Lines: 272

Browsing large graphs with MSAGLJS, a graph draph drawing tool in JavaScript

Lev Nachmanson and Xiaoji Chen

Microsoft Research, US,
levnach@hotmail.com, cxiaoji@gmail.com,
Msagljs github home page: https://github.com/microsoft/msagljs

Abstract. There has been progress in visualization of large graphs recently. Tools appeared that can render a huge graph in seconds. However, if we request that the node labels are rendered, and the edges are routed around the nodes, then the problem is still standing. Interacting with a large graph in an Internet browser with the same ease as browsing an online map, inspecting the high level structure and zooming in to the high level detail, is still a challenging task.

In this paper we describe novel approaches to several aspects of this problem.

We give a new efficient algorithm for edge routing, where the edges are routed around the nodes. The algorithm produces edge paths which are visually appealing and optimal in their homotopy class.

To facilitate graph visualization with DeckGL, or any other viewer supporting tiles, we propose a new simple and fast tiling method. The method guarantees that in every view, except of the highest layer, the number of visible nodes per tile is not larger than a predefined bound.

Our method provides a high level overview of the graph, with the gradual increase of the detail level. We make the node labels of the most important nodes for the current view visible.

The edge routing algorithm mentioned above is reused at the tiling stage to simplify the paths on the lower levels. In addition, we bundle edges per-tile as an optimization heuristic

Introduction

We target our approach to large but not huge graphs. The maximum number of vertices of the graphs we looked at was 28k, and the maximum number of the edges was 237k. There are quite a few algorithms that calculate node positions for such graphs, and work very fast [1, 2]. We look at the node layout as a solved problem.

In the first part of the paper we address edge routing where an edge does not intersects the nodes it is not adjacent to. Our approach works for any node layout, as long as it produces a layout whithout overlap. We build on the edge routing from [3] and improve it. There has been progress in visualization of large graphs recently. Tools appeared that can render a huge graph in seconds. However, the situatiton changes if we request that the node labels are rendered, and the edges overlap only the nodes they are adjacent to. Interacting with a large graph in an Internet browser with the same ease as browsing an online map, inspecting the high level structure and zooming in to the high level detail, is still an unsolved problem. In this paper we describe novel approaches to several aspects of this problem.

We propose a novel and efficient algorithm for edge routing, where each edge can only intersect its source or target. The algorithm produces edge paths which are visually appealing and even optimal in their homotopy class.

To facilitate graph visualization with DeckGL, we propose a new simple and fast tiling method. The method guarantees that in every view, except of the views of the Shighest layer, the number of visible entities is not larger than a predefined bound. The method can be used in other viewers that support tiling.

Our method provides a high level overview of the graph.

The edge routing algorithm mentioned above is reused at the tiling stage to simplify the paths on the lower levels. In addition, we bundle edges per-tile as an optimization heuristic.

Related work

A popular graph drawing tool Graphviz [4] applies Scalable Force-Directed Placement [5] for large graphs, with no support for tiling. Its edge routing for this case builds the whole visibility graph. This can be very slow because the visibility graph can have $O(n^2)$ edges, where n is the number of the nodes in the graph. Interestingly, the funnel algorithm [6, 7], the last step of our approach, is used in Graphviz for the edge routing in the Sugiyama layout. We are not aware of any tool that integrates Graphviz and uses tiling as well.

yWorks [8] has method "Organic edge routing" that produces edge routes around the nodes. We could find only a very general description of the method: "The algorithm is based on a force directed layout paradigm. Nodes act as repulsive forces on edges in order to guarantee a certain minimal distance between nodes and edges. Edges tend to contract themselves. Using simulated annealing, this finally leads to edge layouts that are calculated for each edge separately". It seems the algorithm runs in $O(n+m)log(n+m)$ time, where n is the number of the nodes and m is the number of the edges.

ReGraph [9] uses WebGL as the viewing platform. It can render a large graph using straight lines for the edges. The tool does not support tiling, but instead the user interactively opens the node that is a cluster of nodes.

"graph-tool.skewed" [10] does not implement its own layout algorithms or edge routing algorithms, but instead provides a nice wrapper around the algorithms from other layout tools.

Circos [11] visualizes large graphs in a circular layout. It does not support tiles.

Cosmograph [12] uses a GPU to calculate the layout of a graph and can handle a graph with a million nodes. It renders edges as straight lines. It does not support tiling.

