DOI: 10.1111/cgf.14003
Eurographics Conference on Visualization (EuroVis) 2020
M. Gleicher, T. Landesberger von Antburg, and I. Viola
(Guest Editors)

Volume 39 (2020), Number 3

Sublinear Time Force Computation for Big Complex Network Visualization

Amyra Meidiana[†1], Seok-Hee Hong[‡1], Marnijati Torkel[§1], Shijun Cai[¶1] and Peter Eades[‖1]

[1]School of Computer Science, The University of Sydney, Australia

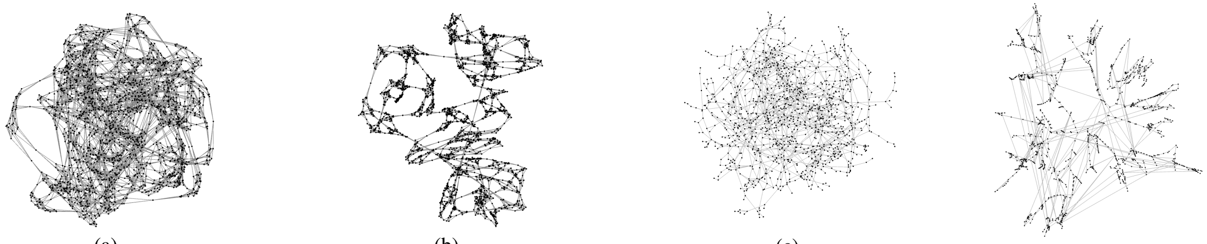

(a) (b) (c) (d)

Figure 1: *Visual comparisons of the results of the linear-time RVS algorithm and our sublinear-time force computation algorithm. (a) and (b) show the sierpinski3d graph drawn using (a) RVS and (b) SLG, our sublinear force computation algorithm with geometric sampling displaying the grid structure of the graph better with a lower runtime; and (c) and (d) show the* 1138_*bus graph drawn using (c) RVS and (d) SSC-G, our sublinear force computation algorithm with spectral sparsification and combinatorial sampling, untangling the structure of the graph better with a lower runtime.*

Abstract

In this paper, we present a new framework for sublinear time force computation for visualization of big complex graphs. Our algorithm is based on the sampling of vertices for computing repulsion forces and edge sparsification for attraction force computation. More specifically, for vertex sampling, we present three types of sampling algorithms, including random sampling, geometric sampling, and combinatorial sampling, to reduce the repulsion force computation to sublinear in the number of vertices. We utilize a spectral sparsification approach to reduce the number of attraction force computations to sublinear in the number of edges for dense graphs. We also present a smart initialization method based on radial tree drawing of the BFS spanning tree rooted at the center.

Experiments show that our new sublinear time force computation algorithms run quite fast, while producing good visualization of large and complex networks, with significant improvements in quality metrics such as shape-based and edge crossing metrics.

1. Introduction

Recently, big complex networks are abundant in various application domains, such as the internet, finance, social networks, and systems biology. Examples include web and autonomous system graphs, social media networks, protein-protein interaction networks and biochemical pathways. Good visualization can be useful in understanding the hidden structure of such big complex networks, emphasizing cognitive perception, leading to new insights and possible future prediction. However, visualization of big complex networks is extremely challenging due to scalability and complexity. For example, visualization of big complex networks often produces hairball-like tangled visualizations, which raises difficulties for human perception and cognition.

Force-directed algorithm is the most popular algorithm for graph visualization. The algorithm consists of computing repulsion force for each pair of vertices, and computing attraction force for each edge. The running time for force computation is $O(|V|^2)$ for re-

† e-mail: amei2916@uni.sydney.edu.au
‡ e-mail: seokhee.hong@sydney.edu.au
§ e-mail: mtor0581@sydney.edu.au
¶ e-mail: scai5619@uni.sydney.edu.au
‖ e-mail: peter.eades@sydney.edu.au

pulsion force, and $O(|E|)$ for attraction force computation for a graph $G = (V,E)$ with a vertex set V and edge set E. However, this quadratic runtime complexity means the algorithm does not scale well to big networks. Faster spring algorithms [QE00, HJ04] use an approximation of the repulsion force computation based on the Barnes-Hut n-body method [BH86], reducing the runtime to $O(n \log n)$ in practice.

Recently, the *RVS (Random Vertex Sampling)* force-directed algorithm which uses random vertex sampling to compute repulsion forces in linear runtime was presented [Gov19b]. Instead of computing repulsion forces for all pairs of vertices, RVS select a subset of the vertex set, the *update set* U with size $|U| = |V|^{0.75}$ and for each vertex u in U, repulsion forces are computed with vertices in a *random sample set* S with size $|S| = |V|^{0.25}$. Furthermore, a *fixed subset repulsion* force computation is performed for each vertex with a fixed-sized subset of $O(1)$ size in each iteration. However, this method does not reduce attraction force computation, resulting in $O(|V|^2)$ runtime for dense graphs.

More recently, a similar random edge sampling to reduce the runtime of attraction force computation was presented [Gov19a]. However, the runtime reduction obtained by the edge sampling approach is small in proportion to the loss in quality of the results.

In this paper, we present a new framework for sublinear time force computation for visualization of big complex graphs, which combines graph analysis and smart initialization with sampling-based force computation to reduce the runtime while producing high quality drawings, improving limitations in [Gov19b, Gov19a].

Roughly speaking, we use U with size $|U| = |V|^{0.5}$ and S with size $|S| = |V|^{0.2}$ to achieve sublinear time repulsion force computation with respect to the number of vertices, where our experiments have shown that these sizes for U and S obtain significant runtime improvement over RVS while still obtaining better quality metrics. We use the *spectral sparsification* method, which reduces the number of edges in a graph to $O(n \log n)$, to achieve sublinear time attraction force computation with respect to the number of edges for dense graphs with $|E| = O(n^2)$.

To maintain the quality of the graph drawing, we use initialization using the radial drawing of the BFS spanning tree rooted at the center, as well as geometric sampling and combinatorial sampling methods to sample more important vertices.

Our experiments demonstrate that our sublinear time force computation algorithms both run faster than existing linear time force computation algorithms and obtain significant improvements in quality metrics.

More precisely, we present the following contributions:

1. We present a framework for sublinear time force computation, which consists of analysis, initialization, and force computation. We use spanning trees and spectral sparsification for analysis, radial tree drawing of BFS spanning tree rooted at the center for initialization, and sampling for force computation.
2. We present the **SL (SubLinear)** family of algorithms, combining BFS (Breadth-First Search) tree rooted at the center of a graph and radial tree drawing for initialization with vertex sampling with size $|U| = |V|^{0.5}$ and S with size $|S| = |V|^{0.2}$ to reduce the runtime of the *repulsion force computation* to sublinear in the number of vertices. We present **SLR (Sublinear Random)**, **SLG (Sublinear Geometric)**, and **SLC (Sublinear Combinatorial)** algorithms which use random sampling, geometric sampling, and combinatorial sampling respectively in the force computation step.
3. We present the **SS (Spectral Sparsification)** family of algorithms, which utilizes spectral sparsification as an analysis step to reduce the runtime of *attraction force computation*. Spectral sparsification reduces the number of edges to $O(n \log n)$, which reduces the attraction force computation to sublinear in the number of edges for dense graphs. The SS family includes **SSR (SS Random)**, **SSG (SS Geometric)**, and **SSC (SS Combinatorial)** algorithms, which combines the different vertex sampling methods. We also present the **SS-G** family of algorithms, which adds back all edges of a graph to the drawings produced by SS algorithms.
4. We implement and evaluate the SL, SS, and SS-G algorithms using experiments with a variety of benchmark datasets and compare with RVS, using runtime, quality metrics and visual comparison. Experiments showed that the SL algorithms attain average runtime improvements on the force computation steps with 20% over RVS, while achieving average improvements of 104.5% for shape-based metrics and 30% for edge crossing. SS algorithms obtain average runtime improvement of 28%, average 57.5% improvement on shape-based metrics, and average 28.5% improvement on edge crossing. SS-G obtains average 130% improvement on shape-based metrics and average 26% improvement on edge crossing.

