CS 106L Winter 2020

Assignment 1: GraphViz

Assignment by Keith Schwartz, revised by Avery Wang
Due date: Jan. 30, 2020, at 11:59 pm on Paperless.
See update log for minor updates. We'll send an email if there are major updates.
Note: We've significantly shortened the handout. Let us know if you find any mistakes!

Introduction

One of the most useful abstractions in computer science is a graph, a means of expressing relationships between objects. Formally, a graph is a collection of *nodes* (sometimes called *vertices*), and *edges* (sometimes called *arcs*), where each node represents some object, and each edge connects two nodes that are somehow related.

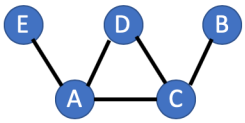
For example, suppose we had a group of students, who were named Alice, Bob, Christine, David, and Eveyln, and their friendships were as follow:

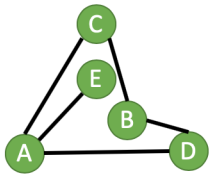
- Alice, Christine, and David, are all friends.
- Bob and Christine are friends.
- Evelyn and Alice are friends.

We can represent this friend group as a graph, where the nodes are the students, and the edges connected pairs of students who were friends. In this example, our graph would be:

```
nodes: {"Alice", "Bob", "Christine", "David", "Evelyn"}
edges: {{"Alice", "Christine"}, {"Alice", "David"}, {"Bob",
"Christine"}, {"David", "Christine"}, {"Evelyn", "Alice"}}
```

It's difficult to understand the friendship structure of this graph by simply listing the nodes and edges. Instead, we visualize the graph by drawing the nodes as points, and edges as lines connecting the nodes. For example, the friendship graph can be visualized in many ways, and three are drawn below. **TODO: Fix the graph, doesn't match the example above.**





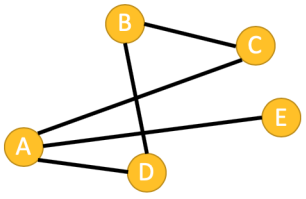


Figure 1 (above): Three possible visualizations for the graph above.
All of these graphs correctly represent the friends between a group of friends. **However, they vary in the locations of the nodes.** Which graph is easiest to interpret? The blue graph clearly shows the local structures within friend groups, especially the central group of friends Alice, Christine, and David. The other two graphs, while still valid visualizations, make the friend group harder to interpret. For example, in the green graph, it looks like Evelyn is closely related to Christine or Bob, even though Evelyn is connected to neither.

In summary, we'll use the following two heuristics to change the locations of the nodes, in order to create a well-visualized graph.

- **Minimize the distances between nodes that are connected.** The idea is that if connected nodes are near each other, then the reader can focus on one part of the graph and see the local structure within that region.
- **Maximize the distances between nodes that are not connected.** The idea is that otherwise, the reader could believe the nodes are somehow related, even if they are not connected.

Solving for an optimal solution with the above constraints is a tough problem. However, we can instead start with a random visualization, and then continually try to improve the visualization. Remarkably, the laws of physics can be applied on these nodes! Each node exerts a force on other nodes, which will push and pull suboptimal nodes until they are in the correct position. More specifically...

- Every edge exerts an attractive force between two connected nodes, pulling the connected nodes together.
- Every node exerts a repulsive force on every other node, pushing the pair of nodes away, whether or not they are connected.

Here is the pseudocode:

```
Assign each node to an initial location around a circle.
Add all the edges within the graph.
While the layout is not yet finished:
      Have each edge exert an attractive force on its endpoints.
      Have each node exert a repulsive force on all other nodes.
      Move the nodes according to the net force
            that node experiences in this iteration.
```

Implementation Details

Here are the steps and details to each part of the assignment.

1. Prompt the user for the name of a file that contains the graph data, and open the file. If unsuccessful, reprompt.

How can you check whether a file was successfully opened? When you declare an ifstream (input file stream, which will be used later to read the file), you pass the filename to the constructor. You can check if a file failed to open correctly by checking the fail bit. If the fail bit is not on, you can assume the file was successfully opened!

2. Prompt the user for the number of microseconds to run the algorithm, which should be a positive integer. If the input is invalid, reprompt.

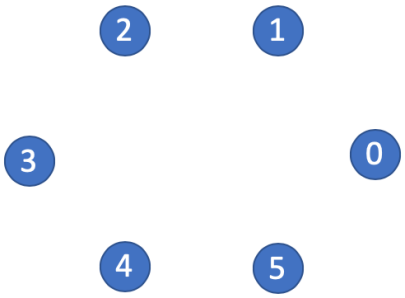
This is a slight twist on the function we wrote in class, getInteger(). Try to not look at the example code there. Try to implement this function yourself!

