Implementing a Spectral Graph Layout in Gephi

Henry Fender

May 8, 2017

Introduction

When working on graph visualizations, the first technique network scientist employ is a graph layout. As the first impression the scientist has, the graph layout is vital to the next few decisions the scientist makes. There are a whole host of graph visualization algorithms that display graphs in unique ways.

In this work I briefly describe spectral variant of graph layout algorithms. Spectral graph layouts use the eigenvectors of graph related matrices, such as the Adjacency Matrix or Laplacian Matrix. Beyond that guiding principle, there are many ways to go about implementing an actual layout algorithm.

I implement a degree normalized Spectral Layout algorithm for use in the popular network analysis software Gephi. The software currently provides particle system based layouts, which, while easy to start and stop, reach equilibrium states after longer amounts of time. As I will mention, Spectral Layouts have optimal solutions and predictable complexity. For these reasons, it is my view that this layout should be made available. This project attempts to accomplish that.

For students interested in implementing their own layout algorithm into Gephi, this paper can be used as a preliminary guide. Before advancing further, one should be comfortable with linear algebra, graph theory, the java programming language, and object oriented design patterns.

Background

In addition to the subjects I mentioned in the introduction, some more specific knowledge must be presented before covering my implementation. This section covers notational conventions we employ in this paper as well as a brief summary of the literature surrounding spectral layouts.

Notation

Below in Table 1 we present notation that is used in this paper.

Symbol	Description	Symbol	Description
D	The degree matrix of a graph	u^k	The kth eigenvector of a matrix
A	The adjacency matrix of a graph	$\deg(i)$	The degree of ith vertex
L	The laplacian matrix of a graph	$N(i)$	The set of vertices directly connected to ith vertex, i.e. its neighborhood
(L, D)	The generalized eigenpair satisfying the equation $Lx = \mu Dx$	$v(i)$	The ith member of a vector

Table 1: Notation

Spectral Layout Literature

Drawing Graphs by Eigenvectors: Theory and Practice

We begin with a seminal paper by Y. Koren [1]. He provides a very detailed overview of graph drawing with Eigenvectors. It provides three different perspectives and develops a visual intuition for the subject that was previously obscured.

The paper is motivated by the drive to create aesthetically pleasing graph layouts. Spectral layouts are nice to look at, but aren't terribly intuitive. But there are distinct advantages to using Spectral layouts. They can be computed very quickly and have an exact solution.

He provides some basic notions for how to compute a spectral layout. He establishes it as the solution to an optimization problem on a symmetric graph related matrix. The Laplacian Matrix L of an associated graph is such a matrix. Say...

$$u^1, u^2, u^3 \dots u^n$$

are the eigenvectors of L. Sometimes, we can directly use its eigenvectors to determine a layout. In that case, the location of the ith node in the spectral layout is $(u^2(i), u^3(i))$ (note: these are the low eigenvectors of L). This is called the *eigenprojection method.*

This method can be framed traditionally as a *force-directed* approach to the layout problem. This type of strategy seeks to minimize distance between nodes based on the weights of edges between them. Koren claims that this approach obscures the intuition behind the layout and presents another related method.

He suggests using the generalized eigenpair (L, D) instead of the Laplacian directly. This degree-normalizes the eigenvectors of the Laplacian. After some manipulating of the eigenpair we can view this in a rather illuminating way. We find that we are calculating the layout based on the eigenvectors of the transition matrix $D^{-1}A$. Since the transition matrix captures the relative importance of the neighbors of a node, we can think of the graph as determined that way instead of by energy.

Koren details an algorithm to solve for the degree-normalized eigenvectors. The basic idea is to use power iteration on a seed vector found by computing the eigenvector of a subgraph. I found this algorithm to be especially appealing because of the intuitive basis Koren built around it. As such, I chose it over all other algorithms to implement a spectral layout in Gephi.

Koren also covers another popular method in the Spectral Layout literature called High-Dimensional Embedding. This idea still uses eigenprojection, but it puts constraints on how. Specifically, the technique eigenprojects into a subspace of the original space the graph lay in.

This paper provides an excellent starting place for multiple research projects. It can be used as an entry point into graph layout problems, but it could also be used to inspire an algorithm written in a number of popular computer languages.

The paper seems to be written contrary to the direction that layout research was heading. For example, years later, in 2017, a popular starting tool for graph rendering, Gephi, contains no spectral layout, but a plethora of force-directed layouts. This indicates that the area does not have as many contributions as force-directed methods.

I speculate that this is because Spectral methods suffer from overlap issues and some clustering issues for unimportant nodes. Force-directed methods can easily be tweaked to overcome this, but the precise solution of an eigenprojection cannot.

The paper is a tour-de-force in the subject. When I approached the subject of spectral graph layouts, I had the scantest idea of what I was embarking on. This paper answered all my questions and then some. It is practically self-contained. It leaves unstated only the most elementary results.