Figure 1 shows example comparisons of our algorithms and RVS. Figure 1 (a) and (b) show the *sierpinski3d* graph drawn with RVS and SLG respectively, showing that our method detangle the mesh structure better. Figure 1 (c) and (d) show the 1138_*bus* graph drawn with RVS and the spectral sparsification of the graph drawn with SSC-G, showing that SSC-G shows the graph structure better while requiring less time to compute.

2. Related Work

2.1. Spring algorithm

Spring, or force-directed, algorithms model a graph as a system with attraction forces between neighboring vertices and repulsion forces between all pairs of vertices. Traditional spring algorithms include those presented by Eades [Ead84] and Fruchterman and Reingold [FR91]. Although spring algorithms are able to produce high quality graph layouts, traditional spring algorithms do not scale well to larger graphs due to the repulsion force computation taking $O(n^2)$ runtime.

Some faster spring algorithms are available. For example, Quigley and Eades presented FADE [QE00] which reduces the repulsion force computation to $O(n \log n)$ runtime. Hachul and Junger presented FM^3 using a method to compute the repulsion forces between vertices, where subgraphs with small diameter, called solar systems, are partitioned and collapsed to obtain a multi-level representation [HJ04].

2.2. Graph Sampling Approach

Graph sampling can be used as a method to address scalability issues in graph drawing. Hong et al [HNM*18] presented a sampling algorithm for drawing graphs, which combines random sampling with biconnected graph decomposition to reduce the runtime of graph drawing algorithms. Another approach utilizes spectral sparsification with topological graph decomposition to attain high quality samples produced by spectral sparsification with better runtime efficiency [MHH*19].

Another way to use sampling is to reduce the number of force computations required to compute a layout. Gove [Gov19b] presented the *RVS (Random Vertex Sampling)* force-directed algorithm. The algorithm utilizes a phyllotaxis initialization. For a graph $G = (V,E)$, a random subset of vertices is selected as the *update set* U with size $|V|^{0.75}$ and for each vertex in the update set, repulsion forces are computed between the vertex and another random subset of vertices, the *sample set* S of size $|V|^{0.25}$. This is repeated at most 300 steps, which was found to achieve convergence by experiment. In addition, each vertex is assigned a fixed-size subset of vertices C_u of size $min(15,|V|)$, where in every iteration of the force computation, repulsion forces are also computed between each vertex and its assigned fixed-size subset.

Gove [Gov19a] also presented edge sampling to reduce the runtime of attraction force computation, where in each computation, a percentage of the edges are selected, and attraction forces are computed only on the selected edges. However, compared to RVS, the runtime reduction obtained by the edge sampling approach is small in contrast to the loss in quality of the resulting drawing.

3. Framework for Sublinear Time Force Computation

This section presents our framework for sublinear time force computation. The framework consists of the following three steps:

1. **Analysis:** This step is a preprocessing step for graph analysis.
2. **Initialization:** This step is to compute an initial layout for the force-directed algorithm.
3. **Force computation:** This step consists of repulsion force computation and attraction force computation.

We now explain the details of each step. Let $G = (V,E)$ be a graph with vertex set V and edge set E. G is called a sparse graph if it has $O(n)$ edges or a dense graph if it has $O(n^2)$ edges.

3.1. Analysis

Graph analysis can be combined with force-directed algorithms to reduce the runtime of the layout computation while retaining quality. The results of the analysis can be used in the initialization step, the force computation steps, or both. The graph analysis methods considered include:

Spanning tree: We compute a BFS spanning tree T rooted at the *center* of a graph G, both of which can be computed in linear time. The center of a graph is a vertex with minimum distance to all other vertices in the graph. In latter steps of the framework, the tree T can be used in creating a good initialisation for drawing the graph G, or the position of the vertices in the hierarchy of the spanning tree can be used in the computation of repulsion forces.

Spectral sparsification: Spectral sparsification samples edges while preserving the structural properties of the original graph. Spielman and Teng proved that every n-vertex graph has a spectral approximation with $O(n\log n)$ edges, and presented a stochastic sampling method using the concept of *effective resistance*, which is closely related to the commute distance [ST11]. Spectral sparsification can be computed in near linear time [T*16].

3.2. Initialization

Initialization is an important part of force-directed algorithms that can greatly affect the resulting layout. The drastically different layout may be computed for the same graph when starting from different initial positions for the vertices, which gives rise to a need for a method to select a good initial position.

For initialization, we use the *radial tree drawing* algorithm [Ead91], which can be computed in $O(n)$ time. We use this algorithm to draw the BFS spanning tree rooted at the center and use the position of the vertices in the drawing as the initial positions for the force-directed algorithm. By putting the graph center and its neighbors near the center of the drawing and peripheral vertices further from the center of the drawing, we expect to obtain an initial position that places the vertices at a distance that is closer to their graph theoretic distance than with random initialization.

3.3. Sublinear Time Force Computation

We present algorithms for sublinear time force computations, operated as the third step of our framework. For repulsion force computation, we present vertex sampling methods which sample a sublinear number of vertices in each iteration, while for attraction force computation, we present a spectral sparsification method that reduces the number of attraction force computations to sublinear in the number of edges.

3.3.1. Repulsion Force Computation

For repulsion force computation, we show *RepulsionSample* (Algorithm 1), a vertex sampling algorithm based on RVS. While RVS uses update size $|U| = |V|^{0.75}$ and sample size $|S| = |V|^{0.25}$, *RepulsionSample* uses sizes such that $|U| \times |S| < |V|$, giving the runtime complexity as sub-linear in the number of vertices. Moreover, fixedSubsetRepulsion in line 10 of Algorithm 1, which represents the fixed subset repulsion routine from RVS, is only run on a subset of vertices of size $|U| \times |S|$ rather than all of $|V|$, keeping the runtime complexity sublinear.

FisherYates in line 3 of Algorithm 1 is used to sample the contents of an array in place in linear time and space complexity with regards to the size of the sample, similar to RVS.

More specifically, we define three versions of the algorithm with different U and S as below:

Algorithm 0702: $U = |V|^{0.7}$, $S = |V|^{0.2}$, $O(n^{0.9})$ runtime.
Algorithm 0602: $U = |V|^{0.6}$, $S = |V|^{0.2}$, $O(n^{0.8})$ runtime.

Algorithm 1: RepulsionSample

```
Input: Vertices $V$, array of indices $I$, starting index $j_{prev}$,
       starting index $l_{prev}$, update size $|U|$, sample size $|S|$
for $i = j_{prev}$ to $j_{prev} + U$ do
    $v = V[i \bmod |V|]$;
    $R$ = FisherYates($I, S$);
    for $r$ in $R$ do
        repulsionForce($v$, $V[r]$);
    end
end
for $i = l_{prev}$ to $l_{prev} + |S| \times |U|$ do
    $v = V[i \bmod |V|]$;
    fixedSubsetRepulsion($v$);
end
```

Algorithm 2: SLR (Sublinear Random)

```
Input: Graph $G = (V, E)$, update size $|U|$, sample size $|S|$
Output: Drawing $D$ of $G$
Initialize $D$ using the radial drawing the BFS tree of $G$
  rooted at the center;
$I$: array of integers from 1 to $|V|$; // for random sampling
$j_{prev}$ = 1; // starting index of update set
$l_{prev}$ = 1; // starting index for fixed set fixed-size subset
  computation
for iterations in 1 to 300 do
    RepulsionSample($V$, $I$, $j_{prev}$, $l_{prev}$, $|U|$, $|S|$);
    AttractionForce($V$, $E$);
    $j_{prev} = (j_{prev} + |U|) \bmod |V|$;
    $l_{prev} = (l_{prev} + |U| \times |S|) \bmod |V|$;
end
return $D$
```

Algorithm 0502: $U = |V|^{0.5}$, $S = |V|^{0.2}$, $O(n^{0.7})$ runtime.

These algorithms run the same as SLR, described in Algorithm 2, only with differing sizes for U and S.

Based on RepulsionSample, we present the *SL (Sublinear)* family of algorithms to compute repulsion forces in sublinear time:

Algorithm SLR (Sublinear Random): uses a random sampling to compute the sample set S.

