**Programmer Guide**

What follows is a comprehensive guide that animation programmers can use in their creation of Galant algorithms. Throughout this guide there will be example algorithms alongside the API. Additionally, there are algorithms that can be downloaded from the test pages.

# **Contents**

[**Contents 1**](#_knw4g3cp7ewm)

[**Understanding the Graph Format 2**](#_rdrw0r2knped)

[**Understanding the API 3**](#_8r6umq9f92d3)

[**Stepping Through the Algorithm 4**](#_238ul0z3qjr8)

[**Print, Display, and Prompt 5**](#_laque25tkm7c)

[**List Getters 6**](#_50ybifz1ygug)

[**Source and Target 7**](#_n579ik8zy23g)

[**Adjacencies 8**](#_pi2fvgs6livo)

[**Marks 9**](#_qnbwnps5pbee)

[**Highlights 10**](#_zffyuglk8366)

[**Colors 11**](#_lo7ioi3fjo42)

[**Labels 12**](#_shjxw4k9f0x3)

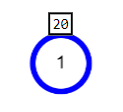
[**Weights 13**](#_2jpdah7ar93i)

[**Hide/Show Node Properties 14**](#_q7mp69wwfmub)

[**Hide/Show Edge Properties 15**](#_6eo87oi2djgt)

# 

# **Understanding the Graph Format**

The input graph file should be a .txt file. The file consists of lines containing nodes and edges.

Node lines follow the form n id x y weight key:value

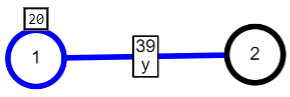
Here is an example line representing the node shown to the right:

n 1 100 150 20 highlight: color:blue

The node has an ID of “1” (shown inside the node), x coordinate 100, y coordinate 150, weight 20, it is highlighted, and its color is blue.

Note that boolean attributes like “highlighted” or “marked” are entered with nothing after the colon in the key/value pairs for the node. The first 3 words for a node have to be the id, x value, and y value. If there is a weight, it should be the 4th word, however this is optional. The order for the key value pairs after that do not matter.

Edge lines follow the form e source target weight key:value

Here is an example line representing the edge shown to the right:

e 1 2 39 color:blue label:y

This line indicates that there should be an edge that goes from node “1” (source) to node “2” (target), with a weight of 39, color blue, and a label called “y”. The first two words for an edge need to be the source and target. If there is a weight, it should be the third word.

By default, graphs are undirected, meaning that the source and target properties are interchangeable. If you want a directed graph, put the line directed anywhere in the file.

Comments can be entered in the input file, by starting a new line with ‘/’ or ‘#’.

There are a number of key-value pairs that can be used.

marked: Boolean value that indicates whether a node’s interior is marked.

highlighted: Boolean value that indicates whether a node or edge is drawn in bold.

color: String that indicates the color of a node or edge. Can be a common color word like red or blue, or a hex code like #00FF00.

label: String that appears above the node or edge, in the same box as its weight.

invisible: Boolean value that makes the node or edge invisible.

invisibleWeight: Boolean value that makes the weight of the node or edge invisible.

invisibleLabel: Boolean value that makes the label of the node or edge invisible.

# **Understanding the API**

Algorithms in Galant are written in JavaScript. They follow the exact same conventions as regular JavaScript code (if JavaScript is not enabled inside of your browser then this will not work, but neither will the website). If you do not know how to code in JavaScript here is a [link](https://www.w3schools.com/js/) to a resource. These algorithms will use our built in API that you, the user, will use to manipulate the graph. These API conventions do not follow the regular Object-Oriented approach. If you would like to get any instances of the graph the API must be used and even then it will not be the full graph “object”. For example, getNodes() is an API call, but it does not return a list of nodes and their fields. It only returns the id’s so that you can then further manipulate these nodes through the API.

# 

# **Stepping Through the Algorithm**

Galant animations run in steps. To move forward one step, press the forward button or the right arrow key. You can use the back button or the left arrow key to undo a step, letting you watch the algorithm multiple times. By default, a step will happen every time the graph view is changed. For instance, coloring a node, adding a label, or updating the display in the corner of the graph view will all cause the algorithm to pause, creating a new step.

