MATH/CPT_S 453 Graph Theory

Final Project Write-Up

A Graph Theorist's Sketchpad: Features and Implementation

By: Nate Gibson

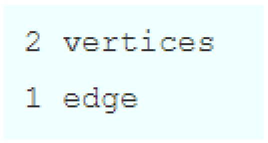
Source Code: https://gitlab.com/nate-gibson/graph-sketchpad

Live version: http://natesgibson.com/graph

Features

1. Graph Information

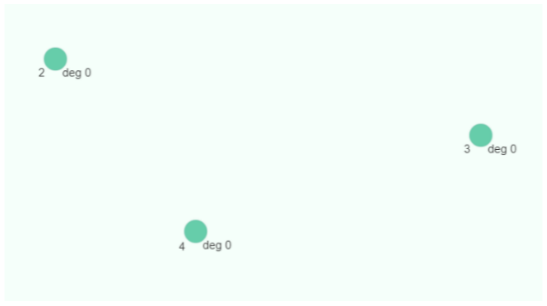
   In the upper left-hand corner of the window, graph information will be displayed and automatically updated. This includes the current number of vertices and the current number of edges in the graph.



```
2 vertices
1 edge
```

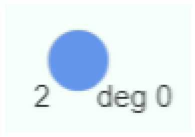
2. Drawing Vertices

   Vertices may be drawn anywhere on the canvas by pressing down the ‘v’ key, provided the cursor is over the canvas and nothing is currently selected. The vertex will be drawn at the position of the cursor.



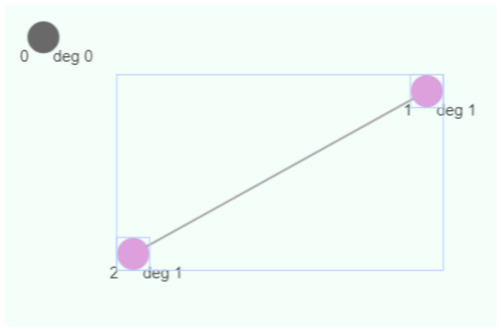
3. Vertex information

   Information about the vertex will be displayed and automatically updated. The unique id of the vertex will appear just below the vertex and to the left. The degree of the vertex will appear just bellow and to the right.



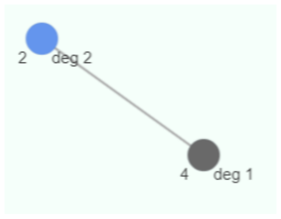
4. Selecting Objects

   One or more objects may be selected by either dragging the blue-purple selection window over the desired object(s), or by holding the shift key and left-clicking on each object the user would like to select. Additionally, selected objects may be un-selected using the shift key method. Once selected, objects will have a light-blue rectangle surrounding them, which is automatically removed should an object become unselected. Selecting objects allows the user to perform actions on the selected objects as a group. Selectable objects include vertices and edges.

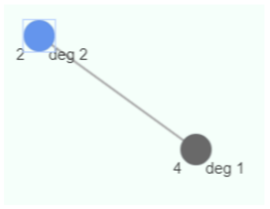


5. Moving Selection

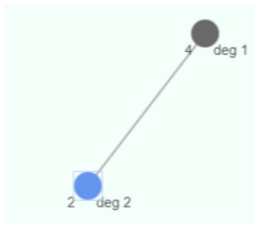
Once selected, objects may be moved by left-clicking and dragging the outer selection rectangle on the canvas. If an object is movable, the cursor will appear as "four arrows" when hovering above the object. If an object is selectable but not movable, the cursor will appear as a "pointer". Note: *drag objects outside of the canvas at your own peril*.



before move



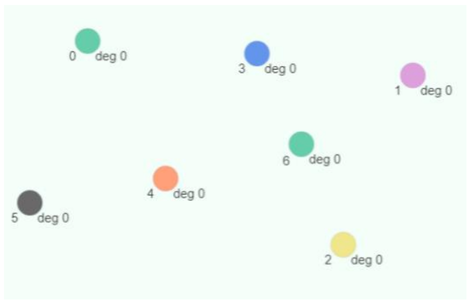
selection made



after move

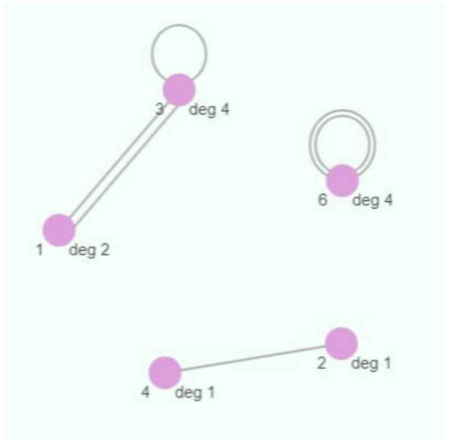
6. Changing the Color of Selected Vertices

If a selection includes one or more vertices, the color of these vertices may be changed by pressing down the 1-6 number keys, with each key corresponding to a different color. The 1-6 numpad keys are supported as well.