Algorithm SLG (Sublinear Geometric): uses a geometric sampling to compute the sample set S.

Algorithm SLC (Sublinear Combinatorial): uses a combinatorial sampling to compute the sample set S.

SLR runs exactly the same as 0502 described previously. In addition, we define two other sampling methods: *geometric* and *combinatorial* sampling. The geometric sampling performs weighted random sampling such that vertices located in denser areas of the drawing are sampled at a higher probability than vertices located in sparser areas, with the aim of better untangling dense areas of the graph. The combinatorial sampling partition the vertices based on their position in the BFS tree level hierarchy, i.e., sample vertices closer to the center c at a higher rate, aiming to untangle the "hairball" structures of vertices close to the center.

Algorithm 2 describes the *SLR* algorithm for computing repulsion forces. The algorithm uses random sampling to select the sample set, and the array of indices I used to sample the vertices simply contains integers from 1 to $|V|$ such that each vertex may be sampled at the same probability. For all SL algorithms, we use $|U| = |V|^{0.5}$ and $|S| = |V|^{0.2}$.

The *SLG* algorithm, described in Algorithm 3 defines a 10-by-10 grid overlaid on the drawing area and group vertices by the grid cell they belong to. A fixed-size grid has been chosen as the space partitioning method as it can be computed in constant time, and determining which cell a vertex falls into when it is moved takes constant time as well.

We represent the grid using an array R, and in position $R[i][j]$, we have an array of indices of vertices located in the grid cell at row i and column j. We then use *weighted random sampling* to select the sample set: vertices in dense cells, where density is defined by the number of vertices in the cell, are sampled at a higher probability than vertices in sparse cells.

Lines 4-6 of Algorithm 3 describe the method used to simulate this weighted random sampling: given an array A_c, we populate the array with indices of the cells, with denser cells having their indices repeated more than sparser cells (e.g. we repeat the indices of the 25% densest cells of R to fill out 65% of A_c). The ratios used to weight the sampling probabilities of cells by their density has been selected through preliminary experiments, where these ratios were seen to obtain drawings with better quality.

During the repulsion force computation stage, we sample $|V|^{0.2}$ indices from A_c, and for every occurrence of the index of a cell r in the sample, we sample that many vertices to be added to the sample set - this is shown in lines 10-15 in Algorithm 3.

At the end of every force computation iteration, we recompute which cell a vertex that had its position changed by the force computation belongs to. We then update A_c as required.

The *SLC* algorithm in Algorithm 4 partitions the set of vertices into groups based on the level of the BFS hierarchy. Similar to geometric sampling, we perform weighted random sampling in selecting the sample set, with vertices belonging to lower levels of the hierarchy (i.e. closer to the root) given more weight than those belonging to higher levels of the hierarchy (i.e. closer to leaves). Vertices closer to the center are likely to be located in the center of "hairball" structures, and by sampling these vertices more, we aim to be able to untangle the hairballs better.

Combinatorial sampling was done using weighted sampling, similar to geometric sampling, but with the weights based on the BFS hierarchy. We first divide the array of vertices into partitions, based on the level in the BFS tree (line 2). Each partition roughly contains $0.2|V|$ of vertices. We then assign percentages to each partition, such that when selecting vertices for the sample set, we sample that percentage of the sample set from that partition - lines 7-9 describes how this is done, with multiple calls to RepulsionSample with different subsets of vertices to be sampled and different sample sizes. From experiments, we use the percentages of each partition, starting from the lowest level of the hierarchy, to be $70, 15, 7, 5, 3$. As with *SLG*, these percentages were selected after preliminary ex-

Algorithm 3: SLG (Sublinear Geometric)

```
Input: Graph $G = (V, E)$, update size $|U|$, sample size $|S|$
Output: Drawing $D$ of $G$
Initialize $D$ using the radial drawing of the BFS tree of $G$ rooted at the center;
$R$: array representing a regular 10-by-10 grid over the drawing area of $D$ where each $r = R[i + j \times 10]$ is the array containing the indices of vertices located in the $i$th row and $j$th column of the grid;
$A_c$: array of size $|V|$; // used for weighted sampling
Repeat the indices of the 25% densest cells of $R$ to fill out 65% of $A_c$;
Repeat the indices of the next 50% densest cells of $R$ to fill out 25% of $A_c$;
Repeat the indices of the 25% least dense cells of $R$ to fill out 10% of $A_c$;
$j_{prev} = 1$;
$l_{prev} = 1$;
for iterations in 1 to 300 do
    Randomly sample $|V|^{0.2}$ indices from $A_c$ into array $I_{geo}$;
    for $r$ in $R$ do
        count: occurrences of the index of $r$ in $I_{geo}$;
        RepulsionSample($V$, $r$, $j_{prev}$, $l_{prev}$, $|V|^{0.5}$, count);
        // sample vertices from $r$ based on the number of times $r$ was sampled
    end
    AttractionForce($V$, $E$);
    $j_{prev} = (j_{prev} + |U|) \bmod |V|$;
    $l_{prev} = (l_{prev} + |U| \times |S|) \bmod |V|$;
    Recompute $A_c$ based on the updated position of vertices;
end
return $D$
```

periments showed that these percentages obtain good quality drawings.

3.3.2. Attraction Force Computation

With dense graphs, attraction force computation takes $O(n^2)$ time. To reduce the runtime of attraction force computation, we compute a *spectral sparsification* G' of a graph G, reducing the number of edges to $O(n \log n)$. By using spectral sparsification, we ensure that important edges are kept in the sparsified graph. This sparsification reduces the runtime of the attraction force computation to sublinear in the number of edges for dense graphs.

We combine this spectral sparsification approach to reduce attraction force computations with the vertex sampling methods used in our SL algorithms to create the *Spectral Sparsification (SS)* family of force computation algorithms. Depending on the vertex sampling method used, we present the following variations:

Algorithm SSR (SS Random): uses random sampling.
Algorithm SSG (SS Geometric): uses geometric sampling.
Algorithm SSC (SS Combinatorial): uses combinatorial sampling.

Algorithm 4: SLC (Sublinear Combinatorial)

```
Input: Graph $G = (V, E)$, update size $|U|$, sample size $|S|$
Output: Drawing $D$ of $G$
Initialize $D$ using the radial drawing of the BFS tree of $G$ rooted at the center;
Partition $V$ into an array of arrays $V_p$ based on the BFS level each vertex falls under; // 20% for each partition
$P$ : array of percentages summing up to 100%; // percentage of sample set sampled from each partition
$j_{prev} = 1$;
$l_{prev} = 1$;
for iterations in 1 to 300 do
    for $i$ in 1 to $|V_p|$ do
        RepulseSample($V$, $V_p[i]$, $j_{prev}$, $l_{prev}$, $U$, $S \times P[i]$); // sample vertices from each partition based on the percentage for that partition
    end
    AttractionForce($V$, $E$);
    $j_{prev} = (j_{prev} + |U|) \bmod |V|$;
    $l_{prev} = (l_{prev} + |U| \times |S|) \bmod |V|$;
end
return $D$
```

Algorithm 5: SSR (Spectral Sparsification Random)

```
Input: Graph $G = (V, E)$, update size $|U|$, sample size $|S|$
Output: Drawing $D'$ of a spectral sparsification $G'$ of $G$
$G'$ : spectral sparsification of $G$;
$T'$ : BFS spanning tree of $G'$ rooted at the center;
Initialize $D'$ using a radial tree drawing of $T'$;
$I$: array of integers from 1 to $|V|$; // for random sampling
$j_{prev} = 1$;
$l_{prev} = 1$;
for iterations in 1 to 300 do
    RepulsionSample($V$, $I$, $j_{prev}$, $l_{prev}$, $|V|^{0.5}$, $|V|^{0.2}$);
    AttractionForce($V$, $E'$);
    $j_{prev} = (j_{prev} + |U|) \bmod |V|$;
    $l_{prev} = (l_{prev} + |U| \times |S|) \bmod |V|$;
end
return $D'$
```

Algorithm 5 describes the SSR algorithm. This algorithm is defined by replacing the analysis and initialization of SLG (Algorithm 2) with computing the spectral sparsification $G' = (V, E')$ of $G = (V, E)$ and the BFS spanning tree of G', as well as by using E' instead of E for the attraction force computation. Similarly, the SSG and SSC algorithms are defined by replacing the analysis and initialization steps of SLG and SSC respectively.