The Step box in between the two arrows represents what step you are on. “Step 3/5” represents that you are on step 3, and 5 total steps have been done (2 steps were undone.)



These functions allow manual control of when the algorithm steps.

disableAutoStep() //Disables the automatic step that happens whenever the graph changes. After this is called, the algorithm will only step when you call a step explicitly.

enableAutoStep() //Re-enables the automatic step if it was disabled.

step() //Pauses the algorithm, creating a step.

step(function) //Runs the passed function without auto-step, pausing at the end.

This is useful for combining multiple API calls into a single step.

**Example Algorithm**//Performs a single step for each node which highlights and marks it simultaneously.

step(() => {

clearNodeMarks();

clearNodeHighlights();

});

disableAutoStep();

for (let node of getNodes()) {

mark(node);

highlight(node);

step();

}

# 

# **Print, Display, and Prompt**

These functions display information to the user, either to the console or the graph window. Displaying to the graph window will pause the program if auto-step is enabled.

print(message) //Prints to the console.

display(message) //Displays a message on the graph pane.

You may also prompt the user for input, opening a pop-up. If the user provides an invalid response, they will automatically be prompted again with an error message.

prompt(message, error?) //Prompts the user for a string with the given message.

The optional error field can be used to inform the user of an invalid prompt input.

promptFrom(message, list, error?) //Prompts for an element of the list.

The optional error field can be used for a custom error message.

promptBoolean(message) //Prompts for a boolean (true or false).

promptInteger(message) //Prompts for a number. Decimal values are rounded down.

promptNumber(message) //Prompts for a number. Decimal values are accepted.

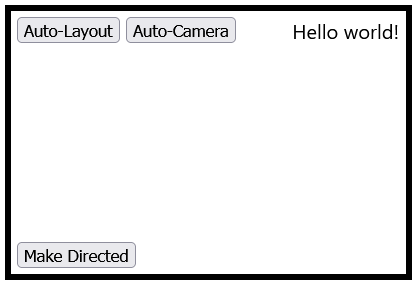
promptNode(message) //Prompts for a node of the graph. Must have at least one node.

promptEdge(message) //Prompts for an edge of the graph. Must have at least one edge.

**Example Algorithm**

//Prompts the user for a message and displays it either to the console or to the graph window.

let message = prompt(“Enter a message:”);

let displayToGraph = promptBoolean(“Display to the graph?”);

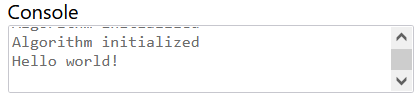
if (displayToGraph) {

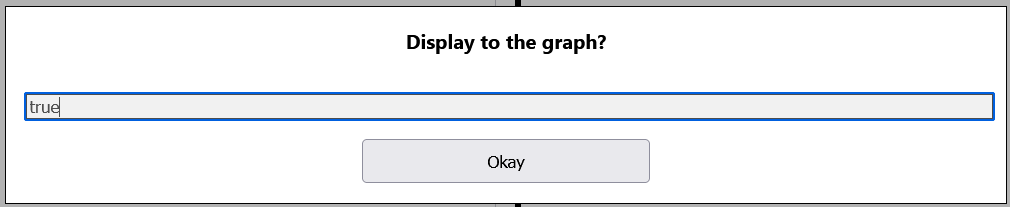
display(message);

} else {

print(message);

}





# 

# **List Getters**

These functions get lists of all nodes or edges in the graph to iterate over.

getNodes() //Returns a list of all nodes in the graph.

getNumberOfNodes() //Returns the number of nodes in the graph.

Can also use getNodes().length

getEdges() //Returns a list of all edges in the graph to iterate over.

getNumberOfEdges() //Returns the number of nodes in the graph.

Can also use getEdges().length

**Example Algorithm**

//Prints each node and then each edge to the console.

for (let node of getNodes()) {

print(node);

}

for (let edge of getEdges()) {

print(edge);

}

# 

# **Source and Target**

These functions select nodes or edges based on the source and target of edges.

Note that for undirected graphs, the source and target are interchangeable.

source(edge) //Returns the source node of the specified edge.

target(edge) //Returns the target node of the specified edge.

getEdgeBetween(source, target) //Returns an edge between two nodes.

