Assignment 1: CS 106L GraphViz

Assignment developed by Keith Schwarz.

Due Thursday, April 19 at 11:59 PM

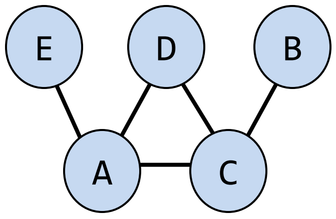
Introduction

One of the most useful abstractions in computer science is the *graph*, a means of expressing relationships between objects. Formally, a graph is a collection of *nodes* (also called *vertices*) joined together by *edges* (also called *arcs*), where each node represents some object and each edge connects two nodes that are somehow related. For example, Facebook represents people and friendships as a graph – each person is node, and two people are connected by edges if they are friends of one another. The Google search engine is powered by the PageRank algorithm, which treats webpages as nodes and has an edge between any two pages that have links to one another.

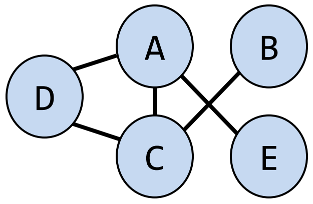
One advantage of using graphs to represent relationships is that graphs have a natural geometric interpretation. In particular, we can visualize the structure of a graph by drawing a dot for each node and a line between any two nodes joined by an edge. For example, suppose that we want to represent friendships in a small group of people using a graph. Let's name the people in this group Alice, Bob, Christine, David, and Evelyn and suppose that their friendships are as follows:

- Alice, Christine, and David are all friends of one another.
- Bob and Christine are friends.
- Evelyn and Alice are friends.

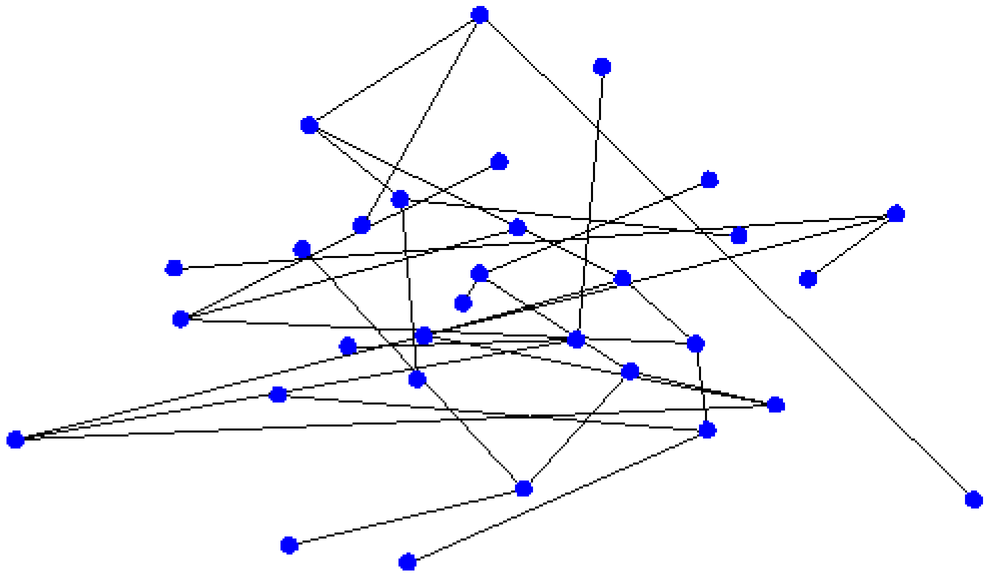
We could then draw these relationships in a graph as follows:



