Spectral Graph Drawing: Building Blocks and Performance Analysis

Shad Kirmani
eBay Inc.
Brisbane, CA
skirmani@ebay.com

Kamesh Madduri
The Pennsylvania State University
University Park, PA
madduri@cse.psu.edu

***Abstract*—The objective of a graph drawing algorithm is to automatically produce an aesthetically-pleasing two or three dimensional visualization of a graph. Spectral graph theory refers to the study of eigenvectors of matrices derived from graphs. There are several well-known graph drawing algorithms that use insights from spectral graph theory. In this work, we experiment with two spectral methods to generate drawings of large-scale graphs. We also combine spectral drawings with the multilevel paradigm, and this leads to a larger collection of implementations. We analyze drawing quality and performance tradeoffs with these approaches.**

1. Introduction

By spectral graph drawing, we refer to algorithms that use eigenvectors of the graph Laplacian matrix or its variants for the purpose of graph drawing. Spectral graph clustering and spectral graph partitioning methods are closely related to drawing algorithms. Informally, the objective of a graph drawing algorithm is to assign coordinates (i.e., a 2/3-dimensional vector) to vertices, given a graph (without vertex coordinates) as input. We consider the problem of drawing unweighted, undirected graphs that are *almost regular* (i.e., all vertices have nearly the same degree). Figure 1 gives an example of three automatically generated drawings or graph layouts. The first drawing is generated using the open-source Graphviz package, the second one using one of the implementations presented in this paper (based on Koren's algorithm [1]), and the third drawing using selected eigenvectors of the Laplacian (Hall's algorithm [2], very closely related to Koren's algorithm). Figure 2 is a drawing of a much larger graph, using the same algorithm as the approach used for Figure 1b.

Spectral graph drawing methods have many advantages: the problem formulations are mathematically sound (we formally introduce the problem in the next Section), the solutions obtained are provably optimal under certain settings, implementations are typically fast, the algorithms are compatible with the multilevel paradigm, and the drawings generated are aesthetically pleasing.

While this work focuses on drawing, the intermediate outputs from our implementations have several other uses, including graph embedding [4], partitioning [5], and clustering [6], [7]. In fact, we began working on this problem motivated by spectral graph partitioning.

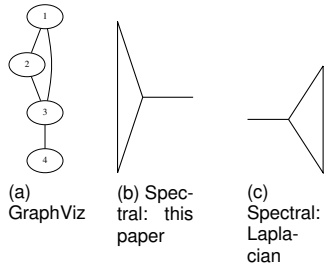


Figure 1. Drawings of a small graph with four vertices and four edges. The drawing in (a) uses the default layout algorithm in Graphviz v2.38. The drawing in (b) is based on an algorithm discussed in this paper. The drawing in (c) uses two eigenvectors of the Laplacian matrix corresponding to the graph.

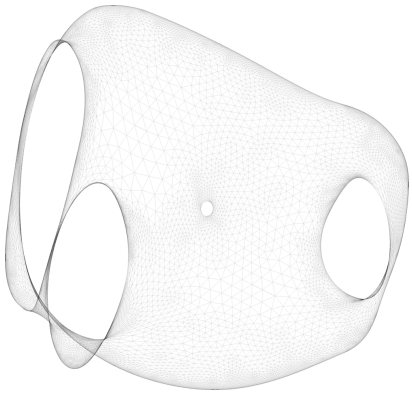


Figure 2. A drawing of the 4elt [3] graph (15 606 vertices, 107 362 edges) using an algorithm discussed in this paper.

Our contributions: This work presents new implementations of the algorithms in [1]. In addition to evaluating these algorithms on a large collection of graphs, we extend these algorithms by using them in a multilevel setting. Our empirical evaluation has many interesting findings.

Workshop relevance: We identify several building blocks for practical and high-performance spectral graph drawing. While we focus on two approaches in this paper, our software (to be open-sourced) already supports more than a dozen variants. Further, we have experimented with

several implementations of the same algorithm, both standalone and library-based. The results presented in this paper are based on implementations that use the Eigen [8] C++ linear algebra library. Further, our implementations can be used for new parallel graph analysis benchmarking.

2. Background

We denote a graph as $G(V, E, W)$, where $V = \{1, \ldots, n\}$ is the set of vertices and E is the set of edges. We assume the graph is undirected and has no self loops or parallel edges. An edge $\langle i, j \rangle$ can have a non-negative real edge weight $W(i, j)$ representing the similarity of vertices i and j. For unweighted graphs, the edge weights are all set to 1. The weighted degree of a vertex is the sum of the weights of its attached edges.

Let A denote the symmetric adjacency matrix corresponding to the graph, with $A(i, j) = W(i, j)$ if $\langle i, j \rangle \in E$, and 0 otherwise. Let D denote the degrees matrix, a diagonal matrix with $D(i, i)$ set to the weighted degree of vertex i. The Laplacian L is another symmetric matrix given by $D - A$. The properties of the Laplacian are well studied. We will state some of its properties relevant to graph drawing in this section, but please see Koren's paper [1] and references therein for more details. Spielman's book chapter on Spectral Graph Theory [9] is a good reference. The degree normalized matrix $D^{-1}A$ is called the transition matrix or the walk matrix.