The SS algorithms only draws the sparsified graph G'. To obtain a drawing of G based on the drawing D' of G produced by the SS algorithms, we present the *SS-G* family of algorithms. *SSR-G* takes D' produced by SSR and adds back all the edges removed by spectral sparsification to produce a drawing of G. Similarly, *SSG-*

G and *SSC-G* adds back the edges to the results of SSG and SSC respectively.

4. Experiments

We implemented the repulsion force computations in Javascript as a replacement for D3's [BOH11] built-in repulsion force implementation, based on Gove's implementation [Gov19b]. We used NetworkX [HSSC08] to compute the BFS tree rooted at the center of the graph, the radial tree layout of Tulip [AAB*17], and the effective resistance implementation of [ENH17]. We used a Dell OptiPlex 7060 desktop with Intel Core i7, 16 GB RAM. We conduct four experiments to compare the performance of our algorithms with RVS:

1. Comparison of 0702, 0602, 0502 with RVS
2. Comparison of SLR, SLG, and SLC with RVS
3. Comparison of SSR, SSG, and SSC with RVS' (RVS on G')
4. Comparison of SSR-G, SSG-G, and SSC-G with RVS

In each experiment, for each dataset we executed one run each per algorithm to determine the runtime.

We use the quality metrics for graph layout based on the shape-based metric and number of edge crossing. The *shape-based metric* [EHNK17] measures the *faithfulness* of graph drawing, i.e., how well the *shape* of the drawing represents the structure of the graph. The shape-based metrics are shown to be effective for measuring quality of drawings of large graphs [EHNK17].

The experiment was conducted with real-world benchmark data sets including social networks and benchmark graphs used in [BGKM10, Gov19b, Wal00]. Table 1 shows the details.

4.1. Comparison of 0702, 0602, 0502 vs. RVS

In this experiment, we compare our SL algorithms using different update sizes to RVS. Based on the analysis, initialization, and update sizes used for the repulsion force computation, we expect our methods to run faster than RVS without a significant loss in quality. We formulate the following hypotheses:

Hypothesis 1: 0702, 0602, and 0502 run faster than RVS; 0502 runs faster than 0602, which runs faster than 0702.

Hypothesis 2: 0702, 0602, and 0502 produce drawings with similar quality to drawings computed by RVS.

We compute the runtime improvement using the formula:

$$RI = \frac{t(RVS) - t(SL)}{t(RVS)}$$

where $t(RVS)$ is the time taken by RVS and $t(SL)$ is the time taken by our algorithms. Meanwhile, for improvement on both shape-based and edge crossing metrics, we use the formula:

$$MI = \frac{m(SL) - m(RVS)}{m(RVS)}$$

where $m(RVS)$ and $m(SL)$ are the metrics computed for RVS and our algorithms respectively.

Figure 2 shows the runtime improvement of 0702, 0602, and 0502 over RVS, with the averages shown in Figure 5(a). It can be

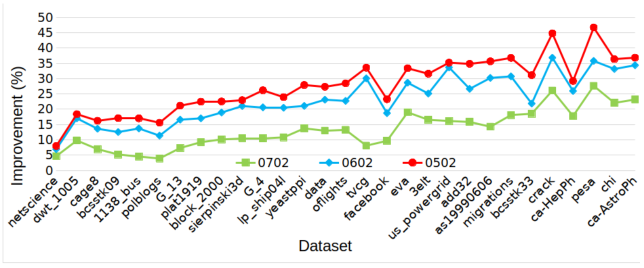


Figure 2: *Runtime improvement of 0702/0602/0502 over RVS*

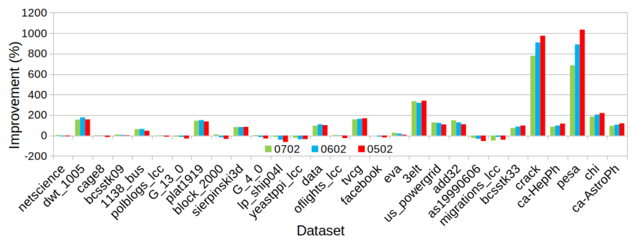


Figure 3: *Shape-based metrics improvement of 0702/0602/0502 over RVS*

seen that all three obtained lower runtime compared to RVS, validating Hypothesis 1. The runtime improvements are more significant on larger graphs, as seen with the increasing trend in Figure 2. Comparing between 0702, 0602, and 0502, 0502 obtained the highest runtime improvements over RVS at an average of 27.5%, compare to 22.5% for 0602 and 18% for 0702.

From the results of shape-based and edge crossing metrics in Figures 3 and 4 with the averages in Figures 5(b) and 5(c), we see that 0702, 0602, and 0502 obtain significant improvement in quality metrics over RVS, at an average of 104.5% for shape-based metrics and 30% for edge crossing, validating Hypothesis 2. The most significant improvements for shape-based metrics are seen with *pesa* and *crack*, while for edge crossing metrics, 1138_*bus*, *pesa*, and *crack* obtain the three largest improvements.

Comparing between 0702, 0602, and 0502, the average improvement on shape-based are better on 0502 and the average improvement on edge crossing metrics of 0502 only has a 3% difference to 0602 while the runtime improvement is 5% higher. Therefore, we can fix the update size to $|V|^{0.5}$ and the sample size to $|V|^{0.2}$ to obtain the lowest runtime without significant loss of quality.

In summary, we validate Hypothesis 1 and Hypothesis 2; 0702, 0602 and 0502 runs faster than RVS, with even better results in quality metrics.

4.2. Comparison of SLR, SLG, SLC vs. RVS

After confirming the efficiency and effectiveness of 0702, 0602, and 0502, we conduct experiments comparing the SL family of algorithms, SLR, SLG, and SLC to RVS. Based on the results of Section 4.1, we use $|U| = |V|^{0.5}$ and $|S| = |V|^{0.2}$ for all experiments. We expect the SL algorithms to obtain both runtime improvement

