CS51 PROBLEM SET 7:
FORCE-DIRECTED GRAPH DRAWING

STUART M. SHIEBER

This problem set is to be done individually. Please review the collaboration policy at the course web site. You can obtain the code at tiny.cc/cs51ps7.

1. Introduction

You'll be familiar with graph drawings, those renderings of nodes and edges between them that depict all kinds of networks – both physical and virtual. These drawings are ubiquitous, in large part because of their fabulous utility. Examples date from as early as the Middle Ages (see Figure 1a), when they were used to depict family trees and categorizations of vices and virtues. These days, they are used to depict everything from molecular interactions to social networks.

To gain the best benefit from visualizing graphs through a graph drawing, the nodes and edges must be laid out well. In this problem set, you'll complete the implementation of a system for *force-directed graph layout*. A modern example of what can be done with force-directed graph drawing is provided in Figure 1b. If you'd like to get a sense of what can be done with force-directed graph drawing, you can play around with the graph visualization from which this snapshot came. In carrying out this project, you'll be making use of the object-oriented programming paradigm supported by OCaml.

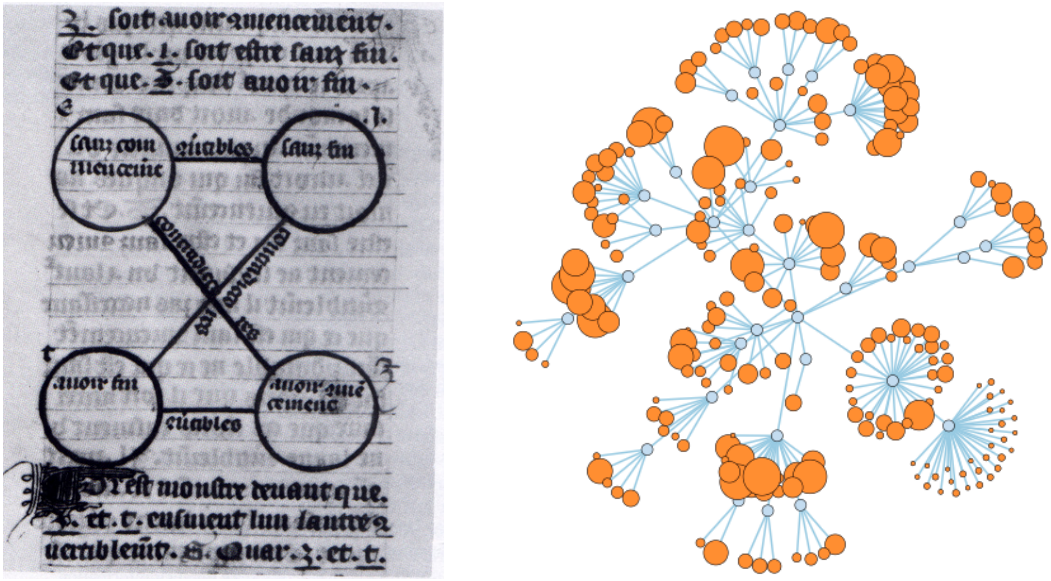
A note of assuagement: Although this problem set document uses a lot of physics terminology, you really don't need to know any physics whatsoever to do the problem set. All of the physics-related code is in portions of the code-base (`graphdraw.ml` and `controls.ml`) that we have provided for you and that you won't need to modify.

2. Background

A GRAPH is a mathematical object defined as a set of NODES and EDGES connecting the nodes. As an example, consider a set of four nodes (numbered 0 to 3) connected with edges cyclically, 0 to 1, 1 to 2, 2 to 3, and 3 to 0, plus an extra edge from 0 to 2. A GRAPH DRAWING is a depiction of a graph in two (or sometimes three) dimensions indicating the nodes in the graph by graphical symbols of various sorts (circles, squares, and the like) and edges with lines drawn between the nodes. Other aspects of the graph are also typically manifested in graphical properties. For instance, groups of nodes might be aligned horizontally or vertically, or grouped with a zone box surrounding them, or laid out symmetrically or in a hub and spoke motif.