The p-dimensional layout of a graph is defined by p vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p \in \mathbb{R}^n$, where $\boldsymbol{x}_k(i)$ corresponds to the coordinate of vertex i in the k^{th} dimension. In practice, p is chosen to be 2 or 3 for screen layouts. The vertices can be connected by straight lines to produce a drawing. The Euclidean distance d_{ij} between vertices i and j in the p-dimensional layout is given by $\sqrt{\sum_{k=1}^{p} \left(\boldsymbol{x}_k(i) - \boldsymbol{x}_k(j)\right)^2}$.

Given any vector $\boldsymbol{y} \in \mathbb{R}^n$, it can be shown that $\boldsymbol{y}'L\boldsymbol{y}$ equals $\sum_{\langle i,j \rangle \in E} W(i, j) \left(\boldsymbol{y}(i) - \boldsymbol{y}(j)\right)^2$. This fact is directly related to an *optimal* layout of a graph. Additionally, the Laplacian satisfies many interesting properties. Since L is symmetric and positive semidefinite, all its n eigenvalues are real and nonnegative, and its eigenvectors are orthogonal. $\mathbf{1}_n$ is an eigenvector of L with a corresponding eigenvalue of 0. If the graph is connected, the multiplicity of the eigenvalue 0 is exactly 1. The eigenvalues of the Laplacian are conventionally ordered from low to high, $0 = \lambda_1 < \lambda_2 \leq \cdots \leq \lambda_n$. Given D and L, the eigenpair $(\boldsymbol{u}, \mu)$ are termed a generalized eigenpair if $L\boldsymbol{u} = \mu D\boldsymbol{u}$. D-normalization is a way to uniquely specify the $\boldsymbol{u}_k$ eigenvectors: $\boldsymbol{u}_k' D \boldsymbol{u}_k = 1, k = 1, \ldots, n$. The vectors $\boldsymbol{u}_k$ are called degree-normalized eigenvectors. It can be easily shown that the degree-normalized eigenvectors are also the (nongeneralized) eigenvectors of matrix $D^{-1}A$, using the fact that $L = D - A$. The eigenvalues are in reverse order. Koren shows that the degree-normalized eigenvectors are the solutions to a constrained minimization problem that is closely related to graph drawing aesthetics (see Equation 15 in [1]). Hall's algorithm refers to the use of the two lowest eigenvectors of the Laplacian for layout. In contrast, Koren's algorithm uses the p lowest degree-normalized eigenvectors. These vectors are also used for image segmentation [10], but with a different motivation. Since $D^{-1}A$ has the same degree-normalized eigenvectors in reverse order, Koren gives a variant of the Power iteration to compute the p leading nondegenerate eigenvectors. We list this approach in Algorithm 1. The familiar Power iteration is augmented with a Gram-Schmidt-like orthogonalization scheme.

Algorithm 1 Algorithm for computing degree-normalized eigenvectors (Koren's algorithm [1] with minor changes).

Input: G, p (desired dimensions), $\boldsymbol{u}_1$ (top (degenerate) eigenvector), ϵ (tolerance)
Output: $\boldsymbol{u}_2, \ldots, \boldsymbol{u}_{p+1}$, the top p nondegenerate eigenvectors of $D^{-1}A$

$M \leftarrow \frac{1}{2}(I + D^{-1}A)$
for $k \leftarrow 2, p+1$ **do**
  $\boldsymbol{v} \leftarrow$ random vector of size n
  Normalize $\boldsymbol{v}$
  repeat
    $\boldsymbol{u}_k \leftarrow \boldsymbol{v}$
    for $l \leftarrow 1, k-1$ **do**
      $\boldsymbol{u}_k \leftarrow \boldsymbol{u}_k - \frac{\boldsymbol{u}_k' D \boldsymbol{u}_l}{\boldsymbol{u}_l' D \boldsymbol{u}_l} \boldsymbol{u}_l$ ▷ D-Orthogonalize
    $\boldsymbol{v} \leftarrow M\boldsymbol{u}_k$ ▷ Similar to Power iteration
    Normalize $\boldsymbol{v}$
  until $\|\boldsymbol{v} - \boldsymbol{u}_k\| < \epsilon$
  $\boldsymbol{u}_k \leftarrow \boldsymbol{v}$

Algorithm 2 Weighted Centroid smoothing of coordinates.

Input: D, A, n_{smooth}, $\boldsymbol{x}$, $\boldsymbol{y}$
Output: $\boldsymbol{x}$, $\boldsymbol{y}$

for $i \leftarrow 1, n_{\text{smooth}}$ **do**
  $\boldsymbol{x} \leftarrow D^{-1}A\boldsymbol{x}$
  $\boldsymbol{y} \leftarrow D^{-1}A\boldsymbol{y}$

In practice, if we want to generate a 2D drawing, we can use the i^{th} elements of the two leading vectors obtained from Algorithm 1 as X and Y coordinates for vertex i. These coordinates can be used to visualize vertices as well as edges (by connecting vertices by straight lines).