Eigensolver Methods for Progressive Multidimensional Scaling of Large Data

Next we cover another spectral layout variant called Multidimensional Scaling presented in [2] The paper presents an eigensolver method with the goal of computing a spectral layout very quickly.

The (MDS) family of techniques is used for high-dimensional data reduction. This is exactly the kind of problem graph drawing is often reduced to. There are two recognized methods called classical scaling and distance scaling. Classical scaling has quadratic computation time and as a result is not used much. A sampling-based adaptation of it can overcome its time complexity.

The authors overview the history of traditional implementations of classical MDS. It details its complexity difficulties and provides some references for general knowledge of the field. They also mention that MDS was the first layout used to plot social networks. This makes sense because neighbor importance/relevance, an important characteristic in social networks, is captured by spectral layouts.

This paper provides an excellent starting place to understanding Spectral Layouts. It does not provide an extremely detailed view of the process, but it provides different perspectives that can be implemented and compared. The study seems easily replicated because they do not use specialized libraries.

This paper is a nice complement to the Y. Koren paper. It covers another method for the spectral layout of a graph besides High Dimensional Embedding. It is also easier to read than Y. Koren. They state elementary results that facilitate the reading of both. With this and Y. Koren, I have a good idea of what I can implement in terms of spectral graph layouts.

SDE: Graph Drawing Using Spectral Distance Embedding

Lastly, we cover a brief paper reinforcing ideas we learned from Koren [3]. This paper is a brief presentation on a graph drawing algorithm using spectral decomposition of graph related matrices. It explains its theory and its performance.

The authors begin with a statement of what a graph layout problem is, which in their words is presenting the graph in a visually intuitive way.

The paper explains spectral graph drawing, which uses the eigenvectors of a graph related matrix (the Adjacency, the Laplacian, or a matrix containing graph theoretic distances, etc.) to draw the graph. Spectral Distance Embedding (SDE) is a particular algorithm to accomplish that task.

The algorithm has two stages:

1. Compute the shortest path lengths for every pair of nodes in the graph, for which Breadth First Search is used. These are then placed in the symmetric matrix L.

2. Then power iteration is used to calculate the eigenvectors and eigenvalues of $M = -\frac{1}{2}\gamma L \gamma$, where $\gamma = I_n - \frac{1}{n} 1_n 1_n^T$.

If D is the matrix of distances, then the entries of L are computed as $L_{ij} = D_{ij}^2$.

The algorithm is reasonably fast for graphs up to $20,000$ nodes. The algorithm computes an exact answer and is quicker than analogous force-directed methods.

This paper is an abbreviated version of a technical report. It is not very detailed, but provides a good overview in lieu of the more technical descriptions of the literature.

I like this paper for what it is. After reading two dense longer papers my head hurt. This paper was a godsend because it spoke about the subject in less technical terms. It functioned as a sort of heuristic in terms of getting through the parts of other papers that I didn't understand as well. I have not read the full technical report yet because I figured that the other two papers I read were a sufficient survey.

Implementation

My goal was to implement the degree normalized version of the constrained optimization problems mentioned by Y. Koren. This involved a translation of an algorithm written by Koren into the source language of Gephi, Java. What follows is the specification and justification of that process. It is my hope that this section in particular can serve as a guide to other students who want to implement layout algorithms into Gephi.

Brief Outline of Layout Design

Graph layouts adhere to a general pattern. We can list it as follows.

1. Takes in a graph as a parameter.
2. Based on a specific property of that graph form a spatial relationship between the nodes.
3. Render that structure visually.

The first point may seem trivial, but in some cases can determine whether the next two steps are feasible. For example, if a layout exploits the connectedness of a graph and a given graph is disconnected, then the resulting layout will not work. Spectral layouts suffer from this exact issue.

The second point determines the types of intuitions a viewer makes about the graph. The properties of a graph reflect different types of importance. Spectral graph layouts stress the importance of a node among its neighbors, a fact I will continue to emphasize. In general, it is important to specialize a layout. If a layout uses more than one property to structure the graph, it can be difficult to understand what nodes are important in the resulting configuration.

The third point relies entirely on computer programming. As such, the designer should choose a language that suits the mathematics behind the layout. Gephi hosts many particle system layouts, which require real time animation. Because of its extensive applications in graphics, Java is a logical choice for these layouts. Spectral layouts do not require such a focus on animation, but do require linear algebra computations. Here, Java is not as useful. I would rather use a scientific programming language like MATLAB or Python to perform the spectral layout. However, for reasons I cover later, I decided to implement the layout in Java anyway.

Algorithm

I present Koren's algorithm [1] for calculating the p top eigenvectors of the transition matrix $D^{-1}A$ below. It is a modified version of power iteration, which is a technique to compute the eigenvectors of a matrix. The general idea is to repeatedly normalize and apply a matrix to a random vector until it transforms into an eigenvector. This process warrants its own examination, but we will not cover it here. For the interested reader, see page 319 of [4].