GRAPH
NODES
EDGES
GRAPH DRAWING

Date: April 4, 2018.



(a) (b)

Figure 1. Two sample graph drawings several hundred years apart. (1a) A graph drawing from the 14th century with nodes depicting logical propositions in an argument and edges depicting relations among them. From Kruja et al. (2001). (1b) Snapshot of a dynamic interactive force-directed graph drawing built using D3 (https://mbostock.github.io/d3/talk/20111116/force-collapsible.html), from the D3 gallery.

For the example four-node graph just presented, if we depict the nodes as small circles, placed more or less randomly on a drawing "canvas", we might get a graph drawing like Figure 2a. It's not particularly visually pleasing.

The force-directed graph drawing method allows the generation of much more attractive layouts by thinking of the positions at which the nodes are to be placed as physical MASSES subject to various kinds of FORCES. The forces encourage the satisfying of graphical constraints, such as nodes being a particular distance from each other, or far away from each other, or horizontally or vertically aligned. For instance, if we imagine a spring with a certain REST LENGTH connecting two masses, those masses will have forces pushing them towards each other if they are farther apart than the rest length or away from each other if they are closer together than the rest length. (See Figure 3 for a visual depiction.) According to Hooke's law, the force applied is directly proportional to the difference between the current distance and the rest length.

We can use this kind of mass-spring physical system to help with graph layout. We imagine that there is a mass for each node initially placed at the locations shown in Figure 2a, and for each edge in the graph there is a Hooke's law spring of a given rest length, 80 pixels, say, connecting the masses representing the nodes at the end of the edge. We refer to a force-generating element like the Hooke's law spring as a CONTROL. If we physically simulate how the forces on the masses generated by the controls would work, eventually the masses will come to rest at locations

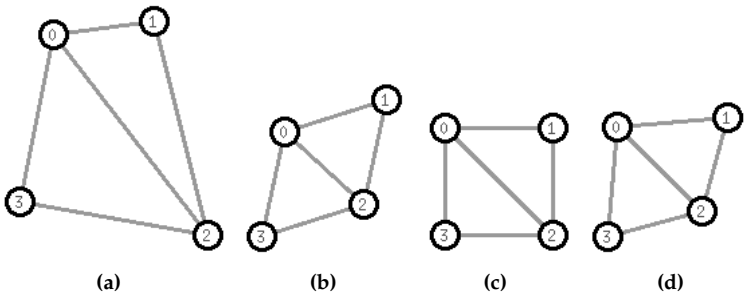


Figure 2. Four different drawings of the same graph. (a) Nodes randomly placed. (b) With fixed length spring constraints between nodes connected by edges. (c) With fixed length spring constraints between nodes connected by outside edges, plus a horizontal alignment constraint on nodes 0 and 1 and a vertical alignment constraint on nodes 0 and 3. (d) An overconstrained layout with the constraints from (c) but with all of the edge constraints from (b), including the fixed length constraint between 0 and 2.

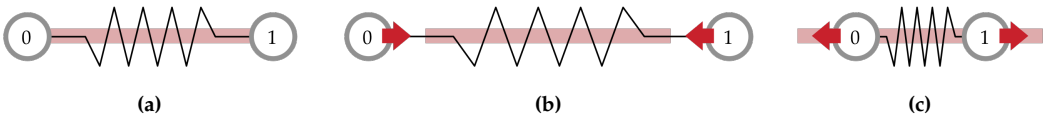


Figure 3. A Hooke's law spring connecting two masses (labeled 0 and 1) and its generated forces. The pale red bar indicates the spring's rest length. (a) The spring at rest. No forces on the masses. (b) When the spring is stretched (the masses are farther apart than the spring's rest length), forces (red arrows) are applied to the two masses pushing them towards each other. (c) Conversely, when the spring is compressed (the masses are closer together than the spring's rest length), forces are applied to the two masses pushing them away from each other.

different from where they started, and indeed, if we place the graph nodes at those locations, we get exactly the layout in Figure 2b. Notice how all of the edge-connected nodes are the same length apart from each other – as it turns out, 80 pixels apart.

