Journal of Graph Algorithms and Applications
http://www.cs.brown.edu/publications/jgaa/
vol. 6, no. 3, pp. 203-224 (2002)

GRIP: Graph Drawing with Intelligent Placement

Pawel Gajer
Department of Computer Science
Johns Hopkins University
Baltimore, MD
http://www.math.jhu.edu/~pgajer/

Stephen G. Kobourov
Department of Computer Science
University of Arizona
Tucson, AZ
http://www.cs.arizona.edu/people/kobourov/

Abstract

This paper describes a system for Graph dRawing with Intelligent Placement, GRIP. The system is designed for drawing large graphs and uses a novel multi-dimensional force-directed method together with fast energy function minimization. The algorithm underlying the system employs a simple recursive coarsening scheme. Rather than being placed at random, vertices are placed intelligently, several at a time, at locations close to their final positions. The running time and space complexity of the system are near linear. The implementation is in C using OpenGL for 3D viewing. The GRIP system allows for drawing graphs with tens of thousands of vertices in under one minute on a mid-range PC. To the best of the authors' knowledge, GRIP surpasses the fastest previous algorithms. However, speed is not achieved at the expense of quality as the resulting drawings are quite aesthetically pleasing.

Communicated by Michael Kaufmann: submitted March 2001;
revised February 2002 and June 2002.

This research partially supported by NSF under Grant CCR–9625289. A preliminary version of this paper appeared in the Proceedings of the 8th Symposium on Graph Drawing (GD 2000).

1 Introduction

Let G be a graph $G = (V, E)$, where V is a set of vertices and E a set of edges, where $|V| = n$ and $|E| = m$. We would like to display G in two or three dimensions so as to show clearly the underlying relationships. This problem is generally known as the graph drawing problem. In a graph drawing we typically use points to represent the vertices and straight-line segments to represent the edges. The quality of the drawings produced by such algorithms is measured by a set of aesthetic criteria such as:

- minimizing the number of edge crossings
- displaying graph symmetries
- distributing the vertices evenly
- uniform edge length.

A large class of graph drawing algorithms (including ours) is based on the force-directed placement technique. The spring embedder of Eades [8] is one of the earliest examples and uses a physical model from Newtonian mechanics. In this model vertices are physical objects (steel rings) and edges are springs connecting them. An initial random placement of the vertices is repeatedly refined until a configuration with low energy has been obtained. The functions modeling the forces in these computations are typically simplified in order to speed up the computation. In Eades' spring embedder, there are two types of forces: attractive and repulsive. Attractive forces exist between vertices connected by edges and are defined by the log of the distance between them,

$$c_1 \log \frac{\mathtt{dist}_{\mathbb{R}^2}(u, v)}{c_2},$$

where c_1 and c_2 are constants and $\mathtt{dist}_{\mathbb{R}^2}(u, v)$ is the distance between vertices u and v in the drawing. Similarly, repulsive forces exist between all pairs of vertices and are defined by the distance between the vertices,

$$\frac{c_3}{\mathtt{dist}^2_{\mathbb{R}^2}(u, v)},$$

where c_3 is a constant.

Kamada and Kawai [15, 16] use a similar approach but specify explicitly the function for the total energy of the graph:

$$\sum_{1 \le u < v \le n} c_{uv}(\mathtt{dist}_{\mathbb{R}^2}(u, v) - \mathtt{dist}_G(u, v))^2,$$

where c_{uv} is a constant associated with the spring between vertices u and v, and $\mathtt{dist}_G(u, v)$ is the length of the shortest path between u and v in G. This algorithm also begins with a random placement of all the vertices. The energy

is minimized by applying a Newton-Raphson method for moving one vertex at a time.

Fruchterman and Reingold [10] present a modification of the spring embedder which yields a faster algorithm. In their method the attractive forces are defined by

$$\frac{\mathtt{dist}^2_{\mathbb{R}^2}(u,v)}{k},$$

where k is the optimum distance between two vertices, defined as $k = \sqrt{(\frac{area}{n})}$. The repulsive forces are defined as

$$\frac{k^2}{\mathtt{dist}_{\mathbb{R}^2}(u,v)}.$$

Davidson and Harel [7] introduce techniques from simulated annealing to the graph drawing process while adding more terms to the energy function to control the distance between vertices and edges. Frick *et al.* [9] present further improvements. Recently, Bruß and Frick [3] and Cruz and Twarog [6] describe several extensions of the drawing algorithms from 2D to 3D.

Most of the above algorithms concentrate on drawing small graphs, typically with 10-50 vertices and produce nice drawing in reasonable time. However, these techniques fail when directly applied to larger graphs. Direct application of these techniques to larger graphs (e.g. graphs with tens of thousands of vertices) fail due to the local nature of the optimization methods used, and the space and time complexity of these techniques. One of the first algorithms to tackle graphs with thousands of vertices is that of Hadany and Harel [12] which uses a multi-scale technique to produce a sequence of approximations to the final layout. The main idea in this approach is to create a hierarchy of graphs, in which each consecutive layer is a coarser version of the previous one. The hierarchy of graphs is created by taking into account the cluster number, the degree number, and the homotopic number.

Several new algorithms for drawing large graphs were presented at the 8th Symposium on Graph Drawing. Harel and Koren [13] present a multi-scale scheme that computes a simpler graph hierarchy. Walshaw [20] describes a similar multilevel algorithm. The n-body simulation method of Quigley and Eades [19] uses the Barnes-Hut [1] hierarchical space decomposition method. A method similar to our intelligent placement is described in the context of incremental drawing by Cohen in [4].

In the remainder of this paper we focus on the GRIP system which is based on a multi-dimensional force-directed technique.

2 The GRIP System

The GRIP system is based on the algorithm of Gajer, Goodrich, and Kobourov [11]. GRIP follows a number of force-directed drawing tools [3, 7, 9, 10, 13, 15] but employs several novel ideas first introduced in [11]: intelligent placement of vertices, drawing in higher dimensions, a fast energy minimization function, and a

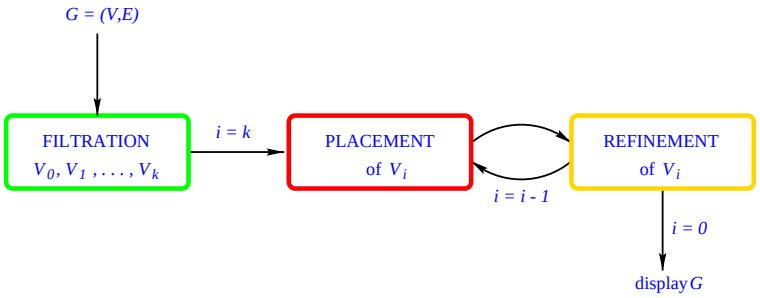

Figure 1: An overview of the algorithm. Given a graph G, the algorithm proceeds in three stages. In the first stage we create a maximal independent set (MIS) filtration. In the second and third stages we use the filtration sets $V_k, V_{k-1}, \ldots, V_0$ to repeatedly add more vertices and refine the drawing.

simple vertex filtration. Carefully put together, these techniques allow GRIP to draw graphs with tens of thousands of vertices in under one minute. While [11] contains the main methodology this paper focuses on the actual system, implementation, examples and experiments.

