Rapid Graph Layout Using Space Filling Curves

Chris Muelder and Kwan-Liu Ma, *Senior Member, IEEE*

Abstract— Network data frequently arises in a wide variety of fields, and node-link diagrams are a very natural and intuitive representation of such data. In order for a node-link diagram to be effective, the nodes must be arranged well on the screen. While many graph layout algorithms exist for this purpose, they often have limitations such as high computational complexity or node colocation. This paper proposes a new approach to graph layout through the use of space filling curves which is very fast and guarantees that there will be no nodes that are colocated. The resulting layout is also aesthetic and satisfies several criteria for graph layout effectiveness.

Index Terms— Information visualization, Graph layout, Space filling curves.

1 Introduction

Applications in many fields employ graph visualization to present data to the user. For example, document visualizations [22] can represent documents and their citation network with a graph. Similarly, social network visualizations [16] can represent people as nodes in a graph. These visualizations are frequently used to show inherent patterns in the data. In order to emphasize these patterns, many algorithms have been developed that determine how to lay out the nodes in space. Many algorithms exist to generate graph layouts, and they all generally share similar goals. Namely, they all strive to produce a layout that meets one or more criteria, such as minimizing edge crossings or having a short average edge length.

While most existing algorithms work well on sparse, mesh-like networks, real world networks such as scale-free networks are often large and dense. As these graphs grow larger and denser, speed and screen space also become issues. As the number of nodes on the screen increases, less screen space can be used for each node, so the more frugal a visualization needs to be in using that space. One way of doing this is by reducing the number of nodes shown in detail at any one time, such as through focus plus context interaction. But in order to allow interactivity, it is imperative that the graph layout is fast, as each interaction requires the graph to be adjusted. Many existing algorithms are not fast enough to interactively adjust, so they rely on distortion techniques such as fisheye lenses [12]. Also, most existing algorithms do not handle dense clusters in the graph very well, as the nodes inside the cluster get placed too close together to discern details.

The graph layout approach we present here avoids these limitations of previous layout algorithms, particularly when working with dense, scale-free networks. It proceeds by using a clustering algorithm to order the nodes of a graph, then using this ordering to arrange these nodes along a space filling curve of arbitrary complexity. This allows the graph to not only be initially laid out quickly, but also to adjust to user interaction even more quickly. Our approach is also very frugal with screen space, as it creates a layout which is space filling and guarantees that nodes inside a cluster are not placed too close together. It also guarantees good aspect ratios of clusters. In this paper, we describe our new approach, and also compare it quantitatively and qualitatively against seven other algorithms.

2 Related Work

This work draws upon several existing techniques in both the fields of graph visualization and space filling curves. Many graph layout algorithms have been developed, and there are several variations of space filling curves.

• *The authors are with the University of California, Davis, 1 Shields Ave, Davis, CA 95616 Email: {muelder, ma}@cs.ucdavis.edu*

Manuscript received 31 March 2008; accepted 1 August 2008; posted online 19 October 2008; mailed on 13 October 2008.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

2.1 Existing Graph Layout Techniques

Sometimes a graph has an intuitive layout where the vertices contain positional information that can be used, such as geographical locations. However, most graphs do not have such information, thereby requiring that the positions of vertices be derived. Special cases, such as trees or directional graphs, have certain properties which can be utilized to generate a layout. But, general graphs require more flexible algorithms. Also, some work has been done in graph layout using parallel hardware [9, 32], but many other layout algorithms could be ported to parallel hardware as well. Hachul and Jünger have produced an in-depth survey of many recent, general graph layouts [14].

The most intuitive, and probably the most often used graph layout algorithms for small graphs, are force-directed layouts such as Fruchterman-Reingold [10], LinLog [30], and Kamada-Kawai[21]. These algorithms position graphs by iteratively refining the positions of vertices in order to incrementally reduce an energy function. This energy function varies between algorithms, but generally has the property that it is a function of the distances between nodes and the weights of the edges between them. While these layouts are intuitive and generally considered aesthetic, they do not scale well to large or dense graphs. Some variants try to alleviate this issue, such as the Grid Variant Algorithm (GVA) of Fruchterman and Reingold [10] accelerates layout by limiting repulsive forces between nodes to those contained in the same area of a grid. However, they are still quite computationally expensive.

There are more efficient algorithms which use a multi-scale approach. Examples of these algorithms include the work of Cohen [3], the Fast Multipole Multilevel Method (FM^3) [13], and the Graph dRawing with Intelligent Placement (GRIP) algorithm [11]. These algorithms start by laying out a small approximation of a graph, then progressively laying out finer approximations of the graph, until the entire original graph is laid out. As this does not take multiple iterations, these algorithms generally perform far better than traditional force-directed approaches, while still producing similar results.

Even faster graph layout algorithms are available in the form of algebraic layouts, such as Algebraic Multigrid Computation of Eigenvectors (ACE) [24] or High Dimensional Embedding (HDE) [15]. These algorithms calculate layouts immediately using linear algebra techniques rather than iteratively or recursively laying out graphs according to force calculations. While not very intuitive, these algorithms can quickly produce layouts that are similar to the force-directed layouts. However, as shown in the survey by Hachul and Jünger [14], these algorithms can fail to produce a good layout in some cases, particularly when the graph is dense.

The graph layout approach most closely related to the one presented here is the treemap based graph layout [25], which does not fall into either category. This approach works by hierarchically clustering a graph, then applying a treemap to this hierarchy to derive placements for the nodes. It can attain the speed of algebraic approaches while avoiding issues such as nodes mapping to the same location. It also uses the entire screen, so that no screen space is wasted. However,

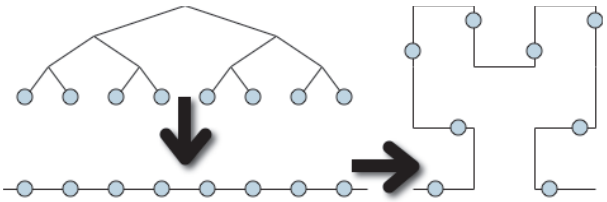


Fig. 1. *Graph layout approach:* First, we hierarchically cluster the nodes of the graph. Then, we traverse this hierarchy to generate an ordering. Finally, we use this ordering to place the nodes on a space filling curve.

when the hierarchy generated by the clustering algorithm is not well balanced, the results often end up with regions with poor aspect ratios. While work has been done on improving treemap aspect ratios, they do not generally apply to binary trees, as are used in the treemap graph layout algorithm.