Dataset	$\|V\|$	$\|E\|$	$\|E'\|$	RVS	0702	0602	0502	SLR	SLG	SLC	RVS'	SSR	SSG	SSC
netscience	379	914	489	5.82	5.55	5.42	5.36	5.36	5.48	5.43	5.43	5.37	5.45	5.40
dwt_1005	1,005	4,813	3,018	7.00	6.31	5.81	5.71	5.71	5.89	5.70	5.70	5.41	5.26	5.60
cage8	1,015	6,009	3,052	6.84	6.37	5.91	5.73	5.73	5.98	5.68	5.68	5.57	5.67	5.58
bcsstk09	1,083	9,760	3,287	7.12	6.75	6.23	5.90	5.90	6.07	5.98	5.98	5.67	5.78	5.61
1138_bus	1,138	2,596	1,305	6.88	6.56	5.93	5.71	5.71	5.72	5.70	5.70	5.57	5.72	5.65
polblogs	1,222	16,714	3,773	7.25	6.96	6.42	6.12	6.12	6.16	6.16	6.16	5.71	5.84	5.72
G_13	1,647	6,487	5,298	7.42	6.87	6.19	5.85	5.85	5.35	5.84	5.84	5.29	5.33	5.80
plat1919	1,919	17,159	6,301	8.06	7.31	6.68	6.25	6.25	5.98	6.45	6.45	5.94	6.09	5.93
block_2000	2,000	9,912	6,603	7.94	7.14	6.44	6.15	6.15	6.87	5.93	5.93	5.98	6.12	5.86
sierpinski3d	2,050	6,144		7.99	7.15	6.31	6.16	6.16	6.38	5.97				
G_4	2,075	4,769	3,442	7.96	7.13	6.32	5.88	5.88	5.44	5.81	5.81	5.78	5.94	5.82
lp_ship04l	2,166	6,379	3,613	8.11	7.23	6.45	6.17	6.17	6.90	5.76	5.76	5.82	5.98	5.83
yeastppi	2,224	6,609	3,723	8.23	7.10	6.50	5.94	5.94	6.30	5.87	5.87	5.90	6.07	5.92
data	2,851	15,093	9,851	8.70	7.57	6.69	6.33	6.33	6.82	6.30	6.30	6.23	6.40	6.20
oflights	2,905	15,645	10,061	8.84	7.67	6.83	6.33	6.33	6.98	6.37	6.37	6.19	6.33	6.20
tvcg	3,213	10,140	5,634	9.22	8.48	6.45	6.13	6.13	8.85	6.24	6.24	6.24	6.48	6.24
facebook	4,039	88,234	14,566	10.46	9.45	8.50	8.02	8.02	8.76	8.97	8.97	6.61	6.86	6.64
eva	4,475	4,652	4,595	9.77	7.91	6.97	6.51	6.51	7.93	6.35	6.35	6.27	6.54	6.34
3elt	4,720	13,722	8,671	9.85	8.22	7.38	6.74	6.74	8.76	6.35	6.35	6.54	6.81	6.73
us_powergrid	4,941	6,594	4,941	10.10	8.47	6.70	6.54	6.54	9.10	6.63	6.63	6.46	6.70	6.42
add32	4,960	14,422	9,165	10.30	8.67	7.56	6.72	6.72	8.84	7.03	7.03	6.57	6.84	6.47
as19990606	5,188	9,930	9,637	10.35	8.88	7.23	6.66	6.66	9.37	6.59	6.59	6.57	6.84	6.63
migrations	6,025	9,378	8,541	10.73	8.79	7.44	6.79	6.79	10.62	6.70	6.70	6.59	6.50	6.58
bcsstk33	8,738	300,321		16.78	13.67	13.12	11.56	11.56	12.86	10.78				
crack	10,240	30,380	20,533	14.27	10.55	9.01	7.88	7.88	10.48	8.20	8.20	8.06	8.63	7.57
ca-HepPh	11,204	117,649	45,370	14.06	11.56	10.41	9.95	9.95	15.50	10.18	10.18	8.90	9.41	8.36
pesa	11,738	45,652	23,885	15.70	11.36	10.08	8.37	8.37	11.65	8.58	8.58	8.24	8.69	7.61
chi	13,174	44,384		14.00	10.91	9.36	8.91	8.91	15.49	9.47				
ca-AstroPh	17,903	197,031		17.70	13.59	11.61	11.18	11.18	27.83	9.94				

Table 1: *Data sets and runtime (in seconds) of algorithms.*

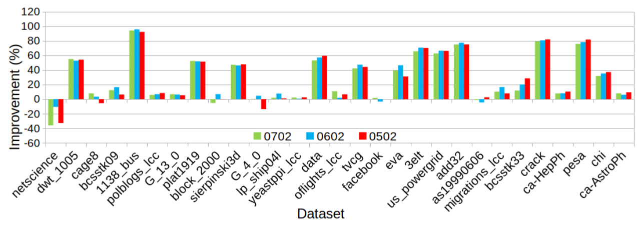
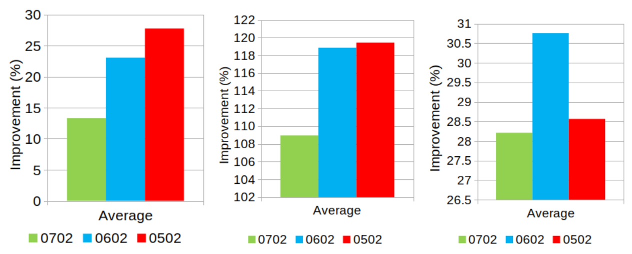


Figure 4: *Edge crossing metrics improvement of 0702/0602/0502 over RVS*



(a) Runtime (b) Shape-based (c) Edge crossing

Figure 5: *Average improvement of 0702/0602/0502 over RVS*

and improvement on quality metrics compared to RVS. We formulate the following hypotheses:

Hypothesis 3: SLR, SLG, and SLC run faster than RVS.

Hypothesis 4: SLR, SLG, and SLC perform better than RVS on shape-based and edge crossing metrics.

Figure 6 displays the improvement in the runtime of SLR, SLG, and SLC over RVS, with the average shown in Figure 9(a). All SL algorithms obtain runtime improvement over RVS, on average 12.5% by SLG and 27.5% for SLR and SLC, supporting Hypothesis 3. In general, the improvement is greater on larger graphs. It should also be noted that the lower average improvement on SLG compared to SLR and SLC is highly affected by an outlier, *chi*, where SLG runs 40% slower than RVS, whereas most other datasets has SLG running faster than RVS.

Figures 7 and 8 shows the improvements of the SL algorithms over RVS on shape-based and edge crossing metrics respectively.

All three SL algorithms obtain significant improvements over RVS on both metrics, as seen from the average improvements in Figure 9 (b) and (c) of around 104.5% for shape-based metrics and 30% for edge crossing metrics, supporting Hypothesis 4. SLG and SLC obtain the highest improvements on edge crossing metrics, with SLC being significantly faster than SLG, showing the strengths of SLC.

crack and *pesa*, two mesh datasets, score particularly well on shape-based metrics. On edge crossings, mesh datasets such as *add32*, *crack*, *chi* and *3elt* also score particularly well as does *1138_bus*. The only outlier to improvements in edge crossing metrics is *netscience*, a smaller, sparse graph, with other datasets producing improvements over RVS.

Figure 17 shows the visual comparison of layouts computed by SLR, SLG, SLC, and RVS. We see that the SL algorithms are able to untangle mesh structures better than RVS, such as *dwt*_1005, *sierpinski*3*d*, and *data*. These results are consistent with the improvements in shape-based metrics. Thus, the results show that the SL algorithms may be better suited for mesh structures.

In summary, we validate Hypothesis 3 and Hypothesis 4; SLR, SLG and SLC run faster than RVS, with significant improvement in quality metrics.

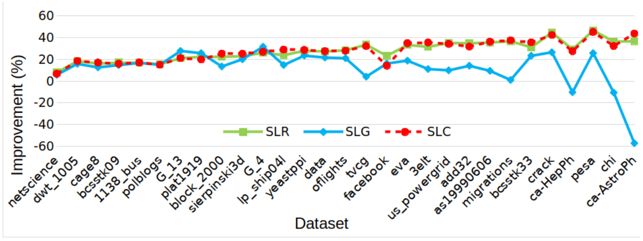


Figure 6: *Runtime improvement of SLR/SLG/SLC over RVS*

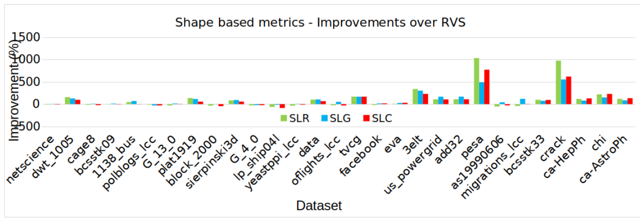


Figure 7: *Shape-based metrics improvement of SLR/SLG/SLC over RVS*

4.3. Comparison of SSR, SSG, SSC vs. RVS’

We also conduct experiments to compare the SS algorithms, SSR, SSG, and SSC, to RVS’ (RVS on G'), to examine the effectiveness of our spectral sparsification-based approach. As with the experiments in Section 4.2, we use $|U| = |V|^{0.5}$ and $|S| = |V|^{0.2}$ for all experiments. In this experiment, we only draw $G' = (V, E')$, a spectral sparsification of a dense graph $G = (V, E)$, with $O(nlogn)$ edges, for attraction force computation in sublinear time. Table 1 shows the details of the differences in size between E and E'.

