c.

3.3 Built-in Properties

The constraint specification can use any properties of the nodes in the original graph. However, we also provide a number of built-in properties that can be defined over the nodes. These properties are automatically computed and added to the graph specification prior to computing the final WebCoLa constraints. These properties are only added to nodes if such a property does not already exist on the nodes.

_id The index of the node in the graph specification.
depth One more than the max depth of the node's parents. This property is only allowed if the graph contains only one root node and does not contain cycles.
parent The parent of the current node. This property is only allowed if the node has one parent or is defined as the parent with the smallest "_id".
firstchild The child node of the current node with the smallest "_id".
parents The list of nodes that have edges where the current node is the target.
children The list of nodes that have edges where the current node is the source.
neighbors The list of nodes that have edges connected to the current node. This property is the join of the "parents" and "children" properties.
degree The number of "neighbors".

4 Implementation

We implemented our constraint language in a system named `HvZ Constraint Layout`[1], which converts the high-level, user-defined constraints to WebCoLa [8] and renders the graph using WebCoLa and D3 [6]. First, we talk about the overall pipeline for specifying and rendering graphs. We then briefly describe three help features included in our system: *schema* description, *configuration*, and *debugging*.

The user begins by providing a specification for a graph and high-level constraints. The system then analyzes the user's high-level constraints and computes any necessary built-in properties before proceeding to the constraint generation. The system works through each high-level constraint, first computing the sets of nodes then using those sets to generate the necessary WebCoLa constraints. Once all of the constraints have been generated, the new graph specification with WebCoLa constraints is passed to WebCoLa, which computes the final layout. The user can then interact with the various options provided by the `HvZ` help features to tune the behavior and display of the layout.

4.1 Generating Constraints

The constraint generation is the main feature of this system. We will briefly describe the process for computing the WebCoLa constraints for each of our high-level constraints. The number of WebCoLa constraints generated for each high-level constraint is shown in Table 1.

[1]`HvZ Constraint Layout` is available online at:
`https://jhoffswell.github.io/human-constraint-layout/`

	Within Set	Between Set
Alignment Constraint	s	$\sum_{i=1}^{s} n_i * \sum_{j=i+1}^{s} n_j$
Order Constraint	$\sum_{i=1}^{s} \binom{n_i}{2}$	$\sum_{i=1}^{s} n_i * \sum_{j=i+1}^{s} n_j$
Position Constraint	$\sum_{i=1}^{s} n_i$	$2 * \sum_{i=1}^{s} n_i * \sum_{j=i+1}^{s} n_j$

Table 1. The number of constraints prodcued for a single high-level, user-defined constraint where s is the number of sets and n_i is the number of nodes in set i.

4.1.1 Alignment Constraints

Generating WebCoLa constraints from our alignment constraints is easy for *within set* constraints. We simply define a new WebCoLa alignment constraint to align the nodes in the set based on the set partioning. We produce one alignment constraint per set for within set constraints.

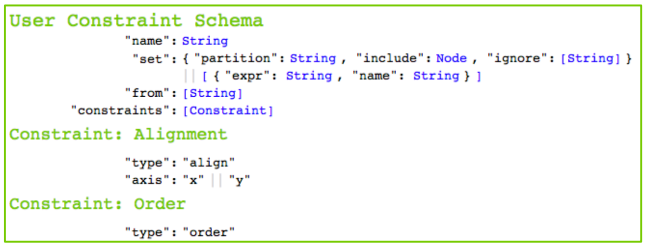
While the user can theoretically define a *between set* alignment constraint, the system will currently generate an alignment constraint for each pair of nodes where their set is not equal. If all the nodes in the graph are contained in a set, this will cause all the nodes in the graph to be aligned on the same axis. Future work is required to determine how to best specify *between set* alignment constraints; one potential expectation for such a constraint would be for the center of mass of each set of nodes to be aligned.

4.1.2 Order Constraints

To generate our high-level order constraints, we produce WebCoLa position constraints that enforce the desired ordering of the nodes. For *within set* constraints, we generate one position constraint for each pair of nodes in the set. We use a similar strategy to generate *between set* constraints in which we generate one constraint for each pair of nodes where their set is not equal. The large number of constraints currently required for *between set* order constraints is highly suboptimal; for future work, we will modify the creation of *between set* constraints to take advantage of the ordering and set nature of constraints to reduce the number of WebCoLa constraints required for this behavior.