2.2 Matrix Ordering

The approach of this paper essentially reduces the problem of graph layout to one of matrix ordering, so it is important to consider what work has been done on ordering of adjacency matrices. Mueller et al. provide a survey of several different matrix ordering algorithms, including breadth-first search, depth first search, RCM, King, Sloan, and spectral decomposition [27]. Fekete et al. also explore matrix orderings in their work on ZAME [8], in which they describe the use of a modified version of HDE and an approximated traveling salesmen algorithm for matrix ordering. Also, any graph clustering can be used to derive a matrix ordering.

2.3 Graph Clustering

Graph clustering is a challenging problem in its own right. There are as many kinds of clustering algorithms as there are kinds of graphs that need to be clustered. However, graph data is frequently clustered in order to create an overview or allow interactions such as semantic zooming [25, 12]. Several clustering algorithms are hierarchical, such as agglomerative or divisive clustering [20]. Others, such as k-means, are not hierarchical. We choose to focus on clustering of small world networks, such as the modularity algorithm of Clauset, Newmann, and Moore [2], and the variant by Huang and Nguyen [18].

2.4 Space Filling Curves

The use of space filling curves in visualization is not that common, but has recently become more popular. PhylloTrees [29] use spirals similar to space filling curves to aid in the layout of hierarchical data. Several recent computer network visualizations have used a space filling curve to map the IP4 address space of the internet [17, 19, 28]. The work of Martin Wattenberg demonstrates the use of a space filling curve as an alternative to a treemap, and proves some nice properties of the resulting visualization, which are due to this curve [33]. The relationship between Wattenberg's 'jigsaw map' and a regular treemap is particularly relevant to this work, as it analogous to the relationship between the space filling graph layout approach of this paper and the treemap based graph layout [25].

3 A Space Filling Graph Layout

This paper proposes an approach to generating a graph layout through the use of space filling curves. As diagrammed in Figure 1, this approach consists of three steps. First, the clustering algorithm groups nodes together into a binary cluster hierarchy. This hierarchy is then traversed to generate a node ordering. Finally, the nodes are spaced out along a space filling curve according to the ordering. An example of a graph layout generated with this approach is shown in Figure 2.

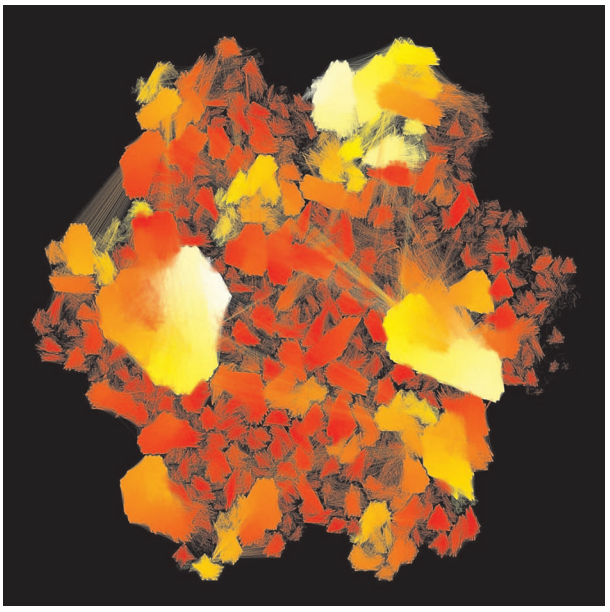


Fig. 2. *A protein homology graph laid out with our space filling curve based approach.* Color corresponds to depth in the clustering hierarchy. $|V| = 28,854, |E| = 1,180,816$

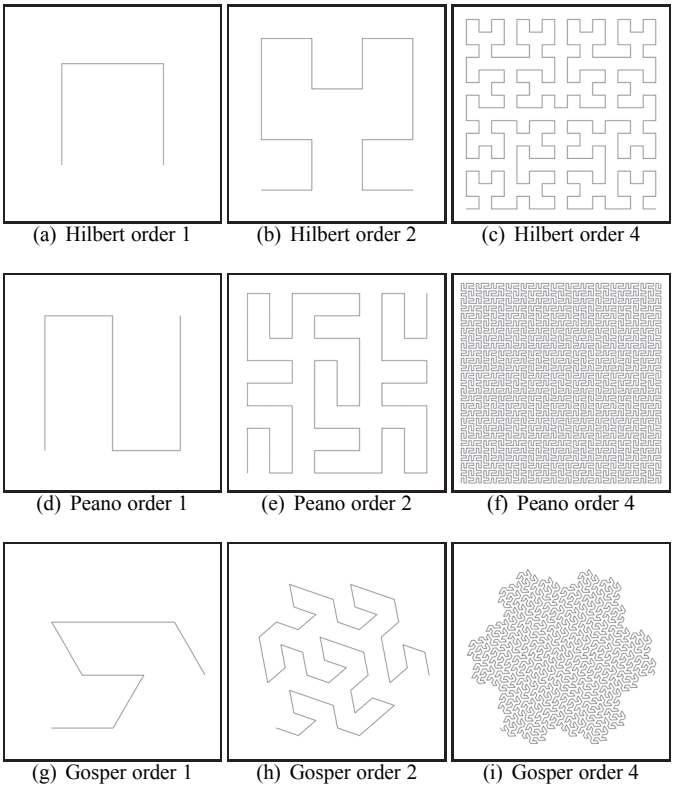
3.1 Node Ordering

In order to lay out the graph nodes along the space filling curve, we must first order them coherently. Several common matrix orderings were tried (including BFS, DFS, RCM, and King), but none of these reliably produced good enough results. We found that using a clustering algorithm reliably produces a good ordering results in many real world datasets. Since we are focusing on small world graphs (also called scale-free or power law graphs), we use the "Fast Modularity" community structure inference algorithm [2], which has been shown to be very effective for graphs of this type. This algorithm works by starting with each node as its own cluster, then iteratively agglomerating them together by merging the two clusters that maximize the increase in the modularity Q, which is defined as:

$$Q = \frac{1}{2|V|} \sum_{i,j} \left(a_{i,j} - \frac{d_i d_j}{2|V|} \right) \delta(i,j)$$

where $|V|$ is the number of nodes, $a_{i,j}$ is 1 if there is an edge between nodes i and j and 0 otherwise, d_i is the degree of node i, and $\delta(i,j)$ is 1 if nodes i and j are in the same cluster and 0 otherwise. As described in [2], this clustering algorithm runs in $O(|E| \times d \times log|V|)$, where d is the depth of the hierarchy (usually $O(log|V|)$).