Edge routing in MSAGLJS

The edge routing starts, as in [3], by building a spanner graph, an approximation of the full visibility graph, and then finding shortest paths on the spanner. The spanner, see Fig 2, is built on a variation of a Yao graph, which was introduced independently by Flinchbaugh and Jones [13] and Yao [14]. This kind of graph is defined by the set of cones with the apices at the vertices. The cones have the same angle, usually in the form of $\frac{2\pi}{n}$, where n is a natural number. The family of cones with the apex at a specific vertex partition the plane as illistrated in Fig. 1. For each cone at most one edge is created connecting the cone apex with a vertex inside of the cone, so the graph has $O(n)$ edges where n is the number of vertices.

The approach of [3] applies local optimizations to shorten an edge path. Namely, it tries to shortcut one vertex at a time from the path, as illustrated in Fig 3. To smoothen a path, it fits Bezier segments into the polyline corners by using a binary search to find a larger fitting segment, see Fig 4. While anylyzing performance of the edge routing in MSAGLJS, we noticed, that for a graph with more than 1k nodes these heuristics sometime create a performance bottleneck, in spite of using R-Trees[15].
In addition, when the naive shortcutting of polyline corners fails, the resulting path might remain not visually appealing, as shown in Fig. 3.

We replace these heuristics with a more precize and efficient optimization described below.

Path optimization

We finalize edge routes by the "funnel" algorithm [6, 7], routing a path inside a simple polygon, that is a polygon without holes.

An application of the 'path in a simple polygon' optimization to edge routing is not a new idea: the novelty of our work is in how we find the polygon and how we use it. The authors of Graphvis used the 'funnel' algorithm [16], but only for hierarchical layouts, where a simple polygon, $\mathcal{P}$, containing the path is available. They write: "If $\mathcal{P}$ does not contain holes ... we can apply a standard "funnel" algorithm ... for finding Euclidean shortest paths in a simple polygon". In general case, for a non-layered layout, they build the visibility graph which is very expensive for a large graph.

Here we find the polygon $\mathcal{P}$ for any layout. We drop the requirement that $\mathcal{P}$ is simple. Indeed, to run the "funnel" algorithm one only needs a "sleeve": a sequence of triangles leading from the start to the end of the path, where each

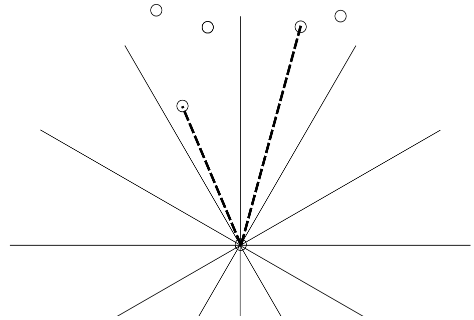

Fig. 1. Yao graph

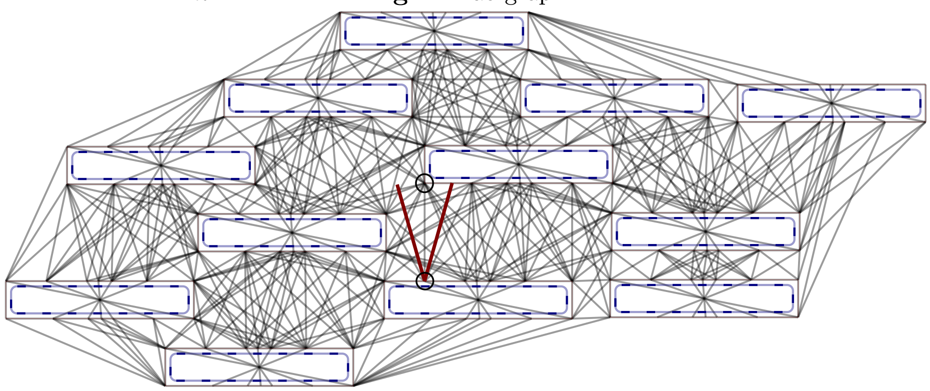

Fig. 2. Spanner graph is built using the idea of Yao graphs. The dashed curves are the original node boundaries. Each original curve is surrounded by a polygon with some offset to allow the polyline paths smoothing without intersecting the former.

The edge marked by the circles is created because the top vertex is inside of the cone and it is the closest among such vertices to the cone apex. The apex of the cone is the lower vertex of the edge.