3. Place all the nodes in a circle.

The first line of the file tells you exactly how many nodes are in the circle. Once you figure out the number of nodes *n*, you can use the unit circle to figure out the positions of each node. The angles are expressed in radians.

```
#include <cmath> // for sin and cos
const double kPi = 3.14159265358979323
for k = 0, 1, …, n-1:
      position = (cos(2*kPi*k/n), sin(2*kPi*k/n))
```

As an example, for the file to the right, after placing the nodes in a circle the graph should look like the following (without the numbers - the numbers represent the k in the equation above). At the end of each step, your SimpleGraph's nodes vector should be filled with n nodes, and indices k = 0, 1, …, n-1 have nodes at the positions stated above.



example file

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

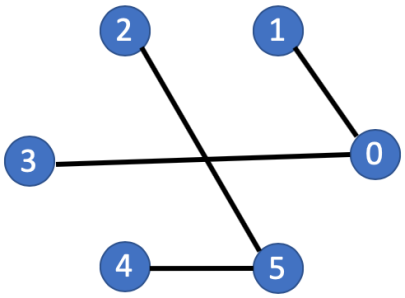
Figure 2: The graph, nodes labeled with indices, immediately after all nodes are placed in a circle. We have only read the first number in the file, 6, the number of nodes.

After you created your graph, the graph will not appear on the canvas until you call the GUI methods, described later in the section "Starter Code Details".

4. Add all the edges to your graph.

The rest of the file gives you pairs of indices that are connected by an edge. Read in the indices two at a time, create an edge with the two indices as endpoints, and add this to the edges vector. For example, after reading the example file, your visualized graph should look like this.

Error-checking whether or not the file is well-formatted is not necessary!



example file

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

Figure 3: The graph, nodes labeled with indices, immediately after all edges have been added to the graph. At this point we have read the entire file.

5. Determine when to continue iterating.

This is the while loop condition. You should keep looping until the elapsed time has exceeded the number of microseconds the user entered. Here is a sample usage of the <chrono> library. Read the documentation for more information!

```
#include <chrono> // fancy timers

// retrieve current times
auto start = std::chrono::high_resolution_clock::now();
auto end = std::chrono::high_resolution_clock::now();

// calculate elapsed time, and convert to milliseconds
auto elapsed =
      std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
int milliseconds = elapsed.count();
```

6. Compute the attractive force between the endpoints of each edge.

Keith's old handout had a ton of explanation of the physics behind the algorithm, which was a bit intimidating. If math terrifies you, here are the equations in code:

```
const double kattract = 0.001;
For each edge:
      Let first endpoint be (x0, y0)
      Let second endpoint be (x1, y1)
      Compute Fattract =
            kattract * ((y1-y0)*(y1-y0) + (x1-x0)*(x1-x0))
      Compute theta = atan2(y1 - y0, x1 - x0)
      delta_x0 += Fattract * cos(theta)
      delta_y0 += Fattract * sin(theta)
      delta_x1 -= Fattract * cos(theta)
      delta_y1 -= Fattract * sin(theta)
```

7. Compute the repulsive force between every pair of nodes.

```
const double krepel = 0.001;
For every pair of nodes (x0, y0) and (x1, y1):
      Compute Frepel =
            krepel / sqrt((y1-y0)*(y1-y0) + (x1-x0)*(x1-x0))
      Compute theta = atan2(y1 - y0, x1 - x0)
      delta_x0 -= Frepel * cos(theta)
      delta_y0 -= Frepel * sin(theta)
      delta_x1 += Frepel * cos(theta)
      delta_y1 += Frepel * sin(theta)
*If you love physics, try to figure out these equations!
```

8. Sum up the forces (delta's) for each node across all calculations in part 7 and 8, and move the positions of that node by the amount of the force.

In each iteration, you need to keep track all of the delta_x's and delta_y's for every node in your graph. At the end of the iteration, you should then adjust the positions (x, y) of each node by delta_x and delta_y respectively.

9. Ask the user if they want to try another file, and loop accordingly.

Don't use recursion here!

Starter Code Details

In the starter code, everything is set up except main.cpp. You will write the entire program, starting from the main function. **Do not modify any other files, such as SimpleGraph.h.** In SimpleGraph.h, the following structures are defined for you.

Structures

In this assignment, we will represent the graph as a struct, defined as follows:

```
struct SimpleGraph {
     std::vector<Node> nodes;
     std::vector<Edge> edges;
};
```

Each node is represented by a Node struct, which simply stores the (x, y) coordinate of each Node. This coordinate is useful when we try to visualize the graph.

