Numerical Optimization-Based Graph Drawing Revisited

Hiroshi Hosobe*
National Institute of Informatics, Japan

ABSTRACT

We present a numerical optimization-based method for visualizing undirected graphs. Our method is a variant of force-directed graph drawing, and has sufficient generality to adopt different basic force models including those for the Kamada-Kawai and Fruchterman-Reingold methods. To achieve efficiency, we use the L-BFGS method for numerical optimization. Our experimental results show that the method gives good performance when combined with the Kamada-Kawai model.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms

1 INTRODUCTION

Graphs provide abstract relationships between objects. In a graph, an object is represented as a vertex, and a relationship between two objects is expressed as an edge that connects two vertices corresponding to the objects. Graphs are categorized into classes such as trees, directed graphs, and undirected graphs, based on their structures and properties associated with vertices and edges. Visualizations of graphs are often useful for intuitively capturing overall structures of the graphs. Therefore, researchers have been studying automatic methods for graph drawing. Such methods are usually developed for particular classes of graphs.

Force-directed methods are often used for drawing undirected graphs. A force-directed method builds from a graph a (pseudo-)physical system of vertices that attract/repel each other by force, and computes an equilibrium state of the system. In general, a force-directed method consists of two key components, a *model* and an *algorithm* [7]; the model defines how to calculate forces, and the algorithm gives the way to actually compute the equilibrium state.

Most of the existing force-directed methods can be viewed as using simulation algorithms. Such an algorithm performs discrete-time simulation of a pseudo-physical system. At each time step, forces exerted on vertices are calculated based on the force model, and then each vertex is moved in the force direction (which assumes the pseudo-physical law $m\mathbf{v} = \mathbf{F}$ instead of Newton's second law $m\mathbf{a} = \mathbf{F}$). To speed up simulation, researchers have been doing considerable work. One common technique is a multi-level method that uses coarse approximations of graphs [2, 13, 19, 30], and another common technique is a tree-code method that reduces force calculation by approximately processing interactions between remote vertices [13, 19, 22, 27].

Numerical optimization [25] is also used for force-directed graph drawing. The Kamada-Kawai method [20] is perhaps the best known in this category. It attaches a set of springs to vertices, and computes the equilibrium state by minimizing the total energy of the springs, which is done by repeatedly applying Newton's method to a selected vertex at a time. Tunkelang [28, 29] developed another numerical optimization method that used a conjugate gradient method. He also pointed out that simulation algorithms used in most of the force-directed methods could be regarded as steepest descent, a straightforward method of numerical optimization. This suggests that numerical optimization approaches to force-directed methods may be promising. However, there have not been many algorithms proposed.

In this paper, we revisit such a numerical optimization approach to force-directed graph drawing. We present a method that is simple but has the following characteristics:

- It has sufficient generality to adopt different basic force models used in previous simulation and optimization methods;
- It uses the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method [24] for efficient numerical optimization;
- It runs at reasonably high speed without using multi-level nor tree-code methods.

Our method is based on the general combination of conservative forces and a scalar potential, and is applicable to different force models using conservative forces. In fact, we show that four basic models, namely Kamada-Kawai [20], Hooke-Coulomb [7], Eades [10], and Fruchterman-Reingold [11], can be incorporated into our method. The L-BFGS method that we use for numerical optimization is an efficient quasi-Newton method that reduces the necessary memory as well as does not require second partial derivatives. Our experimental results show that our method provides good performance especially when combined with the Kamada-Kawai model.

The rest of this paper is organized as follows. After presenting related work in Section 2, we provide preliminary introductions to quasi-Newton methods and the L-BFGS method in Section 3. Next, in Section 4, we provide how to construct our method, and then its implementation in Section 5. In Section 6, we present the results of experiments on our method. After discussing our work in Section 7, we describe conclusions and future work in Section 8.

2 RELATED WORK

Researchers have been studying methods for force-directed graph drawing. Although most of those methods use simulation algorithms (i.e., steepest descent), there are other methods that use numerical optimization approaches. As described in Section 1, the Kamada-Kawai (KK) method [20] is a well-known graph drawing method based on numerical optimization; it uses Newton's method to solve the optimization problem. The same force model as KK's is adopted by the stress majorization method [12], which is an optimization algorithm specialized in the KK model and is usually more efficient than the KK method. Tunkelang [28, 29] developed a graph drawing method using a conjugate gradient method and a variant of the Fruchterman-Reingold force model [11]; his experiments showed that his method was several-times faster than a steepest descent method.

Force-directed methods such as KK are known to be closely related to multidimensional scaling (MDS). Therefore, efficient numerical techniques developed for MDS have been applied to graph drawing [3, 4]. MDS is also used to enable interactive graph drawing [17].

It is possible to restrict force models to obtain problems that can be efficiently handled. Introducing linearly-defined forces derives

*e-mail: hosobe@acm.org

This is the author's version. The final authenticated version is available online at https://doi.org/10.1109/PacificVis.2012.6183577.

Tutte's barycenter method [7]. Methods such as ACE [21] and SDE [5] achieve efficient algorithms by using special formulations of graph drawing that result in eigenvector calculation. A similar approach is adopted by the method using high-dimensional embedding [15].

There have been attempts to obtain more expressive force models. In [6], a method using simulated annealing was proposed and applied to a general force model that handled the boundary of the layout area and the number of edge crossings. In [23], a force model adopting edge repulsion was proposed to better handle graphs with nonuniform degrees.

Combining constraints with graph drawings is another important research direction. The Chorus constraint solver [16] handles graph layout constraints based on the KK model by using the combination of the BFGS method for local search and a genetic algorithm for global search. In [8], stress majorization was extended to support a certain class of linear constraints. In [9], a constrained graph layout method was proposed to handle topology constraints. In [26], mixed integer programming was adopted to handle graph drawings with constraints.

3 Preliminaries

This section provides preliminary introductions to quasi-Newton methods and the L-BFGS method.

3.1 Quasi-Newton Method

Let $\mathbf{p}$ be an n-dimensional real vector, and f be a real-valued two-times partially differentiable function over an n-dimensional real domain. Then minimizing $f(\mathbf{p})$ requires $\nabla f(\mathbf{p}) = \mathbf{0}$, where $\nabla f(\mathbf{p})$ indicates the gradient (or first partial derivative) of f at $\mathbf{p}$.