MSAGLJS uses cone angle $\frac{\pi}{6}$, so the edges of the spanner can deviate from the optimal direction by this angle. Therefore, the shortest paths on the spanner have length that is at most the optimal shortest length multiplied by $\frac{1}{\cos(\frac{\pi}{6})} \simeq 1.155$.

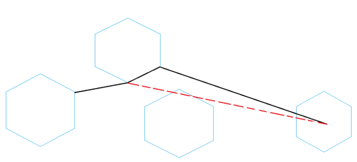

Fig. 3. Unsuccessful shortcut

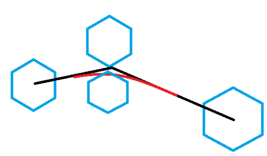

Fig. 4. Fitting a Bezier segment into a polyline corner

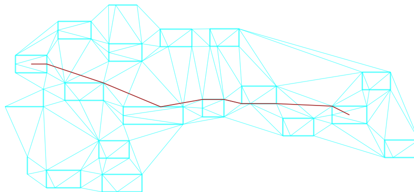

Fig. 5. Path $\mathcal{L}$ with $\mathcal{T}$, a fragment.

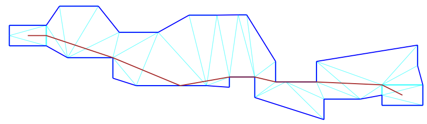

Fig. 6. Polygon $\mathcal{P}$ containing $\mathcal{L}$.

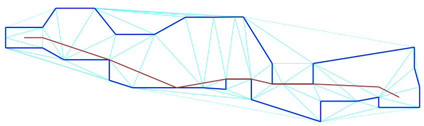

Fig. 7. New triangulation of $\mathcal{P}$.

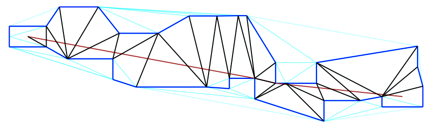

Fig. 8. The optimized path together with the sleeve diagonals.

triangle shares a side with its successor. Let us show how to build polygon $\mathcal{P}$, create a sleeve, and produce an optimized path.

We call obstacles, $\mathcal{O}$, the set of polygons covering the original nodes, see Fig. 2. Before routing edges, we calculate a Constrained Delaunay Triangulation [17] on $\mathcal{O}$. Let us call this triangulation $\mathcal{T}$.

For each edge of the graph we proceed with the following steps.

We route a path, called $\mathcal{L}$, on the spanner, as illistrated by Fig. 5. Let $\mathcal{S}$ and $\mathcal{E}$ be the obstacles containing correspondengly $\mathcal{L}$'s start and end point. To obtain $\mathcal{P}$, let us consider $\mathcal{U}$, the set of all triangles $t \in \mathcal{T}$ such that either $t \subset \mathcal{S} \cup \mathcal{E}$, or t intersects $\mathcal{L}$ and is not inside of any obstacle in $\mathcal{O} \setminus \{S, E\}$. The union of $\mathcal{U}$ gives us $\mathcal{P}$. The boundary of $\mathcal{P}$ comprizes all sides e of the triangles from $\mathcal{U}$ such that e belongs to exactly one triangle from $\mathcal{U}$, see Fig. 6.

To create the sleeve [6, 7], we need to have a triangulation of $\mathcal{P}$ such that every edge of the triangulation is either a boundary edge of $\mathcal{P}$, or a diagonal of $\mathcal{P}$. Because $\mathcal{U}$ might not have this property, as in Fig. 6, we create a new Constrained Delaunay Triangulation of $\mathcal{P}$, where the set of constrained edges is the boundary of $\mathcal{P}$, see Fig. 7.

We trace path $\mathcal{L}$ through the new triangulation and obtain the sleeve. Finally, we apply the funnel algorithm on the sleeve and obtain the path which is the shortest in the homotopy class of $\mathcal{L}$, as illustrated in Fig. 8.

The discussion [18] of the algorithm helped us in the implementation.

Polygon $\mathcal{P}$ is not necessarily simple, as shown in Fig. 9. In this example the path that we calculate with the funnel algorithm is not the shortest path inside of $\mathcal{P}$.