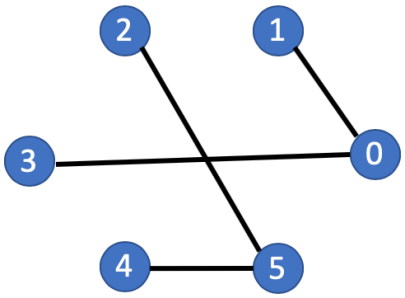
```
struct Node {
     double x, y;
};
```

Finally, an edge identifies the two nodes that it connects. The easiest way to identify a Node is through the index the Node is stored in a vector. Recall that size_t is an unsigned integer representing sizes or indices that must be non-negative. As a result, we define an edge as follows:

```
struct Edge {
      size_t start, end;
};
```

All of our graphs will be undirected, which means that edges are bidirectional. If A is connected to B, then B is connected to A.

For example, if we were to store the following graph in a SimpleGraph struct, we would have the following in our vector:



example file

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

Figure 5: The graph we had before, and how this would be represented in our SimpleGraph struct.

```
SimpleGraph {
      // xi, yi stands for the x or y coordinates of each point
      nodes: { {x0, y0}, {x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}, {x5, y5}}
      edges: { {0, 1}, {5, 2}, {3, 0}, {4, 5} }
}
```

GUI Functions

You do not need to write any code for the graphics. Simply create your graph, and call the following two functions in your code.

Function Name	Description
`void DrawGraph(SimpleGraph& graph)`	Takes in a SimpleGraph, and draws the graph on the canvas. Call once in every iteration of the algorithm, and whenever you want the graph visualizer to update.
`void InitGraphVisualizer(SimpleGraph& graph)`	Sets up the internal state of the graph visualizer. Call once at the beginning of your program.

Advice and Common Mistakes

- Don't hesitate to ask questions! We want to give you practice building a cool program in Standard C++, not to punish you for not knowing a particular library or language feature. Ask on Piazza!
- Create one single SimpleGraph, and use it throughout your program. Do not copy the SimpleGraph, as that may cause bugs in the graphical canvas.
- Be careful about reading files. As mentioned in lecture, eof and good bits are difficult to use and can cause off-by-one errors. If you are seeing an extra garbage edge, there is likely a bug in your loop.
- Be careful when receiving input from the user. As mentioned in lecture, reading directly using >> from cin is a nightmare. See lecture examples for techniques to get around that.
- Label the points exactly as (x0, y0) and (x1, y1). Being clever with names will only confuse you. Don't use (x2, y2) or (x', y').
- There is no square root when calculating Fattract, but there is a square root when calculating Frepel.
- When calculating atan2, make sure you are using y1 - y0 not y0 - y1.
- Do not confuse your +='s vs. -='s. Make sure they match up with the x0's vs. x1's exactly. Yes, they are flipped if you compare attractive with repulsive.
- Use C++ constructs. Do not use: C-strings/functions, functions dealing with memory, #define, printf, etc. Only use material covered in the first two weeks of CS 106B/L (except Stanford libraries, of course!).
- Do not modify any files other than main.cpp. In particular, do not add your own structs inside SimpleGraph.h, and do not change the definitions for SimpleGraph, Node, Edge, or anything else provided in the starter code.

Grading

Here are the minimum functionality requirements to receive credit on the assignment.

- Program must compile (this means you only modified main.cpp).
- Program must not use any Stanford Library functions.
- User must be able to enter a filename and a time. After the time has elapsed, the user must be able to type in "yes" or "no", and the program should reprompt accordingly.
- A reasonable attempt at error-checking and re-prompting for all user input.
- On the input files 'cube' given a large enough time, the program should terminate with the visualizer displaying a cube (orientation isn't important).

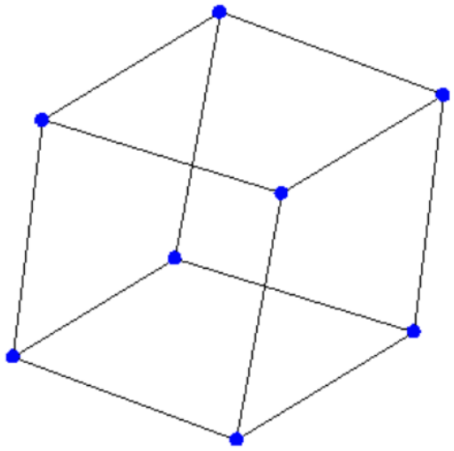


Figure 4: This is what a cube looks like.

- On the input file'10grid', correct inputs when run on various time durations. If a very small time is used, the visualization should look close to a circle. If a medium time is used the visualizer should terminate even if the graph has not converged. If a large time is used, the graph should look like a cube, as shown below. We are purposely vague about what "small", "medium", and "large" are, and we're flexible as long as your program reasonably works correctly.

