

Browsing large graphs with XJS, a graph drawing tool in JavaScript

auth0 and auth1

Institute, US,
 auth0@hotmail.com, auth1@gmail.com,
 X github home page: <https://github.com/X>

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 visible, and the edges are routed
 4 around the nodes, then the problem remains difficult. Interacting with
 5 a large graph in an Internet browser with the same ease as browsing
 6 an online map is still a challenging task. In this paper we describe a
 7 few novel approaches to large graph visualization that we developed in
 8 open-source JavaScript software.
 9 We give a new efficient edge routing algorithm, where the edges are
 10 routed around the nodes. The algorithm produces edge paths which are
 11 visually appealing and shortest in their homotopy class.
 12 To facilitate graph visualization with WebGL, or any other platform
 13 supporting tiles, we propose a new simple and efficient tiling method.
 14 The method guarantees that in every view, except of the highest level,
 15 the number of visible entities per tile is not larger than a predefined
 16 bound.
 17 We make the node labels of the most important nodes of the current
 18 view visible.
 The edge routing algorithm mentioned above is reused at the tiling stage
 to simplify the paths on the lower levels.

19 Introduction

20 Our software is open source, it is represented by a set of NPM packages. It
 21 runs on the client desktop or on a phone, and renders the graph in an Internet
 22 browser. We target large but not huge graphs. The maximum number of vertices
 23 of the graphs we applied our tool at was 28k, and the maximum number of the
 24 edges was 237k.

25 The algorithms described below were discovered while we programmed our
 26 tool. We believe these algorithms can be useful to other developers as well. The
 27 findings seemed to us interesting enough to put them into a paper.

28 The paper has sections Introduction, Related Work, Edge routing in XJS,
 29 Tiling, and Future work.

30 Let us start with a short review of some relevant to us publications.

31 Related work

32 A popular graph drawing tool Graphviz [1] applies method Scalable Force-
33 Directed Placement [2] for large graphs, with no support for tiling. Its edge
34 routing for this method builds the whole visibility graph and routes edges on it.
35 This can be very slow because the visibility graph can have $O(n^2)$ edges, where
36 n is the number of the nodes in the graph. Interestingly, the funnel algorithm [3,
37 4], the last step of our approach, is used in Graphviz for the edge routing in the
38 Sugiyama layout. We are not aware of any tool that integrates Graphviz and
39 uses tiling as well.

40 yWorks [5] has method "Organic edge routing" that produces edge routes
41 around the nodes. We could find only a very general description of the method:
42 "The algorithm is based on a force directed layout paradigm. Nodes act as re-
43 pulsive forces on edges in order to guarantee a certain minimal distance between
44 nodes and edges. Edges tend to contract themselves. Using simulated annealing,
45 this finally leads to edge layouts that are calculated for each edge separately".
46 It seems the algorithm runs in $O(n + m)\log(n + m)$ time, where n is the number
47 of the nodes and m is the number of the edges.

48 ReGraph [6] uses WebGL as the viewing platform. It can render a large graph
49 using straight lines for the edges. The tool does not support tiling, but instead
50 the user interactively opens the node that is a cluster of nodes.

51 "graph-tool.skewed" [7] does not implement its own layout algorithms or edge
52 routing algorithms, but instead provides a nice wrapper around the algorithms
53 from other layout tools.

54 Circos [8] visualizes large graphs in a circular layout. It does not support
55 tiles.

56 Cosmograph [9] uses a GPU to calculate the layout of a graph and can
57 handle a graph with a million nodes. It renders edges as straight lines. It does
58 not support tiling.

59 The authors of [10] implemented GraphMaps, a tool for large graph visu-
60 alization. The tool only runs on Windows. The edge were routed as polylines
61 on a triangulation and were not optimized. The tool supported tiling, but the
62 problem of the limiting number of visible entities was not solved.

63 In [11] an approach to visualize a huge graph is described. The method uses
64 tiles and edge bundling following [12], which is applied at the last moment during
65 the graph browsing. The latter calculation is done on the client side. The rest
66 and the majority of the calculations runs on several servers.

67 Edge routing in XJS

68 We believe that short and smooth edges, that are not obstructed by the nodes,
69 are easier to follow than longer edges with kinks. We believe that such edges
70 help in understanding the graph structure. In addition, we were looking for a
71 fast algorithm. This was our motivation to come up with the routing described
72 here.