Algorithm 1 Spectral Drawing

```
1:  procedure
2:      const $\varepsilon \leftarrow 10^{-8}$
3:      for $k = 2$ to $p$ do
4:          $\hat{u}^k \leftarrow$ random
5:          $\hat{u}^k \leftarrow \frac{\hat{u}^k}{||\hat{u}^k||}$
6:          do
7:              $u^k \leftarrow \hat{u}^k$
8:              for $l = 1$ to $k - 1$ do
9:                  $u^k \leftarrow u^k - \frac{(u^k)^T D u^l}{(u^l)^T D u^l} u^l$
10:             for $l = 1$ to $k - 1$ do
11:                 $\hat{u}^k \leftarrow \frac{1}{2}(u^k(i) + \frac{\sum_{j \in N(i) w_{ij} u^k(j)}}{\deg(i)})$
12:             $u^k \leftarrow \hat{u}^k$
13:         while $\hat{u}^k u^k < 1 - \varepsilon$
14:         $u^k \leftarrow \hat{u}^k$
        return $u^2, \ldots, u^p$
```

The bulk of the logic is uninteresting. It serves to maintain the process; each run through the main for loop corresponds to the creation of one eigenvector. As such there are many assignments and reassignments as the vector converges to the eigenvector. The do while loop monitors that convergence and aborts the procedure when the vector begins to change by marginal amounts. As the algorithm is closely related to power iteration, we should not be surprised.

There are two parts that warrant a detailed consideration, which are the aforementioned modifications. The *for loop* comprising lines 8 and 9 D-orthogonalizes the vector. Essentially, it makes it linearly independent from all the previously computed eigenvectors. This way all the computed eigenvectors will form an orthogonal basis, which is an important feature because we wanted the resulting layout to have as little redundant information as possible.

The *for loop* comprising lines 10 and 11, left-multiplies the random vector by $I + D^{-1}A$. Koren has chosen to represent this operation through summation notation, which presents a different intuition. Instead of transforming the vector into an eigenvector, we are balancing each node at the center of its neighborhood. The observation that these are the same operation is at the core of a spectral layout. Spatial relationships between nodes reflect their relative importance to one another. I recommend manually applying this step of the algorithm to some toy examples, it is the best way to observe that fact.

EJML

The reader will have noticed that the algorithm makes extensive use of linear algebra. Java is not a numerical computing language and thus has no linear algebra functionality built in. I had to employ a specialized library called *Efficient Java Matrix Library* (EJML) to fill the void. From there I translated the algorithm from the previous subsection directly into Java. To avoid repetition, I do not include any excerpt of our code

Alternatively, I could have done the linear algebra calculations in a more suitable computer language, but then the algorithm would have lost time to a transfer between the languages. EJML is a pretty fast implementation of linear algebra computations, so it made sense to keep everything in the domain of Java.

The major downside to the library is its readability. Outside of set theory, mathematical syntax does not carry over into Java. Many awkward conventions show up in the library syntax. Even if they don't compute eigenvectors, most layout algorithms employ linear algebra to some degree. For Programmers who are not experienced in Java I recommend a more "matrix-friendly" environment is used.

Gephi Toolkit

I wanted to implement the spectral layout in Gephi because I used the software frequently in my schooling career. Fortunately, Gephi has an API and SPI library called Gephi toolkit that allows a programmer to manipulate projects from a java environment. Even better, only a small part of the library is needed to successfully program a layout.

To use the library, a programmer writes a script similar to the one displayed in this file: `https://github.com/gephi/gephi-toolkit-demos/blob/master/src/main/java/org/gephi/toolkit/demos/HeadlessSimple.java`. Each line corresponds to an action a user could take in the Gephi GUI, with addition of custom actions that aren't traditionally available.

A layout script with Gephi toolkit follows this pattern:

1. Initialize a project and a project controller object.
2. Create the graph model object (from existing data or by direct construction).
3. Create the layout procedure object and use it to layout the graph model object.
4. (Optional) Export the graph model in a new file.

GitHub Repository

The next section explains my Java implementation of the spectral layout. I recommend looking at my source code while reading it. The code is available at: `https://github.com/hjfender/GephiSpectralLayout`.

Implementation Explained

There are five classes in my implementation. The remaining parts of the project structure involve graph layout examples. We say no more here, for the interested reader we have include an appendix explaining it.

1. `RunGraphLayout.java`

   This class contains the main method of the project. A user can use it to layout different graph files. See the appendix for more implementation.

2. `GephiScript.java`

   This is a specific implementation of the Gephi toolkit script pattern we described on the previous page.

3. `SpectralLayout.java`

   This is a specific implementation of GephiToolkit's `AbstractLayout.java` class. There are several methods that are required for implementation, but are irrelevant to the work. I have tried to denote them with comments in the code.

   As I mentioned, traditional Gephi layouts adhere to a particle system approach. These layouts are animated and iterative. As such they have a "step" method called `goAlgo()` which runs in each iteration.