4.1.3 Position Constraints

To generate our high-level position constraints, we start by creating a new point with the desired properties. If a new point with the desired name has already been created, we reuse that point, otherwise we insert a new "temp" node into the graph. We then generate WebCoLa position constraints with respect to this new point. As seen in Table 1, this process is highly suboptimal for *between set* constraints. Similar to the other *between set* constraints, we start by generating pairs of nodes for which their set does not match. We then produce one WebCoLa constraint for each node in each set, which leads to a large number of repeated constraints. Future work will explore better constraint generation strategies.

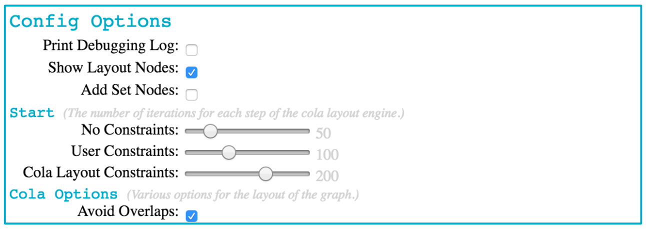


```
User Constraint Schema
                "name": String
                 "set": { "partition": String , "include": Node , "ignore": [String] }
                        || [ { "expr": String , "name": String } ]
                "from": [String]
         "constraints": [Constraint]
Constraint: Alignment

                "type": "align"
                "axis": "x" || "y"
Constraint: Order

                "type": "order"
```

Figure 4. In the schema pane, users can view the schema for our high-level constraint language.



Config Options

Print Debugging Log: ☐

Show Layout Nodes: ☑

Add Set Nodes: ☐

Start (The number of iterations for each step of the cola layout engine.)

No Constraints: 50

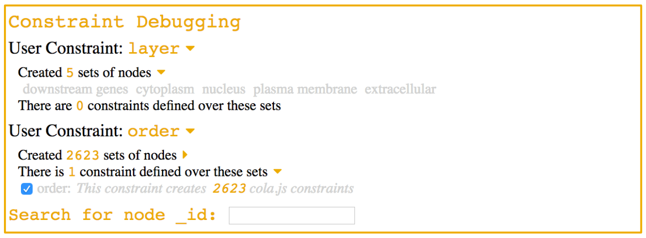
User Constraints: 100

Cola Layout Constraints: 200

Cola Options (Various options for the layout of the graph.)

Avoid Overlaps: ☑

Figure 5. In the configuration panel, users can modify properties of the development environment and graph rendering.



Constraint Debugging

User Constraint: layer

Created 5 sets of nodes

downstream genes cytoplasm nucleus plasma membrane extracellular

There are 0 constraints defined over these sets

User Constraint: order

Created 2623 sets of nodes

There is 1 constraint defined over these sets

☑ order: *This constraint creates 2623 cola.js constraints*

Search for node _id:

Figure 6. In the debugging panel, users can view the sets and WebCoLa constraints generated from their high-level constraint specification.

4.2 Help Features

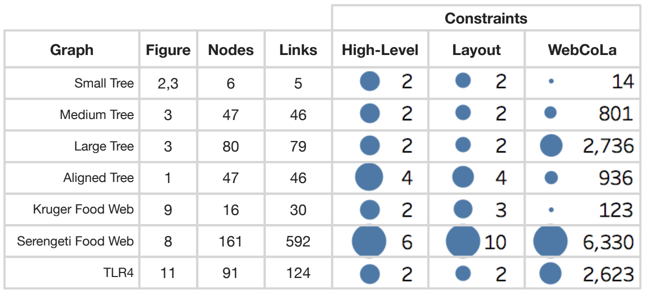
In order to help users understand and modify the graph layout, we include three panels in our development environment to provide different levels of support and customization. A *schema* panel provides a description of our constraint language. A *configuration* panel allows the user to modify aspects of the development environment, constraint generation, and graph rendering. A *debugging* panel provides users with insight into the sets defined in their constraint specification and the number of constraints produced.

4.2.1 Schema

Selecting the question icon (?) opens the schema panel, which shows the schema for our high-level constraint language. A snapshot of the schema panel is shown in Figure 4.