Let $\mathbf{p}_k$ be the approximate solution at step k. Consider the following linear approximation of ∇f around $\mathbf{p}_k$:

$$\nabla f(\mathbf{p}_{k+1}) = \nabla f(\mathbf{p}_k) + B(\mathbf{p}_k)(\mathbf{p}_{k+1} - \mathbf{p}_k),$$

where $B(\mathbf{p}_k)$ indicates the Hessian matrix (or second partial derivative) of f at $\mathbf{p}_k$. Assuming $\nabla f(\mathbf{p}_{k+1}) = 0$ in the linear approximation, Newton's method iteratively solves the following for $\mathbf{p}_{k+1}$:

$$B(\mathbf{p}_k)(\mathbf{p}_{k+1} - \mathbf{p}_k) = -\nabla f(\mathbf{p}_k).$$

A quasi-Newton method [25] further introduces an approximation B_k of $B(\mathbf{p}_k)$. After obtaining $\mathbf{p}_{k+1}$ by using B_k, it computes B_{k+1} from B_k and other known information by adopting an update formula such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula.

To obtain a better convergence, such Newton-based methods usually adopt a line search technique. It updates $\mathbf{p}_{k+1}$ by computing

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \alpha(B(\mathbf{p}_k))^{-1}\nabla f(\mathbf{p}_k)$$

with an appropriate real α. The vector $-(B(\mathbf{p}_k))^{-1}\nabla f(\mathbf{p}_k)$ is called a search direction.

3.2 L-BFGS Method

The limited-memory BFGS (L-BFGS) method [24] is a quasi-Newton method that uses a variant of the BFGS formula. Its advantage is that it reduces the necessary memory because it needs only several vectors to compute the search direction $-B_k^{-1}\nabla f(\mathbf{p}_k)$ (without storing B_k nor B_k^{-1}). Let $\mathbf{g}_k = \nabla f(\mathbf{p}_k)$, $\Delta\mathbf{p}_k = \mathbf{p}_{k+1} - \mathbf{p}_k$, $\Delta\mathbf{g}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$, and $\rho_k = 1/((\Delta\mathbf{g}_k)^{\mathrm{T}}\Delta\mathbf{p}_k)$. Also, let H_k^0 be the initial approximation of the inverse Hessian matrix B_k^{-1} (which is typically a diagonal matrix). Assume that information about the past h (often less than 10) iterations is used. Then the search direction is obtained by the following two steps:

1. Let $\mathbf{q} = \mathbf{g}_k$. For each $i = k-1, \ldots, k-h$, compute $\alpha_i = \rho_i(\nabla\mathbf{p}_i)^{\mathrm{T}}\mathbf{q}$ and $\mathbf{q} = \mathbf{q} - \alpha_i\Delta\mathbf{g}_i$;

2. Let $\mathbf{r} = H_k^0\mathbf{q}$. For each $i = k-h, \ldots, k-1$, compute $\beta_i = \rho_i(\Delta\mathbf{g}_i)^{\mathrm{T}}\mathbf{r}$ and $\mathbf{r} = \mathbf{r} + (\alpha_i - \beta_i)\Delta\mathbf{p}_i$.

After these steps we have $B_k^{-1}\nabla f(\mathbf{p}_k) = \mathbf{r}$.

4 Our Method

This section presents our numerical optimization-based method for force-directed graph drawing. After introducing necessary notations, we describe basics for our method, then how to treat four basic force models in our method, and the underlying numerical optimization algorithm.

Notations. Let $G = (V, E)$ be a graph with a set $V = \{1, \ldots, n\}$ of n vertices and a set $E \subseteq V \times V$ of m edges. We assume only undirected graphs; i.e., $(i, j) \in E$ always implies $(j, i) \in E$. Let $\mathbf{p}_i = (x_i, y_i)$ for each vertex $i \in \{1, \ldots, n\}$ be the position of i. Then a drawing of G is represented as $\mathbf{p} = (x_1, y_1, \ldots, x_n, y_n)$. We also let $\mathbf{p}_{ij} = \mathbf{p}_j - \mathbf{p}_i$. In this paper, we focus on two-dimensional drawings of graphs, but our results will be easily extended to three- or higher-dimensional cases.

4.1 Basics

In a numerical optimization approach such as the Kamada-Kawai method [20], it is typically necessary to introduce an energy that should be minimized. Mathematically, we can explain numerical optimization approaches from the viewpoint of conservative forces and a scalar potential. Let each $\mathbf{F}_i = (F_i^x, F_i^y)$ be the force exerted on vertex i, and let $\mathbf{F} = (F_1^x, F_1^y, \ldots, F_n^x, F_n^y)$. If forces are conservative, we can find a scalar potential Φ that has the following relationship with $\mathbf{F}$:

$$\mathbf{F} = -\nabla\Phi, \tag{1}$$

where $\nabla\Phi$ indicates the gradient of Φ, i.e.,

$$\nabla\Phi = \left(\frac{\partial\Phi}{\partial x_1}, \frac{\partial\Phi}{\partial y_1}, \ldots, \frac{\partial\Phi}{\partial x_n}, \frac{\partial\Phi}{\partial y_n}\right).$$

Equation (1) means that $\mathbf{F}$ is directed toward the steepest descent direction of Φ, and thus an equilibrium state $\mathbf{F} = \mathbf{0}$ is achieved by a local minimum of Φ, i.e.,

$$\mathbf{F} = -\nabla\Phi = \mathbf{0}.$$

Therefore we can find a drawing $\mathbf{p}$ of G by solving the optimization problem with Φ as its objective function, i.e.,

$$\operatorname*{argmin}_{\mathbf{p}} \Phi. \tag{2}$$

In other words, if conservative forces are used, force-directed graph drawing can be reformulated as an optimization problem. It should also be noted that solving (2) does not require steepest descent methods. Instead, more sophisticated numerical optimization methods may be used to solve the generalized problem (2), which we do in this paper.

4.2 Force Models

Next, we show that four basic force models use conservative forces and allow defining scalar potentials. These force models are Kamada-Kawai, Hooke-Coulomb, Eades, and Fruchterman-Reingold.

This is the author's version. The final authenticated version is available online at https://doi.org/10.1109/PacificVis.2012.6183577.

Kamada-Kawai. The Kamada-Kawai (KK) method [20] is perhaps the most popular among those using numerical optimization. The force model of KK, called the spring model, is easy for our method to incorporate since KK explicitly provides a scalar potential in terms of the spring energy.