The *multilevel* paradigm is a heuristic to accelerate graph and linear algebra computations, and can be applied to eigenvector computations as well. In multilevel spectral methods, graphs are recursively coarsened, eigenvectors are computed for the matrix corresponding to the coarsest graph, and the coarse graph vectors are then projected back to the original graph. Multilevel schemes are extensively used because they are very fast and are shown to work well for real-world graphs and matrices.

There are exist alternatives to the heavyweight eigenvector computation of Algorithm 1. To refine a projected

Algorithm 3 High Dimensional Embedding (HDE).

Input: G, L (Laplacian), s (subspace dimension)
Output: $\boldsymbol{x}$, $\boldsymbol{y}$

1: Initialize $S \in \mathbb{R}^{n \times (s+1)}$
2: Initialize $B \in \mathbb{R}^{n \times s}$
3: $\boldsymbol{S}_0 \leftarrow \mathbf{1}$ ▷ first column of S
4: Normalize $\boldsymbol{S}_0$
5: $sv \leftarrow$ randomly-chosen starting vertex
6: Initialize $\boldsymbol{d} \in \mathbb{R}^n$
7: **for** $i \leftarrow 1, s$ **do**
8: $\quad \boldsymbol{B}_i \leftarrow$ BFS(sv)
9: $\quad \boldsymbol{S}_i \leftarrow \boldsymbol{B}_i$
10: $\quad$ Normalize $\boldsymbol{S}_i$
11: $\quad$ **for** $l \leftarrow 0, i-1$ **do**
12: $\quad\quad \boldsymbol{S}_i \leftarrow \boldsymbol{S}_i - (\boldsymbol{S}_l' \boldsymbol{S}_i)\boldsymbol{S}_l$ ▷ Orthogonalize
13: $\quad d \leftarrow \min d_j, \boldsymbol{B}_{ij} \forall j \in [1, n]$
14: $\quad sv \leftarrow \arg\max d$
15: $S \leftarrow S[1, n]$ ▷ Drop degenerate vector
16: $Y_{s \times 2} \leftarrow$ Top two eigenvectors of $S^T L S$
17: $[\boldsymbol{x}, \boldsymbol{y}] \leftarrow BY$

layout generated by a multilevel method, a small number of *smoothing rounds* can be applied, which is shown concisely in Algorithm 2 [11]. In each smoothing round, the coordinate of a vertex is updated to be the weighted average of the coordinates of its adjacencies. We call this the Weighted Centroid smoothing strategy, and this is closely related to Koren's algorithm, a few iterations without normalization or orthogonalization may be applied. In practice, Algorithm 2 works well if we have a layout that is close to global optimality, but has some local perturbations.

2.1. High Dimensional Embedding

Harel and Koren give another speedup heuristic to Algorithm 1. This approach is called High Dimensional Embedding (HDE) and is described in Algorithm 3 [12]. A HDE is computed by constraining the layout algorithm to be in a s-dimensional subspace, $S_{n \times s}$. s is usually chosen to be a very small value, say 10. Constraining the layout to a well chosen s-subspace helps in computing layouts of reasonable quality in very limited amount of time.

The s-subspace is carefully constructed using graph theoretic distances of all vertices starting from s start vertices. More precisely, each vector in the s-subspace is the Breadth-First Search (BFS) distance of the vertices from s different starting vertices. These BFS distance vectors make the matrix $B_{n \times s}$. Another matrix S, derived from the matrix $B_{n \times s}$, whose vectors are orthonormalized BFS distance vectors.

Each successive start vertex for BFS is chosen as the vertex that is the farthest from all previous start vertices. This is achieved by keeping a vector of minimum distance from any start vertex in lines 6, 13, and 14 of the Algorithm 3.

To constrain the layout in the s-subspace, we find the eigenvectors y_s of the matrix $S'LS$. Since $S^{'}LS$ is an $s \times s$ matrix and s is usually very small, eigenvector computation takes negligible time. Once the s-dimensional eigenvectors have been computed, we get the projected coordinates by computing By.

3. Our Spectral Drawing Implementations

Encouraged by the simplicity of Koren's algorithm, we started by implementing this approach as stated in [1]. However, we found that the recommended convergence check of computing the dot product of the vectors $\boldsymbol{u}_k$ and $\boldsymbol{v}$ resulted in unsatisfactory drawings for some of our test inputs. We thus modified the convergence check to the norm of difference of the vectors, as shown on line 11 of Algorithm 1. We then encountered the problem of extremely slow convergence, and this motivated us to implement the High Dimensional Embedding (HDE) scheme. Independently, we were exploring spectral drawings in the multilevel paradigm, where we coarsen a graph, compute the eigenvectors on the coarse graph, and then uncoarsen and refine these eigenvectors. We then realized that our disparate implementations can be expressed and used together in a common framework. Algorithm 4 summarizes how we develop a large number of graph drawing algorithmic variants. Executing Algorithm 1 is akin to going to step 4 in Algorithm 4, avoiding the speedup heuristics steps 1, 2, and 3. The multilevel paradigm neatly fits within this framework because we can use the dominant degree-normalized eigenvectors corresponding to the coarse graph to initialize vectors for the full graph. Koren also mentions the connection of his algorithm to the Weighted Centroid smoothing scheme. If we intentionally choose n_{smooth} to be a small-enough value, we can avoid degenerate solutions while simultaneously improving drawing quality. Finally, HDE is also compatible with coarsening. Instead of doing an aggressive coarsening of the graph, we can generate a moderately-coarse graph, determine the degree-normalized eigenvectors for this coarse graph with HDE initialization, and project the vectors to the full graph.