4.2.2 Configuration

Selecting the gear icon (⚙) opens the configuration panel. The configuration panel provides options for interacting with the development environment and rendering the graph. A snapshot of the configuration panel is shown in Figure 5.



				Constraints		
Graph	Figure	Nodes	Links	High-Level	Layout	WebCoLa
Small Tree	2,3	6	5	2	2	14
Medium Tree	3	47	46	2	2	801
Large Tree	3	80	79	2	2	2,736
Aligned Tree	1	47	46	4	4	936
Kruger Food Web	9	16	30	2	3	123
Serengeti Food Web	8	161	592	6	10	6,330
TLR4	11	91	124	2	2	2,623

Figure 7. The number of nodes, links, and constraints for each example shown in this paper.

4.2.3 Debugging

Selecting the bug icon (🐞) opens the debugging panel. The debugging panel provides insight into the underlying constraint generation system. The debugging panel shows the sets created for each high-level, user-defined constraint. Mousing over the name of the set highlights all nodes in the set in red. Highlighting text in either the original or generated graph specification highlights all nodes referenced in the highlighted text region. Users can also enter a node "_id" into the search box to highlight particular nodes of interest. The debugging panel also shows all of the layout constraints specified for each high-level constraint. Selecting the checkbox removes the WebCoLa constraints corresponding to the constraint from the layout. A snapshot of the debugging panel is shown in Figure 6.

5 Results

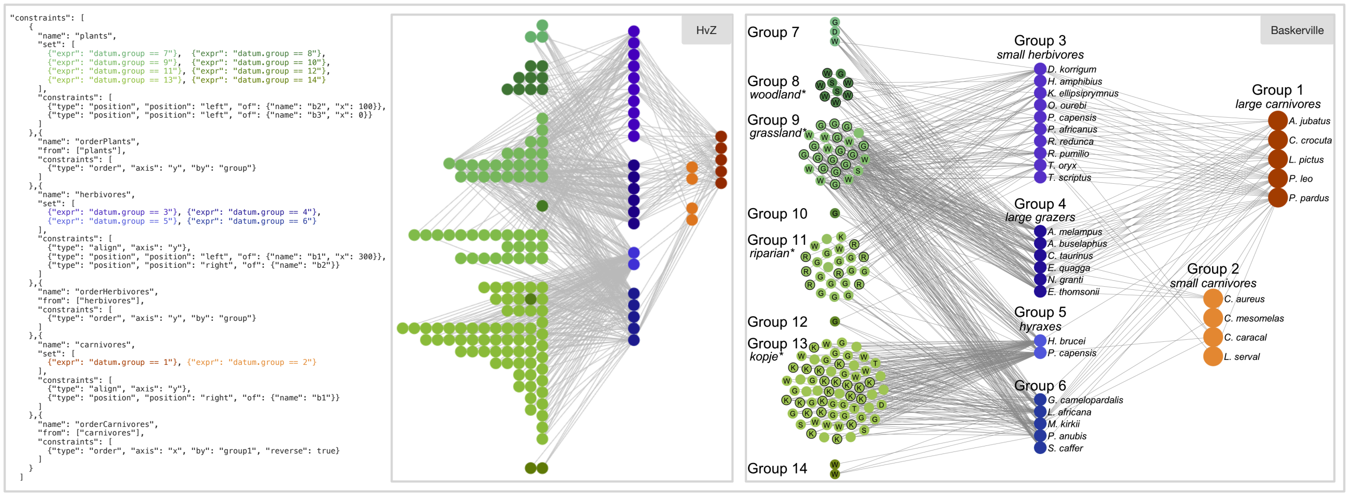
Now that we have described the design and implementation of our high-level constraint language, we will demonstrate the utility of this technique through a number of example use cases. We chose these examples to highlight both common layout techniques and highly domain specific layouts.

5.1 Tree Layouts

Trees are one example of a common layout technique that is often addressed in visualization tools [3, 6, 8, 10]. The basic tree layout from Figure 2 is shown on three trees of varying sizes in Figure 3. Our constraints specifications are highly compact, allowing users to concisely communicate complex layouts. While the layouts produced by these specifications can be highly customized to a particular graph, the layouts are also reusable across graph specifications allowing users to easily reapply the same layout to multiple graphs. Figure 7 shows information about each example shown in this paper, including the number of nodes and links in the graph, the number of high-level and layout constraints specified by the user, and the number of WebCoLa constraints generated.