An overview of the system and its three main stages is given in Fig. 1 and the main algorithm is summarized in Fig. 2. Starting with a graph $G = (V, E)$, we first create a maximal independent set (MIS) or a random filtration $\mathcal{V} : V = V_0 \supset V_1 \supset \ldots \supset V_k \supset \emptyset$ of the set V of vertices of G, so that $k = O(\log n)$, and $|V_k| = 3$. A filtration $\mathcal{V}$ of V is called a *maximal independent set filtration* if V_1 is a maximal independent set of G, and each V_i is a maximal subset of V_{i-1} so that the graph distance between any pair of its elements is at least $2^{i-1} + 1$. Recall that the *graph distance* between a pair of vertices is defined as the length of the shortest path between them in the original graph G. Note that the length of a maximal independent set filtration is $\log \delta(G)$, where $\delta(G)$ is the diameter of the graph. Since $\delta(G) = O(n)$, the depth of the filtration is also $O(\log n)$. Similarly, the expected depth of the random filtration is $O(\log n)$. We ensure that the last set has exactly three elements by modifying the last one or two sets in the filtration.

Once we have obtained the desired vertex filtration, we proceed to the initial placement and refinement stages. First, we generate an initial embedding of V_k. The vertices of V_k are placed in $\mathbb{R}^n$ using their graph distances. More precisely, since $|V_k| = 3$, we find a triangle with sides equal to the graph distances between the three vertices and place the vertices at the endpoints the triangle. Then, we add the vertices of V_{k-1} that are not in V_k, placing them initially at the positions determined by their graph distances to a subset of the elements of V_k. The positions of the vertices in V_{k-1} are modified using a force-directed layout method. This process of adding new vertices and refining their positions is repeated for $V_{k-2}, \ldots, V_1, V_0$. The refined positions of the elements of V_0 constitute the final layout of the vertices of G. Note that we draw only the

```
Main Algorithm
  create a filtration V : V_0 ⊃ V_1 ⊃ ... ⊃ V_k ⊃ ∅
  for i = k to 0 do
    for each v ∈ V_i − V_{i+1} do
      find vertex neighborhood N_i(v), N_{i−1}(v), ... , N_0(v)
      find initial position pos[v] of v
    repeat rounds times
      for each v ∈ V_i do
        compute local temperature heat[v]
        disp[v] ← heat[v] · F_{N_i}(v)
      for each v ∈ V_i do
        pos[v] ← pos[v] + disp[v]
  add all edges e ∈ E
```

Figure 2: After creating the vertex filtration and setting up the scheduling function the algorithm processes each filtration set, starting with the smallest one. Here pos$[v]$ is a point in $\mathbb{R}^n$ corresponding to vertex v and rounds is a small constant. In the refinement stage heat$[v]$ is scaling factor for the displacement vector disp$[v]$, which in turn is computed over a restriction $N_i(v)$ of the vertices of G.

vertices of G up to this point. Only when all the vertices have been placed and their positions refined do we draw the edges of G as straight line segments connecting their endpoints.

3 Building MIS Filtrations

A straightforward algorithm for finding a maximal independent set filtration of a graph G is to compute the distances between all the pairs of vertices of G and then use this information to produce a maximal independent set filtration. A problem with the all-pairs shortest path algorithm is its running time of is $\Omega(nm)$ and storage complexity of $\Omega(n^2)$, e.g., see [5]. When dealing with graphs with tens of thousand of vertices, both the running time and the space complexity of the all-pairs shortest path algorithms pose serious problems.

Our solution is based on the observation that to construct a maximal independent set filtration we do not need the distances between all the pairs of vertices. Indeed, we only need the distances between the vertices in the filtration sets. Moreover, the information that has been used to construct V_i is not needed to construct V_{i+1}. Therefore, we have adopted the following "create then destroy" strategy for construction of MIS filtrations. Suppose we have already constructed set V_i. The next set in the filtration, V_{i+1} is going to contain a proper subset of the vertices in V_i. More precisely, to create V_{i+1}, we build for each vertex of V_i a breadth-first search (BFS) tree up to depth 2^i, but store in it only elements of V_i. We need to keep track of these vertices as they should

not be copied to V_{i+1}. Note that this is all we need to build V_{i+1}.

In the process of creating V_{i+1} we may need to build many BFS trees, but we destroy them immediately once they have been used, so that by the time we enter the next phase (of building V_{i+2}), all memory has been freed. Note that as i decreases, the number of vertices for which we have to perform a BFS calculation increases, but at the same time the depth to which we have to build these BFS trees decreases as well. The storage required for this strategy is

$$\max_i \sum_{v \in V_i} |\mathtt{bfs}_{2^i}(v, V_i)|,$$

where $|\mathtt{bfs}_{2^i}(v, V_i)|$ is the number of elements of V_i that belong to the BFS tree of v of depth 2^i. The time complexity for this strategy in the case of bounded degree graph G is

$$\Theta\Big(\sum_{i=0}^{k} \sum_{v \in V_i} |\mathtt{bfs}_{2^i}(v)|\Big),$$

where $|\mathtt{bfs}_{2^i}(v)|$ is the number of vertices in the BFS tree of depth 2^i for vertex v. Clearly, if we build a complete BFS tree for each vertex of G, then the running time and space complexity of this procedure, even in the bounded degree case, would be $O(n^2)$. For the above MIS filtration construction procedure however, our tests indicate that the running time is near linear as we only construct partial BFS trees and destroy them right away. In all of our experiments, the time spent creating the MIS filtration was less than 3% of the total running time, see Fig 11 and Fig 12.

We store a MIS filtration of a graph of n vertices in an array `misFiltration` of size n so that the first $|V_k|$ entries in the `misFiltration` array are the elements of V_k. The first $|V_{k-1}|$ entries in the array are the elements of V_{k-1}. Similarly, the first $|V_i|$ entries in the array are the elements of V_i. To keep track of where one set ends and another one begins we store the indices indicating the borders of different level sets of the filtration in a separate array `misBorder` of size $\log \delta(G)$, see Fig. 3. Thus the space complexity for storing a MIS filtration is $n + \log \delta(G)$. The same method can be applied to any filtration.

In GRIP we have also implemented a *random filtration*. The random filtration is similar to MIS filtrations and *k-centers filtrations* of Harel and Koren [13], except that the vertices are filtered out at random. We assign each vertex in V_i a 1/2 probability of propagation to V_{i+1}, for $i = 0, 1, \ldots, k-1$. Random filtrations take $O(n)$ time to create, as opposed to the $O(n^2)$ time required by MIS and k-centers filtrations. Random filtrations have expected depth $O(\log n)$ as opposed to $O(\log \delta(G))$ for MIS filtrations. While for most graphs we tested the random filtration produces drawings similar to those created using MIS filtrations, for sparse graphs the performance deteriorates.