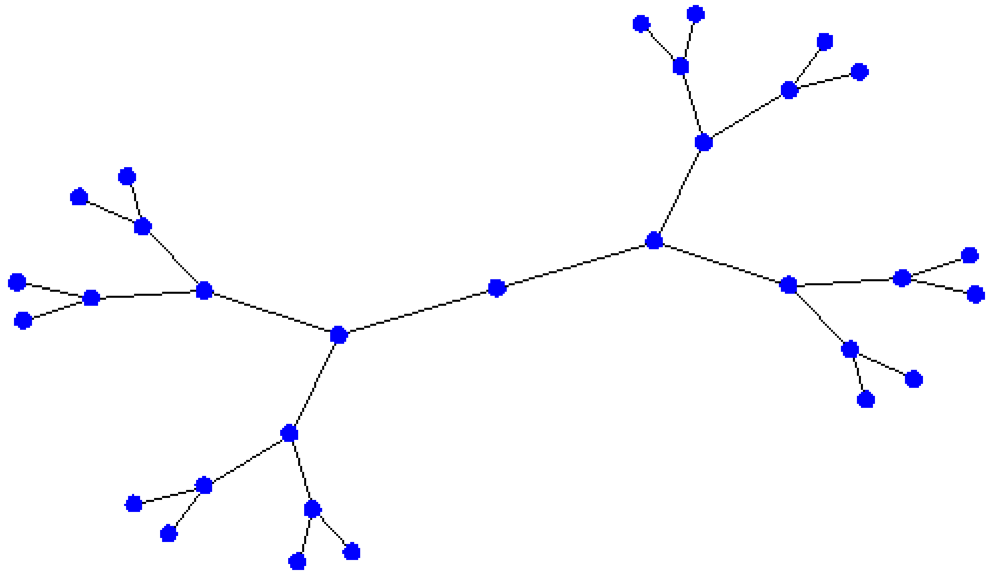
This picture makes it much easier to see who is friends with one another. An important detail to notice, though, is that this is not the only possible way that we could have drawn this graph. Below is an entirely different rendering of the graph:



In both cases, it is possible to discern the group's friendships, though the first drawing is easier to interpret than the second. In the extreme, it is possible to lay out graphs in ways that entirely obscure the relationships between the graph's nodes. For example, consider the following graph layout:



This drawing is a mess! Many of the arcs cross one another, the nodes seem to be scattered around randomly, and the overall drawing is garbled. However, consider this alternative drawing below:



This drawing is for *exactly the same graph* as the previous image, but the structure is much more visible. The drawing is symmetric, the nodes are spaced apart in an aesthetically pleasing manner, and the edges do not cross one another.

In general, it can be difficult to find aesthetically pleasing layouts for graphs. The question then arises – given an arbitrary graph, how can we produce a good drawing for that graph? This is called the *graph drawing problem* and has been studied extensively. Interestingly, while there are many good algorithms for drawing graphs, no one algorithm shines as “the best” graph drawing algorithm. Every algorithm has its strengths and weaknesses, and many algorithms that work marvelously on certain graphs will produce poor layouts for other graphs. In this assignment, you'll learn about one particular type of algorithm called a *force-directed layout algorithm* and will get a chance to see its performance on a variety of graphs. In doing so, you'll cement your skills with the streams library and the STL `vector` class. Plus, you'll end up with an incredibly entertaining piece of software.

Graph Layout Heuristics

Before going into the details of how force-directed layouts work, we need to address the following question: how can a computer, which can only blindly crunch numbers, tell whether a particular graph drawing is aesthetically pleasing to its human masters? This question is both difficult to pose and difficult to answer – what one person finds intuitive might leave another baffled – and in this assignment we won't try to answer it directly. Instead, we will come up with a set of criteria that *in general* lead to nice graph layouts, and then we will design an algorithm to try to pick layouts meeting these criteria. This will not always result in ideal graph drawings, but will do a surprisingly good job laying out many graphs.

There are many heuristics that could be used to determine how good a graph drawing is. For example, we could try to minimize the number of arcs crossing one another, or try to maximize the symmetry of the drawing. In this assignment, however, we'll limit ourselves to the following two heuristics, which work remarkably well in practice:

- **Position connected nodes near one another**. When laying out a graph, it is often a good idea to place nodes that are joined by an edge near one another. The idea behind this metric is that if connected nodes are near each other, a reader can focus on one part of the graph and see the local structure of the graph in that region.
- **Maximize the distance between unconnected nodes**. If two unconnected nodes are placed near one another, the reader can be confused into thinking that they are somehow related when in fact there is no connection between them.