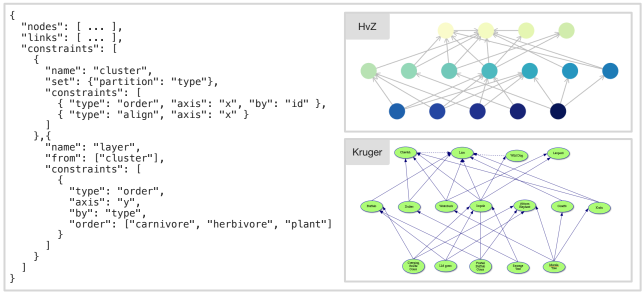
5.2 Food Webs

Food webs visualize complex producer-consumer relationships in ecological systems and are a common presentation



```
"constraints": [
    {
      "name": "plants",
      "set": [
        {"expr": "datum.group == 7"},  {"expr": "datum.group == 8"},
        {"expr": "datum.group == 9"},  {"expr": "datum.group == 10"},
        {"expr": "datum.group == 11"}, {"expr": "datum.group == 12"},
        {"expr": "datum.group == 13"}, {"expr": "datum.group == 14"}
      ],
      "constraints": [
        {"type": "position", "position": "left", "of": {"name": "b2", "x": 100}},
        {"type": "position", "position": "left", "of": {"name": "b3", "x": 0}}
      ]
    },{
      "name": "orderPlants",
      "from": ["plants"],
      "constraints": [
        {"type": "order", "axis": "y", "by": "group"}
      ]
    },{
      "name": "herbivores",
      "set": [
        {"expr": "datum.group == 3"}, {"expr": "datum.group == 4"},
        {"expr": "datum.group == 5"}, {"expr": "datum.group == 6"}
      ],
      "constraints": [
        {"type": "align", "axis": "y"},
        {"type": "position", "position": "left", "of": {"name": "b1", "x": 300}},
        {"type": "position", "position": "right", "of": {"name": "b2"}}
      ]
    },{
      "name": "orderHerbivores",
      "from": ["herbivores"],
      "constraints": [
        {"type": "order", "axis": "y", "by": "group"}
      ]
    },{
      "name": "carnivores",
      "set": [
        {"expr": "datum.group == 1"}, {"expr": "datum.group == 2"}
      ],
      "constraints": [
        {"type": "align", "axis": "y"},
        {"type": "position", "position": "right", "of": {"name": "b1"}}
      ]
    },{
      "name": "orderCarnivores",
      "from": ["carnivores"],
      "constraints": [
        {"type": "order", "axis": "x", "by": "group1", "reverse": true}
      ]
    }
  ]
```

Figure 8. The layout for the Serengeti food web using our constraint language, as compared to Baskerville et al. [2].



```
{
  "nodes": [ ... ],
  "links": [ ... ],
  "constraints": [
    {
      "name": "cluster",
      "set": {"partition": "type"},
      "constraints": [
        { "type": "order", "axis": "x", "by": "id" },
        { "type": "align", "axis": "x" }
      ]
    },{
      "name": "layer",
      "from": ["cluster"],
      "constraints": [
        {
          "type": "order",
          "axis": "y",
          "by": "type",
          "order": ["carnivore", "herbivore", "plant"]
        }
      ]
    }
  ]
}
```

Figure 9. The constraint specification and layout for a small food web using our constraint language, compared to the example provided by Kruger National Park. Note: the image for the Kruger National Park layout was taken from http://kruger-nationalpark.weebly.com/the-food-web.html

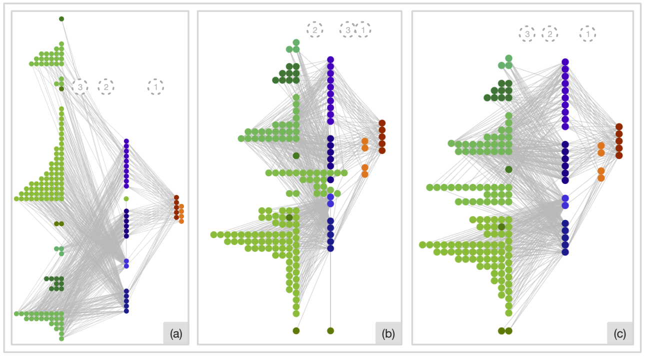


Figure 10. Three steps in the layout of the Serengeti food web from Figure 8. The gray dotted nodes are temporary nodes b1, b2, and b3. (a) The layout when the groups are sorted alphabetically rather than numerically. (b) The original layout computed by WebCoLa. (c) The final layout after some tuning using the temporary nodes.

