MATH/CPT_S 453 Graph Theory

Final Project Write-Up

A Graph Theorist's Sketchpad: Features and Implementation

By: Nate Gibson

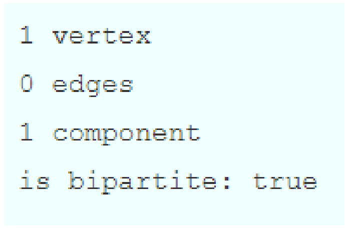
Source code: https://gitlab.com/nate-gibson/graph-sketchpad

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

Features

1. Graph Information

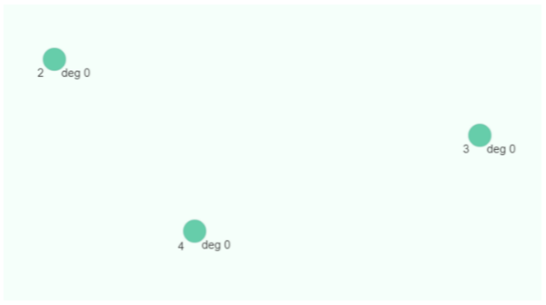
In the upper left-hand corner of the window, graph information will be displayed and automatically updated. This includes the current number of vertices, the current number of edges, the current number of components, and whether the graph is bipartite.



```
1 vertex
0 edges
1 component
is bipartite: true
```

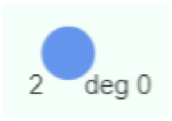
2. Drawing Vertices

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.



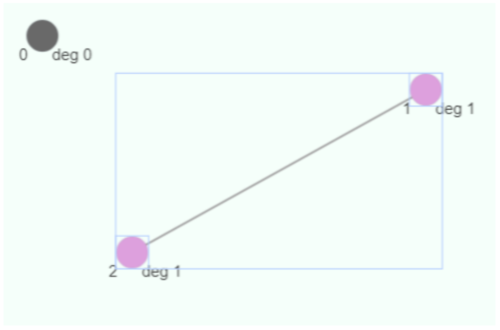
3. Vertex information

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.



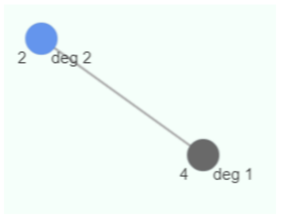
4. Selecting Objects

One or more objects may be selected by either dragging the blue-purple selection window over the desired object(s), or by holding down 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.

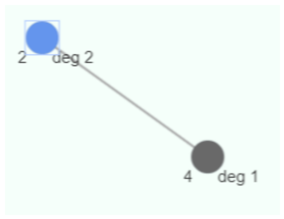


5. Moving Selection

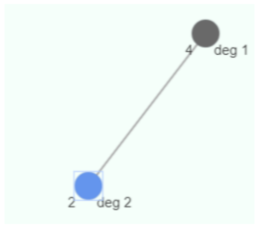
Once selected, objects may be moved by left-clicking and dragging the outer selection rectangle on the canvas. If an object is movable, the cursor will appear as "four arrows" when hovering above the object. If an object is selectable but not movable, the cursor will appear as a "pointer". *Note: drag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



before move



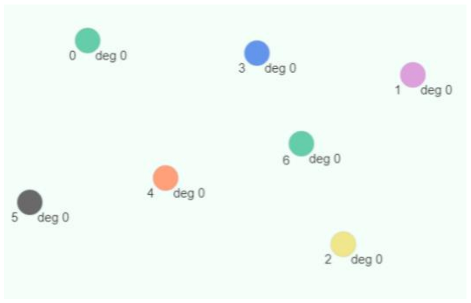
selection made



after move

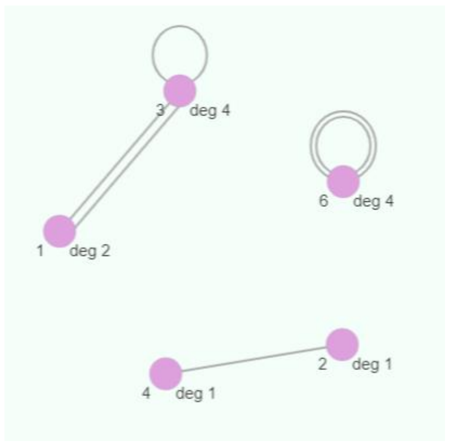
6. Changing the Color of Selected Vertices

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.



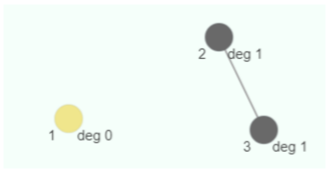
7. Drawing Edges

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.

