# Graph Layout - A5

**Due: Dec 6, 2022 at 11:59PM**

## 1 Graph Layout - A5 Specification

### 1.1 Assignment Instructions

- This is an *individual* assignment.
- Make sure to read through all of the specifications so your submission is complete.
- Follow all the submission steps in the Setup document by the assignment deadline.

### 1.2 Assignment Link

The skeleton code for the assignment is available at `https://classroom.github.com/a/w8Lh0OTv`.

## 2 Background

The goal of this assignment is to draw a weighted graph with a visually clear layout. Graphs do not have an inherent spatial layout: the nodes and edges of a graph can be drawn wherever we want without losing any information in the visualization. Nevertheless, there are certain ways of drawing the graph that are easier to visualize connections. Maybe we want to minimize line crossings, or keep similar nodes next to one another. Getting a computer to automatically produce aesthetically pleasing graph visualizations can be quite challenging and algorithmically intensive!

In this assignment, you will implement a weighted force-directed layout algorithm to organize and visualize a randomly generated weighted graph.

### 2.1 Force-Directed Approach

Force-directed layouts are one of the most common types of algorithms for calculating graph layouts. These algorithms treat the graph as being subject to a set of physical "forces" that move the nodes around the graph until they settle into an equilibrium state. For example, we may treat the nodes as repulsive "magnets" that push each other away, but treat the edges as attractive "springs" that draw connected nodes back to one another. Force-directed layouts are also known as spring-and-damper algorithms.

With this approach, we let each node push each other a little ways, let each edge pull its nodes a little ways, and keep looping these small movements until the graph looks good. Because we're using physical forces as our model (e.g., magnetism, gravity, spring-tension), we can often borrow from simple algorithms like Coulomb's Law (the "inverse square law") and Hooke's Law to determine how far and in what direction each node should move.

These algorithms are fairly simple to implement compared to other graph layout algorithms, but produce interesting behaviors (and nice layouts). However, they have the bad consequence of having a high run-time. Because each node influences every other node, they can often be $O(n^2)$ or $O(n^3)$.

#### 2.1.1 Fruchterman-Reingold Algorithm

The Fruchterman-Reingold algorithm is one of the simplest and earliest force-directed layout algorithms. It works well on small and medium graphs and converges relatively quickly to an equilibrium point. In this model, each node applies a repulsive force upon every other node inversely proportional to the square of the distance between them and edges act as springs that apply an attractive force between nodes that is proportional to the square of the distance between the nodes.

The original pseudocode for this algorithm can be found on page 4 of the paper that introduced the idea. An updated version (below) takes edge weights into account, which the original one does not. This version also skips an outer loop that the original includes – that loop appears in our GUI and enables redraws after each step.

```
FRUCHTERMAN-REINGOLD-STEP()
1:  $area \leftarrow width * height$
2:  $k \leftarrow \sqrt{area/|V|}$
3:
4:  for each vertex $v$ in $V$ do                                   // calculate repulsive forces
5:      $v.disp \leftarrow 0$
6:      for each vertex $u$ in $V$ do
7:          if $u \neq v$ then
8:              $\delta \leftarrow v.pos - u.pos$
9:              $v.disp \leftarrow v.disp + (\delta/|\delta|) * f_r(|\delta|, k)$
10:
11: for each edge $e$ in $E$ with endpoints $v$ and $u$ do           // calculate attractive forces
12:     $\delta \leftarrow v.pos - u.pos$
13:     $v.disp \leftarrow v.disp - (\delta/|\delta|) * f_a(|\delta|, k) * e.weight$
14:     $u.disp \leftarrow u.disp - (\delta/|\delta|) * f_a(|\delta|, k) * e.weight$
15:
16: for each vertex $u$ in $V$ do                                   // update vertex positions
17:     $v.pos \leftarrow v.pos + (v.disp/|v.disp|) * min(|v.disp|, t)$
18:     move $v.pos.x$ into the frame
19:     move $v.pos.y$ into the frame
20:
21: $t \leftarrow cooling(t)$
```

Helper functions:

```
$f_a$(x, k)
1: return $x^2/k$
```

```
$f_r$(x, k)
1: return $k^2/x$
```

## 3 Implementation

Before writing any code, run the project and experiment with the available buttons. Try clicking on nodes and click-dragging them around. The nodes don't move on their own because the `layout()` method doesn't change their positions.

Implement the two `TODO` methods for the assignment. You can write any additional helper methods if you find that they are useful.

There are some important pieces of the algorithm that can use additional detail.

### 3.1 $f_a$ and $f_r$ Functions

The **a**ttractive and **r**epulsive functions borrow from gravitational and electrostatic equations in physics. Nearby nodes push/pull more strongly than distant ones.

### 3.2 Vector Calculations

All of the positions and displacements are represented as mathematical vectors. If you want to refresh your knowledge on vector arithmetic, check out this site.

Some rules:

- The vector between two points can be calculated by separately subtracting the x-coordinates and the y-coordinates
- Points $(x, y)$ can directly be treated as vectors if you just think of the vector as coming out of the origin $(0, 0)$
- The notation $|V|$ indicates the magnitude of a vector $V$
- You can calculate the length of a vector using the basic distance formula
- You can add vectors together by adding their coordinates
- You can multiply vectors by scalars (normal numbers) by multiplying each component independently

### 3.3 Temperature

The temperature variable $t$ determines the "heat" in the system: how energetically nodes move. The smaller the value, the longer the graph will take to converge to an equilibrium point.

The *cooling*() function should decrease the temperature by a small amount, either by a fixed amount or a multiple that's slightly smaller than 1. This makes nodes move slower after the simulation has run for a while. The temperature should have a (positive!) minimum value as well, so it doesn't decrease forever. Depending on the temperature you pick, you might see the graph jitter once it's reached equilibrium. That's a normal behavior.

In order to keep track of this value, you should have a class-level variable that updates during each `layout()` call. Try out some initial values between $1 - 10$ to see what works well for you.

### 3.4 Keep Nodes in the Frame

It's possible that the displacement of a node will move it outside of the frame. We use a frame based on the *width* and *height* values, centered on the origin $(0, 0)$. If a node's update would move it outside the frame, instead move it to the nearby edge so that it's still visible.

## 4 Testing

Incrementally test your program as you write it. For this assignment, there are a few possible ways to approach that.

You can try to update the position of a single node each time `layout()` is called, or try to implement only one of the forces (repulsive or attractive). Run the program after each section is built to see if it behaves how you expect it to. For example, if you have only implemented attractive forces then you should expect all the nodes to pile on top of each other.

Once you have completed both node and edge forces, experiment with the initial temperature and cooling function to find values that balance node movements so they aren't too fast or too slow.

## 5 Grading

Nodes repel each other: $+35\%$

Edges attract each other: $+35\%$

Temperature resets and decreases appropriately: $+5\%$

Excluded node does not move: $+5\%$

Node layout eventually settles (or jitters slightly): $+15\%$

Moving nodes stay within the frame: $+5\%$