   Spectral layouts are not iterative and compute the layout in one run. So we implement the `goAlgo()` method with the expectation that it will be run only once. It employs Y. Koren's algorithm from page 6 to find the "eigenposition" at which each node should be placed (recall positions are determined by the top eigenvectors of the transition matrix). Since the eigenvectors are normalized these placements are incredibly close together. I created a uniform scaling factor called `display_distance` to spread the nodes apart.

   In order to compute the eigenvectors of the transition matrix of the graph I have to construct it. I use the property `graphID` provided by the graph model. It usually is a unique integer from 1 to n, where n is the number of nodes in the graph. These integers can be used to determine a unique position in a Java array, which can be used to store the degree of a node. This array can then be used to create the degree matrix D in EJML. This same property applies to edges and is used to make the adjacency matrix A. Of course, after we have these two matrices we can create the transition matrix $D^{-1}A$.

4. `SpectralBuilder.java`

   This is a helper class for `SpectralLayout.java` required by the Gephi toolkit. It contains no pertinence to my work (in fact I do not even use it), but it must exist in order for the project to compile.

5. `ComputeLayout.java`

   This is another helper class for `SpectralLayout.java`. It is not required by the library, but is of essential importance to my work. It is where I implement Y. Koren's algorithm in java. Since EJML makes linear algebra awkward, the code is somewhat lengthy. Because of this and the prior emphasis I placed on the algorithm as the core of the intuition of the layout, it made sense to give it its own class.

Successes and Shortcomings

I managed to successfully layout some graphs, which I display in the next section. Unfortunately, not all graphs could be drawn in my implementation. Spectral layouts cannot render disconnected graphs because their degree matrix is not invertible. I then had to be careful about what graphs I tried to render.

There are also some bugs graph import bugs that should be handled. Right now the program can only handle `.gml` and `.graphml` files because they ID the nodes with distinct sequential integers, which I use to construct the degree matrix. This should happen with other graph files as well, but, in the case of `.csv` files , Gephi toolkit imports the column headers as a node. This crashes the program because it does not know how to convert a natural language word into an integer.

Additionally, `.gml` and `.graphml` sometimes index their nodes differently. For example, `karate.gml` indexes starting at 1, so when I construct the transition matrix I subtract 1. On the other hand `us-airlines.graphml` indexes starting at 1, so I have to change the rule to **not** subtract 1 before I run the program. This is inconvenient and a more general approach to constructing the transition matrix is desirable.

Results

In this section, we present the initial results of using our implementation. We also explain in greater detail a limitation of the spectral layout.

Examples

We now present some graph drawings. We consider two pedagogical examples: Zachary's Karate Club from [5] and a graph file of US airports. I present Gephi's *ForceAtlas2* layout as a visual base point for both, see Figure 1 and 2. Observe the sorts of clustering that takes place in that layout.

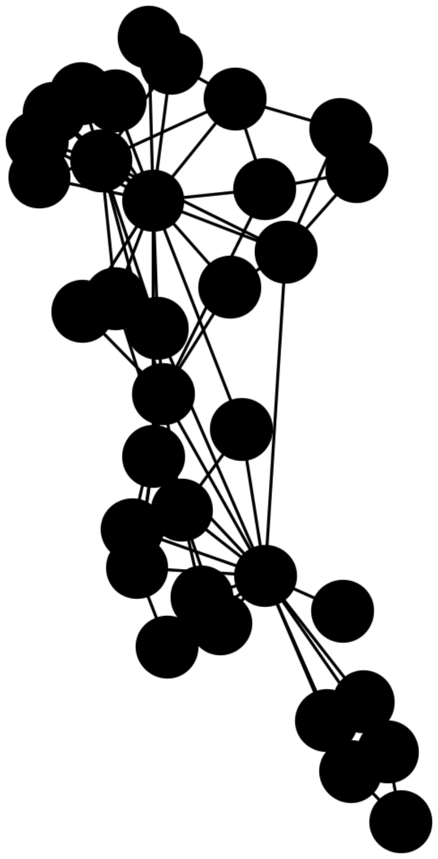


Figure 1: The karate club network after force atlas 2.