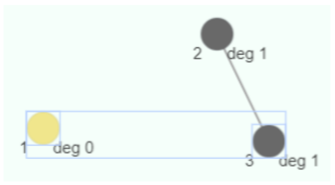


8. Deleting Selection

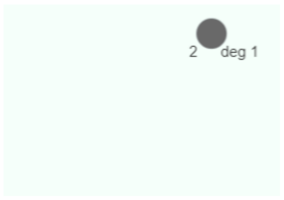
A selection of objects may be deleted by pressing down the delete or backspace key. Edges adjacent to deleted vertices will be automatically deleted. Deletable objects include vertices and edges.



before deletion



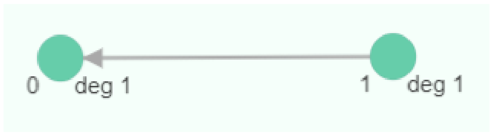
selection made



after deletion

9. Directed Edges

Edges may be directed or undirected. The direction of an edge is indicated by an arrow at the end of that edge pointing "from" one vertex "to" the other.

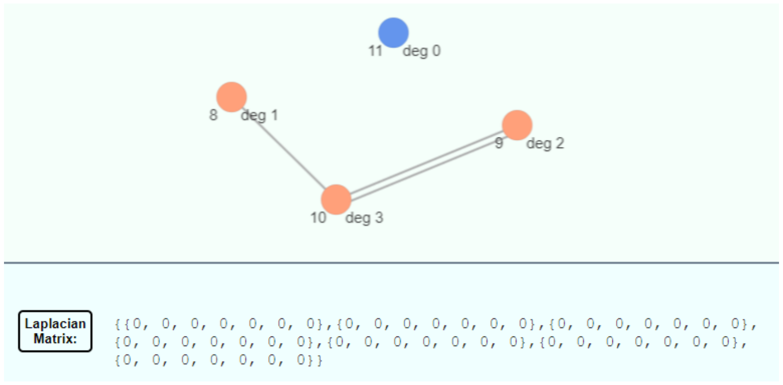


10. Changing the Direction of an Edge

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. An edge adjacent to vertices v_1 and v_2 cycles between $v_1 \to v_2$, $v_2 \to v_1$, and being undirected. Undirected edges have no arrows. Arrows are automatically updated when the direction of the edge is changed. Loops do not have a direction.

11. Displaying the Laplacian Matrix

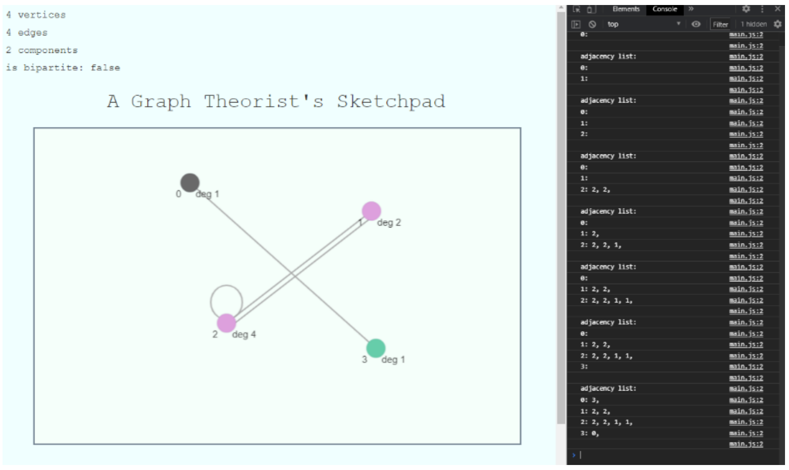
The Laplacian matrix of the graph can be displayed by clicking the "Laplacian Matrix:" button bellow the graph. Computing the associated eigenvalues and eigenvectors is left as an exercise for the user (although the output is WolframAlpha compatible up to 7 vertices, just copy and paste!).



```
{{0, 0, 0, 0, 0, 0, 0},{0, 0, 0, 0, 0, 0, 0},{0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0},{0, 0, 0, 0, 0, 0, 0},{0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0}}
```

12. Console

The console will output the current adjacency list representation of the graph whenever its contents are updated. Access to the console varies by browser. Most browsers allow users to access developer tools, including the console, by pressing down the f12 key.



```
4 vertices
4 edges
2 components
is bipartite: false
```

```
0:                                main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1:                                main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1:                                main.js:2
2:                                main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1:                                main.js:2
2: 2, 2,                          main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1: 2,                             main.js:2
2: 2, 2, 1,                       main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1: 2, 2,                          main.js:2
2: 2, 2, 1, 1,                    main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0:                                main.js:2
1: 2, 2,                          main.js:2
2: 2, 2, 1, 1,                    main.js:2
3:                                main.js:2
                                  main.js:2
adjacency list:                   main.js:2
0: 3,                             main.js:2
1: 2, 2,                          main.js:2
2: 2, 2, 1, 1,                    main.js:2
3: 0,                             main.js:2
                                  main.js:2
>
```