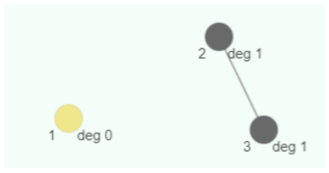
7. Drawing Edges

Edges may be drawn when exactly one or two vertices are selected. If more objects are selected, an edge will not be drawn. If two vertices are selected, a straight edge connecting them will be drawn. More than one edge can connect the same two vertices. These "parallel edges" will be drawn next to each other. If one vertex is selected, an elliptical-shaped loop will be drawn on top of the vertex. Loops add 2 to the degree of their adjacent vertex. More than one loop can be added to a vertex, with each successive loop being drawn larger than the previous. **Edges are undirected by default.**

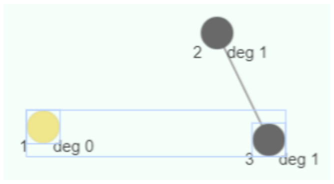


8. Deleting Selection

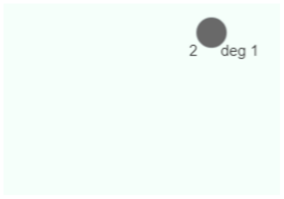
A selection may be deleted by pressing down the delete or backspace buttons. Edges adjacent to deleted vertices will be automatically deleted. Deletable objects include vertices and edges.



before deletion



selection made



after deletion

9. Directed Edges

The

10. Changing Edge-Direction

The direction of an edge can be changed by pressing down the ‘d’ key while exactly one edge is selected. If more objects are selected, the edge direction will not change. The direction of an edge is indicated by an arrow on the end of that edge pointing “from” one vertex “to” the other. An edge adjacent to vertices v_1 and v_2 cycles between $v_1 \rightarrow v_2$, $v_2 \rightarrow v_1$, and being undirected. Undirected edges have no arrows. Arrows are automatically updated when the direction of the edge is changed. Loops cannot have a direction.

11. A

Implementation

Overview:

I implemented this project as a web application, coding it in HTML/CSS and JavaScript. I chose this approach because web apps coded in JavaScript are highly accessible – they will run on pretty much any machine with a sufficiently modern web browser.

Graphics:

The native JavaScript canvas has very limited functionality, which would have required me to code at a very low-level to implement this application's requirements. Instead, I used the JavaScript HTML5 canvas library Fabric.js (http://fabricjs.com).

Fabric.js let me interact with shapes such as lines and circles as objects. These shape objects allow me to define various properties of a shape such as position, size, and color. Once instantiated with properties, adding shape objects to the canvas is as simple as "canvas.add(object)". This greatly streamlined my ability to code the graphics.

Events:

Events allow you to call functions automatically when a certain "event" has taken place. I used the following events in this project:

- On: *object move* – updates the position of edges, **arrows**, and vertex text.
- On: *mouse move* – updates the stored cursor position.
- On: *mouse enter canvas* – updates mouseInCanvas=true, which allows for certain actions (E.g., drawing a vertex).
- On: *mouse exit canvas* – updates mouseInCanvas=false, which removes certain functions (E.g., no drawing a vertex).
- On: *key down* – perform the appropriate action if the pressed key is associated with one (E.g., draw vertex if pressed key is 'v').

Graph, Vertex, Edge:

I implemented Graph, Vertex, and Edge objects as JavaScript classes. Each class stores information pertaining to the object such as degree, position, and id, as well as its associated graphics objects such as a circle for vertices and a line for edges.

The classes contain functions to update the position of the graphics objects, which can be called when certain events happen such as an object moving. For example, if an object is moved, the edges are updated to ensure their end points are still located at the position of their adjacent vertices. The same is true of text objects which display vertex information.

Adjacency List:

The primary graph structure is stored in an adjacency list (“adjList”) in the Graph object. The adjacency list is implemented as a Map, mapping Vertex-object keys to array-of-vertex-objects values. The adjacency list functions as follows:

If an edge connects vertices v_1 to v_2, then adjList.get(v_1) will return an array of adjacent vertices, including v_2. This is reciprocal, so adjList.get(v_2) will return an array which contains v_1. In the case of parallel edges, if there are n parallel edges between v_1 and v_2, each of their associated adjList arrays will contain n copies of the other vertex. In the case of loops, if vertex v contains n loops, adjList.get(v) will contain $2n$ copies of v. If we define loops as adding 2 to the degree of its adjacent vertex, then for any vertex v, the size of its associated adjList array is equal to the degree of v.

Note: Edge information is contained in the adjacency list; however, Edge objects are also stored in a separate array. Importantly, edge objects in this array store and manage graphics objects for each edge. Both the adjacency list and array of edges are updated when edges and vertices are added and deleted.