KK assumes a connected graph, and attaches a spring to any pair of vertices (both adjacent and non-adjacent). Each spring for vertices i and j is assigned the natural length l_{ij} that is the graph-theoretic distance between i and j. The spring constant for the spring between i and j is $1/l_{ij}^2$. Then the force $\mathbf{F}_i$ exerted on each i is calculated by using Hooke's law:

$$\mathbf{F}_i = -\sum_{j\neq i} \frac{|\mathbf{p}_{ij}| - l_{ij}}{l_{ij}^2} \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|}.$$

Then, as known in classical mechanics, the scalar potential Φ is calculated by the following:

$$\Phi = \sum_i \sum_{j>i} \frac{(|\mathbf{p}_{ij}| - l_{ij})^2}{2 l_{ij}^2}.$$

It should be noted that we no longer need the second partial derivatives of Φ, which is partly needed in the original KK method; we will review this point in Subsection 4.3.

Hooke-Coulomb. The combination of springs and electrical forces also derives a force-directed method. In the simplest form (although it seems not to be widely used), such a method can be formulated using laws from classical physics, i.e., Hooke's law for springs and Coulomb's law for electrical forces [7]. Let us call this method Hooke-Coulomb (HC). Unlike KK, HC attaches springs only to adjacent pairs of vertices, and instead introduces repulsive electrical forces for non-adjacent pairs of vertices as follows:

$$\mathbf{F}_i = -\sum_{(i,j)\in E} k_1 (|\mathbf{p}_{ij}| - l_{ij}) \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|} + \sum_{j\neq i} \frac{k_2}{(|\mathbf{p}_{ij}| + \varepsilon)^2} \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|},$$

where l_{ij} is the natural spring length for edge (i,j), k_1 and k_2 are certain constants, and ε is a softening parameter (small positive number) to avoid numerical failure when $\mathbf{p}_{ij}$ is nearly zero. We can compute the scalar potential Φ by the following:

$$\Phi = \sum_{(i,j)\in E\wedge j>i} \frac{k_1 (|\mathbf{p}_{ij}| - l_{ij})^2}{2} + \sum_i \sum_{j>i} \frac{k_2}{|\mathbf{p}_{ij}| + \varepsilon}.$$

Eades. Eades proposed a classical force-directed method called the spring embedder [10]. As HC does, it uses the combination of springs and electrical forces. However, it introduces logarithmic-strength springs instead of ordinary springs based on Hooke's law. Specifically, the forces in Eades's model are calculated as follows:

$$\mathbf{F}_i = -\sum_{(i,j)\in E} \left(k_1 \log \frac{|\mathbf{p}_{ij}|}{l_{ij}} \right) \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|} + \sum_{j\neq i} \frac{k_2}{(|\mathbf{p}_{ij}| + \varepsilon)^2} \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|},$$

where l_{ij} is the natural spring length for edge (i,j), k_1 and k_2 are certain constants (e.g., $l_{ij} = 1$, $k_1 = 2$, and $k_2 = 1$ are used in [10]), and ε is a softening parameter. This force model is also conservative, and we can compute the scalar potential as follows:

$$\Phi = \sum_{(i,j)\in E\wedge j>i} k_1 |\mathbf{p}_{ij}| \left(\log \frac{|\mathbf{p}_{ij}|}{l_{ij}} - 1 \right) + \sum_i \sum_{j>i} \frac{k_2}{|\mathbf{p}_{ij}| + \varepsilon}.$$

Fruchterman-Reingold. The Fruchterman-Reingold (FR) method [11] uses a simulation algorithm, and its force model is often used in other simulation-based methods (e.g., [19, 22]). In this model, attractive forces are given to adjacent pairs of vertices, and repulsive forces are imposed on any pairs of vertices. Specifically, the force $\mathbf{F}_i$ is defined as follows:

$$\mathbf{F}_i = -\sum_{(i,j)\in E} \frac{|\mathbf{p}_{ij}|}{k} \mathbf{p}_{ij} + \sum_{j\neq i} \frac{k^2}{|\mathbf{p}_{ij}| + \varepsilon} \frac{\mathbf{p}_{ij}}{|\mathbf{p}_{ij}|},$$

where k is a certain constant (that can be regarded as an ideal length of an edge [11]), and ε is a softening parameter. This force model is conservative, and therefore we can define the scalar potential Φ as follows:

$$\Phi = \sum_{(i,j)\in E\wedge j>i} \frac{|\mathbf{p}_{ij}|^3}{3k} - \sum_i \sum_{j>i} k^2 \log \left(1 + \frac{|\mathbf{p}_{ij}|}{\varepsilon} \right).$$

4.3 Algorithm

We solve the optimization problem (2) to perform force-directed graph drawing. As shown in the previous subsection, the objective function Φ is a scalar potential that is typically nonlinear. Previous force-directed methods solve such problems by using steepest decent (i.e., simulation-based algorithms), Newton's method (e.g., Kamada-Kawai [20]), and a conjugate gradient method (e.g., Tunkelang's method [28, 29]).

As already mentioned, steepest descent is used by most of the existing force-directed methods that are based on simulation. It is quite simple, and each iteration can be performed at a low cost. However, its convergence is slower than more sophisticated numerical algorithms, and therefore steepest descent is not preferable from the numerical computation viewpoint. It should also be noted that straightforward Newton's method is not desirable. As described in Subsection 3.1, it needs the Hessian matrix (or second partial derivative) of Φ whose computation is expensive. Quasi-Newton methods [25] can resolve this difficulty. Instead of using the actual Hessian matrix, they approximately compute the Hessian matrix by applying a certain update formula.

For our force-directed graph drawing method, we choose a quasi-Newton method called L-BFGS [24]. As described in Subsection 3.2, it reduces the necessary memory by storing only several vectors instead of the approximate Hessian matrix. In spite of the limited memory requirement, it is known to work well especially for large-scale optimization problems, and has been widely used in various fields such as machine learning [1].

Applying the L-BFGS method to our optimization problem (2) is straightforward. We can use Φ and $\nabla\Phi$ ($= -\mathbf{F}$) in our problem as f and ∇f respectively in the L-BFGS method described in Subsection 3.2.

5 Implementation

Using the method described in the previous section, we implemented a graph drawing system called AGI.[1] Its program is written in C++, and currently consists of approximately 3300 lines of code. We adopted the libLBFGS library[2] to perform the L-BFGS method using double-precision floating-point numbers.

5.1 Implementing Force Models

A major characteristic of the AGI system is that it allows implementing a force model easily. It provides a template class `LBFGSBasedDrawer` for implementing L-BFGS-based force-directed methods. An actual method can be built as a subclass of

[1] This is a reimplementation of our previous AGI system [18]. The current version implements only the method presented in this paper.