Once this clustering hierarchy has been generated, the approach derives an ordering of nodes by traversing the tree depth-first. At each level of traversal, it is possible to choose which branch of the hierarchy to traverse next. Currently, our approach chooses the branch to traverse in the order determined by the clustering algorithm, which turns out to be prioritized by the size of cluster. This traversal is very fast, as it takes only $\Theta(|V|)$ operations. However, this does not take into consideration inter-cluster edges, so it is possible for such edges to be stretched out across the graph. A better ordering could probably be achieved by using the inter cluster edges to decide which branch of the hierarchy to descend down at each level, but that is beyond the scope of this paper. Regardless of the order in which the branches are traversed, the clustering information will be preserved. That is, nodes that are in the same branch of the clustering hierarchy will be placed close to each other in the ordering, and hence be in a contiguous area of the resulting layout. Since we are focusing on fairly well clustered graphs, this is the most important property for the datasets shown here.



(a) Hilbert order 1 (b) Hilbert order 2 (c) Hilbert order 4

(d) Peano order 1 (e) Peano order 2 (f) Peano order 4

(g) Gosper order 1 (h) Gosper order 2 (i) Gosper order 4

Fig. 3. Examples of space filling curves.

3.2 Space Filling Curves

Once the graph nodes have been ordered, they are mapped onto the screen according to a space filling curve. The primary property of space filling curves which we wish to utilize is that they exhibit what is referred to as "c-locality" [33]. That is, space filling curves satisfy the equation:

$$Distance(M(p_i), M(p_j)) < c\sqrt{|p_i - p_j|}$$

where p_i is the 1D position of a point i along the line, $M(p_i)$ is the mapping of that point into 2D space, and c is some small constant. Due to this property, graph nodes that are clustered together are guaranteed to be placed nearby on the screen. In particular, for a cluster of n nodes, they are guaranteed to be contained in a circular region of diameter $c\sqrt{n}$. Therefore, a space-filling curve based layout can guarantee a minimum aspect ratio for clusters. Space filling curves also have the property that they do not self-intersect, which means that graph nodes spaced out on a curve will never be placed in the exact same location, as is possible with algebraic layouts. In addition, space filling curves completely fill the area inside their boundary as their order $d \to \infty$. Thus, given a high enough order curve, graph nodes can be placed at any point on the screen, so that screen space is not wasted. However, since displays use discreet pixels, these curves only need to be of an order sufficient to cover every pixel, which is actually quite reasonable.

In this work we use three space filling curves: a Hilbert curve, a Peano curve, and a Gosper curve. All three curves are defined recursively, where the first level is defined by a fairly simple curve, and each segment of the curve of order d is replaced by a copy of the original pattern to create the curve of order $d+1$. Figure 3 shows various orders of all three curves.

The Hilbert curve (shown in Figures 3(a), 3(b), and 3(c)) is the simplest of the three space filling curves used here. At each recursion, every corner of the curve is replaced by 4 new corners. Thus, an order d Hilbert curve has 4^d points and $4^d - 1$ segments of length $\frac{1}{2^d}$.

Pseudo-Code 1 A space filling curve mapping function

```
SFCMap (value, order) -> (x, y) : {
  segment = floor(value * num_segments)
  if(order == 0){
    select the segment of the basis curve
    calculate (x, y) along this segment
  } else {
    subvalue = value * num_segments - segment
    (x', y') = SFCMap(subvalue, order - 1)
    transform (x', y') to (x, y) in right segment
  }
  return (x,y)
}
```

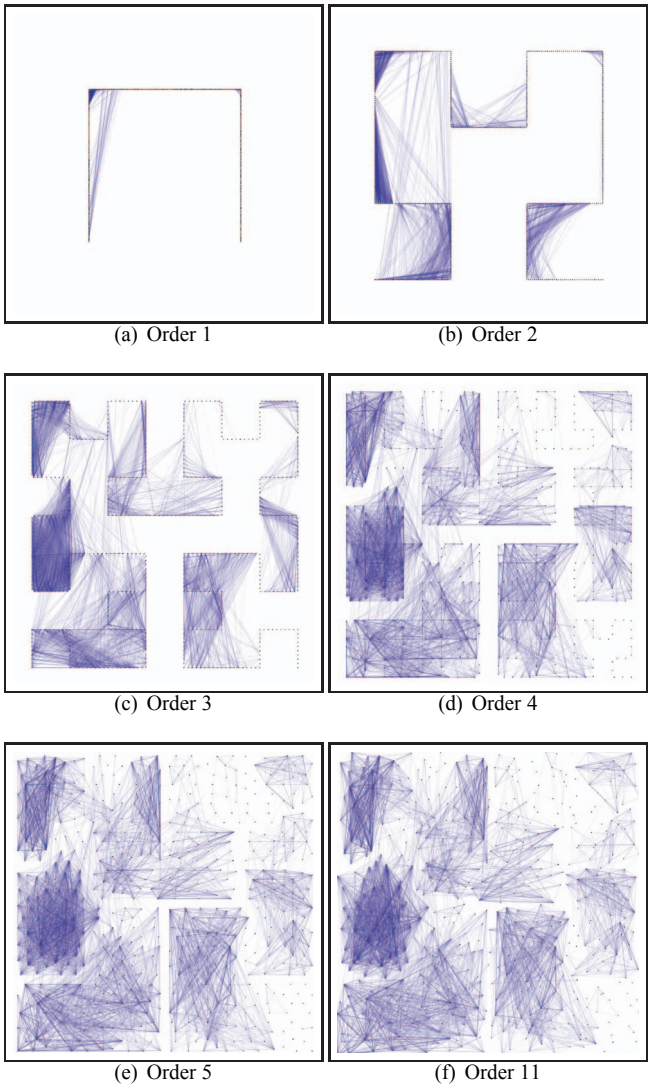
When this length is smaller than the size of a pixel, the Hilbert curve completely covers the screen. In other words, for a screen space with dimensions $S \times S$, the Hilbert curve needs to be of order $d = \Omega(log_2 S)$, which for most standard screen dimensions (up to $2,048 \times 2,048$) is $d = 11$.