83 The edge routing starts, as in [13], by building a spanner graph, an approxi-
84 mation of the full visibility graph, and then finding the edge routes as shortest
85 paths on the spanner. The spanner, see Fig 2, is built on a variation of a Yao
86 graph, which was introduced independently by Flinchbaugh, and Jones [14], and
87 Yao [15]. This graph is built with a help of a set of cones with the apices at
88 the vertices. Each cone of the set has the same angle, usually in the form of $\frac{2\pi}{k}$,
89 where k is a natural number, $k = 12$ in our settings. The family of cones with
90 the apex at a specific vertex partition the plane, as illustrated in Fig. 1. For each
91 cone at most one edge is created connecting the cone apex with a vertex inside
92 the cone. This way the spanner has at most kn edges, where n is the number of
93 the vertices. We cover each node by a polygon with a relatively small number of
94 corners, at most 8. Polygon corners play role of the vertices of the spanner. As a
95 result, the spanner has $O(N)$ edges, where N is the number of the graph nodes.

100 The approach of [13] 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 large fitting segment, see Fig 4.
104 We noticed that when the shortcutting of polyline corners fails, the resulting
105 path might remain not visually appealing, as shown in Fig. 3.

106 We replace the shortcutting with a more precise, but still efficient optimiza-
107 tion described below.

108 Path optimization

109 We finalize edge routes by the “funnel” algorithm [3, 4], routing a path inside a
110 simple polygon, that is a polygon without holes.

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

119 Here we find the polygon \mathcal{P} for any layout. We drop the requirement that
120 \mathcal{P} is simple. Indeed, to run the “funnel” algorithm one only needs a “sleeve”: a
121 sequence of triangles leading from the start to the end of the path, where each
122 triangle shares a side with its successor. Let us show how to build polygon \mathcal{P} ,
123 create a sleeve, and produce an optimized path.

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

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

135 We route a path, called \mathcal{L} , on the spanner, as illustrated by Fig. 5. Let \mathcal{S} and
136 \mathcal{E} be the obstacles containing correspondingly \mathcal{L} ’s start and end point. To obtain



Fig. 1. Yao graph

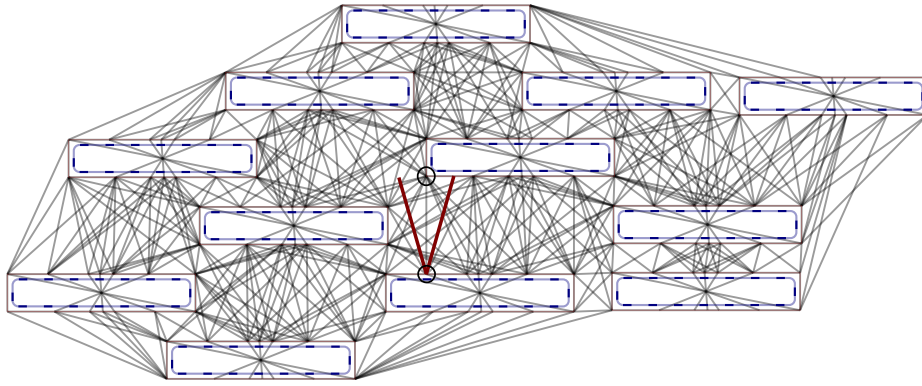


Fig. 2. Spanner graph is built using the idea of Yao graphs. The dashed curves are the original node boundaries. Each original curve is surrounded by a polygon with some offset to allow the polyline paths smoothing without intersecting the former. The edge marked by the circles is created because the top vertex is inside 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. XJS 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 $\frac{1}{\cos(\frac{\pi}{6})} \simeq 1.155$.

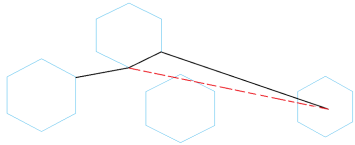


Fig. 3. Unsuccessful shortcut

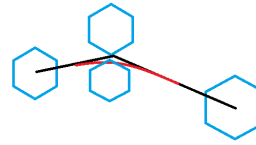
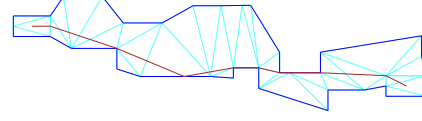


Fig. 4. Fitting a Bezier segment into a polyline corner



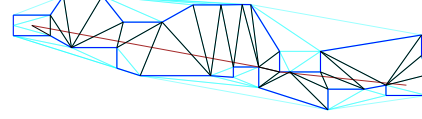
128 **Fig. 5.** Path \mathcal{L} with \mathcal{T} , a fragment.