[2] http://www.chokkan.org/software/liblbfgs/

```
template<> double
LBFGSBasedKamadaKawaiDrawer<2>::
evaluate(const double* p, // current drawing
         double* g, // gradient of the scalar potential
                    // to compute
         const int n, // number of variables
         const double step) {
  for (int k = 0; k < n; ++k) g[k] = 0;
  double phi = 0; // scalar potential to compute
  for (int i = 0; i < n / 2; ++i) {
    // for each vertex i
    double xi, yi;
    get(p, i, xi, yi); // get the position of i
    for (int j = i + 1; j < n / 2; ++j) {
      // for each vertex j > i
      double xj, yj;
      get(p, j, xj, yj); // get the position of j
      double xij = xj - xi;
      double yij = yj - yi;
      double dst = norm2(xij, yij); // current distance
                                    // between i and j
      float lij = distanceMatrix_(i, j); // graph-theoretic
                                         // distance
      double dst_lij = dst - lij;
      phi += dst_lij * dst_lij / (2 * lij * lij);
      double dphi_dxi = -dst_lij * xij / (lij * lij * dst);
      double dphi_dyi = -dst_lij * yij / (lij * lij * dst);
      add(g, i, dphi_dxi, dphi_dyi); // update g for i
      add(g, j, -dphi_dxi, -dphi_dyi); // update g for j
    }
  }
  return phi; // computed scalar potential
}
```

Figure 1: Implementation of the Kamada-Kawai force model.

`LBFGSBasedDrawer` by implementing necessary virtual member functions as well as a constructor and a destructor. The most import function is `evaluate`, which performs the calculation of the scalar potential Φ and its gradient $\nabla\Phi$. Figure 1 shows the implementation of the KK model.

5.2 Memory Requirements

The amount of memory required for drawing a graph depends on the used force model. Among the four force models presented in the previous section, KK requires the largest amount of memory because it needs to store the graph-theoretic distance l_{ij} between each pair of vertices i and j. In our implementation, a single-precision floating-point number is used to store such a distance,[3] and the library used to compute graph-theoretic distances needs a dense square matrix. Therefore, for this purpose, we need $4n^2$ bytes for n vertices (e.g., nearly 400 megabytes for 10000 vertices).

The L-BFGS method does not require such a large amount of memory. In fact, the libLBFGS library (as of version 1.10) stores only $2(h+3)$ n-dimensional vectors, where h indicates the number of the past iterations to memorize and is 6 in our implementation (which is the default value in this library). Since we use double-precision floating-point numbers, we need $144n$ bytes for this purpose (e.g., approximately 1.4 megabytes for 10000 vertices).

6 Experiments

We performed experiments on our method by using the AGI system described in the previous section. We used the same benchmark dataset as [2].[4] The dataset consists of 43 graphs with 34 to 16840 vertices. Although these graphs have concrete names, we also use IDs G01 to G43 to indicate them for brevity. Table 1 shows their IDs as well as the numbers of their vertices and edges.

We compared the use of different force models in our method. In the following, we refer to the instances of our method using the KK, HC, Eades, and FR models as L-BFGS/KK, L-BFGS/HC, L-BFGS/Eades, and L-BFGS/FR respectively.

We also compared these instances of our method with other methods provided as part of the Graphviz system:[5] namely, neato/KK (which implements the KK method [20]), neato/major (which implements the stress majorization method [12]), fdp (which implements the FR method [11]), and sfdp (which implements the method presented in [19]). We used version 2.29.20111123.0545 of Graphviz.

We compiled both AGI and Graphviz by using the GCC 4.6.1 compiler. We executed the experiments on a 3.4 gigahertz quad-core Core i7 processor[6] with 16 gigabytes of memory running the 64-bit Linux 3.0.0 kernel.

Table 1: Graphs in the benchmark dataset.

IDs	Names	# of vertices	# of edges
G01	karateclub	34	78
G02	cylinder_rnd_010_010	97	178
G03	snowflake_A	98	97
G04	spider_A	100	220
G05	sierpinski_04	123	243
G06	flower_001	210	3057
G07	tree_06_03	259	258
G08	dg_617_part	341	797
G09	rna	363	468
G10	Grid_20_20_doublefolded	397	760
G11	Grid_20_20_singlefolded	399	760
G12	Grid_20_20	400	760
G13	protein_part	417	597
G14	516.graph	516	729
G15	flower_005	930	13521
G16	snowflake_B	971	970
G17	cylinder_rnd_032_032	985	1866
G18	grid_rnd_032	985	1834
G19	spider_B	1000	2200
G20	sierpinski_06	1095	2187
G21	ug_380	1104	3231
G22	tree_06_04	1555	1554
G23	Grid_40_40_doublefolded	1597	3120
G24	Grid_40_40_singlefolded	1599	3120
G25	Grid_40_40	1600	3120
G26	esslingen	2075	5530
G27	add20	2395	7462
G28	data	2851	15093
G29	3elt	4720	13722
G30	uk	4824	6837
G31	add32	4960	9462
G32	4970.graph	4970	7400
G33	dg_1087	7602	7601
G34	grid400_20	8000	15580
G35	tree_06_05	9331	9330
G36	cylinder_rnd_100_100	9497	17941
G37	grid_rnd_100	9497	17849
G38	snowflake_C	9701	9700
G39	sierpinski_08	9843	19683
G40	spider_C	10000	22000
G41	crack	10240	30380
G42	4elt	15606	45878
G43	cti	16840	48232

6.1 Experimental Settings

In our experiments, for each combination of a method and a graph, we normally executed the method ten times by giving different random seeds (specifically, $10000k$ for $k = 0, 1, \ldots, 9$), but we executed fdp only for graphs with less than 5000 vertices. In both AGI and Graphviz, the random seeds were given to the srand48 function, and then random numbers were generated by the drand48 function to

[3] While we use double-precision floating-point numbers in the L-BFGS method, we adopt single-precision floating-point numbers to store graph-theoretic distances; this is because graph-theoretic distances are usually simple. Neato/major of Graphviz, which we employed in our experiments (that we describe in Section 6), also uses single- and double-precision floating-point numbers in the same way.

[4] http://ls11-www.cs.tu-dortmund.de/staff/klein/gdmult10

[5] http://www.graphviz.org/

[6] AGI and Graphviz run sequentially although the used processor has four cores. Also, in the experiments, we disabled libLBFGS's support for the SSE/SSE2 SIMD extensions.

