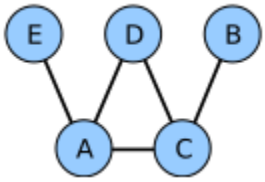
1 Background Information

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, a social network can be represented by people and friendships in 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. Recent work into the smart grid treats the electrical grid as a graph by modeling power stations and substations as nodes and power lines as edges.

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, Celia, Dan, and Edie and suppose that their friendships are as follows:

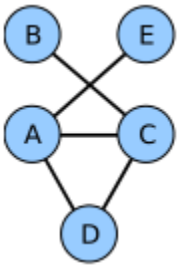
- Alice, Celia, and Dan are all friends of one another.
- Bob and Celia are friends.
- Edie and Alice are friends.

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

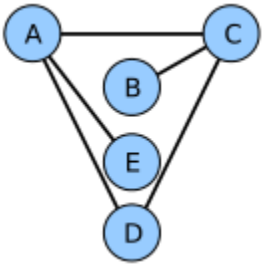


This picture makes it much easier to see who is friends with one another.

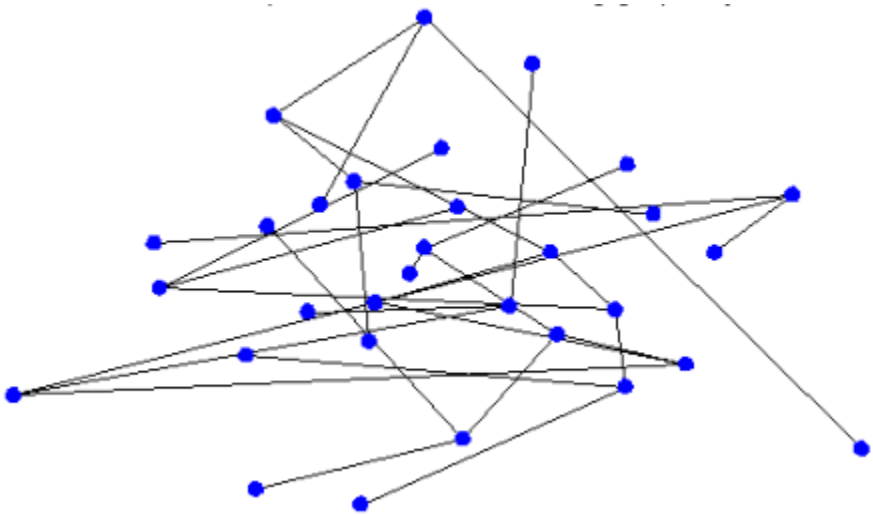
An important detail to notice, though, is that this is not the only possible way that we could have drawn this graph. Below is an entirely different rendering of the graph:



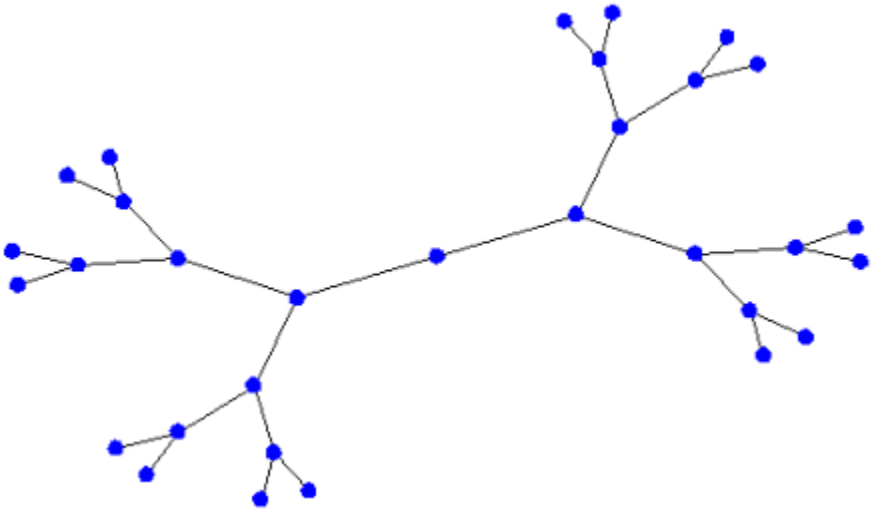
Here is another:



In each case, it is possible to discern the group's friendships, though some of these drawings are easier to interpret than others. 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:



What a mess! Many of the edges cross one another, the nodes are scattered around randomly, and the overall drawing is garbled. However, consider this alternative drawing below:



This rendering 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" 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 sharpen your skills with the streams library and the STL `vector` class. Plus, you'll end up with an incredibly entertaining piece of software.