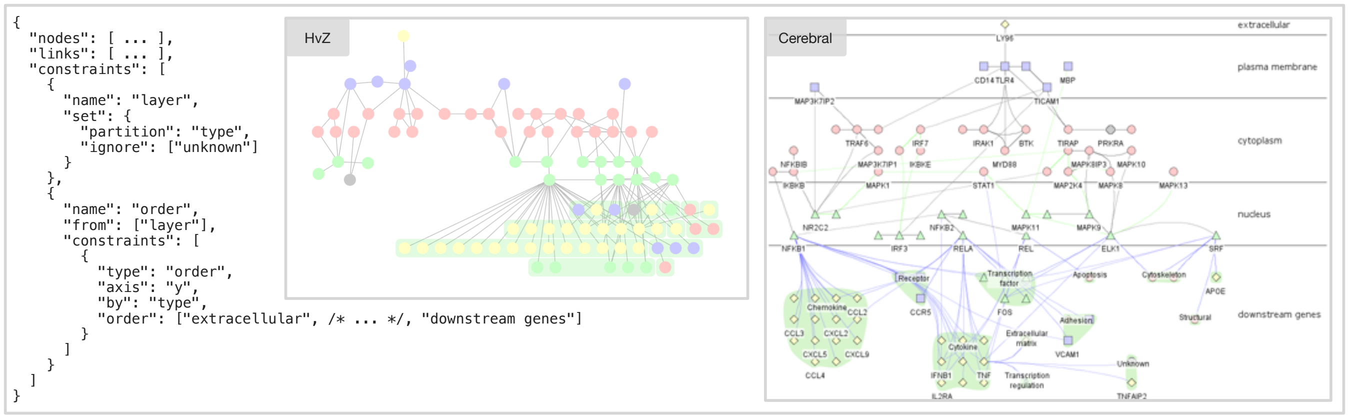
strategy for this information despite the challenge of creating an informative visualization [2, 5, 7, 17, 22, 28]. There are many examples of small food webs, which could easily be created by hand, but we show a simple constraint specification in Figure 9. However, this is just a small example of a food web, whereas the webs are often much more complex.

Baskerville et al. use a Bayesian analysis method to identify group structure for the Serengeti food web and provide a customized layout of their results showing both the hierarchy between groups and clustering [2]. To demonstrate the utility of our technique, we recreate this layout using our constraint language Figure 8. The original food web visualization was produced by the authors using D3; our recreation includes six high-level constraints with a total of 10 layout constraints, which creates 6,330 WebCoLa constraints.

For the final layout shown in Figure 8, we performed a few modifications on the generated layout (Figure 10b,c) to add additional spacing in the hierarchy using the temporary nodes and force some problematic nodes to meet our constraints. This example proved to be a useful debugging use case as well. The original graph specification represented the group as a string (`"group":"13"`), which caused the layout to sort the layers alphabetically (Figure 10a). Using the debugging tools described in Section 4.2.3, we were able to highlight the constraints of interest to verify that they were producing the expected layout and thus observed the alphabetic rather than numeric sorting of the layers. To resolve this issue, we recomputed the graph specification to use numbers for the groups rather than strings. Alternatively, we could have used the `"order"` property to specify our desired ordering. Future work may want to explore more powerful ways to specify the desired ordering of the nodes or sets.

5.3 Biological Networks

Cerebral [1] is a system for the layout and analysis of biological systems. The authors note that existing “tools did not fully meet the needs of our immunologist collaborators”



```
{
  "nodes": [ ... ],
  "links": [ ... ],
  "constraints": [
    {
      "name": "layer",
      "set": {
        "partition": "type",
        "ignore": ["unknown"]
      }
    },
    {
      "name": "order",
      "from": ["layer"],
      "constraints": [
        {
          "type": "order",
          "axis": "y",
          "by": "type",
          "order": ["extracellular", /* ... */, "downstream genes"]
        }
      ]
    }
  ]
}
```

Figure 11. The constraint specification for the TLR4 graph and layout using our constraint language and Cerebral. Note: the image for the Cerebral graph layout was taken from http://www.pathogenomics.ca/cerebral/tutorials.html

as motivation for the design of their system. In order to examine the feasibility of our constraint language for highly domain specific tasks, we recreated the TLR4 graph shown in [1] using our language. The constraint specification, layout, and Cerebral graph are shown in Figure 11.