13. “Features”

- When an object is grouped, its coordinates are defined relative to that group. As a result, objects which update their position based on a group object’s coordinates are updating their positions incorrectly, often resulting in them disappearing off the canvas. This can be remedied by moving a single object.
- Parallel edges do not change their endpoint positions relative to the angle of the edge line. As a result, parallel edges can “disappear” from certain angles.
- Arrows which indicate the direction of an edge do not update their angle and position properly in order to be at the end of an edge, directly pointing to a vertex. Instead, they are fixed to the left-hand side of the vertex that they point to. Arrows do change which vertex they point to when the direction of their edge is modified.
- The Laplacian matrix output is dependent on the ids of all vertices being in range (0, number of vertices – 1), with no gaps. If this is not the case, the matrix will not output correctly. This can be caused by deleting a vertex whose id is not currently the largest in the graph. It is intended that the ids of vertices are automatically decremented by 1 in the event that a vertex with a smaller id is deleted, which would not allow for id gaps to be formed. However, id decrementing is not currently implemented, resulting in incorrect Laplacian matrix output when gaps are present.

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., able to draw a vertex).
- On: *mouse exit canvas* – updates mouseInCanvas=false, which removes certain actions (e.g., not able to draw a vertex).
- On: *key down* – perform the appropriate action if the pressed key is associated with one (e.g., draw a vertex if the 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 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 positions 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, then its associated adjList array 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 its degree.

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 edge array are updated when edges and vertices are added and deleted.

Determining the Number of Components:

The number of components is determined by computing a 2d array of components, with each entry in the "components" array being a "component" array of vertices.

The components array is built by iterating over all vertices. I keep track of the "found" vertices in a separate array. If a vertex is not "found" a new component array is created, consisting of that vertex and all other vertices of the same component. One the component array is constructed, all vertices in that component are marked as "found" by placing them in the found array.

To build each component array, the first vertex's adjacent vertices (from the adjList) are added to the component, and recursively add their adjacent vertices to the component as well, provided they are not already in the component (stopping condition).

The length of the components array is equal to the number of components in the graph, which is displayed. Since I am actually computing the contents of each component, I could use this implementation to display more detailed component information in the future.

Determining if the Graph is Bipartite:

Determining whether or not the graph is bipartite is done with what is essentially a "proof of guilt by contradiction", with an assumption of innocence (bipartite-ness?). I assume that the graph is 2-colorable. I then compute two sets, "set1" and "set2," which represent different colors, and then I check to see if they contradict properties of a 2-colorable graph.

To compute these sets, I iterate over all vertices. For each vertex, I add its adjacent vertices to each set the vertex is not a member of. For example, if a vertex a member of set1 and not a member set2, I add its adjacent vertices to set2.

If the graph is bipartite, these sets should contain none of the same vertices (they are disjoint). To verify this, I compute their intersection and check if its size is 0.

If the graph is bipartite, all vertices should be accounted for in the two sets (every vertex in the graph is in their intersection). JavaScript sets do not allow for duplicate entries, so the multiplicity of each vertex in a set is either 0 or 1. This allows me to use the size of each set to check if they account for all vertices in the graph. I check that the sum of (set1.size + set2.size) is equal to the number of vertices in the graph (the size of adjList). Thus, I am not required to compute the intersection of set1 and set2.

I the sets satisfy both of the above positions, then the graph is determined to be bipartite. Otherwise, the graph is determined to be not bipartite. (There is an additional check for if the graph is empty, as the empty graph is considered to be (trivially) bipartite as well.)

<u>Displaying the Laplacian Matrix:</u>

Displaying the graph's Laplacian matrix is accomplished by converting the graph's adjacency list into a Laplacian matrix, implemented as a 2d array, which I can iterate over to compose the string which is displayed.

To convert adjList to a Laplacian matrix, I first build rows by iterating over the keys of adjList, which represent the vertices of the graph. For each vertex, I construct a "row" array whose size is equal to the number of vertices in the graph, and initialize it with zeros. I then iterate over the vertex's adjacency list, incrementing the element of the row array at the index of the adjacent vertex's id by 1 ("row[adjVertex.id]++"). This assumes that the id of a vertex represents its position in the adjacency list (e.g., the first vertex has id 0, the second vertex has id 1, …, the last vertex has id (number of vertices – 1)).

After each row is constructed, I add the row to the end of the Laplacian matrix 2d array. Thus, the 2d array representing the Laplacian matrix is of the form:

$$[[row_1], [row_2], \dots, [row_{\#vertices-1}]]$$

Using this matrix to compose the string that is to be displayed consists of iterating over each column (the first dimension of the 2d array), then iterating over each row (the second dimension of the 2d array).