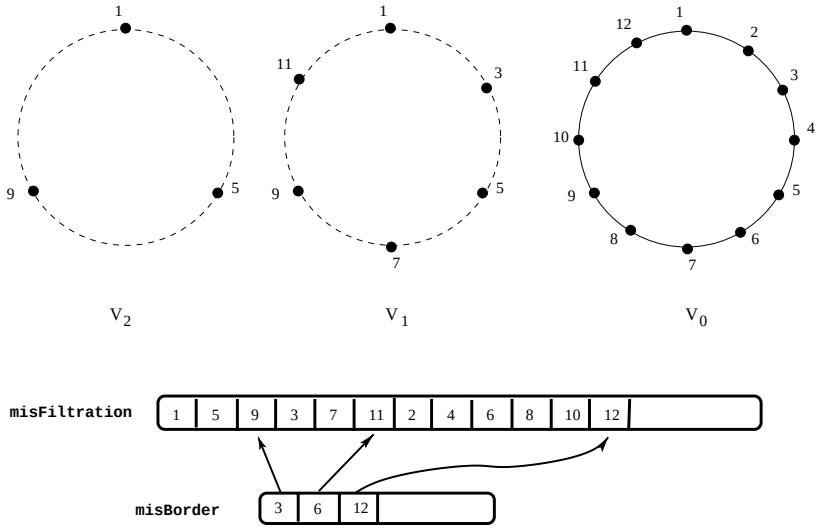

Figure 3: G is a cycle on 12 vertices and the filtrations sets are $V_0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, $V_1 = \{1, 3, 5, 7, 9, 11\}$, and $V_2 = \{1, 5, 9\}$. Note how the filtration is stored in the `misFiltration` array and how the `misBorder` array is used to identify the borders of the filtration sets. The first entry, 3, in the `misBorder` array identifies the first three elements of the `misFiltration` array as V_2. The second entry, 6, identifies the first 6 elements of the `misFiltration` array as V_1 and the third entry, 12, identifies the first 12 elements of the array as V_0.

4 Initial Placement and Refinement

The second and third phases of the algorithm are the placement and refinement stages, respectively. In the i^{th} placement stage, the vertices of set V_i are intelligently placed in $\mathbb{R}^n$. In the i^{th} refinement stage, a local force-directed method is used to obtain better positions for the vertices of V_i. After the placement and refinement stages for V_i have been completed, the process is repeated for V_{i-1}, V_{i-2}, all the way to V_1.

Recall that there are exactly three vertices in V_k. We compute their pairwise graph distances and place them at the endpoints of a triangle with sides of lengths equal to these distances. Consider the general placement case. Suppose the refinement and placement stages for V_i have been completed and we want to begin the placement phase for V_{i-1}. All the vertices in V_i are also in V_{i-1}, since $V_{i-1} \supset V_i$ as defined by the construction of the filtration. Thus we are only concerned with the placement of the vertices in V_{i-1} that are not in V_i. The idea behind the intelligent placement is that every vertex t is placed "close" to its optimal position as determined by the graph distances from t to several already placed vertices. The intuition is that if we can place the vertices close to their optimal positions from the very beginning, then the refinement stages

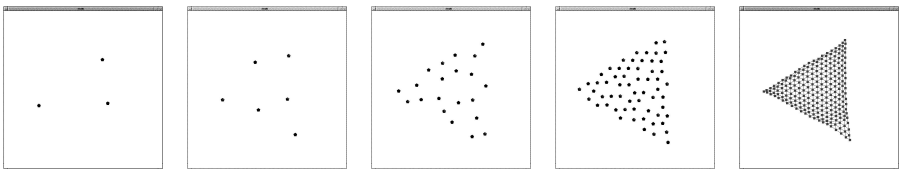

Figure 4: Drawing of the vertices in the filtration sets. Here $\mathcal{V} : V_0 \supset V_1 \supset V_2 \supset V_3 \supset V_4$. The sizes of the sets are 231, 60, 21, 6, 3, respectively. The process begins with a placement for V_4, followed by V_3, etc. Note that edges are drawn only when all the vertices are placed.

need only a few iterations of a local force-directed method to reach a minimal energy state.

After experimenting with several initial placement strategies we decided to use two simple strategies in GRIP. The first strategy, "simple barycenter" begins by setting $\texttt{pos}[t]$, the initial position for a new vertex t, to the barycenter $(\texttt{pos}[u]+\texttt{pos}[v]+\texttt{pos}[w])/3$ of u, v, and w, the three vertices closest to t that are already placed. This is followed by a force-directed modification of the position vector of t with the energy function E calculated only at the three points u, v, w. This makes the procedure very fast, and in our tests it produced good results, see Fig. 4.

The second strategy "three closest neighbors" uses the positions of the three closest already placed neighbors to determine the location of new vertex t. We begin by finding t's three closest neighbors $u, v, w \in V_i$. Since u, v and w have already been placed we can obtain a suitable place for t by solving the following system of equations for u, v, w, and t

$$\begin{cases} (x - x_u)^2 + (y - y_u)^2 = \texttt{dist}_G(u,t)^2 \\ (x - x_v)^2 + (y - y_v)^2 = \texttt{dist}_G(v,t)^2 \\ (x - x_w)^2 + (y - y_w)^2 = \texttt{dist}_G(w,t)^2, \end{cases}$$

where $\texttt{pos}[u] = (x_u, y_u)$, $\texttt{pos}[v] = (x_v, y_v)$, $\texttt{pos}[w] = (x_w, y_w)$, $\texttt{pos}[t] = (x, y)$. Since this system of equations is over-determined and may not have any solutions, we solve the following three pairs of equations instead

$$\begin{cases} \texttt{dist}_{\mathbb{R}^2}(u,t) = \texttt{dist}_G(u,t) \\ \texttt{dist}_{\mathbb{R}^2}(v,t) = \texttt{dist}_G(v,t) \end{cases}$$

$$\begin{cases} \texttt{dist}_{\mathbb{R}^2}(v,t) = \texttt{dist}_G(v,t) \\ \texttt{dist}_{\mathbb{R}^2}(w,t) = \texttt{dist}_G(w,t) \end{cases}$$

$$\begin{cases} \texttt{dist}_{\mathbb{R}^2}(u,t) = \texttt{dist}_G(u,t) \\ \texttt{dist}_{\mathbb{R}^2}(w,t) = \texttt{dist}_G(w,t). \end{cases}$$

Solving these three systems of quadratic equations, we obtain up to six different solutions. We choose the three closest to each other, call them t_1^+, t_2^+, t_3^+, and place t are their barycenter: $\texttt{pos}[t] = (t_1^+ + t_2^+ + t_3^+)/3$.

```
REFINEMENT OF V_i
  repeat rounds(i) times
    for each v ∈ V_i do
      if i > 0 then
        disp[v] ← F⃗_KK(v)
      else
        disp[v] ← F⃗_FR(v)
      heat [v] ← updateLocalTemp(v)
      disp[v] ← heat[v] · disp[v] / ||disp[v]||
    for each v ∈ V_i do
      pos[v] ← pos[v] + disp[v]
```

Figure 5: Pseudocode of the refinement phase of the algorithm.