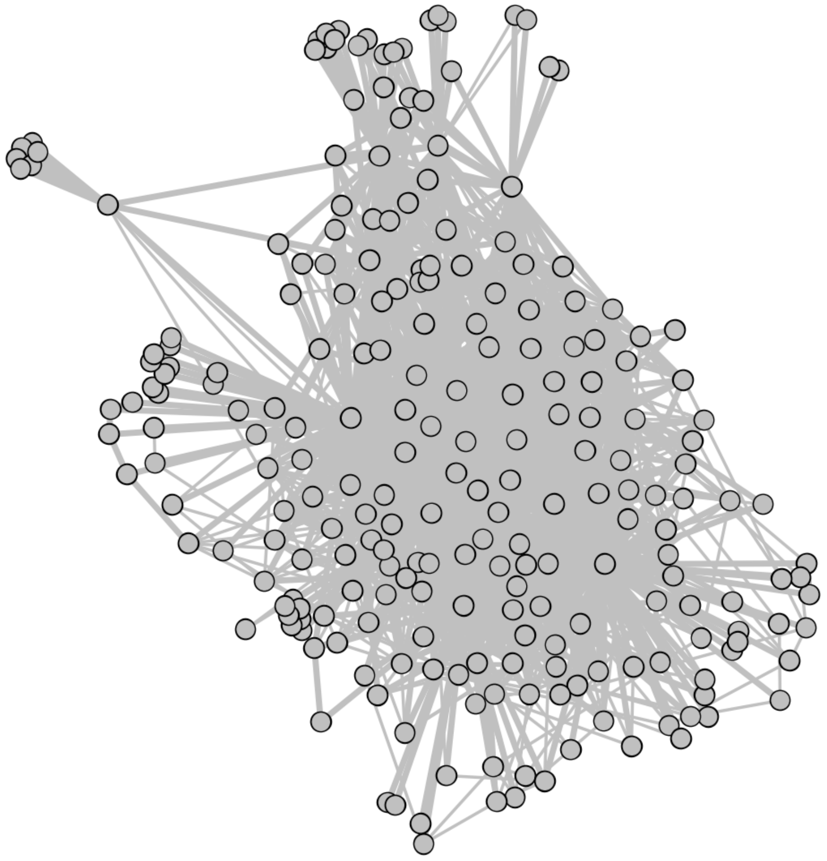


Figure 2: The airports network after force atlas 2.

Now here are the same Networks after an application of the spectral layout, see Figures 3 and 4. Observe the similar structure between the two. There seems to a main cluster that aligns itself in a line and a few outlying nodes that are not at all related to the main cluster.

This does not match the visual intuition promised by the literature. A observer should be able to find out the relative importance of almost every node in the graph, instead of just identifying unimportant nodes.

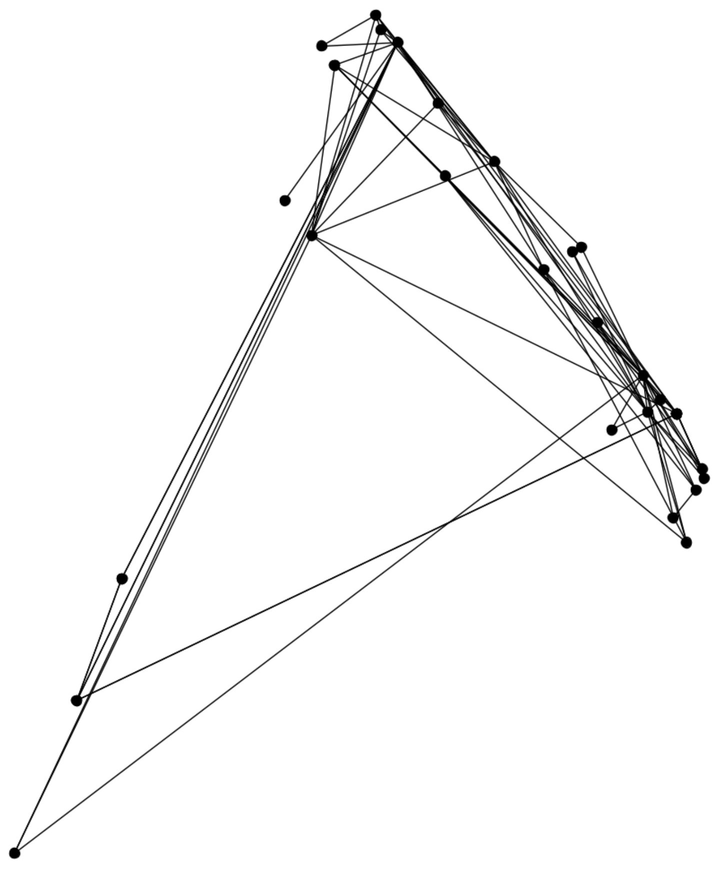


Figure 3: The karate club network after my spectral layout.