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



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



131 **Fig. 8.** The optimized path together
132 with the sleeve diagonals.

137 \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
138 intersects \mathcal{L} and is not inside of any obstacle in $\mathcal{O} \setminus \{S, E\}$. The union of \mathcal{U} gives
139 us \mathcal{P} . The boundary of \mathcal{P} comprizes all sides e of the triangles from \mathcal{U} such that
140 e belongs to exactly one triangle from \mathcal{U} , see Fig. 6.

141 To create the sleeve [3, 4], we need to have a triangulation of \mathcal{P} such that every
142 edge of the triangulation is either a boundary edge of \mathcal{P} , or a diagonal of \mathcal{P} .
143 Because \mathcal{U} might not have this property, as in Fig. 6, we create a new Constrained
144 Delaunay Triangulation of \mathcal{P} , where the set of constrained edges is the boundary
145 of \mathcal{P} , see Fig. 7.

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

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

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

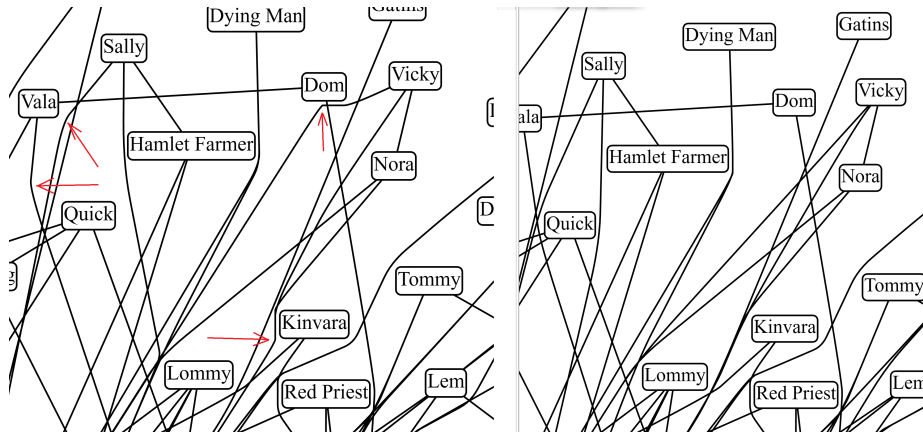
153 Performance and quality comparison

157 In Fig. 10 we compare the paths generated by the old and the new method. We
158 can see that the paths produced by the new method have no kinks. We also
159 know that these paths are the shortest in their 'channels'. Arguably, the new
160 method produces better paths.

161 Our performance experiments are summarized in Table. 1. We see that the
162 older approach outperforms the new one on the smaller graphs; those with the
163 number of nodes under 2000. The new method is faster on the rest of the graphs.
164 We still prefer to use the new method independently of the graph size since
165 the slowdown is insignificant, but the quality of the paths is better. On the
166 larger graphs the new method runs faster and produces better paths, so it is an
167 obvious choice. To load a large graph, for example, `deezer_europe_edges` [19], we



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



154 **Fig. 10.** Comparing the old, on the left, and the new, on the right, paths. The
 155 arrows on the left fragment point to the kinks that were removed by the new
 156 method.

168 start Edge or Chrome with an option that increases the memory limit of their
 169 process: `- max_old_space_size=8192`.

graph	nodes	edges	old method's time	new time
social network [20]	407	2639	1.0	1.4
b103 [21]	944	2438	1.6	2.0
b100 [22]	1463	5806	5.6	5.785
composers [23]	3405	13832	510.5	20.3
p2p-Gnutella04 [24]	10876	39994	375.4	304.2
facebook_combined [25]	4039	88234	132.2	123.7
lastfm_asia_edges [19]	7626	27807	43.3	54.7
deezer_europe_edges [19]	28283	92753	1596.9	1402.6
ca-HepPh [26]	12008	237010	521.2	495.0

Table 1. Performance comparison with time in seconds.