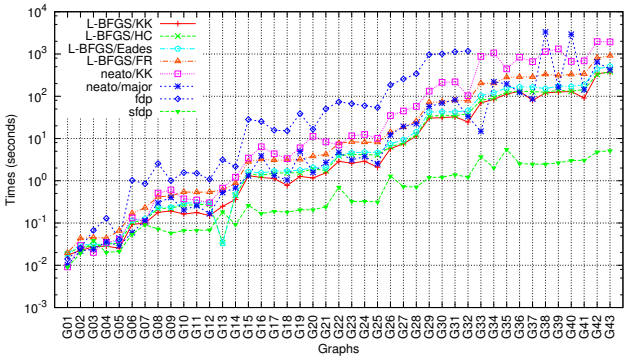


Figure 2: Total execution times.

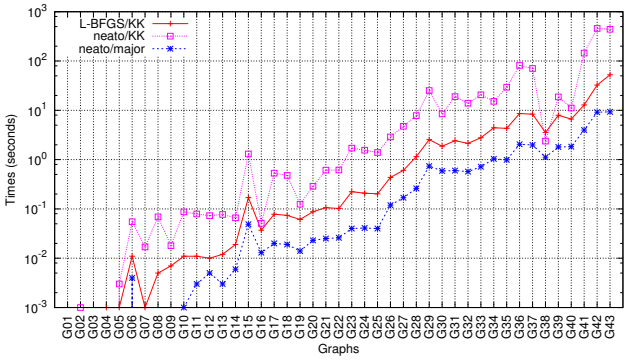


Figure 3: Times for computing graph-theoretic distances.

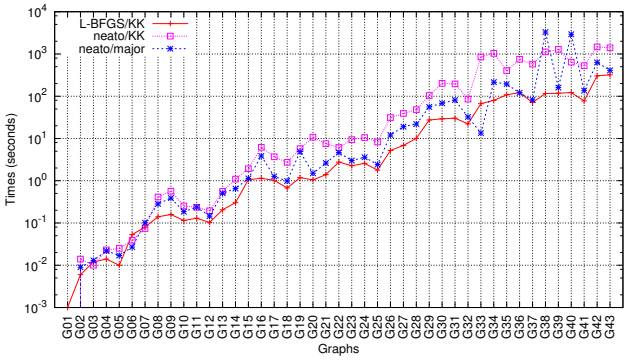


Figure 4: Times for numerical optimization.

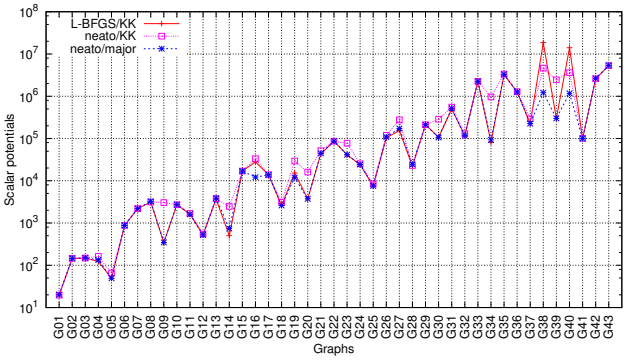


Figure 5: Average final scalar potentials.

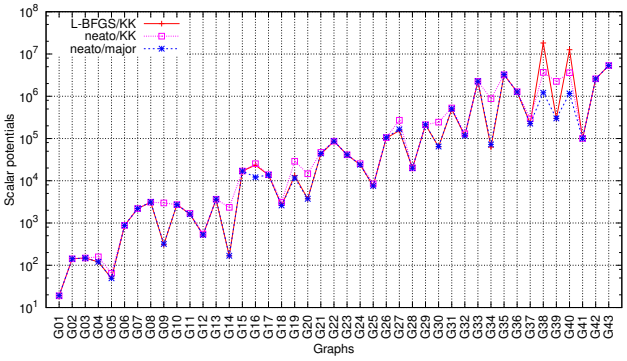


Figure 6: Minimum final scalar potentials.

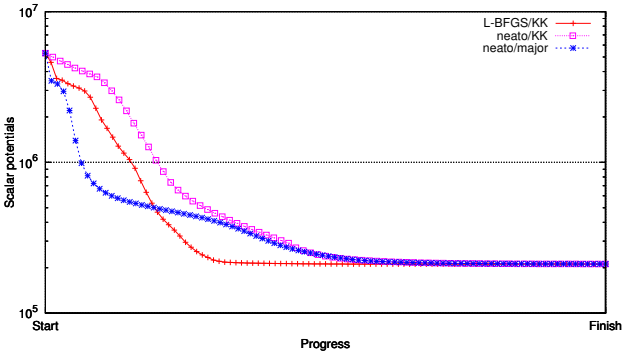


Figure 7: Optimization of scalar potentials for the 3elt graph.

obtain initial graph drawings. In our experiments, L-BFGS/KK and neato/major generated the same initial drawings for the same random seeds. Also, neato/KK generated its initial drawings that were different only in translation from L-BFGS/KK and neato/major's, and the other methods generated their initial drawings that were different only in translation and scaling. In other words, for the same random seed, any of these methods generated the initial drawings that can be transformed into each other only by translation and scaling.

We set the maximum numbers of iterations in L-BFGS/KK, L-BFGS/HC, L-BFGS/Eades, L-BFGS/FR, and neato/major to 100. We did not change the default values for the other parameters including those for the libLBFGS library used in AGI.

