

# 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>

1     **Abstract.** There has been progress in visualization of large graphs re-  
2     cently. Tools appeared that can render a huge graph in seconds. However,  
3     if we request that the node labels are rendered, and the edges are routed  
4     around the nodes, then the problem is still standing. Interacting with a  
5     large graph in an Internet browser with the same ease as browsing an  
6     online map, inspecting the high level structure and zooming in to the  
7     high level detail, is still a challenging task.  
8     In this paper we describe novel approaches to several aspects of this  
9     problem.  
10    We give a new efficient algorithm for edge routing, where the edges are  
11    routed around the nodes. The algorithm produces edge paths which are  
12    visually appealing and optimal in their homotopy class.  
13    To facilitate graph visualization with DeckGL, or any other viewer sup-  
14    porting tiles, we propose a new simple and fast tiling method. The  
15    method guarantees that in every view, except of the highest layer, the  
16    number of visible nodes per tile is not larger than a predefined bound.  
17    Our method provides a high level overview of the graph, with the grad-  
18    ual increase of the detail level. We make the node labels of the most  
19    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

## 20 Introduction

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

26    In the first part of the paper we address edge routing where an edge does  
27    not intersects the nodes it is not adjacent to. Our approach works for any node  
28    layout, as long as it produces a layout whithout overlap. We build on the edge  
29    routing from [3] and improve it. There has been progress in visualization of

30 large graphs recently. Tools appeared that can render a huge graph in seconds.  
 31 However, the situation changes if we request that the node labels are rendered,  
 32 and the edges overlap only the nodes they are adjacent to. Interacting with a  
 33 large graph in an Internet browser with the same ease as browsing an online  
 34 map, inspecting the high level structure and zooming in to the high level detail,  
 35 is still an unsolved problem. In this paper we describe novel approaches to several  
 36 aspects of this problem.

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

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

44 Our method provides a high level overview of the graph.

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

## 48 Related work

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

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

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

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

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

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

## 75 Edge routing in MSAGLJS

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

96  
 100 The approach of [3] applies local optimizations to shorten an edge path.  
 101 Namely, it tries to shortcut one vertex at a time from the path, as illustrated in  
 102 Fig 3. To smoothen a path, it fits Bezier segments into the polyline corners by  
 103 using a binary search to find a larger fitting segment, see Fig 4. While analyzing  
 104 performance of the edge routing in MSAGLJS, we noticed, that for a graph with  
 105 more than 1k nodes these heuristics sometime create a performance bottleneck,  
 106 in spite of using R-Trees[15].

107 In addition, when the naive shortcutting of polyline corners fails, the resulting  
 108 path might remain not visually appealing, as shown in Fig. 3.

109 We replace these heuristics with a more precise and efficient optimization  
 110 described below.

## 111 Path optimization

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

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

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



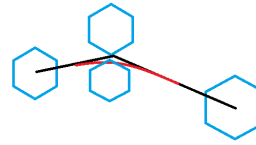
**Fig. 1.** Yao graph



**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  $\frac{1}{\cos(\frac{\pi}{6})} \simeq 1.155$ .



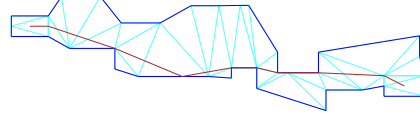
**Fig. 3.** Unsuccessful shortcut



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



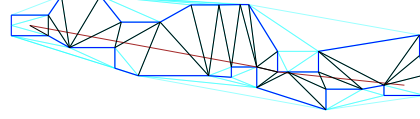
131 **Fig. 5.** Path  $\mathcal{L}$  with  $\mathcal{T}$ , a fragment.



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



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



134 **Fig. 8.** The optimized path together  
135 with the sleeve diagonals.

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

127 We call obstacles,  $\mathcal{O}$ , the set of polygons covering the original nodes, see  
128 Fig. 2. Before routing edges, we calculate a Constrained Delaunay Triangulation  
129 [17] on  $\mathcal{O}$ . Let us call this triangulation  $\mathcal{T}$ .