This constraint specification demonstrates a number of the techniques supported by our constraint language. In particular, the user adds an "ignore" statement to remove the nodes with an "unknown" type from the layout (these are the gray nodes in the graph). To produce the hierarchical layout that reflects the underlying biological system, the user simply applies a custom ordering on the types of the nodes for this graph. The main difference between our layout and the one produced by Cerebral is that our layout more strongly penalizes vertical distance which causes the graph to compress into thinner layers. Future work will explore how to better tune these properties to provide unconstrained movement of nodes within the bounds provided by our constraints.

6 Discussion and Future Work

In the previous section we showed a few example use cases of our constraint language for various types of graph layout. In this section we talk about some of the lessons learned from this process, our current implementation strategies, and areas for future work on this project.

6.1 Adding Customized Annotations

Creating a domain specific layout generally requires both a domain expert and individual with programming expertise. The domain expert provides knowledge about the desired layout motivated by the underlying graph properties whereas the programmer provides the expertise to actually compute the desired layout algorithm. In some cases, the domain expert may also strive to act as the programmer, but taking on such a role may require large amounts of time and effort on behalf of the domain expert.

The goal of this work is to provide a compact language in which to specify domain specific layouts. In the previous section, we showed that we can feasibly recreate complex domain specific layouts with a compact layout specification. However, as is evident from the comparisons (Figure 8, 11), the user may want to incorporate additional information via node labeling, group labeling, or other annotations on the visualization. Our system does not currently provide support for such customizations. Future work may want to examine how to incorporate layout driven annotations.

6.2 Constraint Specification Syntax

In future iterations of this constraint language, we need to revisit some areas of the current constraint syntax to ensure that constraints are both understandable and expressive. There are two aspects of the current syntax that require additional work: specifying constraints and specifying sets.

6.2.1 Specifying Constraints

We identified two styles of high-level constraints, *within set* constraints and *between set* constraints, which represent different techniques for applying constraints over sets of nodes (see Section 3.1). In an earlier iteration of our constraint language, we defined a high-level constraint as follows:

```
{
  "name": NAME,
  "set": SET,
  "within": [CONSTRAINTS...],
  "between": [CONSTRAINTS...]
}
```

In our final constraint language, we decided to standardize the specification of layout constraints using the "constraint" syntax rather than "within" and "between" in order to support more customizable constraint specifications, instead varying the type of constraints via the "set" or "from" definition for the high-level constraint. However, in all of the examples shown in this paper, there is a tight relationship between the sets used for *within set* and *between set* constraints (Figure 2, 3, 8, 9, 11). Future work is required to understand if standardization is appropriate or if a syntax like the one shown here would be more understandable.

6.2.2 Specifying Sets

Future work is also required to explore how best to specify sets in our language. We described a number of techniques for specifying sets including *partitioning* and *manual specification* via predicates (see Section 3.1).

For the *partitioning* approach the user can include additional properties `"include"` and `"ignore"` to modify which nodes are selected for a given set and which sets are excluded from the partitioning; however, while these properties sound similar from a linguistic standpoint, it is important to note that they do not operate on the same aspects of the graph and thus do not have reflexive behaviors. Each of these properties was initially introduced to target particular use cases for graph layout, but in future iterations it may be useful to revisit the set specification to create a more straightforward and understandable syntax.

The process of *manual specification* for sets using the `"expr"` syntax also introduces a potential problem in that the sets do not enforce a disjoint partition of the nodes; in other words, it is currently possible to specify sets using this syntax that include overlapping nodes. This limitation means that it is possible for users to specify constraints that are impossible to satisfy. Additional research is required to determine if there are benefits from supporting overlapping sets or if the manual specification syntax should warn against or disallow such specifications from occurring.