For directed graphs, only the edge in the direction specified will be returned.

If no edge exists between the specified nodes, returns null.

getEdgesBetween(source, target) //Returns a list of edges between two nodes.

For directed graphs, only the edges in the direction specified will be returned.

This function is only needed if a graph contains multiple edges between the same nodes.

other(node, edge) //Returns the other node attached to the edge.

The node and edge parameters to this function may be provided in either order.

**Example Algorithm**

//This function returns the target of the edge A B by finding the node opposite its source.

let edge = getEdgeBetween(“A”, “B”);

let source = source(edge);

print(other(source, edge));

# 

# **Adjacencies**

These functions get lists of edges incident to a node to iterate over. In an undirected graph, all incoming and outgoing functions are equivalent to their nondirectional counterparts.

incident(node) //Returns all edges (incoming and outgoing) incident to a node.

incoming(node) //Returns all incoming edges of a node (the node is their target).

outgoing(node) //Returns all outgoing edges of a node (the node is their source).  
These functions get lists of nodes adjacent to a node to iterate over.

adjacentNodes(node) //Returns all nodes adjacent to the node.

incomingNodes(node) //Returns all nodes connected to the node by an incoming edge.

outgoingNodes(node) //Returns all nodes connected to the node by an outgoing edge.

These functions get the number of edges incident to a node.

degree(node) //Returns the number of edges incident to a node.

inDegree(node) //Returns the number of incoming edges of a node.

outDegree(node) //Returns the number of outgoing edges of a node.

**Example Algorithm**

//This function prints each edge out of the node with the max number of outgoing edges.

let max = 0;

let maxNode = null;

for (let node of getNodes()) {

if (outDegree(node) > max) {

max = outDegree(node);

maxNode = node;

}

}

for (let edge of outgoing(maxNode)) {

let node = other(maxNode, edge);

print(edge);

print(node);

}

# 

# **Marks**

mark(node) //Marks the specified node

unmark(node) //Unmarks the specified node

marked(node) //Returns true if the node is marked, false otherwise

clearNodeMarks() //Clears marks on all nodes

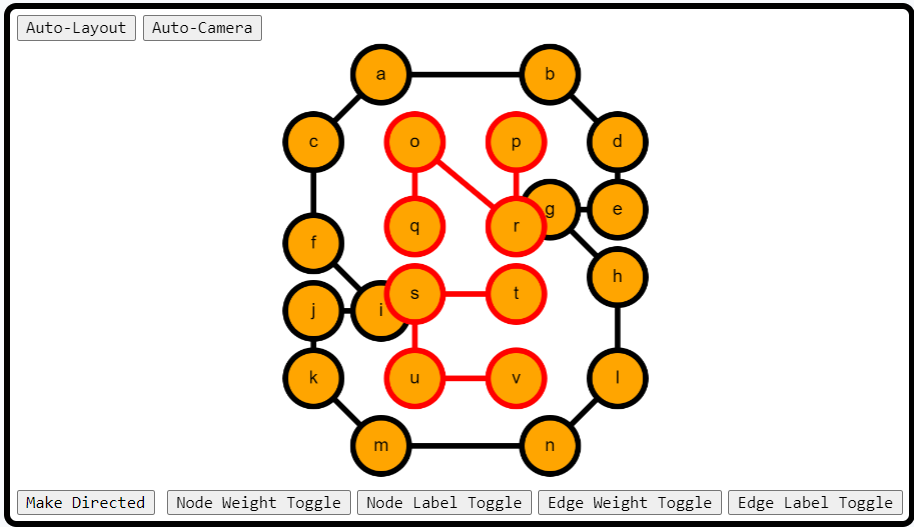
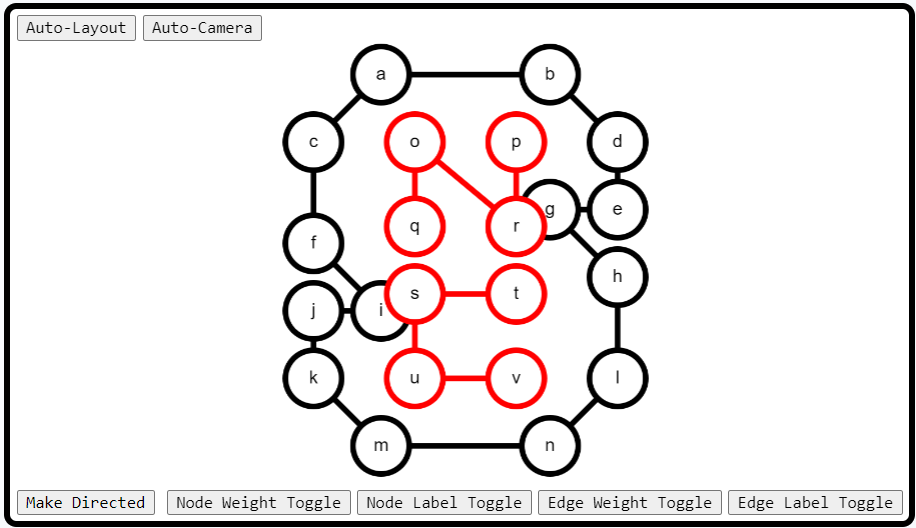
**Example Algorithm**