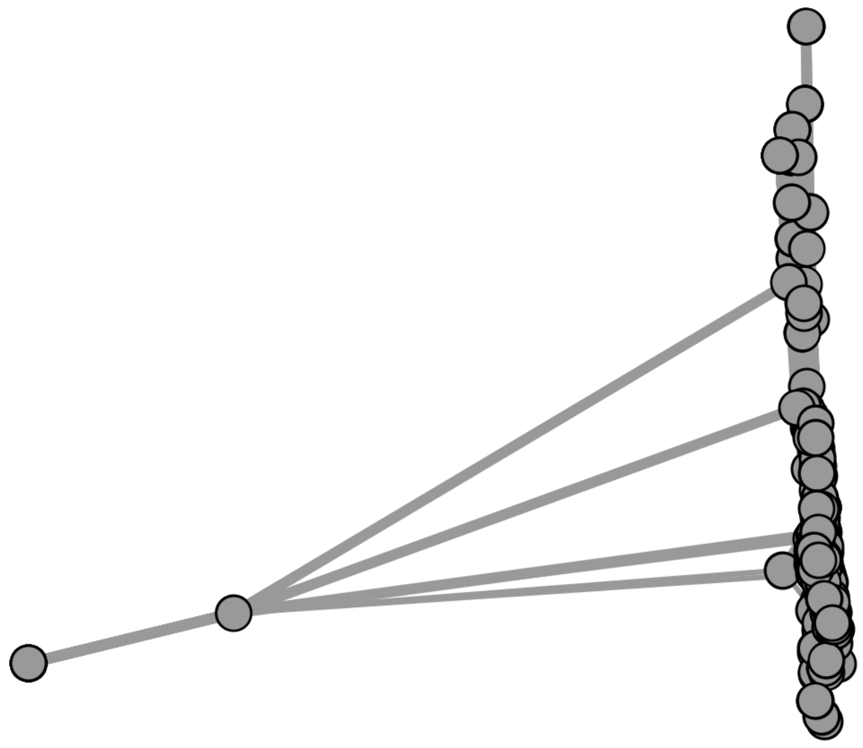


Figure 4: The airports network after my spectral layout.

A possible explanation for this phenomenon is that I am using the wrong eigenvectors to render the graph layout. The literature promises that the optimal layout corresponds to the top eigenvectors **or** the bottom eigenvectors depending on the graph related matrix. The algorithm given by Koren computes the top eigenvectors of the transition matrix, so I thought he was implying that the optimal layout came from them.

To use the bottom eigenvectors using our current method we had to compute all of them, which would slow computation time immensely. I did this for the karate club network (because it was small) and display the resulting layout in Figure 5.

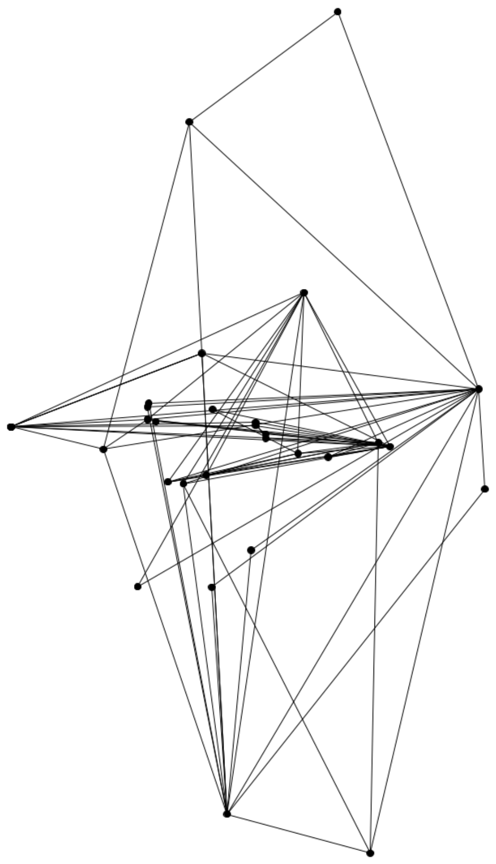


Figure 5: The karate club network after my spectral layout using the bottom eigenvectors.

This layout spreads the structure out better, but it still isn't easy to see the importance of every node. To see if there was any configuration of eigenvectors that resulted in an intuitive layout I randomly selected eigenvectors and drew the graph based on them. The results are in the following figures.

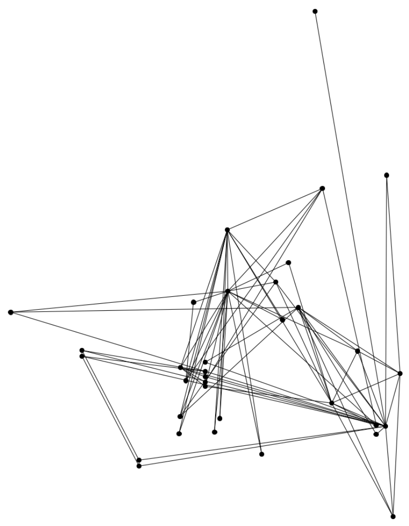


Figure 6: The karate club network after my spectral layout with random eigenvectors.

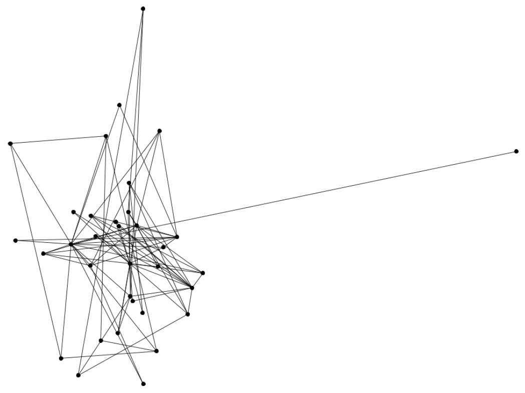


Figure 7: The karate club network after my spectral layout with other random eigenvectors.