6.3 Generating Constraints and Optimizations

The current process of generating WebCoLa constraints from our high-level constraints is unoptimized (see Table 1). There are a few optimizations that can be made in particular that can help to reduce the number of WebCoLa constraints.

Between set alignment constraints currently force all the nodes across all the sets to become aligned. An improvement would be to use an iterative layout that produces the inter-set layout before positioning the set as a whole. This strategy would significantly reduce the number of constraints that need to be solved by the underlying system and produce a more expected layout for the specified constraint.

Order constraints currently produce constraints between pairs of nodes, ignorant of the desired ordering. One optimization is to reflect the ordering in the constraint generation process to only produce constraints for neighboring nodes. For *within set* constraints, this strategy would only apply constraints between adjacent nodes in the final order. For *between set* constraints, we could include an optimization to use temporary nodes to enforce the order rather than combinatorially producing constraints between pairs of nodes.

Between set position constraints will produce many redundant constraints because of the current constraint generation in which the layout first partitions pairs of nodes that require a constraint, and then applies the constraint to each node in the pair. *Position* constraints currently produce the same effect for both *within set* and *between set* constraints since the position is relative to a particular node. However, because of the underlying set generation procedure, *between set* constraints are significantly less optimal than *within set* constraints (Table 1). Future work is required to examine how to optimize this process and what distinctions should be expected between these two types of the constraint.

6.4 Debugging and Defining Constraints

We provide a small amount of debugging support in the current iteration of the system, but we would like to continue exploring improved support for understanding the output of our high-level constraint syntax. When viewing the output of a graph there are a few questions that the user might have: are the constraints generated from my specification what I expected (e.g. correct)? If not, what went wrong and what should I do next? There are a number of possible techniques that can be employed to help users better understand the output of this system and define new constraints.

6.4.1 Previewing the Graph Layout

The current process of specifying graphs relies heavily on clustered abstractions of the underlying nodes. To specify constraints, it may be beneficial for users to dynamically change the coloring of nodes in the graph to show different properties and thus help select the partitions of interest. It may also be beneficial to show approximations of the graph layout by creating a preview using the node abstractions. This technique could help the user understand the high-level relationships for *between set* constraints, while deferring understanding of *within set* constraints to after the final layout has been computed.

6.4.2 Determining Constraint Correctness

While developing examples for this work, we found that there were some instances where the low-level WebCoLa constraints generated by the system would be correct, but that WebCoLa did not perform sufficient iterations to produce a graph layout that accurately reflects the constraints. In future iterations of this work, we would like to provide a way for users to determine how well the final layout reflects the underlying (and high-level) constraints by computing how many of them are satisfied by the final layout. Allowing users to pin nodes or areas of the graph that are correct, while restarting the layout on other may allow users to iteratively examine and correct the layout to produce a result that matches their expectations.

6.5 Evaluation

Finally, for the next version of the system we would like to perform evaluations with domain experts to examine the utility of this system for creating domain specific layouts. We would like to assess how easy it is to learn and use this system for producing complex graph layouts and how expressive the language is for different types of desirable layouts.

7 Conclusion

Visualizations of graph data can provide a powerful way to view and explore complex structural information in large graphs. Such visualizations are often more effective at communicating relevant information if they take advantage of domain specific properties in the layout. But creating customized layouts requires both domain and programming expertise, which may often be infeasible. In this work, we presented a high-level constraint language for specifying customized graph layouts and a system for generating WebCoLa constraints based on our constraint specifications. We showed that our constraint language can produce highly compact, domain specific layouts that closely resemble those produced for real-world use cases. We further outlines design considerations and areas for future work.

Acknowledgments

The author would like to thank Alan Borning, Jeffrey Heer, and the students of UW CSE599F1 for their feedback and support on this work.

References

[1] Aaron Barsky, Tamara Munzner, Jennifer Gardy, and Robert Kincaid. 2008. Cerebral: Visualizing multiple experimental conditions on a graph with biological context. *IEEE transactions on visualization and computer graphics* 14, 6 (2008), 1253–1260.

[2] Edward B Baskerville, Andy P Dobson, Trevor Bedford, Stefano Allesina, T Michael Anderson, and Mercedes Pascual. 2011. Spatial guilds in the Serengeti food web revealed by a Bayesian group model. *PLoS Comput Biol* 7, 12 (2011), e1002321. http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002321

[3] Mathieu Bastian, Sebastien Heymann, Mathieu Jacomy, and others. 2009. Gephi: an open source software for exploring and manipulating networks. *ICWSM* 8 (2009), 361–362.