//Marks all nodes

for (let node of getNodes()) {

mark(node);

}



# 

# **Highlights**

highlight(id) //Highlights the specified node or edge based on the id

unhighlight(id) //Unhighlights the specified node or edge based on the id

highlighted(id) //Returns true if the node or edge is highlighted, false otherwise

clearNodeHighlights() //Clears highlights on all nodes

clearEdgeHighlights() //Clears highlights on all edges

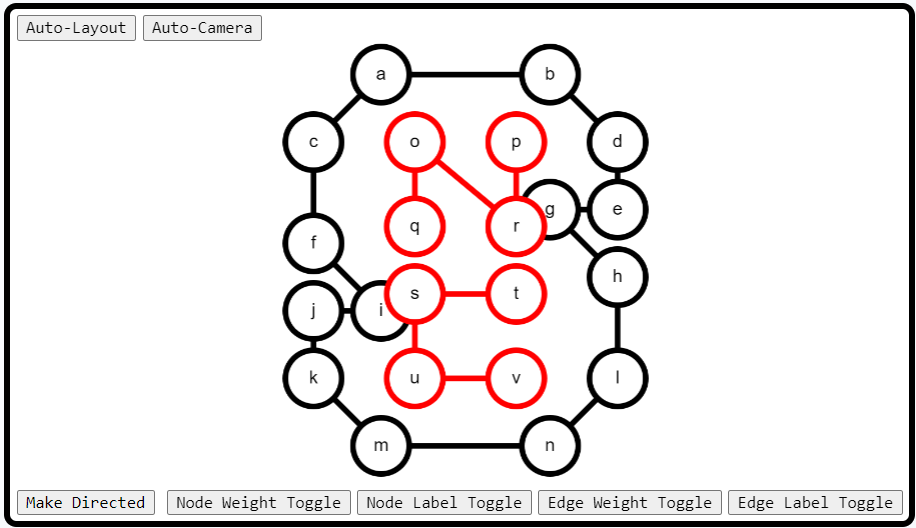
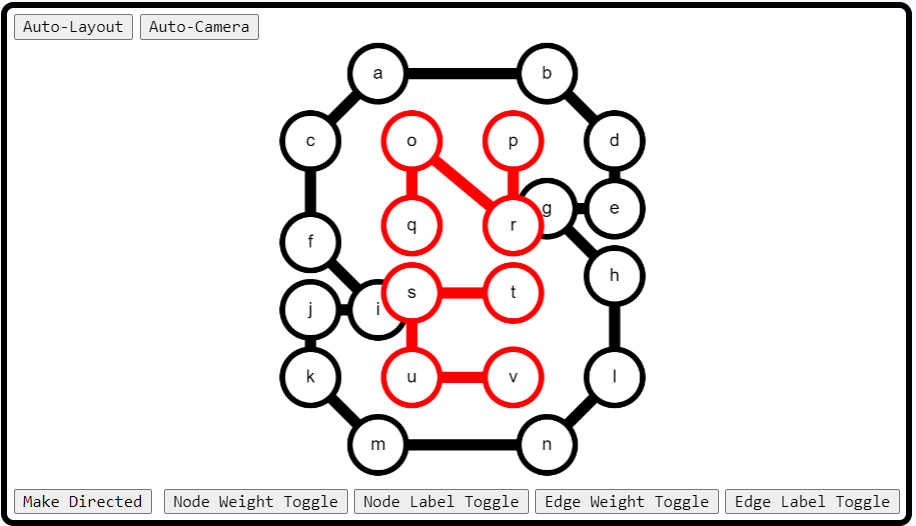
**Example Algorithm**