This methodology for graph layout is called "force-directed graph layout" based on its use of simulated forces to move the nodes and edges around. The method can be generalized to much more expressive graphical constraints than just establishing fixed distances between nodes with Hooke's-law springs. For instance, we can have force-generating controls that push masses to be in horizontal alignment, or vertical alignment. Using these controls, we can generate layouts like the one in Figure 2c. Care must be taken however. If we add too many controls in ways that overconstrain the physical system, the result of finding the resting positions may not fully satisfy any of the constraints, leading to unattractive layouts as in Figure 2d.

3. Building a force-directed graph layout system

We've provided you with most of the components of a force-directed graph layout system, written in an object-oriented design that is particularly appropriate for this task. We start with

POINTS

two-dimensional POINTS. A point has x and y coordinates, and can thus represent a position on the canvas. The signature of a `point` class is specified in the file `points.mli`. The interface file documents the functionality of objects in the class, and we've provided a partial implementation of the class in the file `points.ml`. You'll notice that in addition to retrieving the position of a point, a point can be moved directly to a new position.

Further, operations can be performed on points interpreted as vectors, for instance, adding two points, or multiplying a point by a scaling factor. We describe the various point operations by example. For instance, the sum of points at $(1, 3)$ and $(7, 6)$ would be $(8, 9)$. Scaling the result by 2 would yield $(16, 18)$. Subtracting $(10, 10)$ yields $(6, 8)$. The norm of that point (its distance from the origin) is

$$\sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10 \quad .$$

The unit vector that corresponds to the vector $(6, 8)$ is the vector in the same direction but whose norm is 1, which we can generate by just dividing the vector by its norm: $(0.6, 0.8)$. The distance between two points is the norm of their difference, so the distance between $(5, 5)$ and $(5.6, 5.8)$ is 1.

MASSES

MASSES are like points, except that they have a physical mass and forces can act upon them. Look at the file `masses.mli` for information about the `mass` class, a subclass of `point`. Again, the interface file documents the functionality of objects in the class, and we've provided a partial implementation of the class.

CONTROLS

CONTROLS are force-generating objects – like springs or alignment constraints. We've provided the code for those in `controls.ml`. There is a `control` class along with subclasses for different types of controls: springs, alignment constraints, repulsive forces, and the like. Note how the controls affect masses, so control objects typically have one or more masses as elements. For instance, a spring control (see the `bispring` class) will have two masses as elements, the masses that the spring connects.

GRAPHICAL OBJECTS

GRAPHICAL OBJECTS are the kinds of things that get rendered on the graphics canvas – small circles, squares, or other shapes representing nodes, edges drawn as connecting lines, boxes representing a zone that includes several other graphical objects. These graphical objects have various properties governing how they appear: what color they are drawn in, what line thickness should be used, and so forth. You'll notice that some of these are common to all graphical objects and appear in the `drawable` class. Others are particular to a subclass of `drawable`, like the `radius` property of the `circle` class.

These classes make heavy use of OCaml's ability to have named arguments that are optional, taking a default value if the argument is not provided. You may want to look at the discussion of labeled and optional arguments in *Real World OCaml* to learn about the syntax used.

Finally, the file `graphdraw.ml` implements functionality to "solve" a graph layout problem. The `solve` function is provided a list of masses; a list of constraints generating forces on those masses; and a *scene*, a list of drawable graphical items located relative to those masses. It uses a