Now that we have a set of heuristics for elegant graphs, how do we go about placing nodes according to those heuristics? In general, this problem is difficult. The position of each node influences the position of each other node, so picking a layout that balances the heuristics requires solving a large, complicated system of equations.[1] Fortunately, there is a clever strategy that lets us avoid this complexity. Rather than solving for the answer directly, we'll instead start off with a random guess, and then continuously refine the positions of the nodes to improve upon the heuristics. Over time, the nodes will reach positions that balance the heuristics, and we'll end up with an aesthetically pleasing graph drawing.

1 Later on when we cover the actual math behind node placement, it's a fun exercise to think about solving the system of equations manually. This is very difficult because, as you'll see, the equations governing node positions are nonlinear.

The key step in this algorithm is finding some way to update the node positions. To do this, we'll take a cue from physics. At each step of the graph layout, we'll have the nodes exert *forces* on one another. These forces will push and pull nodes that are in suboptimal positions until they eventually come to rest in a stable location. Because we want unconnected nodes to be far apart from one another, we'll have each node repel each other node with some force. Similarly, because we want connected nodes to remain near one another, we'll have each edge between nodes attract its endpoints. As these forces move the nodes around, the positions of the nodes will tend toward configurations where the net force on each node is minimized; that is, when there is a balance between the forces spacing out unconnected nodes and keeping together connected nodes.

The general skeleton of our algorithm is as follows:

```
Assign each node in the graph an initial location.
While the layout is not yet finished:
    Have each node exert a repulsive force on each other node.
    Have each edge exert an attractive force on its endpoints.
    Move the nodes according to the net force acting on them.
```

The details of each of these steps are quite customizable: we can initialize the positions of the nodes however we choose, decide when to stop based on multiple criteria, and use almost any function to determine the strengths of the attractive and repulsive forces between nodes. In this assignment, you'll implement one particular version of a force-directed layout algorithm, but I highly encourage you to try out your own variations; I've suggested several ideas at the end of this handout.

Let's now go over the specifics of how each of the above steps work.

Assigning initial node positions. There is no one "correct way" to initially lay out the nodes in the graph. Because the algorithm takes an initial guess and improves over time, pretty much any initial layout will do. In this assignment, however, you'll initially position the nodes so that they are evenly spaced apart on a unit circle. In particular, in a graph with *n* nodes, node *k* should be placed at:

$$\left(\cos\frac{2\pi k}{n}, \sin\frac{2\pi k}{n}\right)$$

Here, angles are represented in radians. In C++, you can compute sine and cosine using the `sin` and `cos` functions exported by the `<cmath>` header file. There is no predefined C++ constant for π, but you should be fine with:

```
const double kPi = 3.14159265358979323;
```

Determining whether to continue iterating. The heart of the force-directed algorithm is the iteration. If the iteration cuts off too early, then the nodes may not have moved into good positions; if the iteration runs too long, then the computer will waste time trying to improve upon an already perfect layout. Although there are ways to automatically detect whether to continue or cut off the iteration,[2] in this assignment we'll adopt a simple approach – asking the user to control how long the algorithm runs. In particular, before running the algorithm, you should prompt the user for a number of seconds, and then run the algorithm until that many seconds have elapsed.

In C++, you can retrieve the current time by using the time function, exported by `<ctime>`. The `time` function returns a value of type `time_t` which holds some unique value identifying the current time. In order to track how much time has elapsed in the simulation, you can use the `difftime` function. `difftime` takes as arguments two `time_t` values and reports the number of seconds that have elapsed between the two time points. For example, the following illustrates one way to use `difftime` to measure how long a computation takes:

```
time_t startTime = time(NULL);

/* ... some complex operation ... */

double elapsedTime = difftime(time(NULL), startTime);
```

Here, the first line invokes the `time` function to get the start time, and the last line uses `difftime` to compute the difference in time between now and the start time.