//Highlights all edges

for (let edge of getEdges()) {

highlight(edge);

}

# 

# **Colors**

color(id, color) //Colors the specified node or edge based on its id and given color

uncolor(id) //Uncolors the specified node or edge based on the id

getColor(id) //Returns true if the node or edge is colored, false otherwise

hasColor(id) //Returns true if the node or edge based on the id is colored

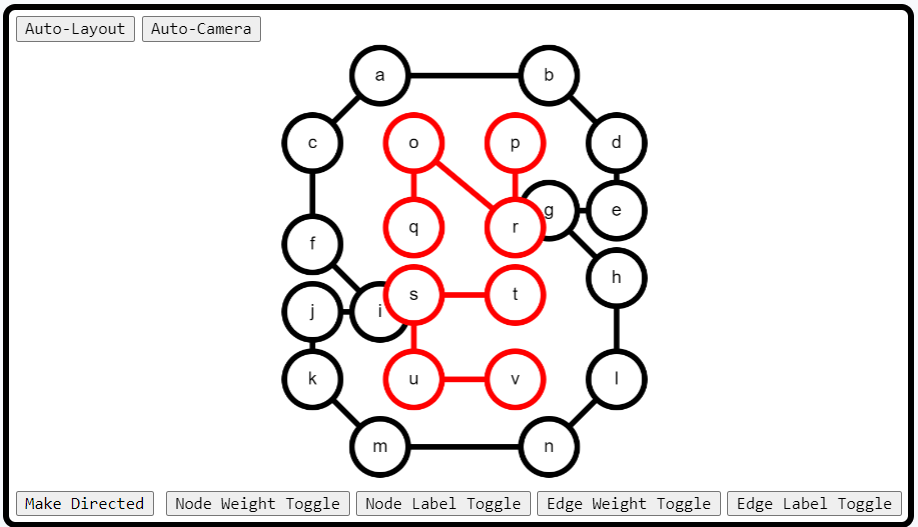
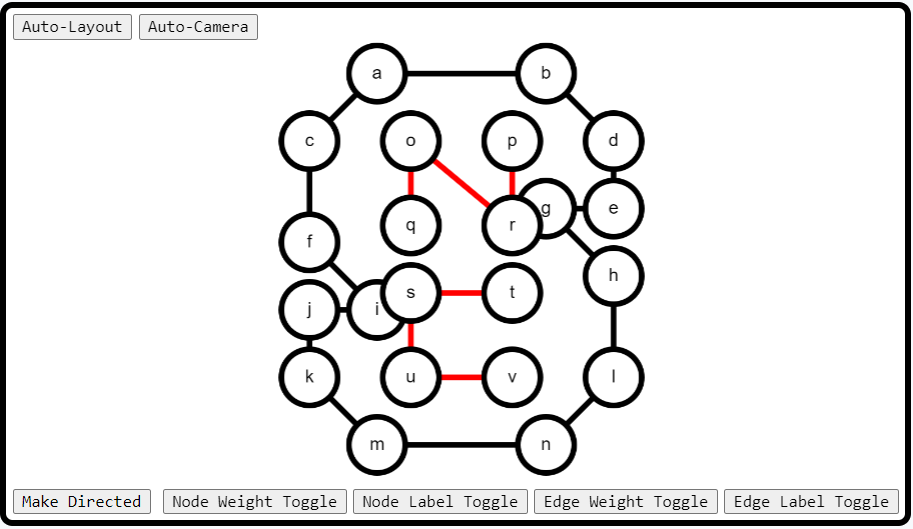
clearNodeColors() //Clears colors on all nodes

clearEdgeColors() //Clears colors on all edges

**Example Algorithm**

//Clears all node colors (the default node color is black)

clearNodeColors()