We can enumerate 24 variants from Algorithm 4. One variant (skipping all four steps) is not useful and can thus be ignored. This leaves us with 23 variants, ranging from step 4 only to performing all four steps. Performing only Step 3 after random initialization can also be omitted, getting us to 22 variants.

The next issue to consider is setting the defaults for steps 1 to 4. For coarsening alone, there exist a number of algorithms that we can employ [13]. We initially experimented with the heavy edge matching routine in Metis v4.0.3 [14]. Even this routine had numerous parameters to set, and so we developed a very simple coarsening scheme to use as a baseline. We coarsen the graph by computing maximal edge matchings. We compute the matching by visiting vertices in order of their identifier (1 to n), and inspecting adjacencies in sorted order. If vertex i is unmatched and has an unmatched neighbor j, we add edge $\langle i, j \rangle$ to the matching and mark both vertices i and j as matched. In the coarser graph, the lower value of i and j is used as the new vertex identifier and the process is repeated. We perform at most 100 iterations of matching or coarsen until the vertex count

is less than 1000. These parameters are also currently fixed in order to reduce the space of implementations to consider. This coarsening scheme generates a coarse graph without any randomization, and may perform poorly on adversarial inputs. However, for the graphs we have experimented with so far, we did not encounter bad cases.

We have not yet fixed a threshold for moderate coarsening. We did some ad-hoc experimentation to combine coarsening and HDE, but we do not report on these results in the current submission.

HDE is parameterized by s, the number of starting points in the graph to perform Breadth-First Searches from. We currently set this value to 10.

For the Weighted Centroid smoothing, we set n_{smooth} to 500 based on limited ad-hoc experimentation.

The key parameter in the degree-normalized eigenvector computation is the tolerance ϵ. We use a tolerance setting of 10^{-6} for the top eigenvector and a setting of 2×10^{-6} for the next eigenvector. The rate of convergence is dependent on the ratio of magnitude of eigenvalues. For nearly all the real-world graphs we experimented with, the dominant eigenvalue was very close to the degenerate solution. The first three digits following the decimal point were all 9's in most cases. This means that we had to perform at least 1000s of iterations for each eigenvector.

While it may be tempting to choose a fast heuristic variant (i.e., activate all or most of steps 1 to 3 before step 4), note that we have no theoretical guarantee that we will converge to the dominant eigenvectors if we use these heuristics. Given that the eigenvalues are so close to each other, we might end up converging to a non-dominant eigenvector. Thus, our baseline in this study is to use Algorithm 1 with a conservative tolerance value.

After obtaining coordinates, the final step of generating the drawing remains. We use a very simple approach in the current work. We scale the X coordinates to integer values that lie in $[0, 10000]$. We set the Y coordinates to integers while preserving the aspect ratio. We use a Portable Network Graphics (PNG) format file writer in Python for storing the image, after connecting vertices using straight lines of fixed thickness. We have not experimented with colors yet. This step of generating the image files is currently not timed. We use the OS X command line utility sips to further compress these images, specifying a resampling setting of 1000. The drawings included in this paper are all generated according to this scheme.

Algorithm 4 A practical framework for spectral graph drawing.

Input: $G(V, E)$

Output: Two vectors $\boldsymbol{x}$, $\boldsymbol{y} \in \mathbb{R}^n$ to use as coordinates for vertices in V.

$\boldsymbol{x}$, $\boldsymbol{y} \leftarrow$ randomly initialized vectors

1: Update $\boldsymbol{x}$, $\boldsymbol{y}$ by using eigenvectors from coarsened G
▷ Two options: coarsen, do not coarsen

2: Update $\boldsymbol{x}$, $\boldsymbol{y}$ by HDE
▷ Three options: HDE on full graph, HDE on coarsened graph, no HDE

3: Update $\boldsymbol{x}$, $\boldsymbol{y}$ using Weighted Centroid smoothing
▷ Two options

4: Use $\boldsymbol{x}$, $\boldsymbol{y}$ to initialize Algorithm 1, and set $\boldsymbol{x}$, $\boldsymbol{y}$ to the output of Algorithm 1
▷ Two options

4. Empirical Evaluation

4.1. Experimental Setup

We evaluate the graph drawing implementations on a collection of sparse graphs. Table 1 lists some of the graphs we experimented with. Most of these graphs are taken from the SuiteSparse matrix collection [15]. Our drawings can be compared with Yifan Hu's visualizations (available at http://yifanhu.net/GALLERY/GRAPHS/) based on the sfdp multilevel force-directed layout algorithm [16]. Hu's visualizations also give the CPU time for the drawing, and these can be compared with our running times. We removed self loops and parallel edges in the input graphs, and ignored any edge weights, if present. Thus, we effectively treated all matrices as unweighted graphs. While the spectral algorithm can handle edge weights, we wanted to start with a simplified setup. Our approaches—with the current default settings—fail to produce good drawings for some large matrices in the SuiteSparse collection, and we suspect this is an artifact of using just two eigenvectors for the drawings.