While the refinement is calculated using a force-directed method, it is important to note that the forces are calculated locally, see Fig. 5. For each level of the filtration V_i, we perform $\texttt{rounds}(i)$ updates of the vertex positions, where $\texttt{rounds}(i)$ is a scheduling function which can be specified at the beginning of the execution. Typically, $5 \leq \texttt{rounds}(i) \leq 30$. At all levels of the filtration except the last one, the displacement vector $\texttt{disp}[v]$ of v is set to a local Kamada-Kawai force vector,

$$\overrightarrow{\texttt{F}}_{\texttt{KK}}(v) = \sum_{u \in N_i(v)} \left(\frac{\texttt{dist}_{\mathbb{R}^n}(u,v)}{\texttt{dist}_G(u,v) \cdot \texttt{edgeLength}^2} - 1 \right) (\texttt{pos}[u] - \texttt{pos}[v]).$$

In the last level of the filtration, $V_0 = V$ all the vertices have been placed. In order to speed up computation, we set the displacement vector for the last level to a local Fruchterman-Reingold force vector,

$$\overrightarrow{\texttt{F}}_{\texttt{FR}}(v) = \sum_{u \in \texttt{Adj}(v)} \frac{\texttt{dist}_{\mathbb{R}^n}(u,v)^2}{\texttt{edgeLength}^2} (\texttt{pos}[u] - \texttt{pos}[v]) +$$
$$+ \sum_{u \in N_i(v)} s \frac{\texttt{edgeLength}^2}{\texttt{dist}_{\mathbb{R}^n}(u,v)^2} (\texttt{pos}[v] - \texttt{pos}[u]),$$

Here, $\texttt{dist}_{\mathbb{R}^n}(u,v)$ is the Euclidean distance between $\texttt{pos}[u]$ and $\texttt{pos}[v]$, and $\texttt{dist}_G(u,v)$ is the graph distance between u and v. In the above equations, $\texttt{edgeLength}$ is the unit edge length, $\texttt{Adj}(v)$ is the set of vertices adjacent to v, and s is a small scaling factor which is set to 0.05 in our program. Note that for a vertex $v \in G$ the force calculation is performed over a restriction $N_i(v)$ of the vertices of G. Each vertex neighborhood $N_i(v)$ contains a constant number of vertices closest to v which belong to V_i. Thus only a constant number of vertices which are near vertex v are used to refine v's position. This is why we call this type of force calculation local.

```
updateLocalTemp(v)
  if ||disp[v]|| ≠ 0 and ||oldDisp[v]|| ≠ 0 then
      cos[v] = disp[v] * oldDisp[v] / (||disp[v]|| * ||oldDisp[v]||)
      r = 0.15, s = 3
      if oldCos[v] * cos[v] > 0 then
          heat[v] = heat[v] + (cos[v] * r * s)
      else
          heat[v] = heat[v] + (cos[v] * r)
      oldCos[v] = cos[v]
```

Figure 6: Pseudocode of the local temperature calculation.

The local temperature heat[v] of v is a scaling factor of the displacement vector for vertex v, similar to that of Fruchterman and Reingold [10] and Frick *et al* [9]. The algorithm for determining the local temperature is in Fig. 6. To speed up the calculation, we maintain two auxiliary arrays oldDisp and oldCos, where oldDisp[v] is the previous displacement vector for v, and oldCos[v] is the previous value of the cosine of the angle between oldDisp[v] and disp[v]. When a displacement vector of v is calculated for the first time, heat[v] is set to a default value edgeLength/6. The local temperature helps speed up convergence by distinguishing between oscillating vertices and vertices that continue moving in one directions. There are three cases for determining the local temperature :

1. if either oldDisp[v] or disp[v] is a zero vector, then the value of heat[v] does not change;
2. if v is oscillating around some stationary point we add to it a factor ($\cos[v] * r * s$);
3. in all other cases we add a factor of ($\cos[v] * r$).

5 Implementation

The GRIP system was originally written in C++ and then re-done in C with OpenGL, with a Tcl/Tk interface, see Fig. 7. GRIP is available for download at http:\\www.cs.arizona.edu/~kobourov/GRIP. The system can read in files and generates several typical classes of graphs parametrized by their number of vertices, e.g. paths, cycles, square meshes, triangular meshes, and complete graphs. GRIP also contains generators for complete n-ary trees, random graphs with parametrized density, and knotted triangular and rectangular meshes. Different types of tori, as well as cylinders and Moebius bends can be generated with parametrized thickness and length. Finally, Sierpinski graphs in 2 and 3

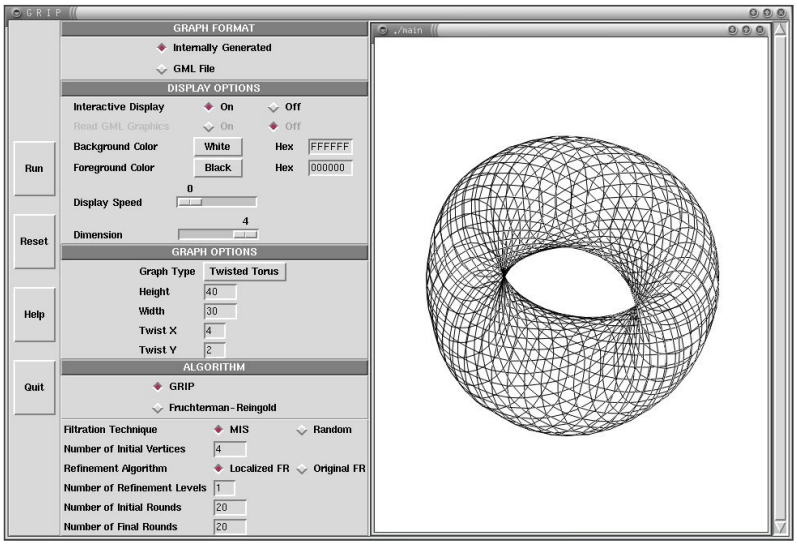

Figure 7: The GRIP system: On the top left we have the control window, which lets us chose the type of graph, the graph parameters, dimension of the drawing, parameters of the algorithm, and display settings. Graphs can be read from a file, or one of the several built-in generators can be used. Currently we support generators for trees, paths, cycles, cylinders, tori, degree 4 meshes, degree 6 meshes, and Sierpinski graphs. Parameters of the algorithm include attractive/repulsive force ratio and initial and final number of rounds, among others. The display settings control the interactive display, speed, and color. Within the OpenGL window the graph can be dragged, rotated, and zoomed.

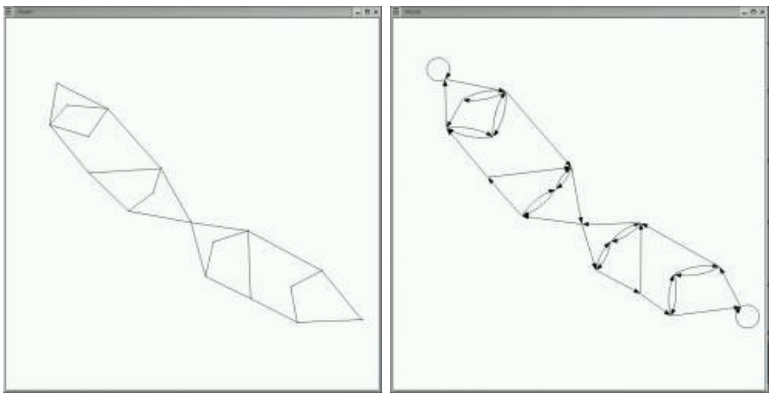

Figure 8: GRIP can read files in the gml format. This allows for displaying properties such as direction of edges, self-loops and colors specified for vertices and edges.

dimensions (Sierpinski triangles and Sierpinski pyramids, respectively) are also available.