Computing forces. The particular approach we will use to compute forces is based off of the *Fruchterman-Reingold algorithm*, an early but effective force-directed layout algorithm. In Fruchterman-Reingold, every node exerts a repulsive force on every other node that is inversely proportional to the distance between those nodes, and each arc exerts an attractive force on its endpoints proportional to the *square* of the distance between those nodes. This means that as connected nodes grow more distant to one another, the attractive force rises quickly and the repulsive force drops off, so connected nodes will have a tendency to “snap back” toward one another. Similarly, as the nodes draw increasingly close, the repulsive force grows rapidly while the attractive force diminishes, so the nodes will move away from each other. Only when the nodes are at a perfect distance from one another will the forces balance and the nodes cease to move.

To keep track of the net forces on each node, you'll maintain a Δx and Δy value for each node which stores the net forces on that node along the x and y axes. It's important to track both, since it's possible that a node might be repelled entirely vertically (in which case it will have a strong force in the y direction but no force in the x direction) or horizontally (strong force in the x direction, no force in the y direction). The net forces in each direction begin at zero, but will be adjusted by the interactions of each node with each other node.

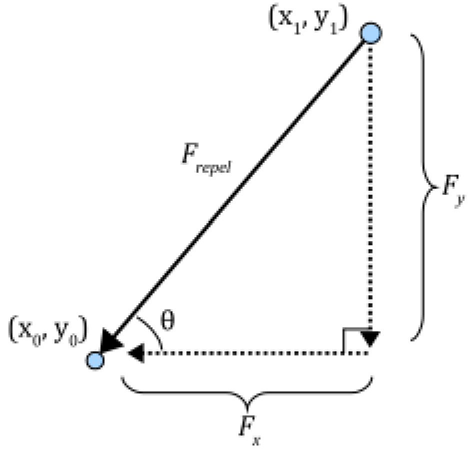
2 One common approach is to compute the *kinetic energy* of the system, which is given by the sum of the squares of the Δx and Δy terms of each of the nodes. When the kinetic energy drops below some particular threshold, the algorithm can assume that the graph is more or less in its final configuration and can stop the iteration.

The first source of force acting on each node is the repulsive force exerted by every node against every other node. For each pair of nodes (x_0, y_0) and (x_1, y_1), the magnitude of the repulsive force F_{repel} between these nodes is

$$F_{repel} = \frac{k_{repel}}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}$$

…where k_{repel} is a constant that controls the strength of the repulsive attraction. If the magnitude of this constant increases, then nodes repel each other more strongly; if it has a small magnitude, then nodes hardly repel each other at all. I advise using a value of 10^{-3} for this constant, but you are free to experiment with this value if you wish.

Once you have computed the magnitude of the repulsive force, you will need to determine how much of that force is in the *y* direction and how much of it is in the *x* direction. To see how this force splits up, consider the following diagram:



Here, F_x and F_y are the forces in the x and y directions exerted on the node (x_0, y_0). Using some simple trigonometry, we get that:

$$F_{x_0} = -F_{repel} \cos\theta$$
$$F_{y_0} = -F_{repel} \sin\theta$$

By Newton's laws, the force exerted against the node (x_1, y_1) are thus

$$F_{x_1} = +F_{repel} \cos\theta$$
$$F_{y_1} = +F_{repel} \sin\theta$$

Here, we've been using the angle θ to represent the angle indicated in the above drawing. Again, simple trigonometry tells us that:

$$\theta = \tan^{-1}\frac{y_1 - y_0}{x_1 - x_0}$$

In C++, you can compute this arctangent using the `atan2` function, also in `<cmath>`. `atan2` takes in two arguments – the first representing $y_1 - y_0$ and the second $x_1 - x_0$ – and returns the value of the above expression in radians. For example:

```
  double theta = atan2(y1 - y0, x1 - x0);
```