# 

# 

# **Labels**

label(id, label) //Labels the specified node or edge based on its id and given label

unlabel(id) //Unlabels the specified node or edge based on the id

getLabel(id) //Returns the labels of the specified node or edge based on its id

hasLabel(id) //Returns true if the node or edge has a label based on the id

clearNodeLabels() //Clears labels on all nodes

clearEdgeLabels() //Clears labels on all edges

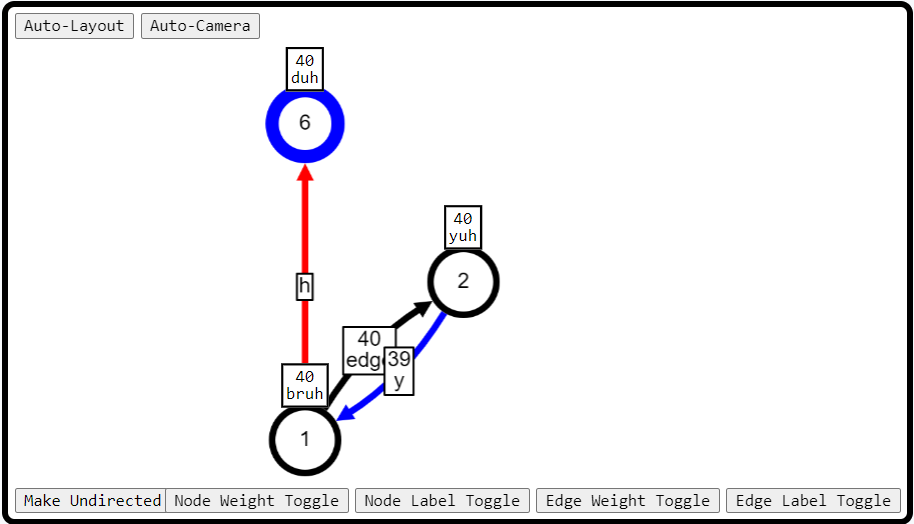
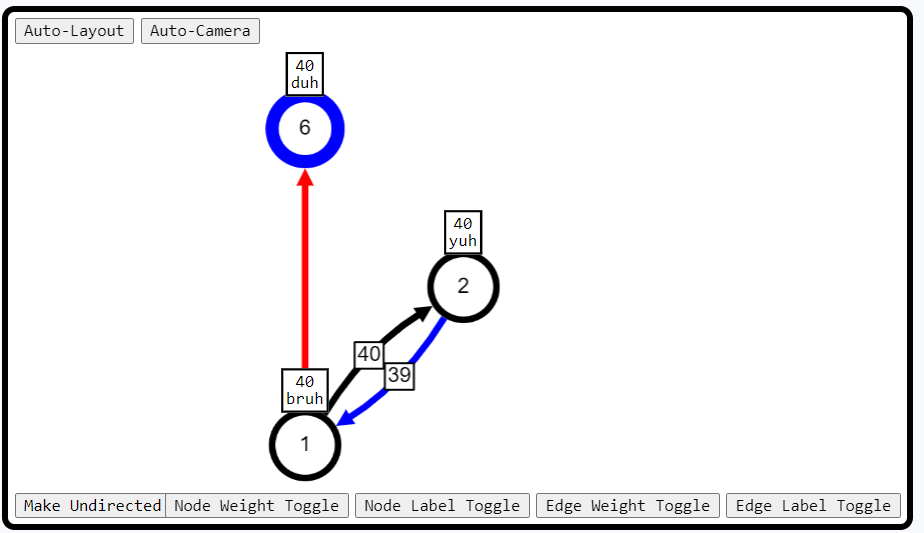
**Example Algorithm**

//Un-labels each label of all edges (40 and 39 are weights on the edges, not labels)

for (let edge of getEdges()) {

unlabel(edge);