The Peano curve (shown in Figures 3(d), 3(e), and 3(f)) is slightly more complex than the Hilbert curve, but is otherwise quite similar. The primary difference between it and the Hilbert curve is that it replaces each corner with 9 new corners instead of 4. Thus, an order d Peano curve has 9^d points and $9^d - 1$ segments of length $\frac{1}{3^d}$. Because of this added complexity, there are several variations of Peano curves, as there are various possible orientations for the recursive steps. Also, this complexity allows the curve to cover all pixels in even fewer iterations - for a screen space with dimensions $S \times S$, the Peano curve needs to be of order $d = \Omega(log_3 S)$, which for most standard screen dimensions (up to $2,187 \times 2,187$) is $d = 7$.

The Gosper curve (shown in Figures 3(g), 3(h), and 3(i)), sometimes called the 'Flow-snake,' is the most complex of the three. It is advantageous in that it does not impose hard borders in each recursion, thus it imposes the least amount of artificial structure onto the graph. However, it is not completely space filling, because it is not square and it leaves white space around the outside of its boundary. At each recursion, the Gosper curve replaces each segment of the curve with a copy of the order 1 Gosper curve. Thus, an order d Gosper curve has 7^d segments of length $\frac{1}{(\sqrt{7})^d}$. To cover a screen space with dimensions $S \times S$, the Gosper curve needs to be of order $d = \Omega(log_{\sqrt{7}} S)$, which for most standard screen dimensions (up to $2,401 \times 2,401$) is $d = 8$.

3.3 Node Mapping

The last step is to map the nodes onto one of the space filling curves. This is done by first arranging the nodes one dimensionally according to their ordering, and normalizing the node's positions to the range $[0,1]$. The simplest way to do this is to space them out evenly, with $\frac{1}{|V|}$ between each node. Once they are spaced out in one dimension, their positions on the space filling curve are calculated with a mapping function $M : \mathbb{Q} \to \mathbb{Q}^2$. Just as the space filling curves are defined recursively, this mapping function is also calculated recursively. At each intermediate level of the recursion, the mapping function calculates which segment of the curve it will be mapped to, then recurses on that segment. In the final level of the recursion, the function calculates which segment the node lands on and places the node along that segment appropriately. However, given a sufficiently high order curve, this final level will be at sub-pixel resolution, so the node could be placed anywhere in the region. Figure 4 shows the result of mapping a graph of similarity between network scans to Hilbert curves of various orders. As the order of the space filling curve increases, the nodes quickly converge to their final positions. As mentioned before, at order 11, the Hilbert curve covers every pixel, so in this example, recursing farther than 11 would not improve the results. By calculating the positions in this manner, we can map the nodes from the ordering to the screen in $O(d \times |V|)$, where d is the order of the fractal, which is logarithmic according to screen resolution, so for our purposes it is essentially constant. Pseudo-code for the basic space filling curve



(a) Order 1 (b) Order 2

(c) Order 3 (d) Order 4

(e) Order 5 (f) Order 11

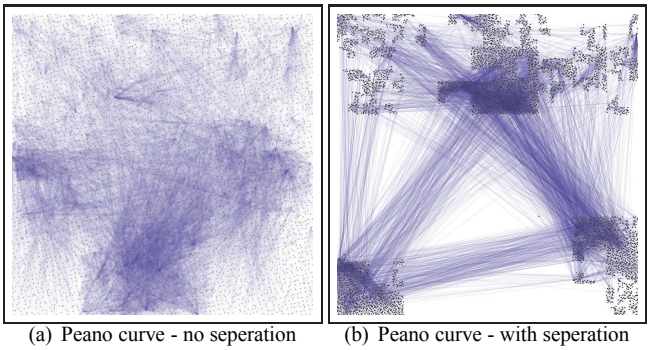
Fig. 4. *Mapping nodes to space filling curves of increasing order.* A graph of network scans shown with Hilbert curves of increasing order. Each increase in order places the nodes closer to their final positions. Order 11 is at the pixel level, so further recursions will have no effect.

mapping function is given in Pseudo-Code 1. While there are details which are specific to each curve, all three curves we use share the same basic algorithm.

3.4 Cluster Spacing

While placing the nodes out evenly on the space filling is simple and guarantees that all nodes are spaced out away from each other, overall the nodes end up with an even distribution across the screen. This is often considered to be not very aesthetic, as it looks as if the nodes were spread out at random. Also, it blurs the distinction between clusters, as there is no separation between them. It is possible to resolve these issues by adjusting the spacing between nodes. In particular, we want to increase the spacing between nodes in different clusters, while decreasing the spacing between nodes in the same cluster. Since we already have clustering information from the layout process, it is straightforward to use this clustering information to space out the nodes.

Nodes that are clustered together are closer in the clustering hierarchy, and therefore have similar depths in the hierarchy. So, the depths of two consecutive nodes in the tree traversal will be nearly the same



(a) Peano curve - no seperation (b) Peano curve - with seperation

Fig. 5. *Separating clusters.* By adjusting the spacing between nodes according to the clustering information, clusters can be separated.

when the two nodes are in the same cluster. Therefore, it is possible to space the nodes out according to the difference in their depths in the hierarchy. That is, when the difference between the clustering depths of two consecutive nodes is small, they should be placed closer together, and when the difference is large, they should be spaced out farther. We do this by calculating the spacing $s_{i,i+1}$ as:

$$s_{i,i+1} = |depth_{i+1} - depth_i|^k$$

where k is user defined. Examples of clusters spaced out this way are shown in Figure 5. As can be seen in the figure, clusters get compacted together and spaced out apart from each other.

3.5 Geometric Zoom

One advantage to using a very fast graph layout is that it can be used in an interactive system. A common interactive technique for graphs is the capability to zoom into a small section of the graph to show it in more detail. Existing approaches often use techniques such as fisheye lenses to distort an initial layout [12]. But such approaches require that the graph be laid out at full resolution to begin with.

By using our space filling curve based approach, the graph can be relaid out rapidly enough to allow interactivity, such as a geometric zoom. One simple way to do this is to increase the spacing factors of nodes that the user selects to focus on. Similar to how clusters can be separated by spreading them out on the curve, focal areas can be expanded to take up more space by increasing their portion of the curve. That is, the layout can perform a geometric zoom by increasing the spaces between nodes that the user selects. In our system, the user to selects nodes to zoom with the mouse, and adjusts the level of zoom with a slider widget. An example is shown in Figure 6.

