```
var width = 1400,
    height = 900;

var color = d3.scale.category20();

// Settings to configure d3 force-layout
var force = d3.layout.force()
  .linkDistance(20)
  .charge(-120)
  .gravity(0.5)
  .size([width, height]);

var svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height)
  .attr("class", "graph");

// Get collaborator JSON data from our Github repository
d3.json("https://raw.githubusercontent.com/benkahle/githubCollab/master/bdCollabsByPerson.json",
  function(error1, fullData) {
  if (error1) {
    throw error1;
  }
  // Get collaborator JSON data in format for d3 from out Github repository
  d3.json("https://raw.githubusercontent.com/benkahle/githubCollab/master/graph.json",
  function(error2, graph) {
    if (error2) {
      throw error2;
    }

    var nodes = graph.nodes.slice(),
      links = graph.links.slice(),
      bilinks = [];

    var sourceList = document.getElementById("source");
    var targetList = document.getElementById("target");

    var nodesByName = {};

    // Get a list of all names in the graph
    var names = [];
    nodes.forEach(function(node, i) {
```

```
43      names.push(node.name);
44    });
45
46    // Sort the names alphabeticly and add them to the drop down menus
47    names.sort();
48    names.forEach(function(name) {
49      var option = document.createElement("option");
50      option.value = name;
51      option.text = name;
52      sourceList.add(option);
53      var option2 = option.cloneNode(true);
54      targetList.add(option2);
55    });
56
57    // Start the d3 force-layout with our graph
58    force
59      .nodes(nodes)
60      .links(links)
61      .start();
62
63    // Get a reference to links and append path elements
64    var link = svg.selectAll(".link")
65      .data(links)
66      .enter().append("path")
67      .attr("class", "link");
68
69    // Make a reference to node groups and append g elements
70    var gnodes = svg.selectAll("g.gnode")
71      .data(graph.nodes)
72      .enter().append("g").classed("gnode", true);
73
74    // Get a reference to all nodes, store them by name, and configure mouse effects
75    var node = gnodes.append("circle")
76      .attr("class", "node")
77      .attr("r", 4)
78      .style("i", function(d) {
79        nodesByName[d.name] = d;
80      })
81      .call(force.drag)
82      .on("mouseover", fade(true)).on("mouseout", resetStyling);
83
84    // Add text labels to node groups
85    var labels = gnodes.append("text").text(function(d) { return d.name; });
```

```
    // Add text labels on hover
    node.append("title").text(function(d) { return d.name; });

    // Style the nodes and links to highlight highly connected nodes more prominantly
    resetStyling();

    // Build reference from node indices to links
    var linkedByIndex = {};
    graph.links.forEach(function(d) {
      linkedByIndex[d.source.index + "," + d.target.index] = d;
    });

    // Utility function to get link if nodes are connected
    function isConnected(a, b) {
      if (linkedByIndex[a.index + "," + b.index]) {
        return linkedByIndex[a.index + "," + b.index];
      } else if (linkedByIndex[b.index + "," + a.index]) {
        return linkedByIndex[b.index + "," + a.index];
      } else if (a.index === b.index) {
        return {value: 1};
      } else {
        return 0;
      }
    }

    // Mouse over effect
    function fade(mouseover) {
      return function(d) {
        var opacity = 1;
        if (mouseover) {
          opacity = 0.1;
        }
        node.style("stroke-opacity", function(o) {
          var connectionWeight = isConnected(d, o).value;
          var thisOpacity = connectionWeight ? 1 : opacity;
          this.setAttribute('fill-opacity', thisOpacity);
          if (mouseover) {
            if (connectionWeight > 3) {
              d3.select(this.parentElement).attr("class", "gnode-connected");
            } else if (connectionWeight > 0) {
              d3.select(this.parentElement).attr("class", "gnode-linked");
            }
          } else {
```

```
129             d3.select(this.parentElement).attr("class", "gnode");
130           }
131           return thisOpacity;
132         });
133
134         if (mouseover) {
135           d3.select(this.parentElement).attr("class", "gnode-active");
136         } else {
137           d3.select(this.parentElement).attr("class", "gnode");
138         }
139
140         link.style("opacity", function(o) {
141           var thisOpacity;
142           if (o.source === d || o.target === d) {
143             if (o.value >= 5) {
144               thisOpacity = 1;
145             } else {
146               thisOpacity = o.value/3;
147             }
148           } else {
149             thisOpacity = opacity;
150           }
151           return thisOpacity;
152         });
153       };
154     }
155
156     // Update the graph position when d3 force-layout calculates a new timestep
157     force.on("tick", function() {
158       link.attr("d", function(d) {
159         return "M"+d.source.x+","+d.source.y + " "+d.target.x+","+d.target.y;
160       });