}

# 

# **Weights**

setWeight(id, weight) //Sets the weight of the specified node or edge based on its id and given weight

clearWeight(id) //Clears the weight of the node or edge based on the id

weight(id) //Returns the weight of the specified node or edge based on the id

hasWeight(id) //Returns true if the node or edge based on the id has a weight

clearNodeWeights() //Clears weights on all nodes

clearEdgeWeights() //Clears weights on all edges

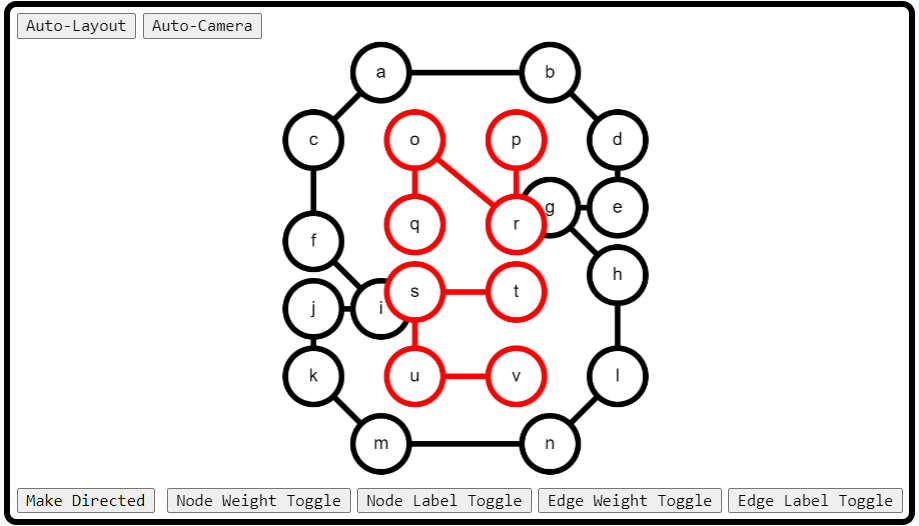
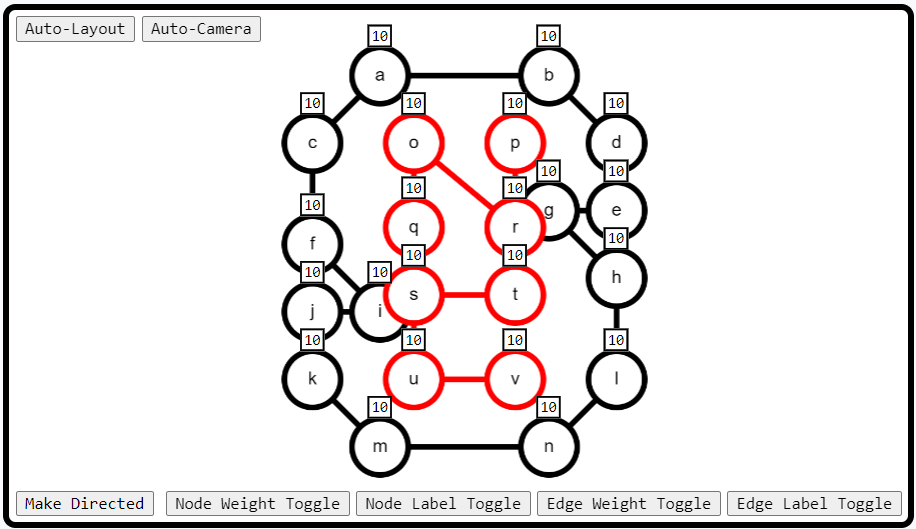
**Example Algorithm**

//Sets the weight of each node to 10

for (let node of getNodes()) {

setWeight(node, 10);

}

# 

# 

# **Hide/Show Node Properties**

hideNode(node) //Hides the node based on the specified node id (also hides connecting edges to that node)