To summarize, the logic for computing repulsive forces is as follows:

```
    For each pair of nodes $(x_0, y_0)$, $(x_1, y_1)$:
        Compute $F_{repel} = k_{repel}$ / sqrt $((y_1 - y_0)^2 + (x_1 - x_0)^2)$
        Compute $\theta$ = atan2$(y_1 - y_0, x_1 - x_0)$
        $\Delta x_0$ -= $F_{repel}$ cos$(\theta)$
        $\Delta y_0$ -= $F_{repel}$ sin$(\theta)$
        $\Delta x_1$ += $F_{repel}$ cos$(\theta)$
        $\Delta y_1$ += $F_{repel}$ sin$(\theta)$
```

The second source of force acting on each node is the attractive forces between nodes joined together by edges. In this case, the attractive force $F_{attract}$ is proportional to the square of the distance between the nodes:

$$F_{attract} = k_{attract}\ ((x_1 - x_0)^2 + (y_1 - y_0)^2)$$

Again, $k_{attract}$ is a constant controlling the strength of the attractive force. As with k_{repel} we suggest using the value 10^{-3}, but you should feel free to experiment with this value as well.

Using logic similar to the above reasoning with repulsive forces, we can compute how this force divides over the *x* and *y* components as follows:

```
    For each edge:
        Let the first endpoint be $(x_0, y_0)$
        Let the second endpoint be $(x_1, y_1)$
        Compute $F_{attract} = k_{attract}$ * $((y_1 - y_0)^2 + (x_1 - x_0)^2)$
        Compute $\theta$ = atan2$(y_1 - y_0, x_1 - x_0)$
        $\Delta x_0$ += $F_{attract}$ cos$(\theta)$ // Note that this is a += and not a -=!
        $\Delta y_0$ += $F_{attract}$ sin$(\theta)$
        $\Delta x_1$ -= $F_{attract}$ cos$(\theta)$ // Note that this is a -= and not a +=!
        $\Delta y_1$ -= $F_{attract}$ sin$(\theta)$
```

Moving Nodes According to the Forces. Once you've computed the net Δx and Δy for each node, moving the nodes is an easy step – simply increment each node's x coordinate by its Δx and each node's y coordinate by its Δy. Do **NOT** update the x and y coordinates for each node until you have calculated the Δx and Δy for **ALL** the nodes. Otherwise, your force calculations for later nodes will be incorrect.

Note that in a true physical simulation the forces on each object would change the *velocity* of the object rather than directly modifying its position, but for our simplified example changing position alone is sufficient. If you'd like to experiment with each node having a velocity and a position, feel free to do so.

The Assignment

Your assignment is to implement a simple program which reads in a graph from a file, then runs the force-directed layout algorithm described above to find an appealing layout for the graph. In particular, your program should do the following:

1. Prompt the user for the name of a file containing the graph to visualize. (The graph file format is discussed below.)
2. Prompt the user for the number of seconds to run the algorithm, which should be positive.
3. Place each node into its initial configuration.
4. While the specified number of seconds has not elapsed:
    1. Compute the net forces on each node.
    2. Move each node by the specified amount.
    3. Display the current state of the graph using the provided library.
5. (optional) Ask the user if they want to display another graph and loop accordingly.

In the provided starter code, we have provided you a header file, `SimpleGraph.h`, which exports a set of types for representing a graph. In particular, you're given:

- `Node`, a struct representing a node in a graph. It simply stores an x and y position.
- `SimpleGraph`, which stores two `vector`s, one of `Node`s and one of `Edge`s.
- `Edge`, a struct representing an edge in the graph. Each `Edge` has two fields, `start` and `end`, which correspond to the indexes of the edge's endpoints in the `Node` `vector`.

The provided starter code also contains many sample graphs you can run your program on. Each graph is represented as a file, where the first line contains the number of nodes and each remaining line defines an edge. For example, here is a graph file where the nodes and edges represent the corners and edges of a cube:

```
8
0 1
1 2
2 3
3 0
4 5
5 6
6 7
7 4
0 4
1 5
2 6
3 7
```

Here, the first line of the file indicates that there are eight nodes in the graph, and the remaining lines consist of two numbers denoting which nodes are connected by edges. For example, the line `0 1` indicates that there is an edge from node 0 to node 1, the line `3 7` that there is an edge from node 3 to node 7, etc. You can assume that every file the user opens is well-formed, though if you wanted to make this program more resilient you may want to add some code to check that this is the case.