In addition to the set of graphs that GRIP can generate, other graphs can be read from a file in gml format [14]. This allows the display of options such as directed edges, different color vertices and edges, self-loops, etc., see Fig. 8. GRIP can draw graphs directly in 2D or 3D or projected down from higher dimensions. Drawings produced in higher dimensions and projected down to 2D or 3D usually produce better results. At this time, the projections are done at random. We are working on incorporating projections based on spectral analysis similar to [2].

Figures 9-10 show how the MIS filtration method compares to the random filtration method. In both the final drawing for each level of the filtration is captured. Note that all but the last drawings show only vertices as the induced graphs are never computed. For a given filtration level V_i, the vertices in $V_i \setminus V_{i-1}$ are shown as bigger dots, while those that came down from the previous level (V_{i-1}) are shown as small dots. While random filtrations take a small fraction of the time required to create MIS filtrations in general they induce more refinement levels. Most drawings produced with random filtrations and MIS filtrations are similar, see Fig. 9. Random filtrations perform worse for sparse graphs as the variation in the distances between vertices in the same level becomes greater, see Fig. 10.

Running times for the MIS creation and for the entire drawing process as shown in Fig 11 and Fig. 12. Most of the drawings in the following examples took less than 1 second. The Sierpinski pyramid of order 8 (with 32,770 vertices, 196,608 edges) was the most time consuming: it took 58 seconds on a 500MHz Pentium III machine.

The parameters discussed in this paper can be changed via GRIP's interface, thus allowing for experimentation with different scheduling functions, scaling parameters, filtrations, etc. There are controls for the drawing dimension and the drawing speed. The drawings produced by default are three dimensional, interactive, and use color and shading to aid three dimensional perception. For faster results, the interactive display can be turned off so that only the final drawing is shown. The size and colors of the vertices and edges can also be modified.

6 Examples

In the next few pages we provide several examples of graphs produced by GRIP. None of the drawings have been additionally modified. We focus mostly on larger graphs but also provide several "classical" smaller graphs to illustrate the versatility of the system. Fig. 13 shows drawings of a dodecahedron, C60 (bucky ball), and a 3D cube mesh. Fig. 14 shows 3D drawings of a 4D cube, 5D cube, and 6D cube. Fig. 15 shows binary, 3-ary, and 4-ary trees. Fig. 16 shows several cycles. Fig. 17 shows three "real-world" graphs from the Stanford GraphBase [17]. Fig. 18 shows regular degree 4 meshes of up to 10,000 vertices.

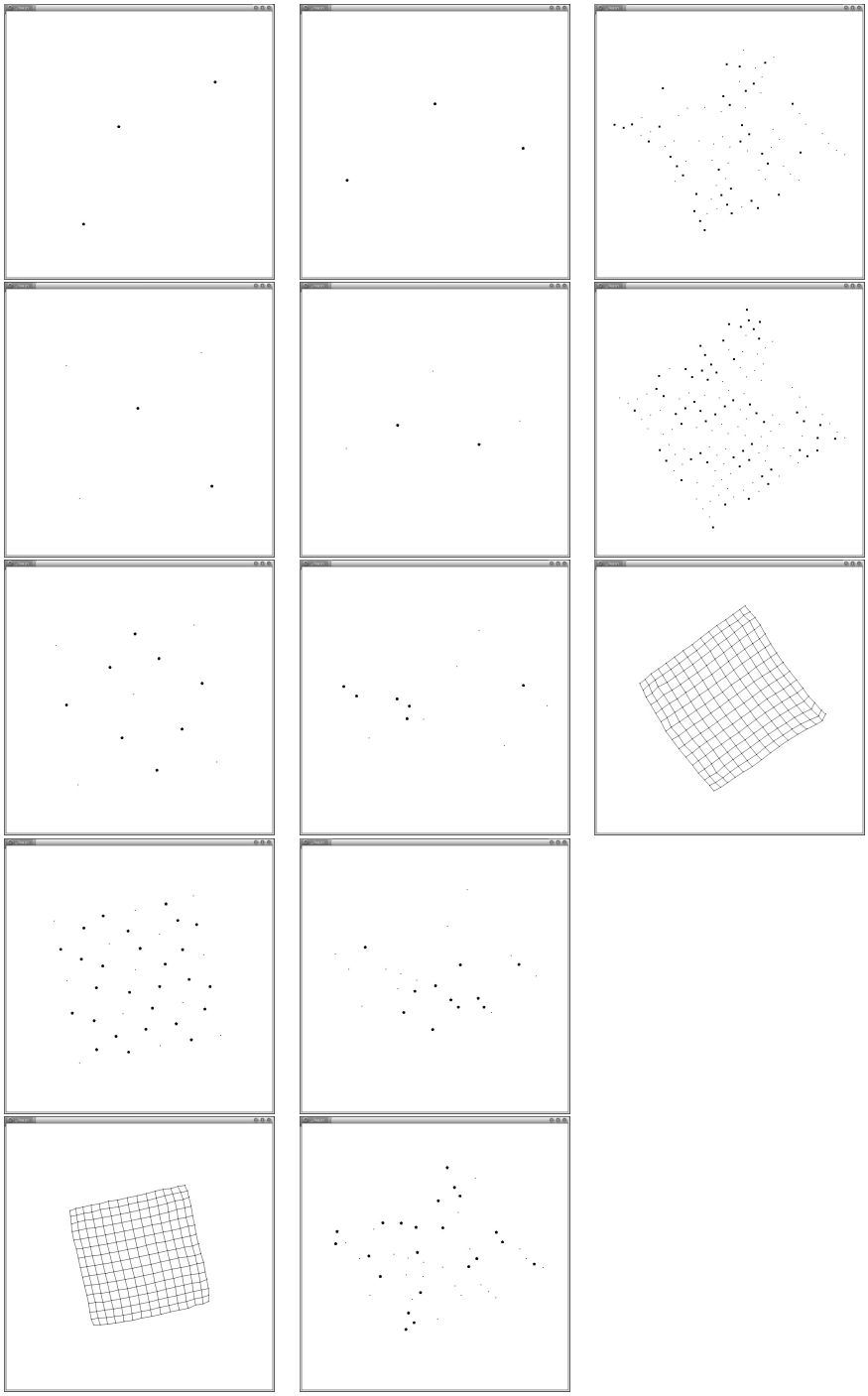

Figure 9: Mesh on 225 vertices using MIS filtration (1st column) and random filtration (2nd and 3rd columns).