showNode(node) //Shows the node based on the specified node id

hideNodeWeight(node) //Hides the node weight based on the specified node id

hideAllNodeWeights() //Hides all node weights

showNodeWeight(node) //Shows the node weight based on the specified node id

showAllNodeWeights() //Shows all node weights

hideNodeLabel(node) //Hides the node label based on the specified node id

hideAllNodeLabels() //Hides all node labels

showNodeLabel(node) //Shows the node label based on the specified node id

showAllNodeLabels() //Shows all node labels

**Example Algorithm**

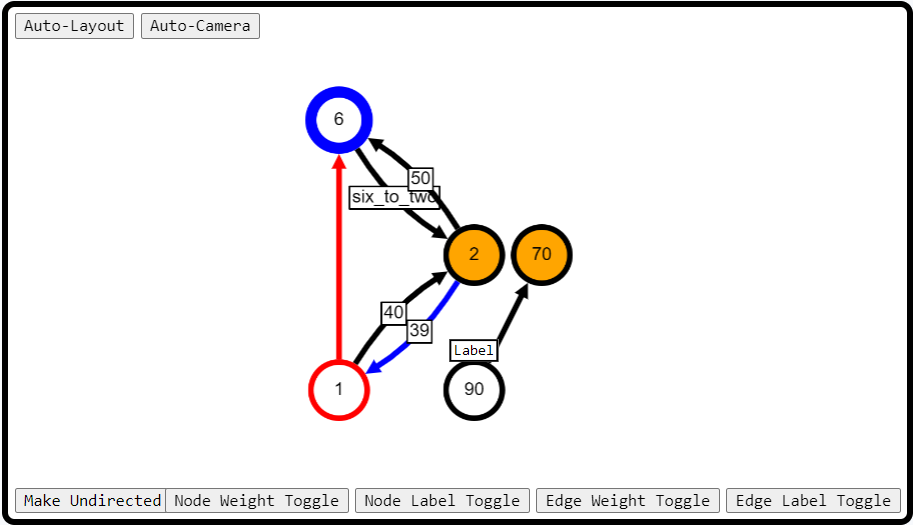
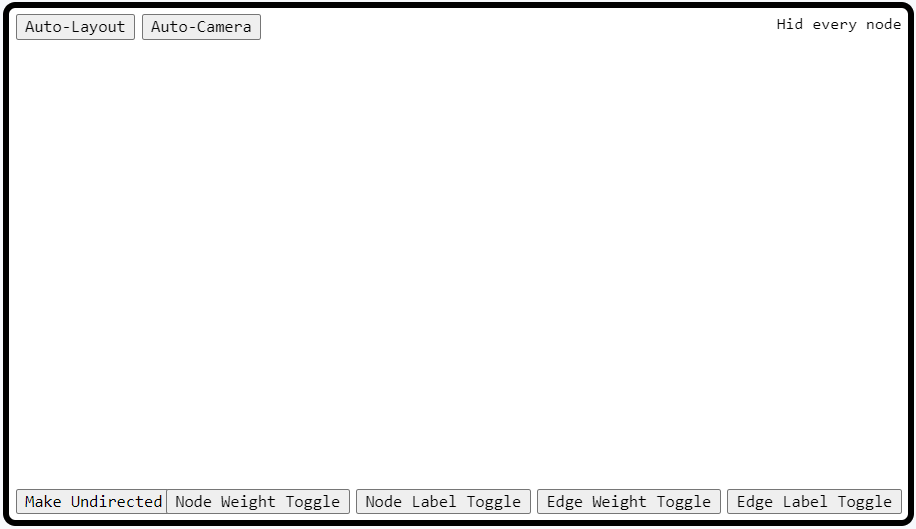
//Hides each node in the graph individually.

for (let node of getNodes()) {

hideNode(node);

}

display("Hid every node");

# 

# **Hide/Show Edge Properties**

hideEdge(edge) //Hides the edge based on the specified edge id

showEdge(edge) //Shows the edge based on the specified edge id

hideEdgeWeight(edge) //Hides the edge weight based on the specified edge id

hideAllEdgeWeights() //Hides all edge weights

showEdgeWeight(edge) //Shows the edge weight based on the specified edge id

showAllEdgeWeights() //Shows all edge weights

hideEdgeLabel(edge) //Hides the edge label based on the specified edge id

hideAllEdgeLabels() //Hides all edge labels

showEdgeLabel(edge) //Shows the edge label based on the specified edge id

showAllEdgeLabels() //Shows all edge labels

**Example Algorithm**

//Hides each edge weight in the graph individually.

for (let edge of getEdges()) {

hideEdgeWeight(edge);

}

display("Hid every edge weight");