161       gnodes.attr("transform", function(d) {
162         if (d.x > width) {
163           d.x = width;
164         }
165         if (d.x < 0) {
166           d.x = 0;
167         }
168         if (d.y > height) {
169           d.y = height;
170         }
171         if (d.y < 0) {
```

```
172           d.y = 0;
173         }
174         return "translate(" + d.x + "," + d.y + ")";
175       });
176     });
177
178     // Click handler for "Find Path" button
179     document.getElementById("run").onclick = function() {
180       var type = document.querySelector('input[name="search-type"]:checked').value;
181       var results = document.getElementById("results");
182       results.textContent = "Searching....";
183       var resultsList = [];
184       var source = sourceList.value;
185       var target = targetList.value;
186       // Run shortest path or longest path algorithm
187       if (type === "shortest") {
188         resultsList = dijk(source, target);
189       } else {
190         resultsList = bellmanFord(source, target, JSON.parse(JSON.stringify(fullData)));
191       }
192       resetStyling();
193       var stringResults;
194       // If a path is found, show links on side bar and highlight nodes and links in graph
195       if (resultsList.length > 0) {
196         stringResults = "Path: (Length: "+resultsList.length+")\n\n"+resultsList.join("\n");
197         focusPath(resultsList);
198       } else {
199         stringResults = "No path found";
200       }
201       results.textContent = stringResults;
202     };
203
204     // Reset styling on "Clear" button click
205     document.getElementById("clear").onclick = function() {
206       resetStyling();
207     };
208
209     // Function to style graph highlighting highly connected nodes
210     function resetStyling() {
211       node.style("stroke-opacity", function(o) {
212         d3.select(this.parentElement).attr("class", "gnode");
213         return 1;
214       }).style("fill-opacity", 1);
```

```
215       link.style("stroke", function(d) {
216         d3.select(this).attr("class", "link");
217         var colors = [
218           "#0002ff",
219           "#0064ff",
220           "#00a4ff",
221           "#00ffd0",
222           "#00ff36",
223           "#65ff00",
224           "#b0ff00",
225           "#fdff00",
226           "#FFf000",
227           "#FFb400",
228           "#FFa000",
229           "#FF8c00",
230           "#FF7800",
231           "#FF6400",
232           "#FF5000",
233           "#FF3c00",
234           "#FF2800",
235           "#FF2800",
236           "#FF1400",
237           "#FF0000",
238           "#FF0000",
239           "#FF0050",
240           "#FF0050",
241           "#FF0050",
242           "#FF0050",
243           "#FF0050",
244           "#FF0050",
245           "#FF0050",
246         ];
247         return colors[d.value];
248       })
249       .style("opacity", function(d) {
250         if (d.value > 4) {
251           return 1;
252         } else if (d.value > 2) {
253           return d.value/5; //3/5 (.6) or 4/5 (.8)
254         } else if (d.value === 2){
255           return d.value/6; //2/6 (.3)
256         } else {
257           return 0.2; //(.2)
```

```
258         }
259       })
260       .style("stroke-width", function(d) {
261         var base = 3;
262         if (d.value > 4) {
263           return 1*base+"px";
264         } else if (d.value > 2) {
265           return (d.value/5)*base+"px"; //3/5 (.6) or 4/5 (.8)
266         } else if (d.value === 2){
267           return (d.value/6)*base+"px"; //2/6 (.3)
268         } else {
269           return 0.2*base+"px"; //(.2)
270         }
271       });
272     }
273
274     // Function to highlight a path of nodes and links
275     function focusPath(pathList) {
276       node.style("stroke", function(o) {
277         if (pathList.indexOf(o.name) !== -1) {
278           d3.select(this.parentElement).attr("class", "gnode-connected");
279         }
280       });
281       link.style("stroke", function(l) {
282         for (var i = 0; i < pathList.length-1; i++) {
283           var link = isConnected(nodesByName[pathList[i]], nodesByName[pathList[i+1]]);
284           if (l === link) {
285             d3.select(this).attr("class", "connected-link")
286               .style("opacity", 1)
287               .style("stroke-opacity", 1)
288               .style("stroke-width", "2px");
289           }
290         }
291       });
292     }
293
294     // Bellman-Ford Algorithm modified for longest path with dynamic cycling pruning
295     function bellmanFord(source, target, vertices) {
296         var cycles = false;
297         var dists = {};
298         var prev = {};
299         var path = [];
300         //Set inital distances to 1 assuming they are next to each other and set neighbor on shortest path fr
```

```
301         Object.keys(vertices).forEach((key) => {
302                 dists[key] = Infinity;
303                 prev[key] = undefined;
304         });
305
306         //set distance from source to source to be 0
307         dists[source] = 0;