[4] Moritz Y Becker and Isabel Rojas. 2001. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics* 17, 5 (2001), 461–467.

[5] Austin R Benson, David F Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[6] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics* 17, 12 (2011), 2301–2309.

[7] Joel E Cohen, Tomas Jonsson, and Stephen R Carpenter. 2003. Ecological community description using the food web, species abundance, and body size. *Proceedings of the National Academy of Sciences* 100, 4 (2003), 1781–1786.

[8] Tim Dwyer. 2017. cola.js: Constraint-Based Layout in the Browser. http://marvl.infotech.monash.edu/webcola/. (2017). Accessed: 2017-03-12.

[9] Tim Dwyer, Kim Marriott, and Michael Wybrow. 2008. Dunnart: A constraint-based network diagram authoring tool. In *International Symposium on Graph Drawing*. Springer, 420–431.

[10] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. 2001. Graphvizfi!?open source graph drawing tools. In *International Symposium on Graph Drawing*. Springer, 483–484.

[11] Linton C Freeman. 1978. Centrality in social networks conceptual clarification. *Social networks* 1, 3 (1978), 215–239.

[12] Jean Mark Gawron. 2016. Python for Social Science. http://gawron.sdsu.edu/python_for_ss/course_core/book_draft/index.html. (2016). Accessed: 2017-03-14.

[13] Nils Gehlenborg, Seán I O'donoghue, Nitin S Baliga, Alexander Goesmann, Matthew A Hibbs, Hiroaki Kitano, Oliver Kohlbacher, Heiko Neuweger, Reinhard Schneider, Dan Tenenbaum, and others. 2010. Visualization of omics data for systems biology. *Nature methods* 7 (2010), S56–S68.

[14] Mark S Granovetter. 1973. The strength of weak ties. *American journal of sociology* 78, 6 (1973), 1360–1380.

[15] Sarah Harper-Smith, Eric L Berlow, Roland A Knapp, Richard J Williams, and Neo D Martinez. 2006. Dynamic Food Webs. In *Elsevier Inc.*

[16] Jefferson Hinke, Isaac Kaplan, Kerim Aydin, George Watters, Robert Olson, and James FK Kitchell. 2004. Visualizing the food-web effects of fishing for tunas in the Pacific Ocean. *Ecology and Society* 9, 1 (2004).

[17] Kelly A. Kearney. 2016. Food webs as network graphs. http://kellyakearney.net/2016/01/19/food-webs-as-network-graphs-1.html. (2016). Accessed: 2017-03-14.

[18] Kelly A. Kearney. 2017. d3-foodweb. https://github.com/kakearney/d3-foodweb. (2017). Accessed: 2017-03-14.

[19] Kelly A. Kearney. 2017. foodwebgraph-pkg. https://github.com/kakearney/foodwebgraph-pkg. (2017). Accessed: 2017-03-14.

[20] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. 2013. Incremental Grid-Like Layout Using Soft and Hard Constraints.. In *Graph Drawing*. 448–459.

[21] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. 2016. Hola: Human-like orthogonal network layout. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 349–358.

[22] David Lavigne. 1996. Cod Food Web. http://www.visualcomplexity.com/vc/project.cfm?id=47. (1996). Accessed: 2017-03-14.

[23] Purvi Saraiya, Chris North, and Karen Duca. 2005. Visualizing biological pathways: requirements analysis, systems evaluation and research agenda. *Information Visualization* 4, 3 (2005), 191–205.

[24] John Scott. 1988. Social network analysis. *Sociology* 22, 1 (1988), 109–127.

[25] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. 2003. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research* 13, 11 (2003), 2498–2504.

[26] Jeffrey Travers and Stanley Milgram. 1967. The small world problem. *Phychology Today* 1 (1967), 61–67.

[27] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of fismall-worldfinetworks. *nature* 393, 6684 (1998), 440–442.

[28] Peter Yodzis. 1998. Local trophodynamics and the interaction of marine mammals and fisheries in the Benguela ecosystem. *Journal of Animal Ecology* 67, 4 (1998), 635–658.