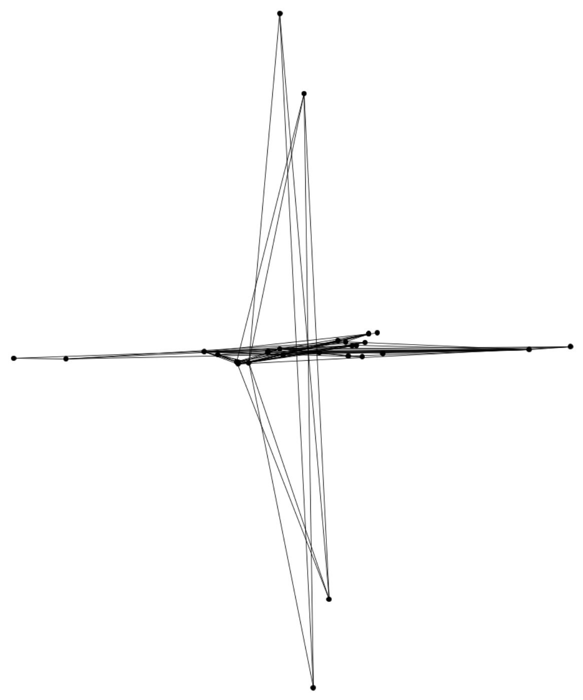


Figure 8: The karate club network after my spectral layout with yet other random eigenvectors.

Each layout represented the graph in a unique way by placing importance in different nodes. As an aggregate, they could be used to learn about the karate club. However, our goal was to learn about the graph from a single rendering. These case examples only really speak towards small graphs, however.

Limitations

Clearly, the method has some hiccups when it comes to small graphs. The literature I reviewed worked on large connected graph datasets. I hypothesize that spectral layouts are thus much better in that domain. This conjecture is supported by runtime comparisons. Spectral layouts run fairly quickly [3]. From personal experience, I can say particle system layouts take much longer to render large graphs.

There is a fundamental limitation of this method that is circumvented by particle systems. Spectral layouts cannot render a disconnected graph. This can be roughly overcome this issue by applying a layout on each connected component, but this is not a very realistic solution in a graph that contains many isolated nodes. It also does not say anything about how each component should be place in relation to the others. The particle systems do a much better job because they just push isolated small nodes and components to the periphery.

Conclusion

The layout implementation successfully rendered some graphs, but failed to draw many others. It is not very robust in its current form. From a software development point of view, there are many elements missing. Not the least of which is that there are no unit tests. If these tests were implemented, other issues in the layout would undoubtedly be identified. Work should be done in the future to implement them.

Now that the initial layout is implemented, we can easily implement other spectral layouts. We simply need to code the correct algorithm and insert it in place of the current algorithm in the `ComputeLayout` class. This is because the input of these spectral computation algorithms is always a graph (in our implementation represented by a degree list and edge list) and the output is always eigenvectors.

In order to be practically usable, there should be a GUI or some other interface. It is possible to import new features into Gephi for use. Work should be done to implement the layout in this way, so that students have access to it.

Our spectral layout did not seem to render small graphs in a meaningful way. It could not capture the relative importance of all the nodes in a graph. The literature suggests that the method is very efficient and visually intuitive on large networks. Future work could test our implementation on larger networks.

References

[1] Y. Koren *Drawing Graphs by Eigenvectors: Theory and Practice.* AT&T Labs–Research, 2004.

[2] Ulrik Brandes and Christian Pich *Eigensolver Methods for Progressive Multidimensional Scaling of Large Data.* University of Konstanz, 2006.

[3] Ali Civril, Malik Magdon-Ismail, and Eli Bocek-Rivele *SDE: Graph Drawing Using Spectral Distance Embedding.* Rensselaer Polytechnic Institute, 2005.

[4] David C. Lay *Linear Algebra and Its Applications.* Reading: Addison-Wesley, 1994. Print.