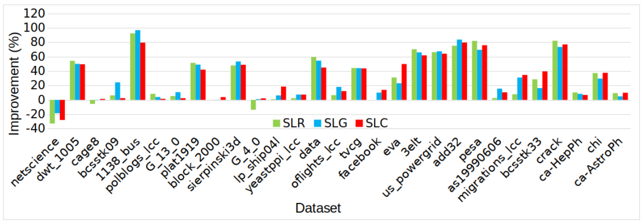
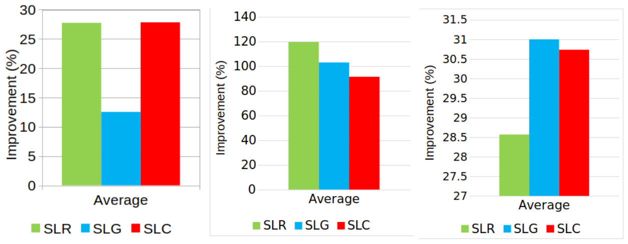


Figure 8: *Edge crossing metrics improvement of SLR/SLG/SLC over RVS*



(a) Runtime (b) Shape-based (c) Edge crossing

Figure 9: *Average improvement of SLR/SLG/SLC over RVS*

As with the SL experiments, we expect that the SS algorithms will obtain improvements on runtime, shape-based metrics, and edge crossing metrics over RVS, with visual comparisons showing similar quality to RVS. We formulate the following hypotheses:

Hypothesis 5: SSR, SSG, and SSC run faster than RVS’.
Hypothesis 6: SSR, SSG, and SSC achieve significant improvement on shape-based and edge crossing metrics over RVS’.

Figure 10 shows the runtime improvement of SSR, SSG and SSC over RVS’, and Figure 13(a) shows the average runtime improvement. All SS algorithms achieve significant runtime improvement of average 28% over RVS’, supporting Hypothesis 5. This improvement is more notable among large datasets, as shown in Figure 10 where the curve shows an increasing trend in general. One dataset, *G*_13, is an outlier with less improvement than other datasets, however this is less drastic than that of *chi* in the SL experiments.

Figures 11 and 12 show the improvement on the shape-based and edge crossing metrics over RVS’, where on average all SS algorithms obtain significant improvement over RVS, with 58% on shape-based metrics (Figure 13 (b)) and 28% on edge crossing metrics (Figure 13 (c)), which supports Hypothesis 6. SSC and SSR obtain the two highest average improvements in metrics.

The highest shape-based metrics improvements can be seen on mesh datasets such as *crack*, *pesa*, and 3*elt*, while *us*_*powergrid* also obtain one of the highest edge crossing metrics improvements.

Figure 18 shows visual comparisons between the layouts computed by RVS’, SSR, SSG and SSC. In these visual comparisons, only edges that are included in G' are drawn.

Clearly, SSR, SSG and SSC layouts perform significantly better than RVS’, highlighting the mesh structure of *bcsstk09* and *crack*,

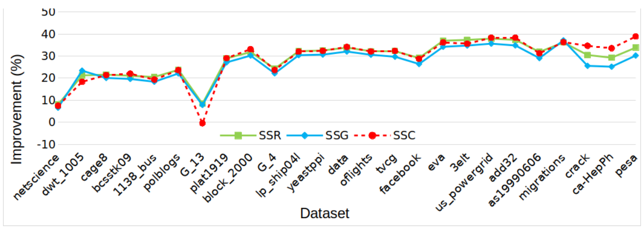


Figure 10: *Runtime improvement of SSR/SSG/SSC over RVS'*

which is consistent with the improvement in quality metrics and exceeds the expectations. With regards to the comparison between SSR, SSG, and SSC, there are not much differences in quality metrics and visual comparison.

In summary, the experiments support Hypotheses 5 and 6, showing that SSR, SSG, and SSC run faster than RVS' with significant improvement in shape-based and edge crossing metrics. Visual comparisons show that the SS algorithms can visualize mesh structures better than RVS.

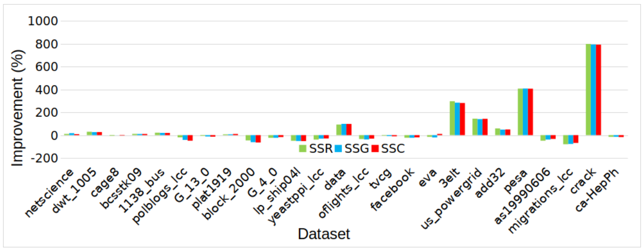


Figure 11: *Shape-based metrics improvement of SSR/SSG/SSC over RVS'*

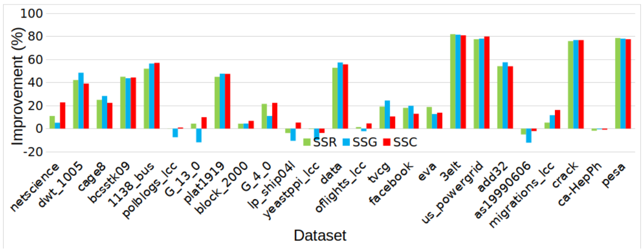


Figure 12: *Edge crossing metrics improvement of SSR/SSG/SSC over RVS'*

4.4. Comparison of SSR-G, SSG-G, and SSC-G vs. RVS

After confirming the effectiveness of our SS algorithms, we conduct experiments with the SS-G algorithms to examine the performance of the spectral sparsification-based approach to draw the original graphs. We take the results of the SS experiments presented in Section 4.3 and add back all the edges that were removed. We then compute the shape-based and edge crossing metrics and compare them to RVS.

Figures 14 and 15 shows the improvements on the shape-based

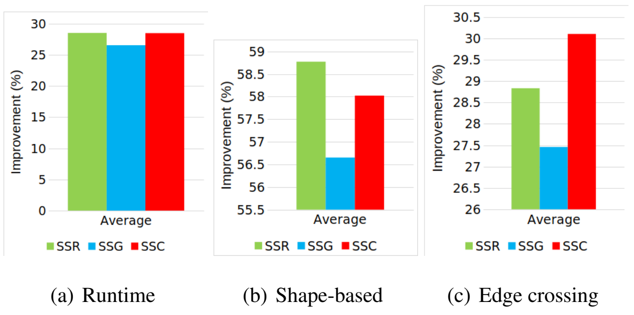


Figure 13: *Average improvement of SSR/SSG/SSC over RVS'*

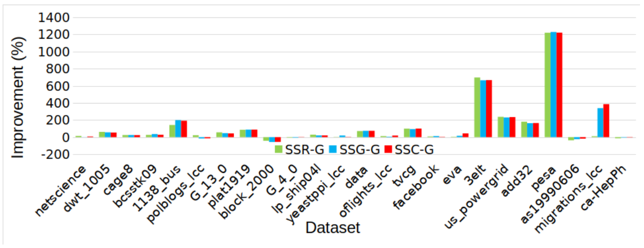


Figure 14: *Shape-based metrics improvement of SSR-G/SSG-G/SSC-G over RVS*

and edge crossing metrics over RVS, with average improvements shown in Figures 16 (a) and (b). SS-G on average obtains shape-based metrics improvements of around 130% and edge crossing metrics improvements of about 26% over RVS - these shape-based metrics improvements are higher than that of SL (Figure 9 (b)) and SS (Figure 13 (b)). Overall, SSC-G obtains the highest average improvements in metrics, followed by SSG-G.

Mesh datasets *pesa* and 3*elt* obtain the highest improvements on shape-based metrics, while 1138*_bus* obtain the highest edge crossing metrics improvement.

Figure 19 shows the drawings obtained by SSR-G, SSG-G, and SSC-G compared to RVS. It can be seen that by drawing all edges, we obtain drawings that display the shapes of the graph more clearly than RVS, especially for the graphs 1138*_bus* and *us_powergrid* which look almost tree-like in the SS versions where only G' was drawn (see Figure 18).

4.5. Discussion

Our experiments show that our sublinear force computation algorithms draw graphs faster than RVS, with SL and SS achieving average runtime improvements of 20% and 28% respectively over RVS. Our algorithms also obtain significant improvement on quality metrics over RVS, with 104.5% on shape-based metrics and 30% on edge crossing metrics for SL and 130% on shape-based metrics and 26% on edge crossing metrics for SS-G.