We report performance results on a single compute node of the Comet supercomputer, operated by the San Diego Supercomputer Center at UC San Diego. Comet is available through the Extreme Science and Discovery Environment (XSEDE) program [17]. The compute node we use has two 12-core Intel(R) Xeon(R) CPU E5-2680 v3 (Haswell) processors clocked at 2.50 GHz. Each node has 128 GB DDR4 DRAM memory. The STREAM Triad bandwidth is 104 GB/s. We use the Intel C++ compiler (version 16.0.3) to build our code. We compile our code with OpenMP support and the optimization flags -xHOST -O3.

We use the Eigen C++ linear algebra library (v3.3.4), primarily for the sparse matrix-vector multiplication and orthogonalization required in Algorithm 1. Before using Eigen, we developed standalone implementations for some of the functionality in Algorithm 4, but the code started to become quite unwieldy. Using Eigen greatly simplified our work. We are also able to leverage Eigen's parallelization and vectorization of underlying routines.

4.2. Performance of Koren's algorithm

In Table 2, we report on the performance of Koren's algorithm without any speedup heuristics. For the last five graphs listed in the table, we use a tolerance of 1e-5 because of slow convergence. We report overall time in seconds, including file I/O, memory allocation, and initialization costs. We also report the aggregate times for computing the

TABLE 1. THE COLLECTION OF UNDIRECTED GRAPHS (SYMMETRIC SPARSE MATRICES) USED IN THIS STUDY. THE NUMBER OF VERTICES (n), NUMBER OF EDGES (m), AND SOURCE ARE GIVEN.

Graph	n	m	Source
grid6400	6400	12 640	own
crack	10 240	30 380	[15]
4elt	15 606	45 878	[3]
finance256	37 376	130 560	[15]
oilpan	73 752	1 761 718	[15]
apache1	80 800	230 692	[15]
filter3D	106 437	1 300 371	[15]
para-5	155 924	2 630 217	[15]
d_pretok	182 730	756 256	[15]
turon_m	189 924	778 531	[15]
mario002	389 874	933 557	[15]
pa2010	421 545	1 029 231	[15]
ecology1	1 000 000	1 998 000	[15]
G3_circuit	1 585 478	3 037 674	[15]
kkt_power	2 063 494	6 482 320	[15]

two leading eigenvectors. It is immediately apparent that the eigenvector computation time is the main contributor to the overall time. The times reported are for 24-thread runs, and so we also report speedup based on the times obtained with a single-threaded run. Finally, we list the number of loop iterations required to satisfy the desired tolerance, and compute a per-iteration performance rate, by normalizing running time by number of edges. Inspecting this Table by column, it is unsurprising that the graph size is correlated with running time. The two largest graphs by vertex count also take the longest time to finish.

Observe that the 24-core parallel speedup is relatively low. This may be because of the small matrix sizes and OpenMP overheads in matrix-vector multiplication. Now that we have a reference implementation, we can optimize performance by substituting a tuned sparse matrix-vector multiplication routine. Speedup also appears to be correlated with average vertex degree and is higher for denser graphs.

The number of iterations required varies from input to input, and is dependent on the ratio of the eigenvalues and the quality of our initial random estimate. filter3D has the largest iteration count for the second eigenvector, but with the caveat that we used a higher threshold for the last five graphs. 4elt was a particularly challenging input and the residual for the third eigenvector briefly shows an increasing trend, before again reducing.

Inspecting the rate provides more insight into performance and room for improvement. The rate varies as much as $10\times$ from input to input. It is highest for the graph with the largest m/n ratio, suggesting that the sparse matrix vector multiply operations offset any overhead due to non-parallelization of the orthogonalization-related steps.

4.3. Sensitivity to default parameters

In Figure 3, we observe a *warping effect* when the threshold for convergence in Algorithm 1 is set higher than the default threshold parameter of 1e-6. In Figure 3a, notice that the drawing for 4elt is not as smooth as the drawing in Figure 2. As we can observe with Figure 3b, if the threshold is raised higher, warping effect gets more acute. It is difficult to identity beforehand when the graph drawing may be warped, and so we had to experiment with a large collection of tolerance values, before settling on the defaults used in this work.

4.4. Impact of Coarsening and HDE

In Table 3, we present the running time overhead of just the initialization strategies. Instead of starting from random vectors, we can use projected coarsened coordinates or HDE initialization. The time taken to compute projected coarsened coordinates is in general very low, but for some graphs, such as mario002 and pa2010, the time is very high. This is because of the stringent tolerance criterion (1e-7) used to assess convergence when using the coarser graph. On the other hand, HDE initialization is relatively fast for all graphs, because eigenvector computation is performed on a tiny graph.

In Table 4, we present the performance, running time improvement, and iteration counts of Algorithm 1, if initialized with projected coarsened coordinates. To compute second eigenvector, we achieve speedups ranging from $1.4\times$ to $988\times$. The iteration count drops if the projected coarsened graph coordinates are used. For instance, filter3D requires $152\,426$ iterations to compute second eigenvector with random initialization, but requires only $16\,602$ iterations when initialized with projected coarsened graph coordinates. We observe similar results for computing the third eigenvector as well.