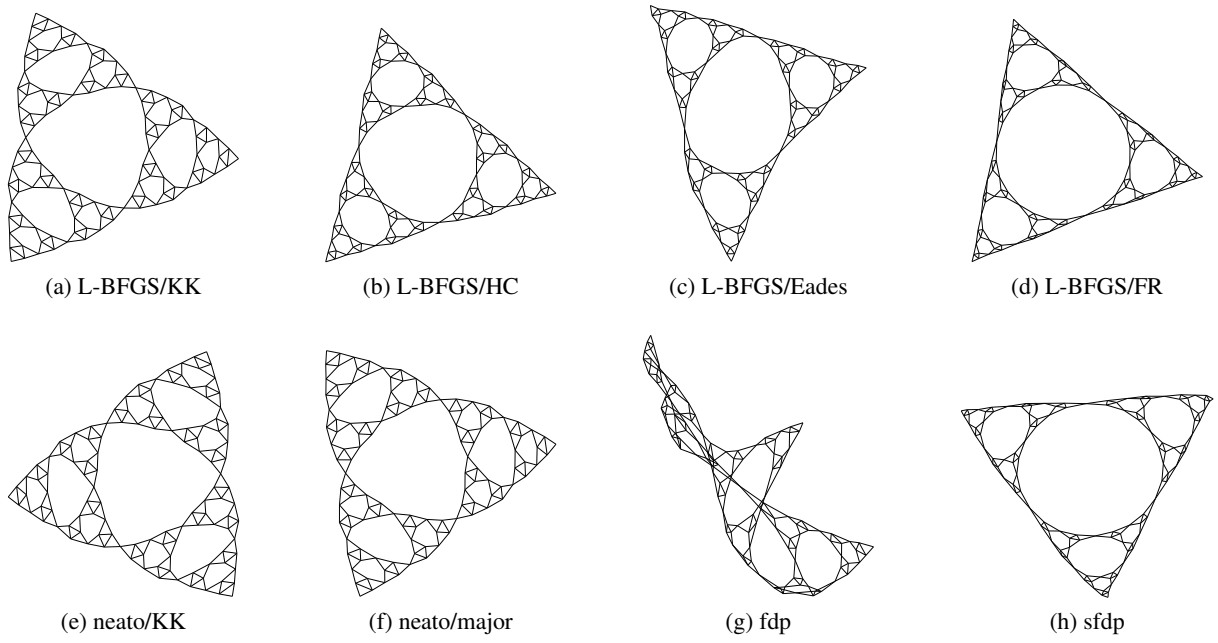
6.2 Execution Times

Now we present the execution times of these methods. Figure 2 shows the averages of the total execution times. Regarding L-BFGS/KK, neato/KK, and neato/major, we also measured the execution times of main algorithmic components. Since all these methods are based on the KK model, they need to compute graph-theoretic distances between any pairs of vertices before numerical optimization. Figure 3 presents the average times for computing graph-theoretic distances, and Figure 4 gives the average times for numerical optimization.

From these execution times we can see that, among the instances of our method, L-BFGS/KK often provides the best performance. Also, it should be noted that L-BFGS/KK is usually faster than neato/KK, neato/major, and fdp, although it is much slower than sfdp (which incorporates both multi-level and tree-code methods).

6.3 Scalar Potentials

We can quantitatively evaluate the results of L-BFGS/KK, neato/KK, and neato/major by comparing the scalar potentials since all of them use the KK model. Figure 5 presents the averages of final scalar potentials, and Figure 6 gives the minimum final scalar potentials.



(a) L-BFGS/KK (b) L-BFGS/HC (c) L-BFGS/Eades (d) L-BFGS/FR

(e) neato/KK (f) neato/major (g) fdp (h) sfdp

Figure 8: The sierpinski_04 graph drawn by (a) L-BFGS/KK, (b) L-BFGS/HC, (c) L-BFGS/Eades, (d) L-BFGS/FR, (e) neato/KK, (f) neato/major, (g) fdp, and (h) sfdp.

From these results we can see that L-BFGS/KK and neato/major usually achieve almost equivalent final scalar potentials. However, for the snowflake_C and spider_C graphs, L-BFGS/KK did not work well. Although we have not yet analyzed this problem in detail, these graphs are identified as "challenging" graphs in the literature [14]. It should also be noted that neato/major took long time for these graphs.

We can compare these methods by observing the changes of scalar potentials during numerical optimization. Figure 7 plots the scalar potentials that they obtained while drawing the 3elt graph by using the same random seed. It should be noted that this plot does not consider the actual times. L-BFGS/KK, neato/KK, and neato/major performed 100, 152008, and 93 iterations. For this graph all of them finally achieved equivalent scalar potentials.

6.4 Quality of Drawings

Now we show actual graph drawings. Figures 8, 9, and 10 show the graph drawings produced by these methods. Among the instances of our method, L-BFGS/KK usually provided the most pleasing drawings, and L-BFGS/FR gave the next most pleasing drawings. However, the quality difference between these two became larger as they handled larger graphs.

It should be noted that L-BFGS/KK and neato/major usually produced similar drawings. This is because they use the KK model and usually achieved equivalent final scalar potentials as shown in the previous subsection.

6.5 L-BFGS/KK versus Sfdp

As these results suggest, among the instances of our method, L-BFGS/KK often gives the best performance both in execution times and drawing quality. Finally, for comparison of L-BFGS/KK with sfdp, we show in Figure 11 drawings of larger graphs computed by these methods.

7 Discussion

A major advantage of our method is that it is more general than most existing methods; as shown in Subsections 4.2 and 5.1, our method is easily applicable to different force models as far as their forces are conservative. By contrast, the stress majorization-based method [12] is a quite opposite to our method; it is specialized in the Kamada-Kawai model, and it seems not to be easy to apply it to other force models. Because of the generality, our method will also be good as a tool for testing new force models.

Another advantage of our method is that it can easily incorporate existing numerical optimization methods and implementations. Because of the long history and large community of numerical optimization, we can expect opportunities for obtaining good methods and implementations. In fact, we already did it by using the existing L-BFGS implementation libLBFGS. If we had used a more specialized method, it would have been more difficult to implement its algorithm.

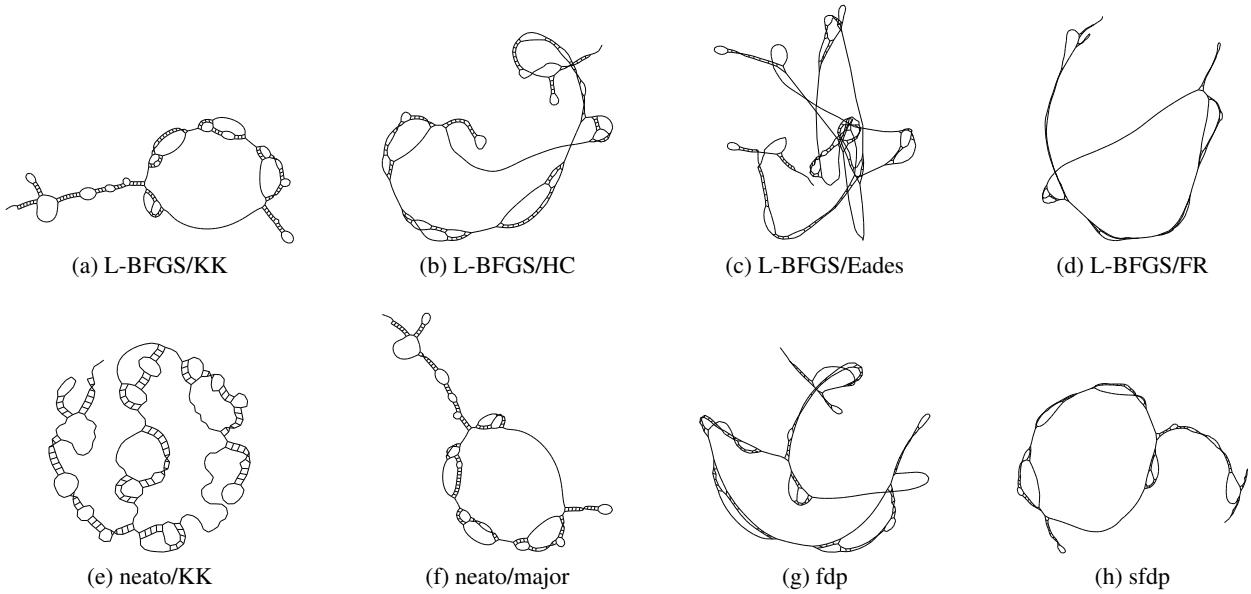
We have not yet included standard performance improvement techniques such as multi-level methods [2, 13, 19, 30] and tree-code methods [13, 19, 22, 27]. However, we foresee that multi-level methods will be effective for our method. At least, the combination of our method with a multi-level method is not difficult to implement since multi-level methods do not impose strong requirements on associated single-level methods [2]. Also, tree-code methods seem worth applying to our method. However, the combination of our method with a tree-code method is not as clear as its combination with a multi-level method.