Our smart initialization may have contributed to the improved quality metrics. As opposed to RVS's phyllotaxis initialization which does not consider the topology of the graph when assigning

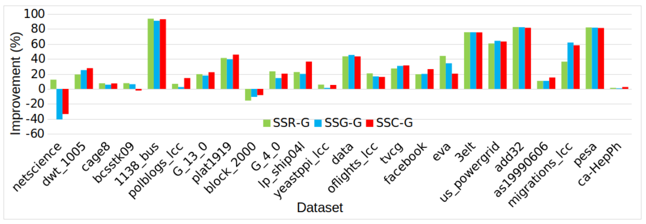
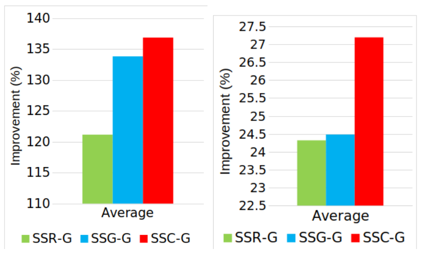


Figure 15: *Edge crossing metrics improvement of SSR-G/SSG-G/SSC-G over RVS*



(a) Shape-based (b) Edge crossing

Figure 16: *Average improvement of SSR-G/SSG-G/SSC-G over RVS*

initial coordinates, our initialization with the radial tree drawing of the BFS spanning tree places the graph center in the middle of the drawing and its neighbours close by, with peripheral vertices further away. This may help to position vertices with smaller graph theoretic distance closer to each other.

From visual comparisons in Figures 17, 18, and 19, we see that our SL, SS, and SS-G methods generally produce more aesthetically pleasing results compared to RVS, in particular for grid-like structures such as 3*elt* and *dwt*_1005. This may have arisen also from the use of the radial tree drawing of the BFS spanning tree rooted at the center, giving an initialization that is closer to the grid structure of the graph. We also see that in some cases, geometric sampling manage to unfold the grid structures better, such as with the SLG results for *data* and *crack* in Figure 17.

Our experiments demonstrate the strengths of our geometric and combinatorial sampling methods. With the SL algorithms, SLG and SLC obtain the two highest average improvements in edge crossing metrics, at 31% and 30.7% respectively.

The improvements over random sampling are more evident in the results for SS-G, which combines the sampling methods with spectral sparsification, where SSC-G clearly obtains the highest improvements in shape-based and edge crossing metrics at 137% and 27% respectively and SSG-G with the second highest - most notably, the shape-based metrics improvements are at around 15% better than SLR, which only uses random sampling, for SSC-G and 10% better for SSG-G. Therefore, we demonstrate that our geometric and combinatorial sampling methods, combined with spectral sparsification, perform significantly better than random sampling.

5. Conclusion

We present a framework for sublinear time force computation for big complex network visualization. We utilize a smart initialization method based on radial drawing of a spanning tree rooted at the center and vertex sampling methods for reducing the runtime of repulsion force computation, and spectral sparsification approach for reducing the runtime of attraction force computation.

Experiments showed that our SL, SS, and SS-G algorithms achieves significant runtime improvements over RVS on force computation, while achieving significant improvement on quality metrics (i.e., shape-based and edge crossing metrics) and visual comparisons. We also show that the combination of our new sampling methods, geometric and combinatorial, with spectral sparsification obtains higher improvements in metrics than random sampling.

Our future work is to consider a variety of analysis and initialization methods to further improve quality metrics. Furthermore, extensive experiments are required to draw more specific conclusions based on the structures of graph data sets. Another direction is to incorporate weighted edge sampling using effective resistance as the weights to reduce attraction force computations.

Acknowledgments

This research was supported by an ARC (Australian Research Council) DP grant.

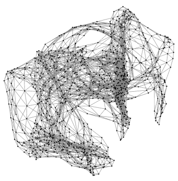
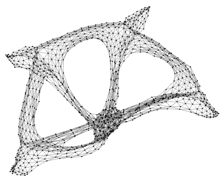
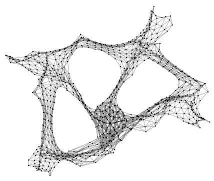
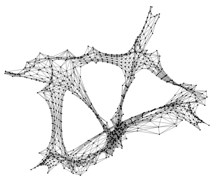
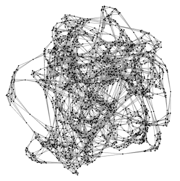
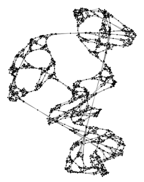
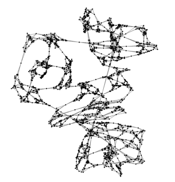
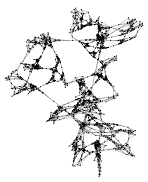
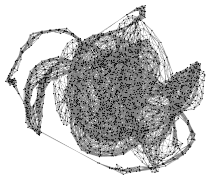
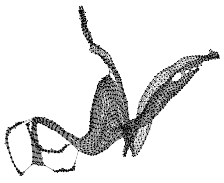
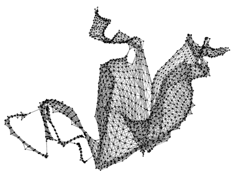
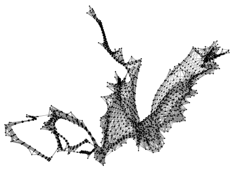
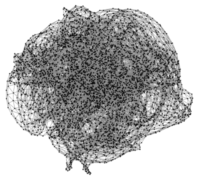
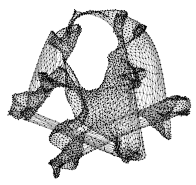
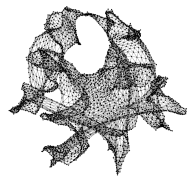
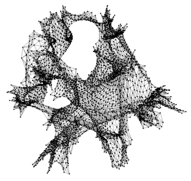
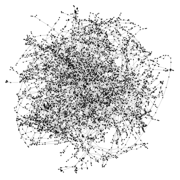
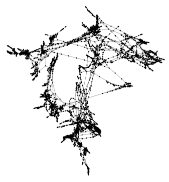
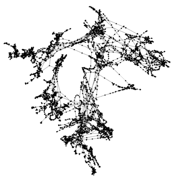
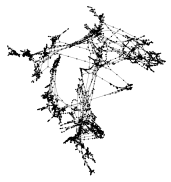
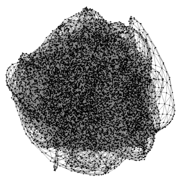
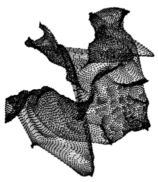
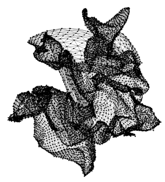
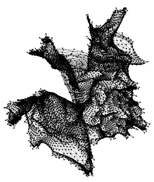
dwt_1005_RVS	dwt_1005_SLR	dwt_1005_SLG	dwt_1005_SLC
			