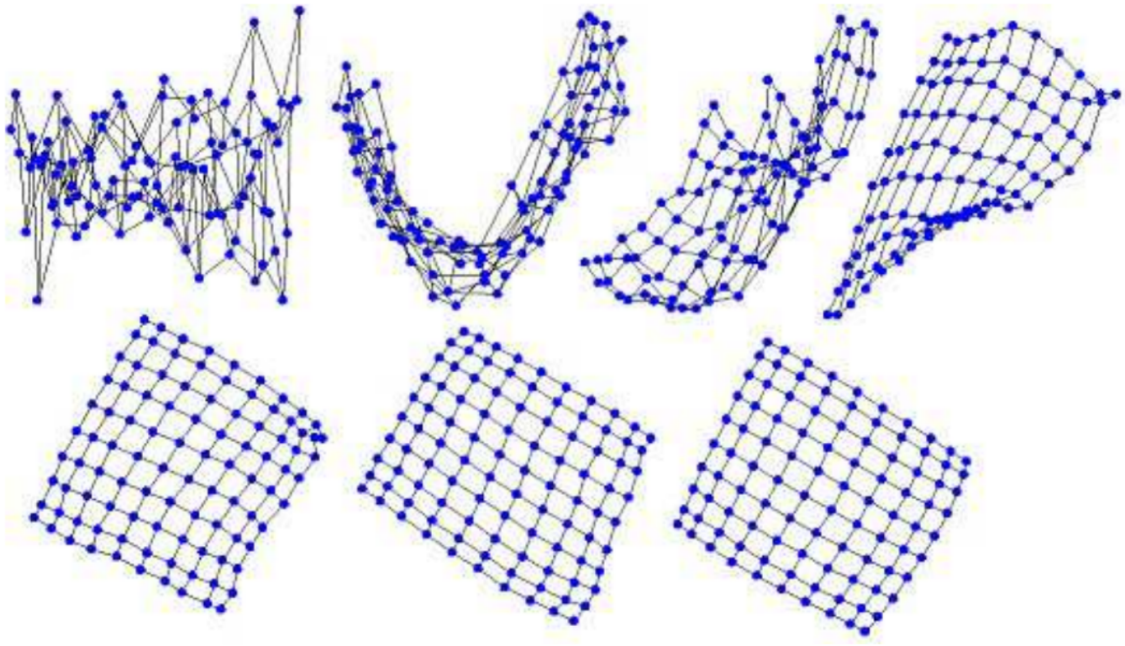


Figure 5: Sample runs with the '10grid' file. The program should converge at the graph in the bottom right.

Style Guidelines

We will offer feedback on style, though we won't include style as part of your grade. Here are some things to keep in mind.

- We said this once above, and we'll say it again. Stick with Standard C++! Particularly if you've coded in C, you may be tempted to use C constructs such as C-strings, arrays, macros (#defines), and memory manipulations. Resist that temptation, and code with the powerful language that is C++. You will find the code much easier to write in C++.
- Think about when a pair, tuple, struct, and vector are best used. For example, if you wanted to store a coordinates (x, y), a vector of size two should not be used, since vectors are meant to be used as dynamically sized containers.
- Everything from CS 106B still matter! Decomposition, avoiding redundancy, naming your variable and functions well - these are all important!
- Avoid using anything past lecture 4 (vectors). This assignment is quite simple, and we don't want you diving into iterators and algorithms until you've mastered the material from the first four lectures.

Extensions

Here are a few possible extensions you may want to experiment with.

1. **Add node velocities.** In our current algorithm, the amount each node moves is independent in each iteration. If we really implemented this as a force-directed algorithm, the forces should cause the nodes to speed up or slow down over multiple iterations.

2. **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. 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.

3. **Add penalties for low resolution.** The resolution of a graph drawing is the smallest angle between any two edges incident to a single node. Graphs that have low resolution can be hard to understand, since the edges emanating from a source node will all be bunched together. One way to do this might be to add a repulsive force between the endpoints of arcs that have a small angle between them.

Submission Instructions

Visit Paperless, click on CS 106L, and submit main.cpp. You should not have modified any other files. If you submit extensions, please submit them in the same submission named as main-extra-1.cpp, main-extra-2.cpp, etc. Let us know if you have trouble submitting.

Good luck with the assignment! Don't hesitate to ask on Piazza if you have questions.

Update Log

Every time there is an update to this document, I will log it below, so you can easily see if any changes were made.
Jan. 23, 2:13 AM - added update log, added note that krepel = kattract = 0.001
Feb. 2, 12:54 PM - changed microseconds to milliseconds in the chrono class. We will accept both, although you'll find yourself typing less 0's if you use milliseconds.