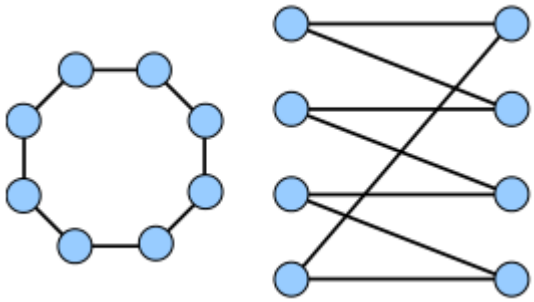
2 Graph Layout Heuristics

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 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, then will design an algorithm to try to pick lay-outs meeting

these criteria. This will not always result in ideal graph drawings, but will do a surprisingly good job laying out many graphs.

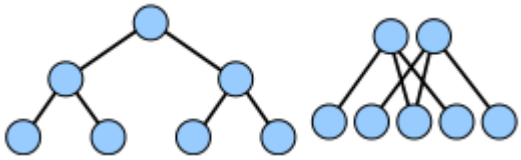
There are many heuristics that could be used to determine how good a graph drawing is. For example, we could try to minimize the number of arcs crossing one another, or try to maximize the symmetry of the drawing. In this assignment, however, we'll limit ourselves to the following two heuristics, which 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. For example, below are two renditions of the same graph, where the graph on the left-hand side has connected nodes placed near each other and the right-hand side has connected nodes spaced far apart:



  As you can see, the left-hand graph is much easier to interpret.

- **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. For example, consider the two drawings below:



  In the left-hand graph, nodes that are not connected are spaced apart from one another, providing insight into the structure of the graph. In the right-hand side, no two horizontally adjacent nodes are connected, and the structure of the graph is much harder to intuit.

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 of with a random guess, then continuously refine the positions of the nodes to improve upon the heuristics. Over time, the nodes will reach a positions that balance the heuristics, and we'll end up with an aesthetically pleasing graph drawing.

The key step in this algorithm is finding some way to update the node positions, and 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

[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

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 speciics 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 the unit circle. In particular, in a graph with n nodes, node k should be placed at

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

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 of 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 of 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, 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 s`, then 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.

[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.

Computing forces. The particular approach we will use to compute forces is based of 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.

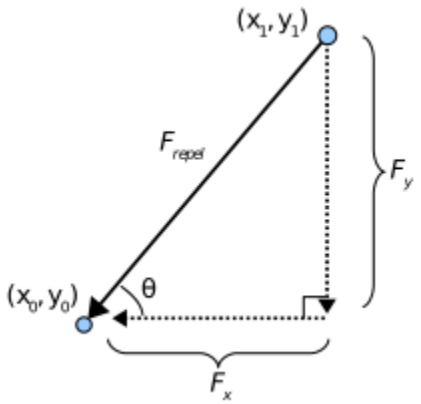
To keep track of the net forces on each node, you'll maintain a Δx and Δy value for each node which stores the net forces on that node along the x and y axes. It's important to track both, since it's possible that a node might be repelled entirely vertically (in which case it will have a strong force 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.

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 between these nodes is

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

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

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

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

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

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

tells us that

$$\theta = \arctan\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$$

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 $F_{\text{repel}} = k_{\text{repel}} / \sqrt{(y_1 - y_0)^2 + (x_1 - x_0)^2}$
  Compute $\theta = \text{atan2}(y_1 - y_0, x_1 - x_0)$
  $\Delta x_0 -= F_{\text{repel}} \cos(\theta)$
  $\Delta y_0 -= F_{\text{repel}} \sin(\theta)$
  $\Delta x_1 += F_{\text{repel}} \cos(\theta)$
  $\Delta y_1 += F_{\text{repel}} \sin(\theta)$

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} I 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 (x₀, y₀)
    Let the second endpoint be (x₁, y₁)
    Compute F_attract = k_attract * ((y₁ - y₀)² + (x₁ - x₀)²)
    Compute θ = atan2(y₁ - y₀, x₁ - x₀)
    Δx₀ += F_attract cos(θ) // Note that this is a += and not a -=!
    Δy₀ += F_attract sin(θ)
    Δx₁ -= F_attract cos(θ) // Note that this is a -= and not a +=!
    Δy₁ -= F_attract sin(θ)
```

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. 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.

3 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:
   (a) Compute the net forces on each node.
   (b) Move each node by the specified amount.
   (c) Display the current state of the graph using the provided library.
   (d) (optional) Ask the user if they want to display another graph and loop accordingly.

Your solution should use the `SimpGraph.h` header file (see calendar) and be submitted by email to ghagopian@collegeofthedeser
You may work alone or in a group of two or three people. If you work in a group, submit an accompanying statement of who did what.