Figure 4 shows two different drawings of the 4elt input, both using Algorithm 1 for refinement, but with different initialization strategies. We observe that these drawings are nearly identical to the drawing with random initialization.

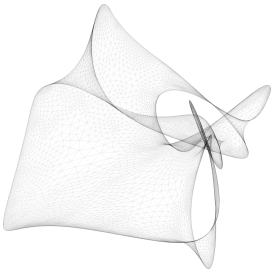
4.5. Performance of Fast Variants

In Table 6, we present the performance of a variant of Algorithm 4, with HDE initialization and Weighted Centroid smoothing Algorithm 2. We see that the time taken to compute drawings is considerably smaller, and the speedups range from $22\times$ for small graphs to $130\times$ for larger graphs. As we see next, the quality of the drawing computed with this algorithm is almost similar for small graphs and better for larger graphs.

4.6. Drawing Quality

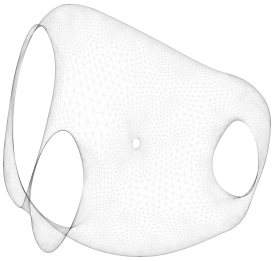
In Figure 5, we compare the drawings computed using the exact method described in Algorithm 1 and the approximate variant using HDE along with Weighted Centroid smoothing described in Algorithm 2. In general, we observe that HDE with weighted centroid smoothing computes good global layouts, i.e., no warping effects in the drawing. The details are also preserved well. On the other hand, exact Algorithm 1, computes locally smoother drawings, but can

(a) eps $= 5 \times 10^{-5}$

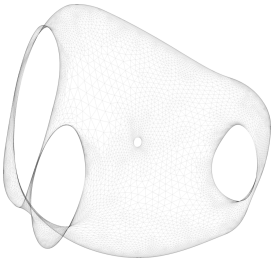


(b) eps $= 10^{-4}$

Figure 3. The drawing quality is sensitive to the convergence threshold used in our algorithm. We give drawings for the 4elt input with two different eps settings. These can be compared to the drawing in Figure 2, which uses the default tolerance value.



(a) Coarsening



(b) HDE

Figure 4. Drawings of the 4elt graph when using initialized vectors from coarsening and HDE.

suffer from warping effects when the graphs are large with large convergence thresholds. For smaller graphs, such as crack, finance256, oilpan, etc. we can observe that using Algorithm 1 alone computes more symmetric drawings.

5. Conclusions and Future Work

This work presents a number of spectral graph drawing algorithmic variants, all focused on computing the two dominant degree-normalized eigenvectors. We find that we need to compute the vectors to a high degree of accuracy in order to generate aesthetically-pleasing layouts. An extremely fast heuristic works surprisingly well. We also observe different algorithmic variants highlighting different features of the graph in the layout.

We foresee a number of directions for future research. An immediate goal would be to extend the evaluation and make it more comprehensive, along the lines of Hu's graph visualization gallery. Identifying good defaults to use for various parameters is another short-term goal. Finally, we want to explore parallel performance and scalability bottlenecks in greater depth.

There are several closely-related problems areas where we can directly use the computed degree-normalized eigenvalues. Graph embedding and partitioning problems are the most promising avenues.

Acknowledgments

This work is supported by the US National Science Foundation grant #1253881. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

References

[1] Y. Koren, "Drawing graphs by eigenvectors: theory and practice," *Computers & Mathematics with Applications*, vol. 49, no. 11, pp. 1867–1888, 2005.

[2] K. M. Hall, "An r-dimensional quadratic placement algorithm," *Management science*, vol. 17, no. 3, pp. 219–229, 1970.

[3] C. Walshaw, "Walshaw's graph partitioning archive," 2016, http://chriswalshaw.co.uk/partition, last accessed Feb 2018.

[4] H. Cai, V. W. Zheng, and K. C. Chang, "A comprehensive survey of graph embedding: Problems, techniques and applications," 2017, arXiv preprint abs/1709.07604.

[5] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proc. Int'l. Conf. on high performance computing, networking, storage and analysis (SC)*, 2013.

[6] M. Filippone, F. Camastra, F. Masulli, and S. Rovetta, "A survey of kernel and spectral methods for clustering," *Pattern Recognition*, vol. 41, no. 1, pp. 176–190, 2008.

[7] S. White and P. Smyth, "A spectral clustering approach to finding communities in graphs," in *Proc. SIAM Int'l. Conf. on Data Mining (SDM)*, 2005.

[8] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[9] D. Spielman, "Spectral graph theory," in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, Eds. CRC Press, 2012, ch. 18.

[10] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[11] W. T. Tutte, "How to draw a graph," *Proc. London Mathematical Society*, vol. 3, no. 1, pp. 743–767, 1963.

[12] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," in *Proc. Int'l. Symp. on Graph Drawing*, 2002.