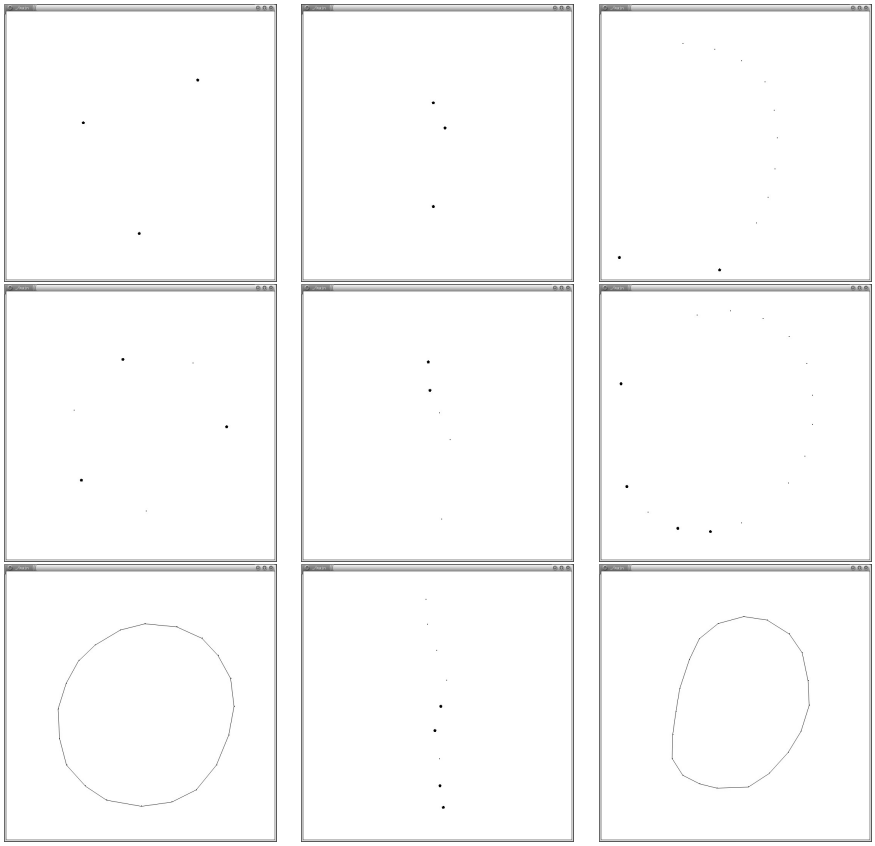

Figure 10: Cycle on 20 vertices using MIS filtration (1st column) and random filtration (2nd and 3rd columns).

Fig. 19 shows drawings of the same meshes with their endpoints attached (knotted meshes). Fig. 20 shows several cylinders and Fig. 21 shows tori. Fig. 22 shows regular degree 6 meshes and Fig. 23 shows the same meshes with their endpoints attached (knotted meshes).

The Sierpinski triangle (pyramid) is a classic fractal [18]. Several 2D, 3D, and 4D drawings are shown on Fig. 24 and Fig. 25. Whereas traditionally the image is defined with fixed vertices and edges, we make ours a fractal graph with no specific embedding. The Sierpinski pyramid graph is created by a recursive procedure, parametrized on the order of the recursion. As in the 2-D case, at each iteration, every pyramid is divided into five congruent smaller pyramids with the central pyramid removed. In a Sierpinski pyramid of order k the number of vertices is $|V_k| = \frac{4^k}{2} + 2$ and the number of edges is $|E| = 6(|V| - 4) + 12 = 3 \times 4^k$. Fig. 24(c) shows a Sierpinski pyramid of order 7 with 8,194 vertices and 49,152 edges. Fig. 25 shows a Sierpinski pyramid of order 8 with 32,770 vertices and 196,608 edges. Given the parameter k we generate

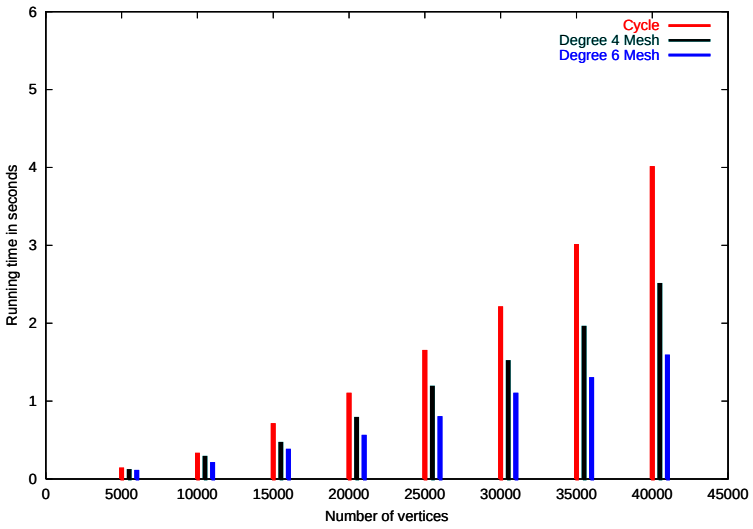

Figure 11: The chart contains the running times for construction of a MIS filtration for cycles, meshes of degree 4, and meshes of degree 6. As can be expected, the depth of the filtration is the largest for the sparsest graphs, in this case, the cycles.

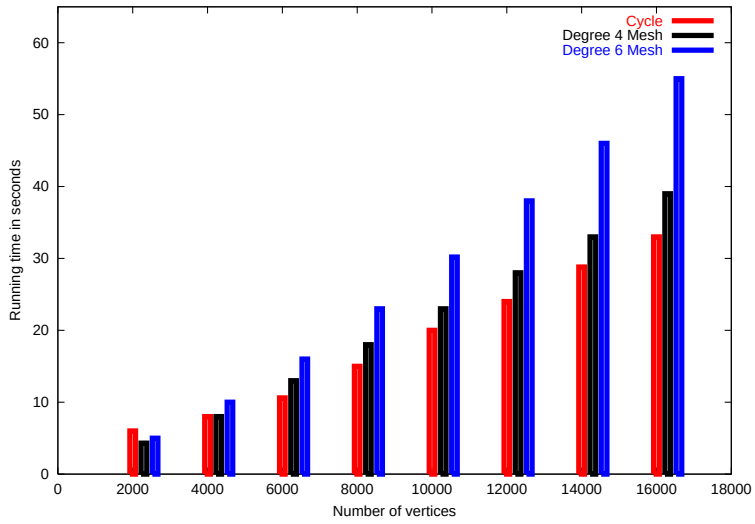

Figure 12: The chart contains the total running time for the three classes of graphs from Fig 11. Note that the construction of the MIS filtration takes less than 3% of the total running time.

the adjacency matrix of a graph which corresponds to the Sierpinski pyramid of order k. The drawings have been taken directly from the output of GRIP, without any modifications.

7 Conclusion and Future Work

In the process of writing this program several interesting questions arose. Some we answered in this paper, others we address in [11] but quite a few still remain:

- How should the type of the graph affect the number of rounds?
- What can we do about dense graphs and graphs with small diameter? (The MIS filtration is very shallow for such graphs.)
- What filtrations (other than MIS) could produce good results?
- Can the MIS filtration be created in provable subquadratic time, in the number of vertices and edges of the graph? (Currently, we have a modified MIS filtration, with similar properties which can be built in linear time. However, we are not aware of a subquadratic algorithm for the creation of a standard MIS filtration, as defined in the paper.)

8 Acknowledgements

We would like to thank Michael Goodrich, Christian Duncan, and Alon Efrat for helpful discussions. The conversion from C++ to C is due to Roman Yusufov who also wrote the user manual.