Performance and quality comparison

In Fig. 10 we compare the paths generated by the old and the new method. We can see that the paths produced by the new method have no kinks. We also

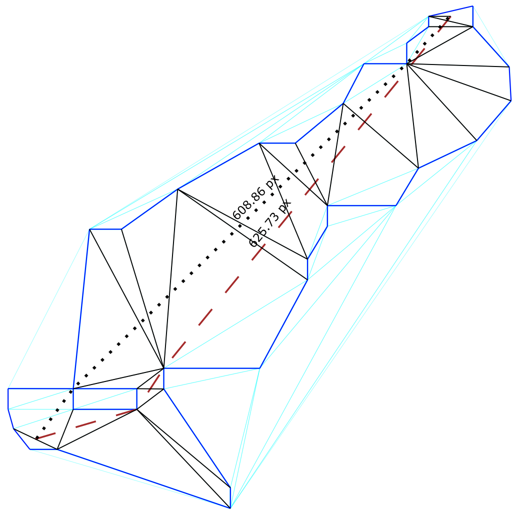

Fig. 9. $\mathcal{P}$ is not simple. The dotted path is shorter than the dashed one that was found by the routing.

know that these paths are the shorterst in their 'channels'. Arguably, the new method produces better paths.

Our performance experiments are summarized in Table. 1. We see that the older approach outperforms the new one on the smaller graphs; those with the number of nodes under 2000. The new method is faster on the rest of the graphs. We still prefer to use the new method independently of the graph size since the slowdown is insignificant, under half of a second in our experiments, but the quality of the paths is better. On the larger graphs the new method runs faster and produces better paths, so it is an obvious choice.

graph	nodes	edges	old method's time	new time
social network [19]	407	2639	1.0	1.4
b103 [20]	944	2438	1.6	2.0
b100 [21]	1463	5806	5.6	5.785
composers [22]	3405	13832	510.5	17.5
p2p-Gnutella04 [23]	10876	39994	375.4	293.8
facebook_combined [24]	4039	88234	132.2	119.1
lastfm_asia_edges [25]	7626	27807	43.3	41.4
deezer_europe_edges [25]	28283	92753	1596.9	1209.3
ca-HepPh [26]	12008	237010	521.2	495.0

Table 1. Performance comparison with time in seconds.

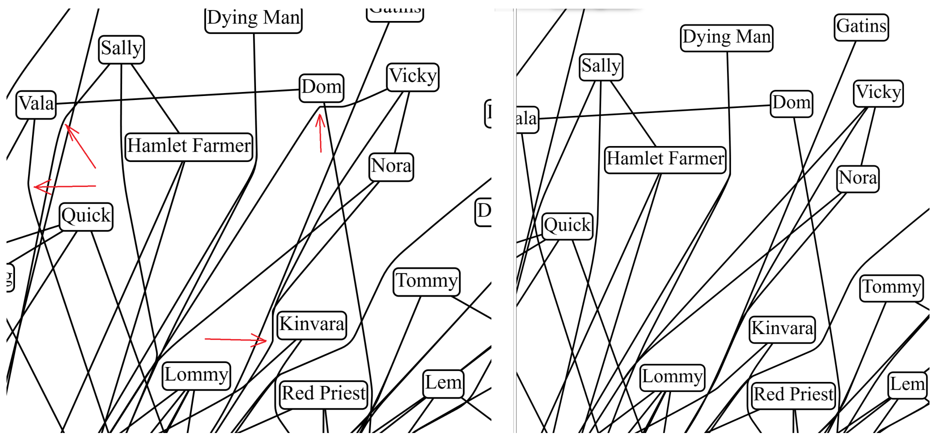

Fig. 10. The difference in the paths between the old, on the left, and the new, on the right, paths. The arrows on the left fragment point to the kinks that were removed by the new method.

1 Tiling

The algorithm works in three phases. The first phase builds the levels starting from the lowest level and proceeding to higher and more detailed levels, with smaller tiles, until no more tile subdivision is required. The second phase filters out the entities from the layers to satisfy the capacity quota. Finally, the third phase simplifies the edge routes to utilize the space freed by the filtered out entities.

A tile, in our settings, is a pair (*rect, tiledata*), where *rect* is the rectangle of the tile and *tiledata* is a set of *tile elements* visible in *rect*. A *tile element* could be a node, an edge label, an edge arrowhead, or an *edge clip*. An edge clip is a pair (e, p), where e is an edge and p is a continuous piece of the edge curve c_e. Sometimes we need several edge clips to trace an edge through a tile.