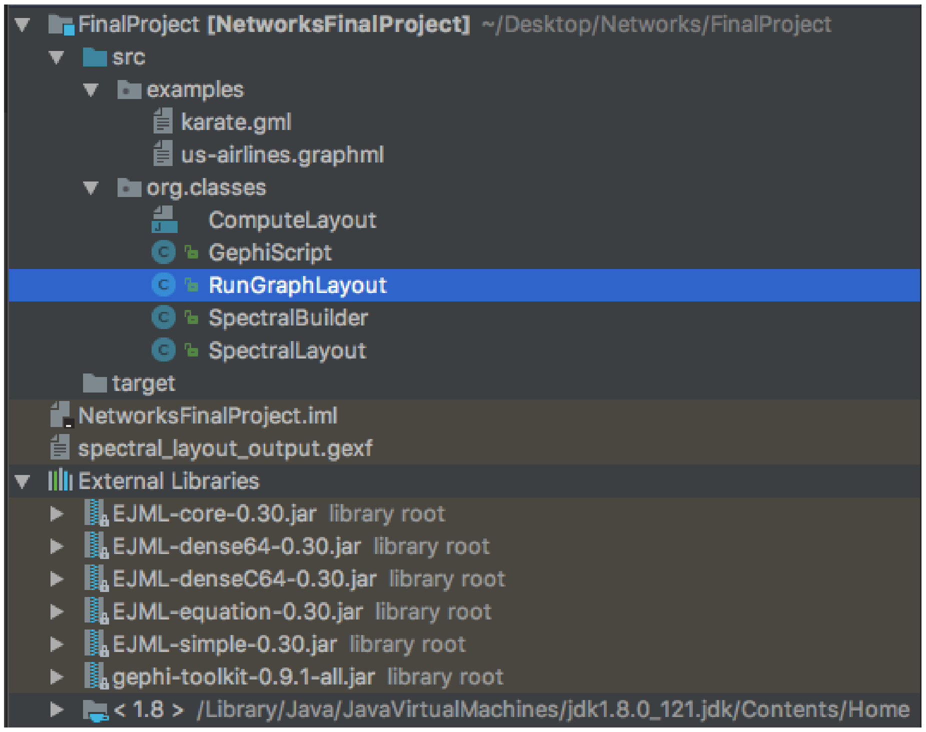
[5] Wayne W. Zachary *An Information Flow Model for Conflict and Fission in Small Groups.* Journal of Anthropological Research 33.4 (1977): 452-73.

Appendix: Use Guide

This section explains how to install and run the spectral layout in its current form. The source code is available at: `https://github.com/hjfender/GephiSpectralLayout`.

First, clone the project locally and open it in a suitable IDE, i.e. one that support Java development. I used IntelliJ community edition, which is available at `https://www.jetbrains.com/idea/download/`. Make sure you have JDK 1.8+ installed, available at `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`. Associate it to the project in your chosen IDE.

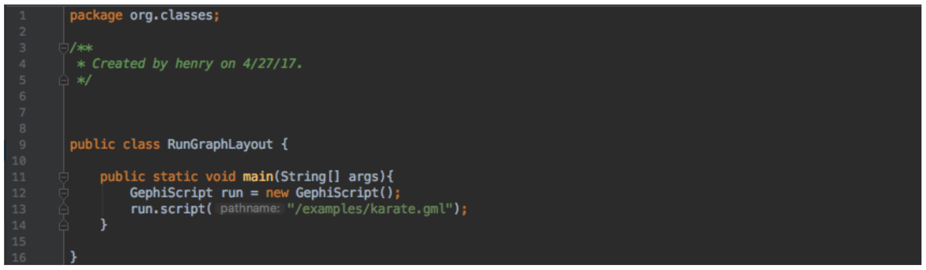
Now, in order for the project to compile the project need access to EJML, available at `https://sourceforge.net/projects/ejml/files/v0.30/`, and Gephi Toolkit at `https://gephi.org/toolkit/`. Download these libraries as `.jar` files and add them as modules to the external libraries of the project.



```
▼ FinalProject [NetworksFinalProject] ~/Desktop/Networks/FinalProject
  ▼ src
    ▼ examples
        karate.gml
        us-airlines.graphml
    ▼ org.classes
        ComputeLayout
        GephiScript
        RunGraphLayout
        SpectralBuilder
        SpectralLayout
    target
  NetworksFinalProject.iml
  spectral_layout_output.gexf
▼ External Libraries
  ▶ EJML-core-0.30.jar library root
  ▶ EJML-dense64-0.30.jar library root
  ▶ EJML-denseC64-0.30.jar library root
  ▶ EJML-equation-0.30.jar library root
  ▶ EJML-simple-0.30.jar library root
  ▶ gephi-toolkit-0.9.1-all.jar library root
  ▶ < 1.8 > /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home
```

Figure 9: The project structure.

The project should compile now! Go to the main method and run it. Right now it is set to create a layout of the karate club example. So see the result open the `spectral_layout_output.gexf` file in Gephi. This file should be at the same hierarchical level as the src folder of the project.



```
package org.classes;

/**
 * Created by henry on 4/27/17.
 */



public class RunGraphLayout {

    public static void main(String[] args){
        GephiScript run = new GephiScript();
        run.script( pathname: "/examples/karate.gml");
    }

}
```

Figure 10: The main method of the implementation.

You can run the layout algorithm on other .gml and .graphml files provided they represent connected graphs. Deposit your favored graph in the examples file and change the path name to "/examples/filename.filetype" in the main method (see figure 10). After running the program, the resulting layout will be returned in spectral_layout_output.gexf, open it in Gephi to see it.

There is an issue with how .gml and .graphml index their nodes. We covered it in the main body of the paper. It will be apparent that it is happening if upon running the program there is an ArrayIndexOutOfBoundsException in line 52 of the SpectralLayout class. To resolve it, change the NodeIDasInt method as described by its comment.