8 Conclusions and Future Work

In this paper, we presented a numerical optimization-based method for force-directed graph drawing. Our method was applicable to different basic force models, and was implemented by using the L-BFGS method for numerical optimization. Our experimental results showed that our method gave good performance when combined with the Kamada-Kawai model.

Our future work includes introducing more general force models that treat edge directions and other kinds of constraints [7]. Another future direction is to incorporate a multi-level method and a tree-code method to further improve the performance of our method.

This is the author's version. The final authenticated version is available online at https://doi.org/10.1109/PacificVis.2012.6183577.



(a) L-BFGS/KK (b) L-BFGS/HC (c) L-BFGS/Eades (d) L-BFGS/FR

(e) neato/KK (f) neato/major (g) fdp (h) sfdp

Figure 9: The rna graph drawn by (a) L-BFGS/KK, (b) L-BFGS/HC, (c) L-BFGS/Eades, (d) L-BFGS/FR, (e) neato/KK, (f) neato/major, (g) fdp, and (h) sfdp.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by JST CREST.

REFERENCES

[1] G. Andrew and J. Gao. Scalable training of L_1-regularized log-linear models. In *Proc. ICML*, pages 33–40, 2007.

[2] G. Bartel, C. Gutwenger, K. Klein, and P. Mutzel. An experimental evaluation of multilevel layout methods. In *Proc. GD*, volume 6502 of *LNCS*, pages 80–91, 2010.

[3] U. Brandes and C. Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Proc. GD*, volume 4372 of *LNCS*, pages 42–53, 2007.

[4] U. Brandes and C. Pich. An experimental study on distance-based graph drawing. In *Proc. GD*, volume 5417 of *LNCS*, pages 218–229, 2009.

[5] A. Civril, M. Magdon-Ismail, and E. Bocek-Rivele. SDE: Graph drawing using spectral distance embedding. In *Proc. GD*, volume 3843 of *LNCS*, pages 512–513, 2005.

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

[7] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[8] T. Dwyer and K. Marriott. Constrained stress majorization using diagonally scaled gradient projection. In *Proc. GD*, volume 4875 of *LNCS*, pages 219–230, 2007.

[9] T. Dwyer, K. Marriott, and M. Wybrow. Topology preserving constrained graph layout. In *Proc. GD*, volume 5417 of *LNCS*, pages 230–241, 2009.

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

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

[12] E. R. Gansner, Y. Koren, and S. C. North. Graph drawing by stress majorization. In *Proc. GD*, volume 3383 of *LNCS*, pages 239–250, 2004.

[13] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Proc. GD*, volume 3383 of *LNCS*, pages 285–295, 2004.

[14] S. Hachul and M. Jünger. Large-graph layout algorithms at work: An experimental study. *J. Graph Algorithms Appl.*, 11(2):345–369, 2007.

[15] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. In *Proc. GD*, volume 2528 of *LNCS*, pages 207–219, 2002.

[16] H. Hosobe. A modular geometric constraint solver for user interface applications. In *Proc. ACM UIST*, pages 91–100, 2001.

[17] H. Hosobe. A high-dimensional approach to interactive graph visualization. In *Proc. ACM SAC*, pages 1253–1257, 2004.

[18] H. Hosobe. An interactive large graph visualizer. In *Proc. Smart Graphics*, volume 5166 of *LNCS*, pages 271–272, 2008.

[19] Y. Hu. Efficient, high-quality force-directed graph drawing. *Mathematica J.*, 10(1):37–71, 2005.

[20] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inf. Process. Lett.*, 31(1):7–15, 1989.

[21] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvectors computation for drawing huge graphs. In *Proc. IEEE InfoVis*, pages 137–144, 2002.

[22] T. Matsubayashi and T. Yamada. A force-directed graph drawing based on the hierarchical individual timestep method. *Intl. J. Elect. Comput. Eng.*, 2(10):686–691, 2007.

[23] A. Noack. Energy-based clustering of graphs with nonuniform degrees. In *Proc. GD*, volume 3843 of *LNCS*, pages 309–320, 2006.

[24] J. Nocedal. Updating quasi-newton matrices with limited storage. *Math. Comput.*, 35(151):773–782, 1980.

[25] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

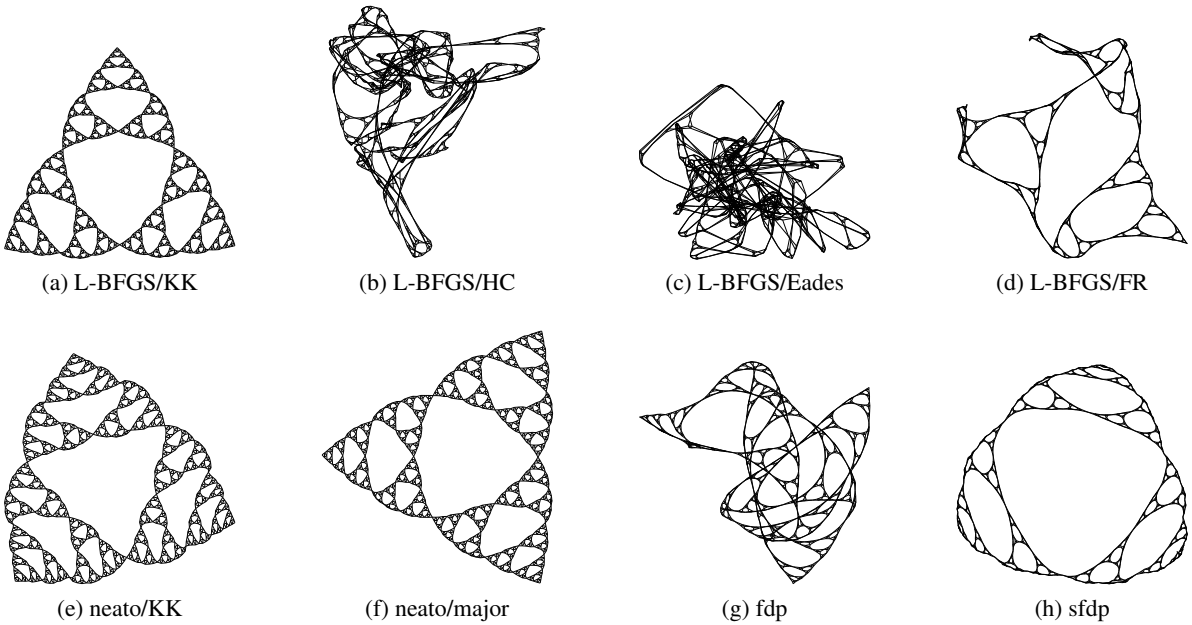
[26] M. Nöllenburg and A. Wolff. Drawing and labeling high-quality metro maps by mixed-integer programming. *IEEE Trans. Visual. Comput. Gr.*, 17(5):626–641, 2011.