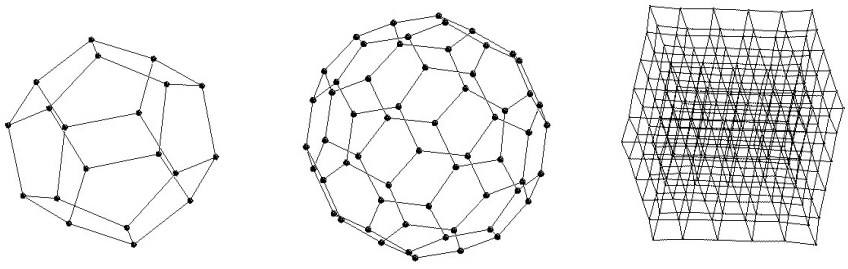

Figure 13: Small graphs: (a) dodecahedron (20 vertices); (b) C60 – bucky ball (60 vertices); (c) 3D cube mesh (216 vertices).

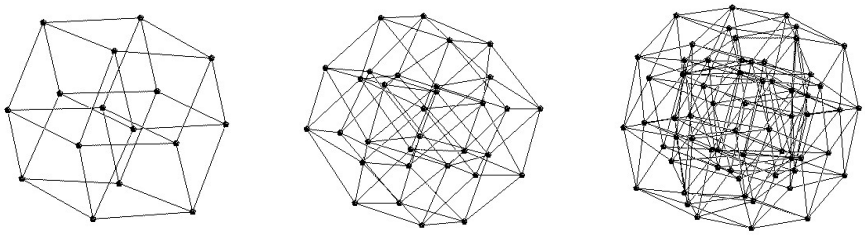

Figure 14: Cubes in 3D: (a) a 4D cube (16 vertices); (b) 5D cube (32 vertices); (c) 6D cube (64 vertices).

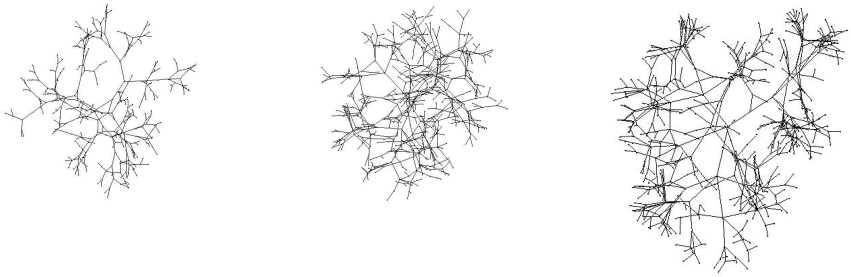

Figure 15: Trees: (a) a complete binary tree of depth 9 (511 vertices); (b) complete 3-ary tree of depth 7 (1093 vertices); (c) complete 4-ary tree of depth 6 (1365 vertices).

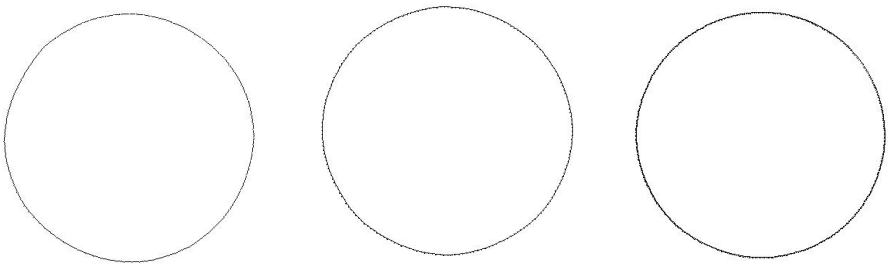

Figure 16: Cycles of 100, 200, and 400 vertices.

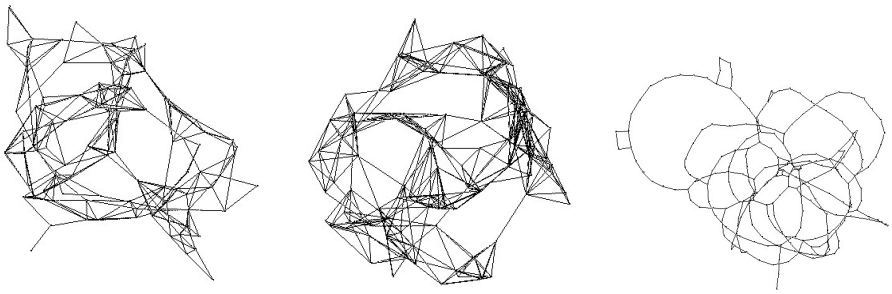

Figure 17: Graph from "real-world data", as given by Knuth [17]: (a) miles2, (128 vertices, 368 edges); (b) miles3 (128 vertices, 518 edges) (c) miles4 (256 vertices, 312 edges

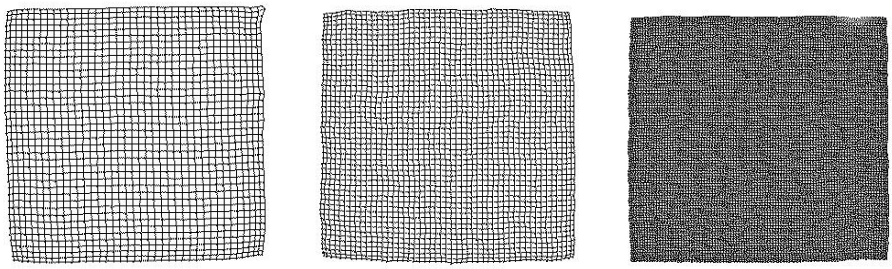

Figure 18: Rectangular (degree 4) meshes of 1600, 2500, and 10000 vertices.

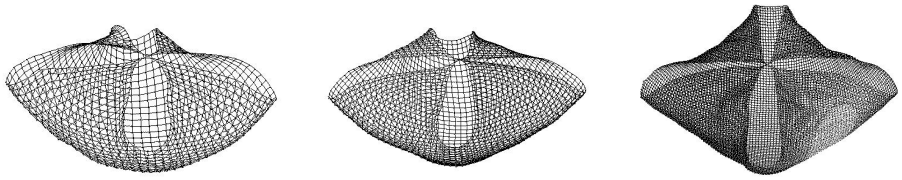

Figure 19: Knotted rectangular (degree 4) meshes of 1600, 2500, and 10000 vertices.

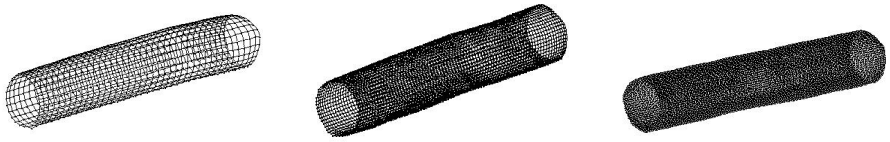

Figure 20: Cylinders of 1000, 4000, and 10000 vertices.