4 Results

In order to test the effectiveness of our graph layout approach, we ran it on several graph datasets of various sizes. These datasets are summarized in Table 1. The "netscans" dataset (shown in Figure 4) is a complete weighted graph representing the similarity between network scans. However, to more clearly depict patterns in this graph, edges with a weight below a threshold value are not shown [26]. The "california" dataset (shown in Figures 5 and 8) consists of the links between the webpages found from a search for the word 'California' [4]. The "pgraph" dataset (shown in Figures 2 and 6) is a protein homology graph, which is relatively dense [7]. Finally, the "usafla" dataset (shown in Figure 7) is of the intersections and the streets between them in the state of Florida [5].

4.1 Scalability

One primary application for rapid graph layout is for very large graphs, which would take a very long time to lay out with traditional force-directed layouts. In order to be useful for this task, the approach has to be scalable to large datasets. As mentioned before, the actual layout

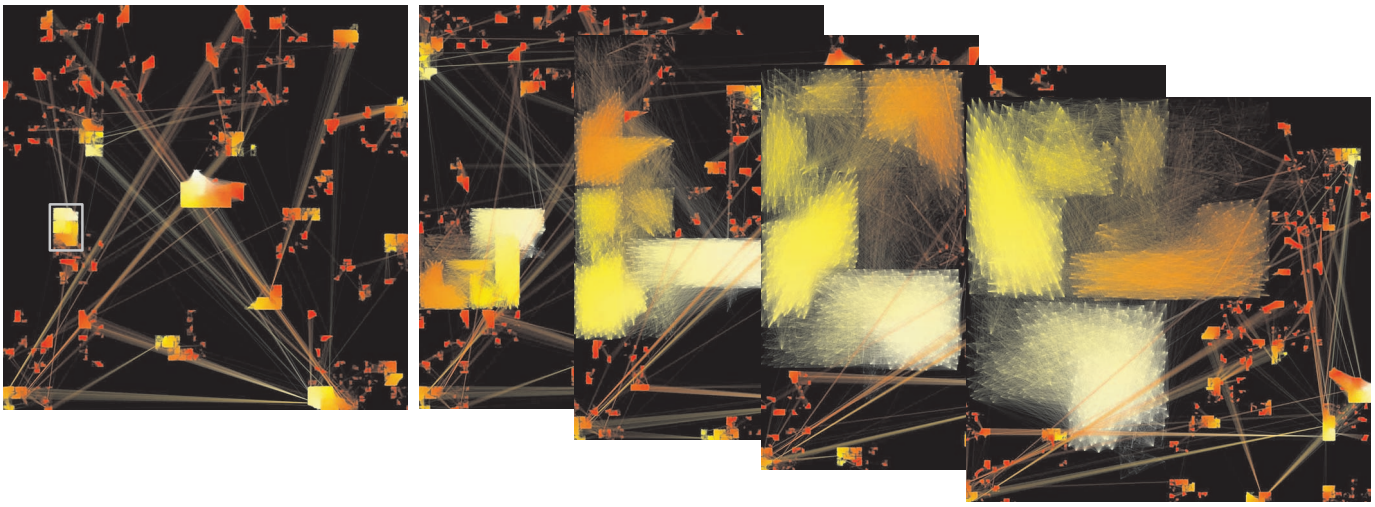


Fig. 6. *Geometric Zoom.* The graph can be distorted by increasing the spacing between selected nodes along the space filling curve. The left image shows the original graph and the other ones show the result of expanding the selected region by different zoom factors.

Table 1. *Results:* Time to generate a usable layout for several graphs

Graph	$\|V\|$	$\|E\|$	Clustering	Hilbert	Peano	Gosper	Total time
netscans	848	22,462	.0739s	0.0007s	0.0007s	0.0007s	0.074s
california	6,107	15,160	0.655s	0.0043s	0.0039s	0.0043s	0.659s
pgraph	28,854	1,180,816	9.169s	0.0212s	0.0206s	0.0220s	9.190s
usafla	1,070,376	2,712,798	20.531s	0.8185s	0.7689s	0.8318s	21.337s

process only takes $\Theta(|V|)$ operations, so this approach is limited by the clustering algorithm's complexity of $O(|E| \times d \times log|V|)$, which is still quite fast. The extra space used by this approach is only $\Theta(|V|)$ to store the hierarchy, which makes it quite efficient in terms of memory usage. As shown in Figure 7, we have tested our approach on graphs up to $|V| = 1,070,376, |E| = 2,712,798$ and still been able to quickly and reliably produce results with commodity hardware.

4.2 Comparison

The primary advantage to the approach presented here is that it is very fast. However, even if a graph layout is fast, it is useless unless the resulting layout is good. For instance, a randomized layout can be generated very fast, but the resulting layout will rarely be useful. Force-directed layouts, such as LinLog, generally produce quite good results, but take a long time to do so. Algebraic layouts are much faster, but can fail to produce good results. The treemap based layout is fast and often effective, but imposes structural artifacts such as poor aspect ratios, since everything is rectangular. The space filling curve based approach presented here attains speed comparable to the treemap layout, and provides the same guarantees of filling the screen and no node colocation, but it does not impose such a rigid structure or have the issues with 'skinny regions' that the treemap layout does. In order to demonstrate the quality of a space filling curve based layout, a comparison between it and several other layouts is given in Figure 8, where they are applied to the "California" graph [4]. We used the Boost Graph Library's implementation of GVA [1] Yehuda Koren's own implementations of ACE and HDE [23], Yusufov's implementation of GRIP src-grip, and the Open Graph Drawing Framework's implementation of FM^3 [31]. While several algorithms have parameters that can be adjusted, this would involve a trial and error process which would need to be included in the timing tests, so we use default parameters. Timing results were generated by running the programs on one core of a 2.66GHz Intel Xeon Mac Pro with 8GB of RAM and are presented in Table 2.