The initial tile, the only tile on level 0, is represented by pair $(0, 0)$. For $z = 1$, there are four tiles: $(0, 0), (0, 1), (1, 0)$, and $(1, 1)$. Each tile (i, j) can be subdivided into four subtiles for level $z + 1$: $(2i, 2j), (2i, 2j + 1), (2i + 1, 2j)$, and $(2i + 1, 2j + 1)$.

Each z-level is represented by a map L_z, so $L_z(i, j)$ gives us a specific tile. Empty tiles correspond to undefined $L_z(i, j)$.

We use edge clips to represent the edge intersections with the tiles and provide the renderer with the minimal geometry that is sufficient to render a tile. To achieve this we require property $\mathcal{F}$:

a) For each tile t, for each curve clip $(e, p) \in t.\mathit{tiledata}$, we have: $p \subset t.\mathit{rect}$ and p might cross the boundary of the $t.\mathit{rect}$ only at endpoints of p.

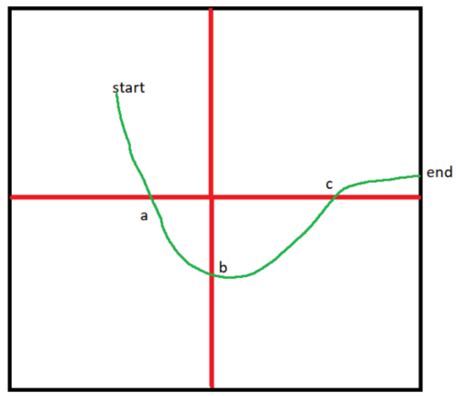

Fig. 11. Intersect curve [start,end] with the midlines. Sort the intersections parameters together with start, and end into array $u = [start, a, b, c, end]$. Split the curve to sub-curves [start,a], [a,b],[b,c],[c,end].

b) For each edge e we have : the union of all p for all $(e, p) \in t.tiledata$ is equal to $c_e \cap t.rect$.

First phase of tiling

The first phase starts with $L_0 = \{(0, 0) \to (rect, tiledata)\}$: and *tiledata* comprising curve clips (e, c_e), for all edges e of the graph, all graph nodes, all edge labels, and all edge arrowheads. We ensure property $\mathcal{F}$ by setting *rect* to a padded bounding box of the graph, so each edge curve does not intersect the boundary of *rect*.

Let us assume that L_z is already constructed and $\mathcal{F}$ holds for its tiles. To build level L_{z+1} we divide each tile $t = L_z(i, j)$ into four subtiles of equal size. For each node, arrowhead, or edge label of $t.tiledata$, if the bounding box of the element intersects the subtile's rectangle then we add the element to the subtile *tiledata*.

The edge clip treatment is more involved. Let (e, p) be a curve clip belonging to tile t. We find all intersections of curve p with the horizontal midline and the vertical midline of $t.rect$. Each intersection can be represented as $p[t_j]$. We sort sequence $u = [start, \ldots, t_j, \ldots, end]$, where $[start, end]$ is the parameter domain of p, in ascending order, and remove the duplicates.

Next we create curve clips $(e, l_k) = (e, trim(p, u_k, u_{k+1}))$, as shown in Fig 11. We assign each curve clip (e, l_k) to the subtile with the rectangle containing the bounding box of l_k.

Because, by the induction assumption property $\mathcal{F}$ is true on L_z, and by construction, each new curve clip can cross the boundary of the subtile only at the clip endpoints. We also cover all the intersections of p with the subtiles with the new edge clips, so the property $\mathcal{F}$ holds for L_{z+1}.

Two parameters control the algorithm: tile capacity, $\mathcal{C}$, and the minimal size of a tile: ($\mathcal{W}$, $\mathcal{H}$). If for each (i, j) the number of elements in $L_z(i, j).tiledata$ is not greater than $\mathcal{C}$, or, if $w \leq \mathcal{W}$ and $h \leq \mathcal{H}$, where w (h) is the current tile width (correspondengly, height), then the second phase starts.

In our setting $\mathcal{C} = 500$, and $(\mathcal{W}, \mathcal{H}) = 3(w, h)$, where w is the average width and h is the average height of the nodes of the graph.