308         //start algorithm at the source
309         //while vertices are still graph
310
311         for(var i=0; i<Object.keys(vertices).length-1; i++){
312                 Object.keys(vertices).forEach((currentVertex)=>{
313                         //store edges before deleting object
314                         var currentEdges = vertices[currentVertex];
315                         //find new shortest paths to all neighboring vertices if available
316                         Object.keys(currentEdges).forEach((neighbor) => {
317                                 var testDist = (-1.0/currentEdges[neighbor])+dists[currentVertex];
318                                 if(testDist < dists[neighbor]){
319                                         prev[neighbor] = currentVertex;
320                                         dists[neighbor] = testDist;
321                                 }
322                         });
323                 })
324         }
325         //prepend the target to the list
326         currentVertex = target;
327         path.unshift(currentVertex);
328         // console.log(prev);
329         while(prev[currentVertex] != source){
330                 // console.log("VERTEX:",currentVertex);
331                 // console.log("PREV:",prev[currentVertex]);
332                 //preprend prev to list and set new current to be the previous
333                 if(path.indexOf(prev[currentVertex])==-1){
334                         path.unshift(prev[currentVertex])
335                         currentVertex = prev[currentVertex]
336                 } else{
337                         // console.log("CYCLE IS:",currentVertex, "to", prev[currentVertex]);
338                         // console.log(vertices[prev[currentVertex]][currentVertex]);
339                         delete vertices[prev[currentVertex]][currentVertex] //= -10;
340                         // console.log(vertices[currentVertex][prev[currentVertex]]);
341                         cycles = true;
342                         break
343                 }
```

```
344         }
345         if(!cycles){
346                 path.unshift(source)
347                 return path;
348         } else {
349                 console.log("Found a cycle with path length", path.length);
350                 return bellmanFord(source, target, vertices)
351         }
352     }
353
354     // Dijkstra's Algorithm for shortest path finding
355     function dijk(source, target){
356         var dists = {};
357         var prev = {};
358         var path = [];
359       var vertices = JSON.parse(JSON.stringify(fullData));
360         //Set inital distances to infinity (and beyond) and set neighbor on shortest path from the source to
361         Object.keys(vertices).forEach((key) => {
362                 dists[key] = Infinity;
363                 prev[key] = undefined;
364         });
365
366         //set distance from source to source to be 0
367         dists[source] = 0;
368         //start algorithm at the source
369         var currentVertex = source;
370         //while vertices are still graph
371         while (Object.keys(vertices).length >0){
372                 //preset minDistance not visted yet to infinity
373                 var minDist = Infinity;
374                 //find unvisited node with minimum distance from source
375                 Object.keys(vertices).forEach((vertex) =>{
376                         var vertDist = dists[vertex];
377                         if(dists[vertex] < minDist) {
378                                 minDist = dists[vertex];
379                                 currentVertex = vertex;
380                         }
381                 });
382
383                 //store edges before deleting object
384                 var currentEdges = vertices[currentVertex];
385                 //delete the current vertex from graph because we have now visited it
386                 delete vertices[currentVertex];
```

```
387
388         // If no more edges, no path is possible
389         if (!currentEdges) {
390           return [];
391         } else {
392           //find new shortest paths to all neighboring vertices if available
393           Object.keys(currentEdges).forEach((neighbor) => {
394             var testDist = currentEdges[neighbor]+dists[currentVertex];
395             if(testDist < dists[neighbor]){
396               prev[neighbor] = currentVertex;
397               dists[neighbor] = testDist;
398             }
399           });
400
401           //if our current vertex is the target, go down the prev tree to find the whole path
402           if(currentVertex === target){
403             //prepend the target to the list
404             path.unshift(currentVertex);
405             while(prev[currentVertex] != undefined){
406               //preprend prev to list and set new current to be the previous
407               path.unshift(prev[currentVertex])
408               currentVertex = prev[currentVertex]
409             }
410             return path;
411           }
412         }
413         }
414     }
415   });
416 });
```