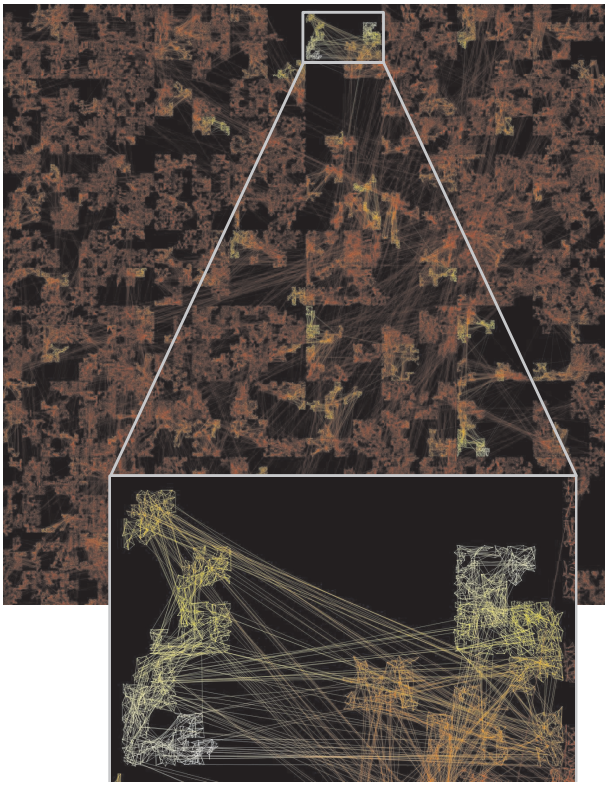


Fig. 7. *Scalability.* Our approach can scale to large graphs. This graph is of the streets in the state of Florida, $|V| = 1,070,376, |E| = 2,712,798$ One small region is expanded to show detail.

The LinLog layout, shown in Figure 8(a), was easily the slowest, taking 10,737 seconds to compute 200 iterations. In this layout, it is clear that there is a very tightly connected group of nodes, and many weakly connected or disconnected subgraphs. While it can be seen that there are actually three clusters in the center of the graph, the internals of these clusters can not be seen, since the nodes are so close together. Also, much of the space around the outside of the region is not utilized.

The Grid-Variant Algorithm (GVA) [14], shown in Figure 8(b), is a heuristically accelerated layout based on the Fruchterman-Reingold layout algorithm [10]. As can clearly be seen, this algorithm is much faster than LinLog, taking only 45 seconds to do 400 iterations. However, the results are not as good as LinLog, since the three clusters in the middle are not distinguishable from each other. It also induces a grid-like arrangement of many of the nodes due to the heuristic, which is not related to the graph. Also, GVA leaves even more of the surrounding area as whitespace. While the force-directed graph layouts may be intuitive and aesthetically pleasing, neither of them show the internal structures of the clusters.

The Fast Multipole Multilevel Method (FM^3) [13], shown in Figure 8(c), uses a multi-scale approach to generate a layout more quickly than traditional force directed layouts. It produced results similar to GVA, but in a fraction of the time. Also, the resulting layout does not have the grid-like artifact that GVA does. However, the results are even less useful than either of the previous two. The cluster in the center is packed even tighter than in GVA, and any internal properties of this cluster are indiscernible. In particular, there is no way to tell that there are three clusters inside. Also, even more space is left white around the borders, so that the disconnected components can be placed around the large one.

GRIP [11], shown in Figure 8(d), is another multi-scale algorithm, similar to FM^3. As can be seen in the figure, it performed even better than FM^3 in terms of speed, taking just over a second, yet produced very similar results. That is, the majority of the nodes are compacted into one small cluster, inside of which very few details can be seen, while the majority of screen space is allocated to the disconnected components.

The algebraic method ACE [24], shown in Figure 8(e), while very fast, produced the least useful results. Not only are the vast majority of nodes concentrated in a very small area, but it does not even separate disconnected components. It also leaves most of the screen blank. However, the worst issue was that it often failed to complete running. When it did finish running, it finished very quickly, but it often locked up and never reached completion. Similar results were presented in Hachul and Jünger's survey paper [14], though it could be an implementation bug.

Similarly to ACE, HDE [15], shown in Figure 8(f), also produced results very quickly, but which were not as useful as the force-directed algorithms. As in ACE, the majority of the nodes are grouped in one large cluster, with a few outliers on the other side of the screen. The three clusters are not distinct at all, and the disconnected components are not separated. Also, just like all the previously mentioned examples, HDE leaves a large portion of the screen empty.

The treemap based layout (Figure 8(g)) was made by applying the "Fast Modularity" algorithm [2]. splitting the regions such that edges are shortened, and randomizing node placement within their regions. The whole layout took less than a second to compute. The resulting layout very clearly shows three large clusters of nodes, with many internal nodes, and a large number of other nodes distributed around the screen. Within these clusters, interesting features, such as nodes of high degree, are easily visible. Similar features are also visible in the rest of the graph. However, there are artifacts, particularly near the edges of clusters, where nodes are spread out over skinny regions of the treemap. The treemap also imposes a fairly artificial rectangular shape to the graph, which can distract from other patterns that are actually present in the graph.

The space filling curve layouts (8(h), and 8(i)) were generated as described in this paper. The graph was clustered with the "Fast Modularity" algorithm [2], traversed to generate an ordering, then mapped to the screen with a space filling curve of sufficient order. That is, the Hilbert curve was order 11, and the Gosper curve was order 7. In all cases, the layout time is greatly dominated by the initial clustering calculation, just as in the treemap layout. However, this is done offline and only once, so that when the user interacts with the graph it can be updated very quickly. As with the treemap based layout, all three clusters can be seen clearly in each space filling curve based layout, and the internals of all three clusters can also be easily seen. Unlike the treemap based layout, these layouts do not encounter a problem with skinny clusters, due to the c-locality property of the space filling curves. While the Hilbert curve does still impose square boundaries on the clusters, the effect is much less pronounced than in the treemap layout. The Gosper curve, on the other hand, imposes no square boundaries, but it sacrifices filling the entire screen, as it leaves some space empty around its border. A Peano curve based layout was also generated, but was very similar in time and quality to the Hilbert curve based layout, and is omitted due to space constaints.

Table 2. *Speed comparison* of various algorithms on the "California" graph [4], $|V| = 6,107, |E| = 15,160$

	Iters or Depth	Time/itr	Clustering	Time
LinLog	200	53.7s/itr	N/A	10,737s
GVA	400	0.113s/itr	N/A	45.5s
FM^3	N/A	N/A	N/A	12.9s
GRIP	N/A	N/A	N/A	1.51s
ACE	N/A	N/A	N/A	0.19s or ∞
HDE	N/A	N/A	N/A	0.19s
Treemap	N/A	0.076s	0.655s	0.731s
Hilbert	11	0.0043s	0.655s	0.659s
Peano	6	0.0039s	0.655s	0.659s
Gosper	7	0.0043s	0.655s	0.659s

Overall, only the treemap based layout and the space filling curve based layouts clearly showed the three separate clusters in the graph. Further exploration reveals that the three clusters are in fact of different groups of websites - one for university sites, one for government sites, and one for all other sites.