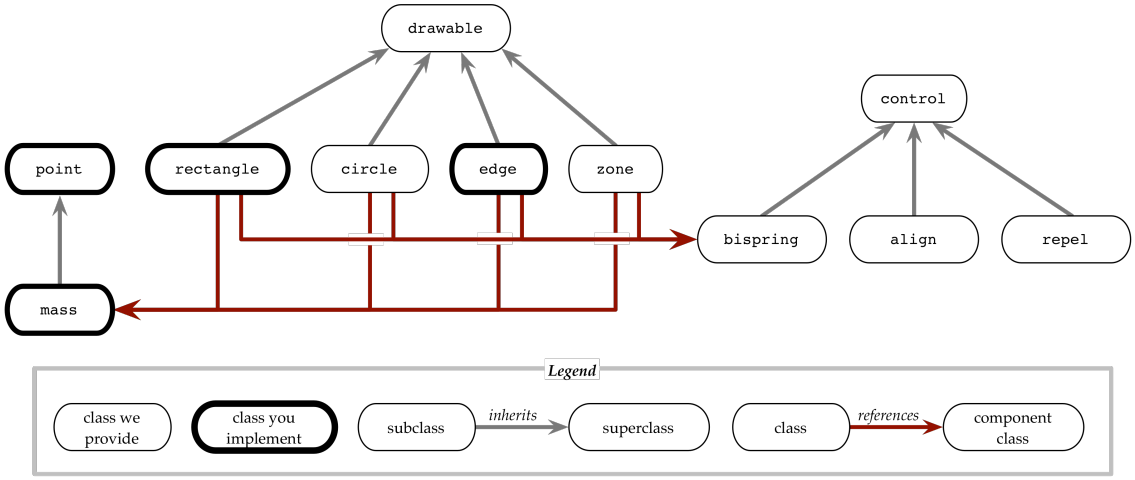


Figure 4. A map of the various classes involved in this problem set.

renderer (which we also provide) to display the scene in OCaml's graphics window (using the X11 windowing system that you installed at the start of term[1]), and animates the scene, moving the graphical objects around the canvas as the forces apply to them, until the scene settles into a final state. To get a sense of what the system looks like as it performs the physical simulation of masses and forces, you can view the demo movies at http://tiny.cc/cs51demo or http://tiny.cc/cs51demo-ext.

Figure 4 depicts the interrelationships among the various classes in the problem set (as a graph drawing!).

4. Completing the graph drawing system

The following files make up the system:

- `points.ml` and `points.mli`: The `point` class.
- `masses.ml` and `masses.mli`: The `mass` class.
- `controls.ml`: The various controls classes, for instance, `bispring`, `align`, `logisticrepel`.
- `graphobj.ml`: Graphical object classes, including `drawable` and its subclasses. We provide the `circle` subclass. You'll add classes for `rectangle`, `square`, `edge`, and `zone`.
- `graphdraw.ml`: Code for carrying out the physical simulation and for rendering graphs in the graphics window. All of the physics happens here, so you won't need to deal with that part.
- `testXXX.ml`: Files that provide tests that you can run to see the system in action. You may want to add your own tests.

The parts that you'll need to do are as follows:

(1) Implement the `point` class in `points.ml` consistent with the interface in `points.mli`. Test it thoroughly.

[1] If you followed our Windows 10 instructions, you'll need to make sure that XMing is running in your Windows environment.

(2) Complete the implementation of the `mass` class in `masses.ml` consistent with the interface in `masses.mli`. Test it thoroughly.
(3) You should already be able to test the system on examples that only use the circle graphical objects that we've provided. For instance, the example in `testuniformcentered.ml` should already work. Try building and running it and verify that the graphics window launches and you see the animated layout process.
(4) Add the additional graphical objects to `graphobj.ml` – rectangle and square nodes, edges, and zone boxes – and try the other tests we provided.
(5) Construct an example graph drawing of your own, placing the code in a file `example.ml`. The file should culminate in a function `example : unit -> unit`, which when called uses the system to lay out the graph nicely and display the result. We've provided a file `example.mli` that it should be consistent with. If you'd like, post a screenshot or video to Piazza. We'll award a prize for the best graph posted.

5. Submission

Before submitting, please estimate how much time you spent on the problem set by editing the lines in `points.ml`, `masses.ml`, and `graphobj.ml` that look like

```
let minutes_spent_on_part () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) each part of the problem set took you to complete. Make sure it still compiles.

Then, to submit the problem set, follow the instructions found here. Please note that only one of your partners needs to submit the problem set.

References

Eriola Kruja, Joe Marks, Ann Blair, and Richard Waters. A short note on the history of graph drawing. In *International Symposium on Graph Drawing*, pages 272–286. Springer, 2001.