TABLE 2. PERFORMANCE OF THE 'EXACT' APPROACH (WITHOUT COARSENING OR HDE). RATE IS CALCULATED AS $\frac{m}{1e6*t*c}$. NUMBERS EMPHASIZED IN BOLD FONT ARE DISCUSSED IN THE PAPER.

Graph	Overall Time (s)	Parallel Speedup	Second eigenvector Iter c	Second eigenvector Time t (s)	Second eigenvector Rate	Third eigenvector Iter c	Third eigenvector Time t (s)	Third eigenvector Rate
grid6400	4	1.44	25 169	1.6	197	28 643	2.0	184
crack	13	1.60	74 027	9.4	238	28 313	3.6	239
4elt	31	1.70	66 664	11.7	261	**100 348**	19.0	242
finance256	23	1.78	30 999	12.1	335	23 560	10.2	301
oilpan	140	**4.66**	115 420	110.3	**1843**	26 564	28.7	**1630**
apache1	56	1.80	43 286	34.4	291	23 170	20.5	260
filter3D	213	3.96	**152 426**	180.1	1101	24 074	31.7	986
para-5	166	4.07	37 651	91.3	1084	27 504	72.4	999
d_pretok	230	2.19	47 102	83.5	426	75 795	144.5	397
turon_m	352	2.23	126 801	232.5	425	56 908	117.2	378
mario002	323	2.61	39 017	165.5	220	32 593	153.7	198
pa2010	335	2.71	54 516	227.7	246	21 000	103.7	208
ecology1	1286	1.81	85 370	**955.9**	178	24 748	321.1	154
G3_circuit	**1417**	2.07	47 819	917.6	158	21 941	484.9	137
kkt_power	**1448**	1.75	18 121	584.4	201	23 448	**843.8**	180

TABLE 3. THE TIME TAKEN FOR COARSENING AND HDE WITH DEFAULT SETTINGS. NOTE THAT THESE TIMES INCLUDE EIGENVECTOR COMPUTATION FOR THE SMALLER GRAPH.

Graph	Time (s) Coarsening	Time (s) HDE
grid6400	0.44	0.19
crack	0.29	0.13
4elt	0.34	0.12
finance256	0.70	0.14
oilpan	0.35	0.19
apache1	0.24	0.15
filter3D	1.88	0.13
para-5	0.60	0.29
d_pretok	0.10	0.24
turon_m	0.12	0.24
mario002	**388.69**	0.41
pa2010	**129.19**	0.45
ecology1	5.34	0.70
kkt_power	**28.80**	2.88

TABLE 4. PERFORMANCE OF THE EIGENVECTOR COMPUTATION STEPS WITH COARSENED GRAPH INITIALIZATION.

Graph	Second eigenvector Time (s)	Second eigenvector Speedup	Second eigenvector Iter c	Third eigenvector Time (s)	Third eigenvector Speedup	Third eigenvector Iter c
grid6400	0.2	7.6	169	0.0	151.0	157
crack	1.1	8.9	7314	1.9	1.9	13 166
4elt	3.8	3.1	21 378	4.9	3.9	25 385
finance256	5.0	2.4	12 220	3.5	2.9	8103
oilpan	15.1	7.3	13 791	25.8	1.1	20 947
apache1	3.9	8.9	5448	38.7	0.5	47 784
filter3D	0.2	988.6	16 602	1.7	18.7	148 191
para-5	64.6	1.4	22 291	35.9	2.0	11 406
d_pretok	54.3	1.5	34 558	388.6	0.4	216 209
turon_m	66.2	3.5	39 979	66.2	1.8	35 116
mario002	8.6	19.2	2311	4.7	32.5	1045
pa2010	10.5	21.8	2562	11.9	8.7	2389
ecology1	37.8	25.3	3104	56.0	5.7	3777
kkt_power	390.8	1.5	13 360	459.4	1.8	13 698

TABLE 5. PERFORMANCE OF THE EIGENVECTOR COMPUTATION STEPS USING THE HDE VECTORS FOR INITIALIZATION.

Graph	Second eigenvector Time (s)	Second eigenvector Speedup	Second eigenvector Iter c	Third eigenvector Time (s)	Third eigenvector Speedup	Third eigenvector Iter c
grid6400	0.6	2.6	3335	0.5	4.2	2777
crack	4.1	2.3	13 263	4.4	0.8	13 353
4elt	12.7	0.9	36 260	7.5	2.5	22 181
finance256	8.0	1.5	16 969	6.4	1.6	13 854
oilpan	23.2	4.8	16 404	19.1	1.5	12 890
apache1	7.3	4.7	7133	6.4	3.2	5660
filter3D	205.8	0.9	168 588	11.5	2.8	8347
para-5	126.8	0.7	45 136	87.9	0.8	28 084
d_pretok	50.3	1.7	23 741	179.8	0.8	82 147
turon_m	56.7	4.1	25 949	37.5	3.1	15 282
mario002	165.3	1.0	38 168	70.7	2.2	13 480
pa2010	309.6	0.7	65 791	208.5	0.5	36 194
ecology1	256.5	3.7	19 192	62.3	5.2	4172
kkt_power	490.5	1.2	15 734	568.4	1.5	15 586

TABLE 6. PERFORMANCE OF A FAST VARIANT (HDE AND WEIGHTED CENTROID REFINEMENT) AND SPEEDUP OVER THE 'EXACT' APPROACH IN TABLE 2.