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

138 We route a path, called  $\mathcal{L}$ , on the spanner, as illustrated by Fig. 5. Let  $\mathcal{S}$  and  
139  $\mathcal{E}$  be the obstacles containing correspondingly  $\mathcal{L}$ 's start and end point. To obtain  
140  $\mathcal{P}$ , 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  $t$   
141 intersects  $\mathcal{L}$  and is not inside of any obstacle in  $\mathcal{O} \setminus \{\mathcal{S}, \mathcal{E}\}$ . The union of  $\mathcal{U}$  gives  
142 us  $\mathcal{P}$ . The boundary of  $\mathcal{P}$  comprizes all sides  $e$  of the triangles from  $\mathcal{U}$  such that  
143  $e$  belongs to exactly one triangle from  $\mathcal{U}$ , see Fig. 6.

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

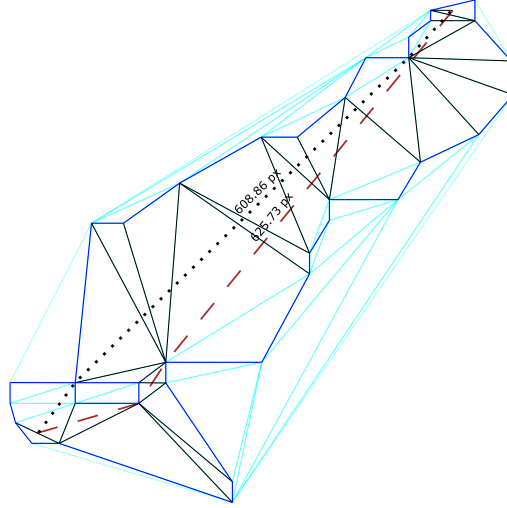
149 We trace path  $\mathcal{L}$  through the new triangulation and obtain the sleeve. Finally,  
150 we apply the funnel algorithm on the sleeve and obtain the path which is the  
151 shortest in the homotopy class of  $\mathcal{L}$ , as illustrated in Fig. 8.

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

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

## 156 Performance and quality comparison

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



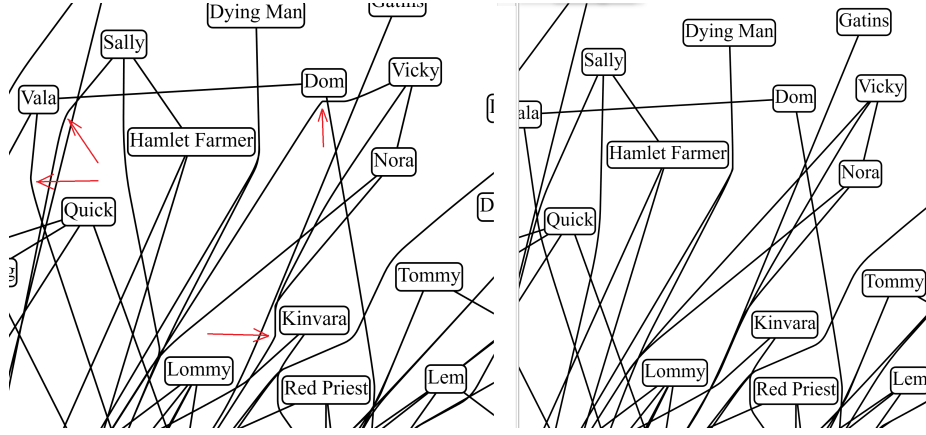
**Fig. 9.**  $\mathcal{P}$  is not simple. The dotted path is shorter than the dashed one that was found by the routing.

know that these paths are the shortest 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.



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

## 181 1 Tiling

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

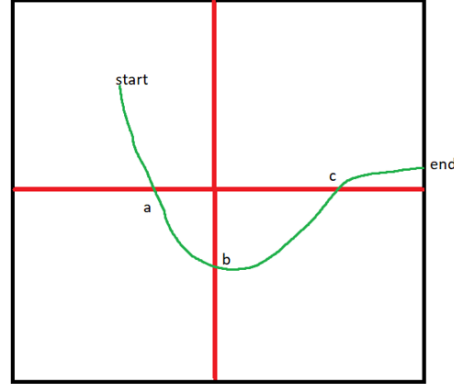
188 A tile, in our settings, is a pair  $(rect, tiledata)$ , where  $rect$  is the rectangle of  
 189 the tile and  $tiledata$  is a set of *tile elements* visible in  $rect$ . A *tile element* could  
 190 be a node, an edge label, an edge arrowhead, or an *edge clip*. An edge clip is a  
 191 pair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

193 The initial tile, the only tile on level 0, is represented by pair  $(0, 0)$ . For  
 194  $z = 1$ , there are four tiles:  $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ , and  $(1, 1)$ . Each tile  $(i, j)$  can be  
 195 subdivided into four subtiles for level  $z + 1$ :  $(2i, 2j)$ ,  $(2i, 2j + 1)$ ,  $(2i + 1, 2j)$ , and  
 196  $(2i + 1, 2j + 1)$ .