sierpinski3d_RVS	sierpinski3d_SLR	sierpinski3d_SLG	sierpinski3d_SLC
			
data_RVS	data_SLR	data_SLG	data_SLC
			
3elt_RVS	3elt_SLR	3elt_SLG	3elt_SLC
			
us_powergrid_RVS	us_powergrid_SLR	us_powergrid_SLG	us_powergrid_SLC
			
crack_RVS	crack_SLR	crack_SLG	crack_SLC
			

Figure 17: *Visual comparison of RVS, SLR, SLG and SLC*

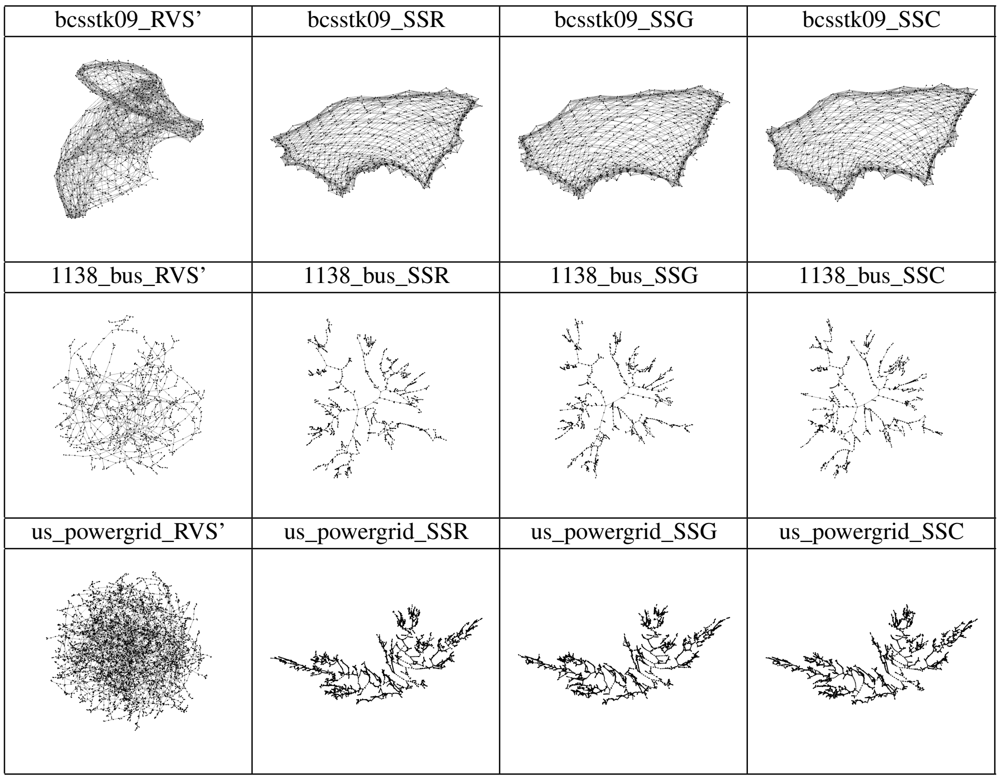


Figure 18: *Visual comparison of RVS', SSR, SSG and SSC*

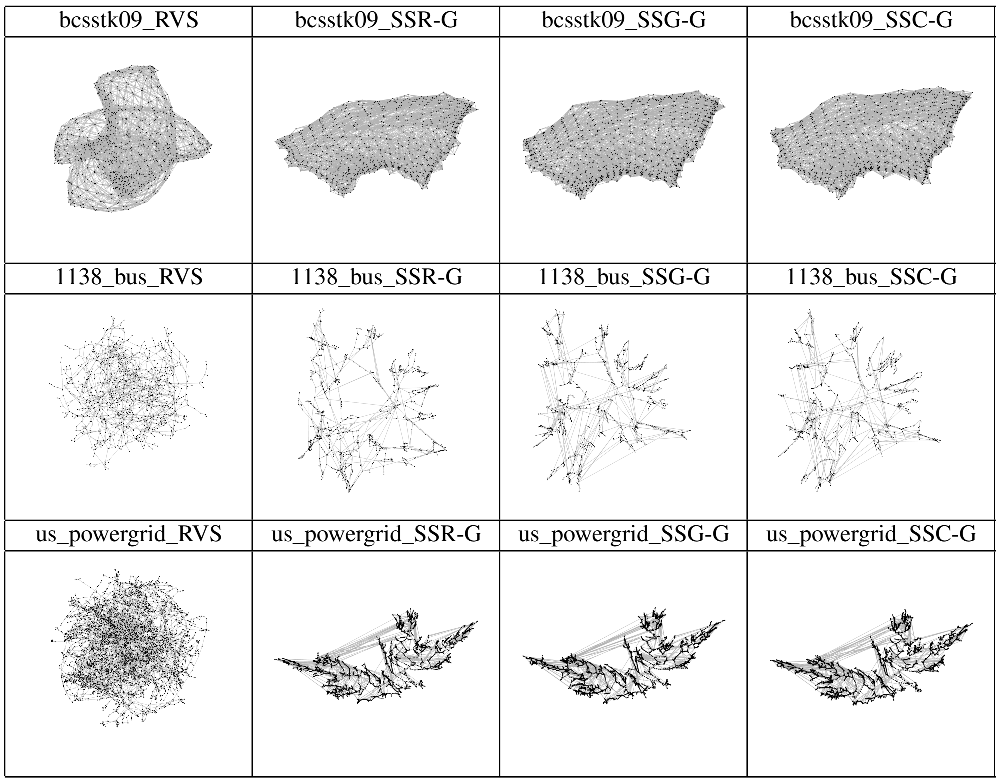


Figure 19: *Visual comparison of RVS, SSR-G, SSG-G and SSC-G*

References

[AAB*17] Auber D., Archambault D., Bourqui R., Delest M., Dubois J., Lambert A., Mary P., Mathiaut M., Melançon G., Pinaud B., Renoust B., Vallet J.: TULIP 5. In *Encyclopedia of Social Network Analysis and Mining*, Alhajj R., Rokne J., (Eds.). Springer, Aug. 2017, pp. 1–28. URL: https://hal.archives-ouvertes.fr/hal-01654518, doi:10.1007/978-1-4614-7163-9_315-1. 6

[BGKM10] Bartel G., Gutwenger C., Klein K., Mutzel P.: An experimental evaluation of multilevel layout methods. In *International Symposium on Graph Drawing* (2010), Springer, pp. 80–91. 6

[BH86] Barnes J., Hut P.: A hierarchical o (n log n) force-calculation algorithm. *nature 324*, 6096 (1986), 446. 2

[BOH11] Bostock M., Ogievetsky V., Heer J.: D^3 data-driven documents. *IEEE transactions on visualization and computer graphics 17*, 12 (2011), 2301–2309. 6

[Ead84] Eades P.: A heuristic for graph drawing. *Congressus numerantium 42* (1984), 149–160. 2

[Ead91] Eades P.: *Drawing free trees.* International Institute for Advanced Study of Social Information Science . . . , 1991. 3

[EHNK17] Eades P., Hong S.-H., Nguyen A., Klein K.: Shape-based quality metrics for large graph visualization. *J. Graph Algorithms Appl. 21*, 1 (2017), 29–53. 6

[ENH17] Eades P., Nguyen Q., Hong S.-H.: Drawing big graphs using spectral sparsification. In *International Symposium on Graph Drawing and Network Visualization* (2017), Springer, pp. 272–286. 6

[FR91] Fruchterman T. M. J., Reingold E. M.: Graph drawing by force-directed placement. *Software: Practice and Experience 21*, 11 (1991), 1129–1164. doi:10.1002/spe.4380211102. 2

[Gov19a] Gove R.: Force-directed graph layouts by edge sampling. In *2019 IEEE LDAV* (2019), IEEE. 2, 3

[Gov19b] Gove R.: A random sampling o (n) force-calculation algorithm for graph layouts. 2, 3, 6

[HJ04] Hachul S., Jünger M.: Drawing large graphs with a potential-field-based multilevel algorithm. In *International Symposium on Graph Drawing* (2004), Springer, pp. 285–295. 2

[HNM*18] Hong S.-H., Nguyen Q., Meidiana A., Li J., Eades P.: Bc tree-based proxy graphs for visualization of big graphs. In *2018 IEEE Pacific Visualization Symposium (PacificVis)* (2018), IEEE, pp. 11–20. 3

[HSSC08] Hagberg A., Swart P., S Chult D.: *Exploring network structure, dynamics, and function using NetworkX.* Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008. 6

[MHH*19] Meidiana A., Hong S.-H., Huang J., Eades P., Ma K.-L.: Topology-based spectral sparsification. In *2019 IEEE LDAV* (2019), IEEE. 3

[QE00] Quigley A., Eades P.: Fade: Graph drawing, clustering, and visual abstraction. In *International Symposium on Graph Drawing* (2000), Springer, pp. 197–210. 2

[ST11] Spielman D. A., Teng S.-H.: Spectral sparsification of graphs. *SIAM Journal on Computing 40*, 4 (2011), 981–1025. 3

[T*16] Teng S.-H., et al.: Scalable algorithms for data and network analysis. *Foundations and Trends® in Theoretical Computer Science 12*, 1–2 (2016), 1–274. 3

[Wal00] Walshaw C.: A multilevel algorithm for force-directed graph drawing. In *International Symposium on Graph Drawing* (2000), Springer, pp. 171–182. 6