[27] A. J. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In *Proc. GD*, volume 1984 of *LNCS*, pages 197–210, 2000.

[28] D. Tunkelang. JIGGLE: Java interactive graph layout environment. In *Proc. GD*, volume 1547 of *LNCS*, pages 412–422, 1998.

[29] D. Tunkelang. A numerical optimization approach to general graph drawing (Ph.D. thesis). Tech. Rep. CMU-CS-98-189, Sch. Comput. Sci., Carnegie Mellon Univ., 1999.

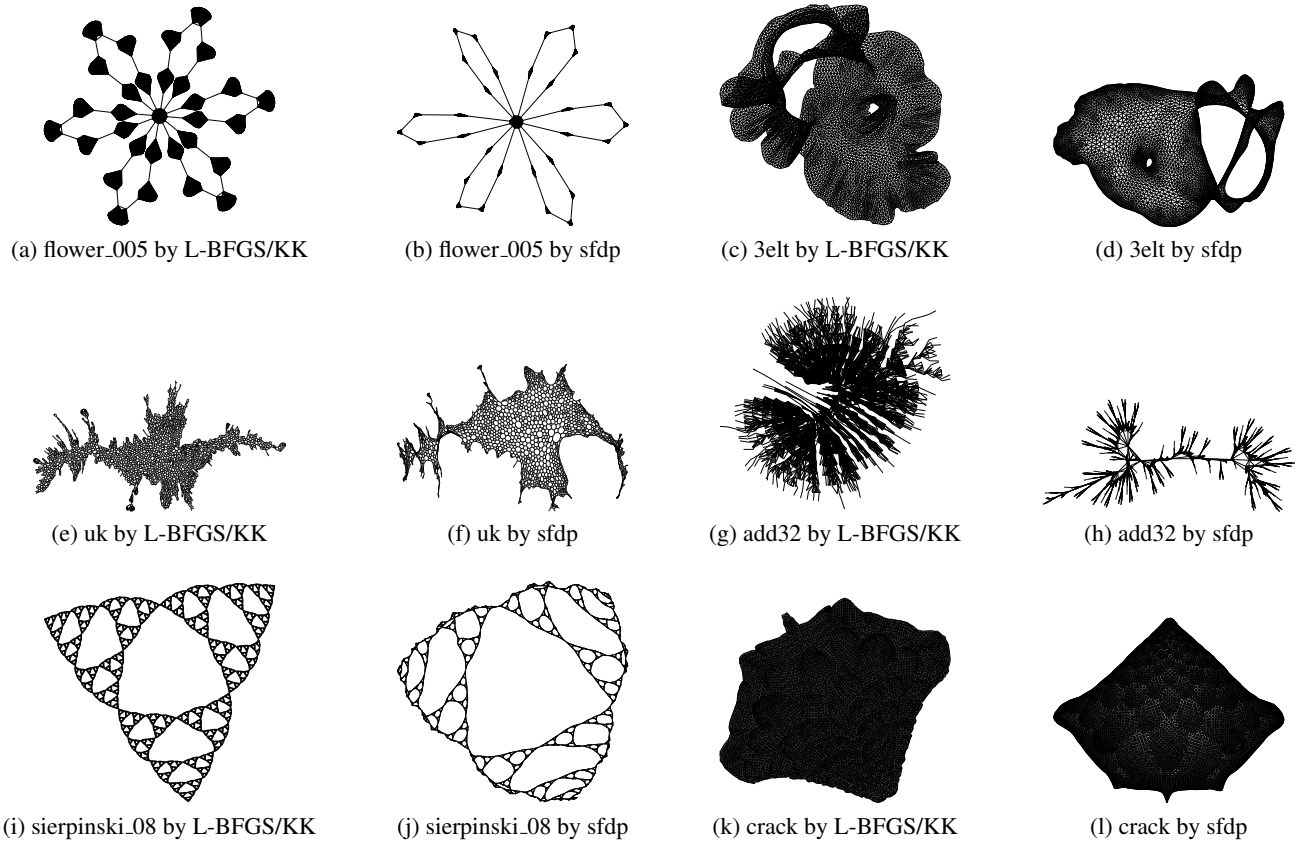
[30] C. Walshaw. A multilevel algorithm for force-directed graph-drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.



(a) L-BFGS/KK (b) L-BFGS/HC (c) L-BFGS/Eades (d) L-BFGS/FR

(e) neato/KK (f) neato/major (g) fdp (h) sfdp

Figure 10: The sierpinski_06 graph drawn by (a) L-BFGS/KK, (b) L-BFGS/HC, (c) L-BFGS/Eades, (d) L-BFGS/FR, (e) neato/KK, (f) neato/major, (g) fdp, and (h) sfdp.



(a) flower_005 by L-BFGS/KK (b) flower_005 by sfdp (c) 3elt by L-BFGS/KK (d) 3elt by sfdp

(e) uk by L-BFGS/KK (f) uk by sfdp (g) add32 by L-BFGS/KK (h) add32 by sfdp

(i) sierpinski_08 by L-BFGS/KK (j) sierpinski_08 by sfdp (k) crack by L-BFGS/KK (l) crack by sfdp

Figure 11: Graphs drawn by L-BFGS/KK and sfdp: (a)(b) flower_005, (c)(d) 3elt, (e)(f) uk, (g)(h) add32, (i)(j) sierpinski_08, and (k)(l) crack.