Graph	Time (s)	Speedup
grid6400	0.17	22.2
crack	0.47	28.2
4elt	0.47	65.7
finance256	0.80	28.3
oilpan	1.75	80.3
apache1	1.28	43.6
filter3D	2.31	92.5
para-5	4.84	34.3
d_pretok	2.42	95.1
turon_m	2.67	131.6
mario002	5.54	58.4
pa2010	6.05	55.4
ecology1	13.55	94.9
G3_circuit	21.34	66.4
kkt_power	35.75	40.5

[13] C. Chevalier and I. Safro, "Comparison of coarsening schemes for multilevel graph partitioning," in *Learning and Intelligent Optimization*, T. Stützle, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 191–205.

[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme

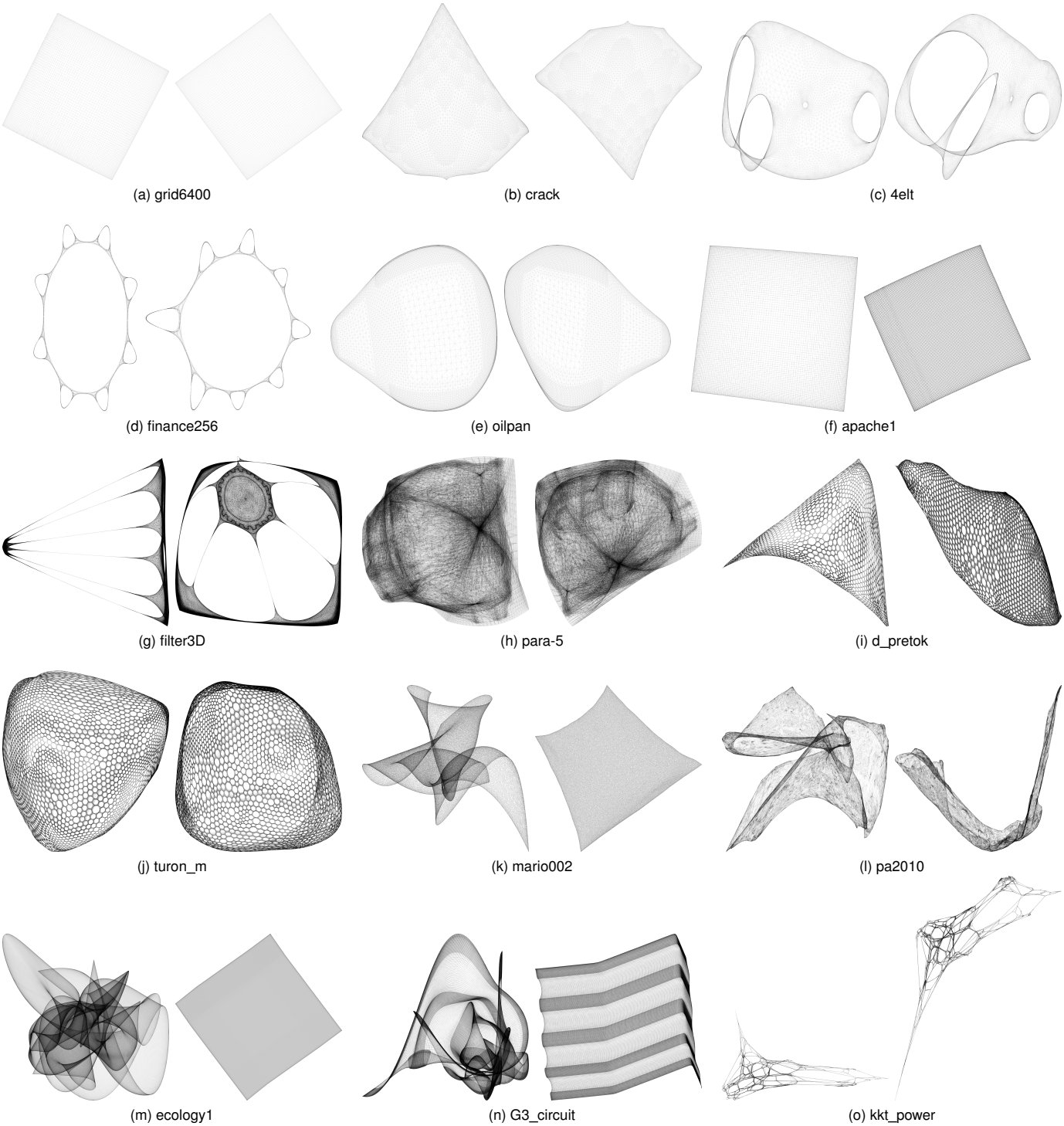


Figure 5. Drawings of all 15 test graphs using two methods: Exact (left) and HDE + weighted centroid refinement (right).

for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998, http://glaros.dtc.umn.edu/gkhome/fsroot/sw/metis/OLD, last accessed Feb 2018.

[15] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011, http://faculty.cse.tamu.edu/davis/matrices.html, last accessed Feb 2018.

[16] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[17] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr, "XSEDE: Accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.