The starter code also contains a module called `GraphVisualizer.h/.cpp`. This file contains two functions: `InitGraphVisualizer`, which sets up the internal state of the visualizer, and `DrawGraph`, which accepts a `SimpleGraph` and displays it on screen. At the start of your program, you should call `InitGraphVisualizer`, and after each iteration of the main algorithm, you should call `DrawGraph` on the current graph configuration. This will let you watch as the computer incrementally finds a good layout for the graph.

When writing this program, you should **not** use any of the CS 106B/X libraries. That is, you should not use anything from the `strlib.h`, `scanner.h`, or `simpio.h` headers. However, you are free to use anything off of the CS 106L course website, including the implementations of the functions contained in these headers. This may seem like an arbitrary restriction, but the idea is to get you to write your entire program using purely portable C++ code.

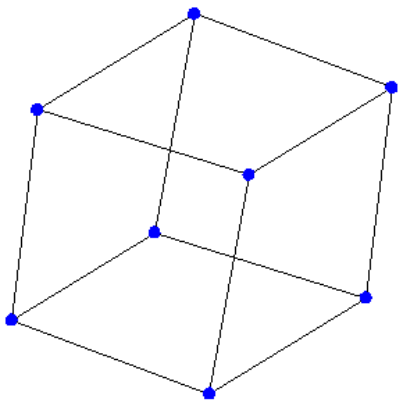
Task Breakdown

Although this handout is fairly lengthy, *don't panic* – you don't need to write very much code to complete this assignment. To help you get started, I've constructed one possible breakdown of the required tasks. You should feel free to implement the assignment in any way you see fit, so don't worry if this ordering sounds unusual.

- *Implement code to read a graph from disk*. As a first step, I suggest writing a set of functions that will prompt the user for the name of a graph file and then read that file from disk. The actual file reading should not be particularly difficult, and depending on how you go about reading the file it may actually require more code to prompt the user for a valid filename than to actual read in the graph. You should be sure to use `getline` (or a wrapper function) to read input from the user.

- *Implement the logic to initially position the nodes.* Before you can run the force-directed algorithm, you will need to position all of the nodes in the graph in a circle. I suggest getting this step working before writing the iterative part of the algorithm since it's easy to check whether it's working. In particular, you can always call `DrawGraph` and check whether all of the nodes are arranged in a circle.

- *Implement the force-directed layout algorithm*. This is the core of the assignment, but fortunately the code necessary to get it working is not too daunting. You will ultimately need to prompt the user for how long to run the algorithm, but initially I suggest just putting the iteration in a `while (true)` loop and just letting the algorithm run forever. As a test of whether your code is working, if you run the algorithm on the graph `10line`, your algorithm should eventually end up arranging the points in a straight line. Similarly, running the algorithm on the file `cube` should arrange the points in a way that looks like a 3D perspective drawing of a cube, as shown here:



- *Implement the code to time the simulation.* Once you have all the rest of the program working, add in the code to prompt the user for the length of time to run the algorithm, and then update your iteration to terminate once that amount of time has passed. Feel free to use lecture code from the CS 106L website to read a number of seconds from the user.

Advice, Tips, and Tricks

Here are a few specific pointers that might make your life a lot easier as you go through this assignment:

- *Don't hesitate to ask questions!* **The point of this assignment is to give you a chance to play around with C++ and build cool software, not to punish you for not understanding a particular library or language detail**. If you're having trouble understanding the starter code, run into inexplicable runtime errors, or can't seem to get some part of the assignment working, please send me an email or talk to me after lecture. I genuinely love this material and want to help you learn it, so don't hesitate to ask questions if you need to.

- *Make use of stream extraction when reading from a file.* If you can load data using `>>`, take advantage of it. Don't try to parse it yourself if you don't have to.