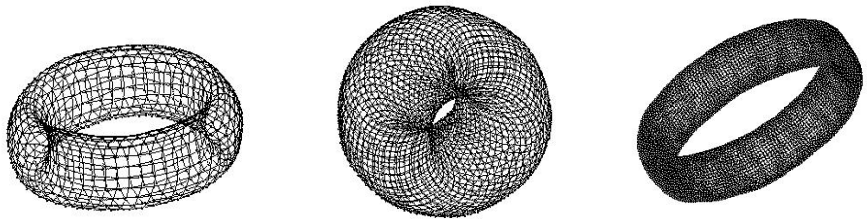

Figure 21: Tori of various length and thickness: 1000, 2500, and 10000 drawn in four dimensions and projected down to three dimensions.

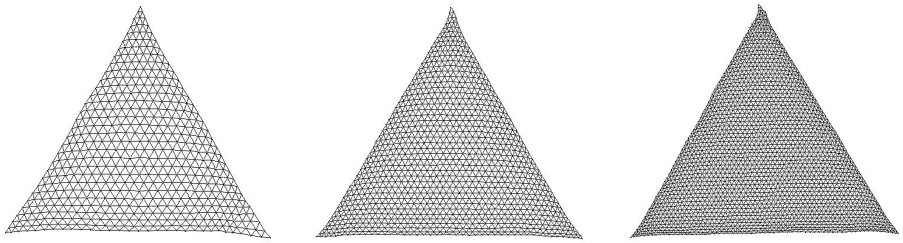

Figure 22: Triangular (degree 6) meshes of 496, 1035, and 2016 vertices.

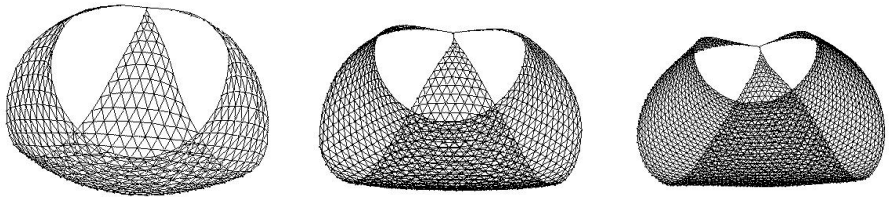

Figure 23: Knotted triangular (degree 6) meshes of 496, 1035, and 2016 vertices.

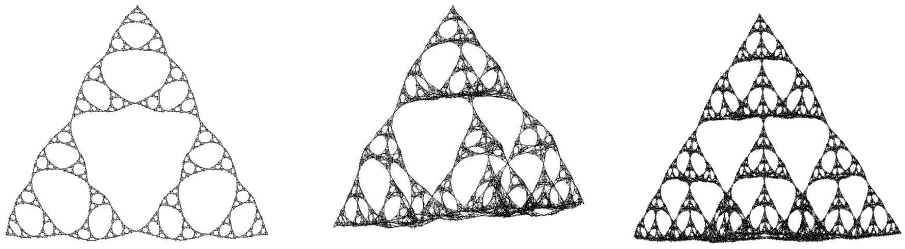

Figure 24: Sierpinski graphs in 2D and 3D (a) 2D Sierpinski of order 7 (1,095 vertices); (b) 3D Sierpinski of order 6 (2,050 vertices); (c) 3D Sierpinski of order 7 (8,194 vertices). The last drawing took 15 seconds on a 500Mhz Pentium III machine.

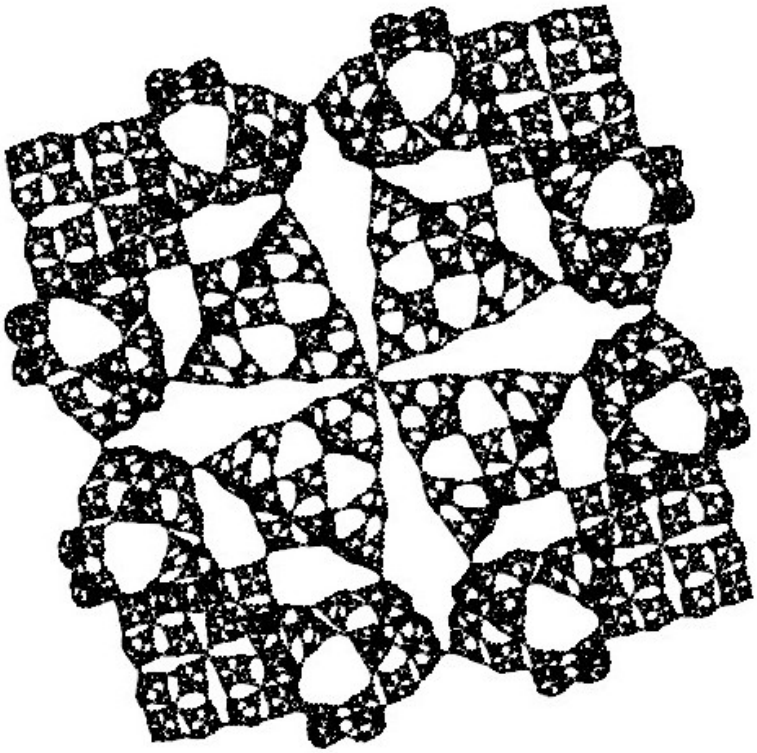

Figure 25: 4D Sierpinski graph of order 8 on 32,770 vertices and 196,608 edges. The drawing took 58 seconds on a 500Mhz Pentium III machine.

References

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, 324:446–449, Dec. 1986. Institute for Advanced Study, Princeton, New Jersey.

[2] Brandes and Cornelsen. Visual ranking of link structures. In *WADS: 7th Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 222–232, 2001.

[3] I. Bruß and A. Frick. Fast interactive 3-D graph visualization. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Computer Science*, pages 99–110. Springer-Verlag, 1996.

[4] J. D. Cohen. Drawing graphs to convey proximity: An incremental arrangement method. *ACM Transactions on Computer-Human Interaction*, 4(3):197–229, Sept. 1997.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[6] I. F. Cruz and J. P. Twarog. 3d graph drawing with simulated annealing. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Computer Science*, pages 162–165, 1996.

[7] R. Davidson and D. Harel. Drawing graphics nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

[8] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[9] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, LNCS 894, pages 388–403, 1995.

[10] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.

[11] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A multi-dimensional approach to force-directed layouts. In *Proceedings of the 8th Symposium on Graph Drawing (GD 2000)*, pages 211–221, 2000.

[12] R. Hadany and D. Harel. A multi-scale algorithm for drawing graphs nicely. In *Proc. 25th International Workshop on Graph Teoretic Concepts in Computer Science (WG'99)*, 1999.

[13] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *Proceedings of the 8th Symposium on Graph Drawing (GD 2000)*, pages 183–196, 2000.

[14] M. Himsolt. Gml: A portable graph file format. Technical report, Universität Passau, 1994.

[15] T. Kamada and S. Kawai. Automatic display of network structures for human understanding. Technical Report 88-007, Department of Information Science, University of Tokyo, 1988.

[16] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.*, 31:7–15, 1989.

[17] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, NY 10036, USA, 1993.

[18] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Co., New York, rev 1983.

[19] A. Quigley and P. Eades. FADE: graph drawing, clustering, and visual abstraction. In *Proceedings of the 8th Symposium on Graph Drawing (GD 2000)*, pages 197–210, 2000.

[20] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proceedings of the 8th Symposium on Graph Drawing (GD 2000)*, pages 171–182, 2000.