4.3 Limitations

While we have shown this approach to be fast and effective at visualizing dense networks, it does still have some limitations. The approach as presented here is dependent on a good clustering to generate the ordering. So, if a dataset has no clear clustering, or if a poor clustering algorithm is chosen, there is a good chance that this approach will not work well. In particular, our approach will not perform well on artificial 'grid-like' graphs, which are prevalent in many graph drawing works. However, many real world networks do actually exhibit clustering. Finally, the space filling graph layout approach would run into a problem if the number of nodes matches exactly with the complexity of the space filling curve, as it would suddenly create many collinear points. For instance, if the number of nodes is exactly 4^n for some n, then the Hilbert curve based layout would place every node on the corner of the Hilbert curve of order n, resulting in a poor layout. However, this could be solved by adjusting the spacing factors or randomly jittering the nodes along the curve. While our current cluster separation approach has been effective on the data sets we have tried, it would theoretically fail in the pathological case of a graph with two identically sized clusters that are adjacent in the clustering hierarchy, as they would have zero spacing between them. However, this situation is very rare in real world data. While our approach guarantees that nodes are not placed in exactly the same location, it is still possible for them to get arbitrarily close, which can be a problem when nodes are of large or non-uniform size, or if they are being labeled. Finally, a space filling curve layout might not be as aesthetically pleasing as traditional force directed layouts, due to the artificial structure imposed

by the curve. However, just like the treemap based layout [25], the space filling curve based layout could be used as an initial layout for a force directed approach, which would refine the layout in very few iterations.

5 Future Work

While our space filling curve based approach to graph layout is effective, there are still improvements that can be made. Accounting for inter-cluster edges is yet to be solved by taking these edges into account when traversing the hierarchy. Alternately, other matrix ordering algorithms could be applied here, in which case the clustering hierarchy would not be necessary. There are many other space filling curves which could be explored, such as H-curves and Sierpenski curves, as some of them have better c-locality than others. Other interactive techniques such as semantic zooming could also be added to this framework, and they would work well due to the very low amount of time it takes to recalculate a layout.

6 Conclusion

We have described how graph layouts generated through the use of a space filling curve guarantee several nice properties. Using a clustering algorithm to generate the ordering guarantees that each cluster is located in a contiguous region of space. Because the curves never self-intersect, the layout guarantees that graph nodes are never placed on top of each other. Furthermore, due to the c-locality property of the space filling curves, this layout guarantees that clusters of nodes are arranged with a good aspect ratio. The scalability and efficiency of space filling curves has been demonstrated using several real world datasets, and the performance has been compared to seven existing algorithms. We have shown that this new approach to graph layout is quite effective at dealing with dense graphs and capable of clearly presenting features that many other algorithms could not show.

Acknowledgements

This research was supported in part by the Intel Corporation, the National Science Foundation through grants CCF-0634913, CNS-0551727, OCI-0325934, CNS-0716691, CCF-0808896, OCI-0749227, and OCI-0749217, and the Department of Energy through the SciDAC program with Agreement No. DE-FC02-06ER25777. We would also like to thank those who provided the data sets [4, 6, 7, 26] as well as Yehuda Koren, Andreas Noack, Roman Yusufov, and the authors of the Boost graph library and the Open Graph Drawing Framework for making the source of their graph layout implementations available online [1, 23, 31, 34].

References

[1] Boost graph library's GVA: `http://boost.org/libs/graph/doc/fruchterman_reingold.html`.

[2] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70:066111, 2004.

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

[4] Data. 'California' search results graph, `http://www.cs.cornell.edu/Courses/cs685/2002fa/`, accessed 12/10/07.

[5] Data. Graph of Florida streets from the 9th dimacs implementation challenge, `http://www.dis.uniroma1.it/~challenge9/download.shtml`, accessed 12/10/07.

[6] Data. Graph of San Francisco Bay Area streets from the 9th dimacs implementation challenge, `http://www.dis.uniroma1.it/~challenge9/download.shtml`, accessed 12/10/07.

[7] Data. Protein homology graph from Large Graph Layout project site, `http://bioinformatics.icmb.utexas.edu/lgl/`, accessed 12/10/07.

[8] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete. Zame: Interactive large-scale graph visualization. In *Proceedings of IEEE VGTC Pacific Visualization Symposium (PacificVis)*, 2008.

[9] Y. Frishman and A. Tal. Multi-level graph layout on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

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

[11] P. Gajer and S. G. Kobourov. Grip: Graph drawing with intelligent placement. In *International Symposium on Graph Drawing*, pages 222–228, London, UK, 2001. Springer-Verlag.

[12] E. R. Gansner. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, July 2005.

[13] S. Hachul. *A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs*. PhD thesis, Universitaet zu Koeln, 2002.

[14] S. Hachul and M. Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *International Symposium on Graph Drawing*, pages 235–250, 2005.

[15] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. In *International Symposium on Graph Drawing*, pages 207–219, London, UK, 2002. Springer-Verlag.

[16] J. Heer and d. boyd. Vizster: Visualizing online social networks. In *Proceedings of the 2005 IEEE Symposium on Information Visualization (InfoVis)*, 2005.

[17] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister. Census and survey of the visible internet (extended). technical report isi-tr-2008-649, usc/information sciences institute, february, 2008.

[18] M. L. Huang and Q. V. Nguyen. A fast algorithm for balanced graph clustering. *Proceedings of the 2007 IEEE Symposium on Information Visualization (InfoVis)*, pages 46–52, 2007.

[19] B. Irwin and N. Pilkington. High level internet level traffic visualization using hilbert curve mapping. In *ACM VizSEC 2005 Workshop*, pages 29–38, 2005.

[20] S. C. Johnson. Hierarchical clustering schemes. In *Psychometrika*, pages 2:241–254, 1967.

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

[22] W. Ke, K. Borner, and L. Viswanath. Major information visualization authors, papers and topics in the acm library. In *InfoVis04 Contest*, 2004.

[23] Y. Korens algorithm implementations (ACE and HDE): `http://research.att.com/~yehuda/index_programs.html`.

[24] Y. Koren, L. Carmel, and D. Harel. Ace: A fast multiscale eigenvectors computation for drawing huge graphs. *Proceedings of the 2002 IEEE Symposium on Information Visualization (InfoVis)*, 00:137, 2002.