- *Be careful when reading input from the user.* As mentioned in class, directly reading data from `cin` is dangerous and can completely break input in the program. You should instead use a wrapper around `cin` using functions like `getline`, as I showed in class. If you'd like to use lecture code in your program, that's completely fine, but you should be sure to cite it.

- *Watch out for integer truncation.* When you are working with numbers, you need to make sure to avoid dividing two integers if you need a double. 3/2 = 1, but what you really want is 3.0/2 = 1.5.

- *Make sure to reset Δx and Δy to 0 after you update the x and y coordinates*. Otherwise, when you go to update x and y in the next iteration, your Δx and Δy will be incorrect.

- *Don't call* `InitGraphVisualizer` *more than once.* Calling `InitGraphVisualizer` multiple times can result in the screen flickering, which will make your visualization not run as smoothly.

Extensions

This assignment implements one of many algorithms that can be used to draw graphs. Here are a few possible extensions you may want to experiment with:

1. *Change the relative strength of the attractive and repulsive forces*. The aforementioned algorithm uses two constants, k_{repel} and $k_{attract}$, which control how strong the attractive and repulsive forces are. My suggestion is to set both of these values to 10^{-3}, but there's no reason that they stay at this value, or that they even stay in a 1:1 ratio. Try changing these values independently of one another. What do you notice happening to the graph layouts? Do any values cause the algorithm to completely fail to work?

2. *Add random pertubations*. One potential problem with this force-directed algorithm is that the graph can get stuck in a *local minimum*, a configuration in which all of the forces are balanced but which is not the optimal configuration. Often, this manifests itself in a situation in which if any of the nodes were to move even a slight amount, the graph would collapse into a much better configuration. For example, consider a graph in which three nodes are each connected to one another in a triangle. One possible layout for the graph would be to put them all in a straight line. If this happens, then the repulsive and attractive forces on the nodes would always push and pull along that line, and so the nodes would always be collinear. However, if one of the nodes were to get bumped slightly out of place, then the

forces between the nodes would no longer be along a line and the nodes will quickly arrange themselves in a triangle. Try modifying the current algorithm by giving each node a slight “push” in a random direction at each step of the process. Does this result in better layouts in any cases?

3. *Add node velocities*. In this particular force-directed algorithm, the amount each node moves on each iteration is completely independent of the amount that the node moved on the previous steps. That is, if a node moves a great distance on one step, it has no “memory” of this and on the next step it will move based solely on the current configuration. In an actual physical system, each node would have an associated velocity which would keep it moving in a particular direction until the net forces slowed it down. Consider modifying the algorithm to incorporate information about the previous iteration's velocities into the current step. One way to do this would be to track the Δx and Δy of each object from one iteration to the next, rather than resetting it to zero each time. What do you notice about the graph layouts?

4. *Add penalties for crossing edges*. One aspect of graph drawings we did not take into account when computing forces is the number of edges that are crossing in a particular drawing. A graph drawing without crossings is more likely to be aesthetically pleasing than a graph drawing with crossings. Modify the algorithm to detect these crossings and adjust the layout accordingly. One common approach for doing this is to pretend that there is an invisible node at the center of each edge that exerts a repulsive force against the center of each other edge, thus pushing edges apart from one another.

5. *Generalize the definition of an attractive force*. In our current algorithm, only nodes directly connected by edges have an attractive force between them. The rationale behind this was that nodes that are connected to one another ought to be close to each other. However, given this same reasoning, it also makes sense to have nodes that are *almost* connected to one another exert a force on each other. For example, if node A is connected to node B and node B is connected to node C, then A and C are likely to be near each other in the final graph layout. Similarly, if C is then connected to D, perhaps A and D should be near each other as well. Update the algorithm so that nodes that are more than one hop away from each other still exert an attractive force. One option might be to have each node exert an attractive force on each other node proportional to how “close” they are to one another in the initial graph.

There are a whole host of extensions you could try out, so don't be limited just by these.

Deliverables

To submit the assignment, please submit on paperless.stanford.edu. Then pat yourself on the back – you've just completed the first assignment of the quarter!

Good luck!