Edge bundling In our settings each edge clip is uniquely defined, module direction, by its start and end point. We can use this property to bundle the edges. In each tile we keep a map from unordered pairs of points to the set of edge clips that have these points as start and end points. Each such pair defines an edge bundle. For all edge clips in a bundle we create only one curve segment, avoiding the expensive trimming. We alse count a bundle as one element in the tile, as in most of the cases the drawing attributes of the edges in the bundle are the same.

In our experiments, the number of edge bundles is about 50% of the number of edge clips, so the edge bundling is a significant optimization.

Second phase of tiling

The second phase of tiling filters out the entities from the lower layers. We do not change the highest, the most detailed layer. We sort the nodes of the graph into array N by PageRank [27]. For each layer L, except of the highest, we proceed as follows.

```
1: procedure FILTER(L)
2:     r ← removeEntities(L)
3:     for all n in N do
4:         if !addNodeToLayer(n, r, N) then break
5:         end if
6:     end for
7: end procedure
```

Here removeEntities(L) empties all the tiles of layer L, but returns map r allowing to restore the tiles. Function $addNodeToLayer(n)$ returns false and does not change L when one of the tiles intersecting n already has more elements than $\mathcal{C}$. Otherwise, the function adds n to all tiles intersected by n. It also adds the tile elements for self edges of n, and the tile elements for the edges connecting n with the nodes appearing in N before n, i.e. the nodes with the rank not lesser than the rank of n.

This procedure guarantees that each tile of L has no more than $\mathcal{C}$ nodes, but a tile can have more than $\mathcal{C}$ elements in general.

Third phase of tiling

In the third phase we use a fact that some nodes are not present on the layer. For all layers except of the highest we reroute the edges but only around the nodes that are present it the layer. We do not calculate edge routes from scratch, but use the existing routes and only apply the "funnel" heuristic in larger channels. This gives us simpler edge routes but still has a visual stability during the layer change while browsing.

2 Future work

- Find a tiling method that guarantees that each tile has no more than $\mathcal{C}$ elements. One approach could be to use a more aggressive edge bundling to reduce the number of edge clips in the tiles.

References

1. Y. Hu and L. Shi, "Visualizing large graphs," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.
2. U. Brandes and C. Pich, "Eigensolver methods for progressive multidimensional scaling of large data," in *Graph Drawing: 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers 14*, pp. 42–53, Springer, 2007.
3. T. Dwyer and L. Nachmanson, "Fast edge-routing for large graphs," in *Graph Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September 22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.
4. "Graphviz." http://www.graphviz.org/.
5. "sfdp." https://graphviz.org/docs/layouts/sfdp/.
6. B. Chazelle, "A theorem on polygon cutting with applications," in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE, 1982.
7. J. Hershberger and J. Snoeyink, "Computing minimum length paths of a given homotopy class," *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
8. "yworks." https://yworks.com/products/yed.
9. "Regraph." https://cambridge-intelligence.com/regraph/.
10. "Skewed." https://graph-tool.skewed.de.
11. "Circos." http://circos.ca/.
12. "Cosmograph." https://cosmograph.app.
13. B. Flinchbaugh and L. Jones, "Strong connectivity in directional nearest-neighbor graphs," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463, 1981.
14. A. C.-C. Yao, "On constructing minimum spanning trees in k-dimensional spaces and related problems," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736, 1982.
15. A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57, 1984.

16. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, “Implementing a general-purpose edge router,” in *Graph Drawing: 5th International Symposium, GD’97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer, 1997.
17. B. Delaunay, “Sur la sphere vide, bull. acad. science ussr vii: Class,” *Sci. Mat. Nat*, pp. 793–800, 1934.
18. “Funnel algorithm.” https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf.
19. A. Beveridge and M. Chemers, “The game of game of thrones: Networked concordances and fractal dramaturgy,” in *Reading Contemporary Serial Television Universes*, pp. 201–225, Routledge, 2018.
20. “b103.” https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot.
21. “b100.” https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot.
22. “Skewed.” http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml.
23. “p2p-gnutella04.” https://snap.stanford.edu/data/p2p-Gnutella04.html.
24. “facebookcombined.” https://snap.stanford.edu/data/facebook_combined.txt.gz.
25. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM ’20)*, p. 1325–1334, ACM, 2020.
26. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, pp. 2–es, 2007.
27. L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web,” *Stanford InfoLab*, vol. 249, no. 373, pp. 1–17, 1999.