[25] C. Muelder and K.-L. Ma. A treemap based method for rapid layout of large graphs. In *Proceedings of IEEE VGTC Pacific Visualization Symposium (PacificVis)*, 2008.

[26] C. Muelder, K.-L. Ma, and T. Bartoletti. A visualization methodology for characterization of network scans. In *ACM VizSEC 2005 Workshop*, pages 29–38, 2005.

[27] C. Mueller, B. Martin, and A. Lumsdaine. A comparison of vertex ordering algorithms for large graph visualization. *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on*, pages 141–148, 5-7 Feb. 2007.

[28] R. Munroe. Map of the internet. `http://www.xkcd.com/195/`.

[29] P. Neumann, M. S. T. Carpendale, and A. Agarawala. Phyllotrees: Phyllotactic patterns for tree layout. In *EuroVis*, pages 59–66, 2006.

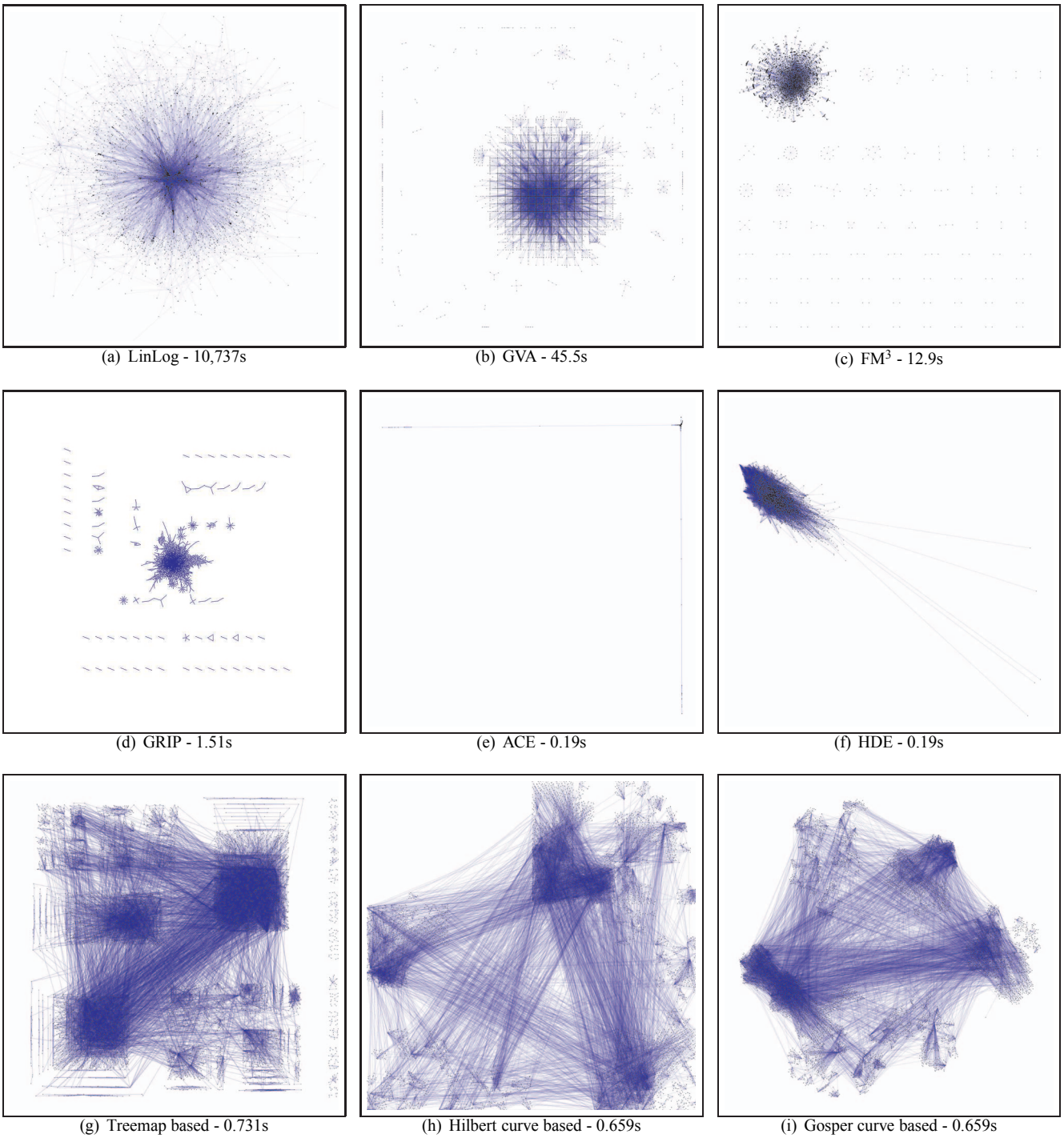
[30] A. Noack. An energy model for visual graph clustering. *Lecture Notes in Computer Science*, 2912:425–436, Mar. 2004.

[31] The open graph drawing framework: `http://www.ogdf.net/`.

[32] A. Tikhonova and K.-L. Ma. A scalable parallel force-directed graph layout algorithm. In *Proceedings of Parallel Graphics and Visualization Symposium*, April 2008.

[33] M. Wattenberg. A note on space-filling visualizations and space-filling curves. In *Proceedings of the 2005 IEEE Symposium on Information Visualization (InfoVis)*, page 24, Washington, DC, USA, 2005. IEEE Computer Society.

[34] R. Yusufovs implementation of GRIP: `http://www.cs.arizona.edu/~kobourov/GRIP`.



(a) LinLog - 10,737s (b) GVA - 45.5s (c) FM^3 - 12.9s

(d) GRIP - 1.51s (e) ACE - 0.19s (f) HDE - 0.19s

(g) Treemap based - 0.731s (h) Hilbert curve based - 0.659s (i) Gosper curve based - 0.659s

Fig. 8. *Space filling curve layouts versus existing layouts*. Existing methods (a-f) range from very slow to very fast, and produce layouts of various qualities, with the faster ones generally producing less aesthetic or less useful layouts, and devoting smaller regions of the screen to the majority of nodes, which obscures details such as the number of subclusters and their contents. The treemap layout (g) is both fast and effective at showing structures such as the three distinct clusters in the graph, but introduces problems such as poor aspect ratios. The space filling curve based layouts (h,i) solve this by guaranteeing good aspect ratios, while also clearly showing the distinct clusters and maintaining speed.