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

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

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



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

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

## 206 First phase of tiling

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

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

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

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

228 Because, by the induction assumption property  $\mathcal{F}$  is true on  $L_z$ , and by  
 229 construction, each new curve clip can cross the boundary of the subtile only at  
 230 the clip endpoints. We also cover all the intersections of  $p$  with the subtiles with  
 231 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 (correspondingly, 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 also 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:    $r \leftarrow removeEntities(L)$ 
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. The function 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  $n$ .

## References

1. Y. Hu and L. Shi, "Visualizing large graphs," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.

- 263 2. U. Brandes and C. Pich, “Eigensolver methods for progressive multidimensional  
264 scaling of large data,” in *Graph Drawing: 14th International Symposium, GD*  
265 *2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers 14*, pp. 42–  
266 53, Springer, 2007.
- 267 3. T. Dwyer and L. Nachmanson, “Fast edge-routing for large graphs,” in *Graph*  
268 *Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September*  
269 *22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.
- 270 4. “Graphviz.” <http://www.graphviz.org/>.
- 271 5. “sfdp.” <https://graphviz.org/docs/layouts/sfdp/>.
- 272 6. B. Chazelle, “A theorem on polygon cutting with applications,” in *23rd Annual*  
273 *Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE,  
274 1982.
- 275 7. J. Hershberger and J. Snoeyink, “Computing minimum length paths of a given  
276 homotopy class,” *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
- 277 8. “yworks.” <https://yworks.com/products/yed>.
- 278 9. “Regraph.” <https://cambridge-intelligence.com/regraph/>.
- 279 10. “Skewed.” <https://graph-tool.skewed.de>.
- 280 11. “Circos.” <http://circos.ca/>.
- 281 12. “Cosmograph.” <https://cosmograph.app>.
- 282 13. B. Flinchbaugh and L. Jones, “Strong connectivity in directional nearest-neighbor  
283 graphs,” *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463,  
284 1981.
- 285 14. A. C.-C. Yao, “On constructing minimum spanning trees in k-dimensional spaces  
286 and related problems,” *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736,  
287 1982.
- 288 15. A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Pro-*  
289 *ceedings of the 1984 ACM SIGMOD international conference on Management of*  
290 *data*, pp. 47–57, 1984.
- 291 16. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, “Implementing a  
292 general-purpose edge router,” in *Graph Drawing: 5th International Symposium,*  
293 *GD’97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer,  
294 1997.
- 295 17. B. Delaunay, “Sur la sphere vide, bull. acad. science ussr vii: Class,” *Sci. Mat. Nat*,  
296 pp. 793–800, 1934.
- 297 18. “Funnel algorithm.” <https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf>.
- 298 19. A. Beveridge and M. Chemers, “The game of game of thrones: Networked con-  
299 cordances and fractal dramaturgy,” in *Reading Contemporary Serial Television*  
300 *Universes*, pp. 201–225, Routledge, 2018.
- 301 20. “b103.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot)  
302 [layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot).
- 303 21. “b100.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot)  
304 [layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot).
- 305 22. “Skewed.” <http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml>.
- 306 23. “p2p-gnutella04.” <https://snap.stanford.edu/data/p2p-Gnutella04.html>.
- 307 24. “facebookcombined.” [https://snap.stanford.edu/data/facebook\\_combined.txt.gz](https://snap.stanford.edu/data/facebook_combined.txt.gz).
- 308 25. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a  
309 Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the*  
310 *29th ACM International Conference on Information and Knowledge Management*  
311 *(CIKM ’20)*, p. 1325–1334, ACM, 2020.

- 312 26. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification  
313 and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data*  
314 (*TKDD*), vol. 1, no. 1, pp. 2–es, 2007.
- 315 27. L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking:  
316 Bringing order to the web,” *Stanford InfoLab*, vol. 249, no. 373, pp. 1–17, 1999.