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.

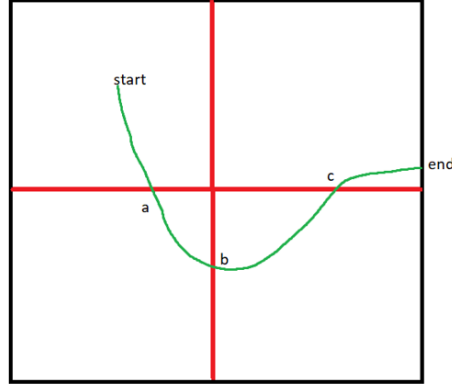
188 A tile, in our settings, is a pair $(rect, tiledata)$, where $rect$ is the rectangle of
 189 the tile and $tiledata$ is a set of *tile elements* visible in $rect$. A *tile element* could
 190 be a node, an edge label, an edge arrowhead, or an *edge clip*. An edge clip is a
 191 pair (e, p) , where 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 .
- 204 b) For each edge e we have : the union of all p for all $(e, p) \in t.tiledata$ is
 205 equal to $c_e \cap t.rect$.



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 $[start, a]$, $[a, b]$, $[b, c]$, $[c, end]$.

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} .

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

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

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

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

248 Second phase of tiling

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

```

1: procedure FILTER( $L$ )
2:    $r \leftarrow \text{removeEntities}(L)$ 
3:   for all  $n$  in  $N$  do
4:     if ! $\text{addNodeToLayer}(n, r, N)$  then break
5:   end if
6: end for
7: end procedure

```

252
 253 Here $\text{removeEntities}(L)$ empties all the tiles of layer L , but returns map r allow-
 254 ing to restore the tiles. Function $\text{addNodeToLayer}(n)$ returns false and does not
 255 change L when one of the tiles intersecting n already has more elements than
 256 \mathcal{C} . Otherwise, the function adds n to all tiles intersected by n . It also adds the
 257 tile elements for self edges of n , and the tile elements for the edges connecting n
 258 with the nodes appearing in N before n , i.e. the nodes with the rank not lesser
 259 than the rank of n .

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

262 Third phase of tiling

263 In the third phase we use a fact that some nodes are not present on the layer. For
 264 all layers, except of the highest, we reroute the edges but only around the nodes
 265 that are present in the layer. We do not calculate edge routes from scratch, but

266 use the existing routes and only apply the "funnel" heuristic in larger channels.
 267 This gives us simpler edge routes but still has a visual stability during the layer
 268 change while browsing.

269 2 Future work

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

273 References

- 274 1. "Graphviz." <http://www.graphviz.org/>.
- 275 2. "sfdp." <https://graphviz.org/docs/layouts/sfdp/>.
- 276 3. B. Chazelle, "A theorem on polygon cutting with applications," in *23rd Annual*
 277 *Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE,
 278 1982.
- 279 4. J. Hershberger and J. Snoeyink, "Computing minimum length paths of a given
 280 homotopy class," *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
- 281 5. "yworks." <https://yworks.com/products/yed>.
- 282 6. "Regraph." <https://cambridge-intelligence.com/regraph/>.
- 283 7. "Skewed." <https://graph-tool.skewed.de>.
- 284 8. "Circos." <http://circos.ca/>.
- 285 9. "Cosmograph." <https://cosmograph.app>.
- 286 10. L. Nachmanson, R. Prutkin, B. Lee, N. H. Riche, A. E. Holroyd, and X. Chen,
 287 "Graphmaps: Browsing large graphs as interactive maps," in *Graph Drawing and*
 288 *Network Visualization: 23rd International Symposium, GD 2015, Los Angeles, CA,*
 289 *USA, September 24-26, 2015, Revised Selected Papers 23*, pp. 3–15, Springer, 2015.
- 290 11. A. Perrot and D. Auber, "Cornac: Tackling huge graph visualization with big data
 291 infrastructure," *IEEE Transactions on Big Data*, vol. 6, no. 1, pp. 80–92, 2018.
- 292 12. C. Hurter, O. Ersoy, and A. Telea, "Graph bundling by kernel density estimation,"
 293 in *Computer graphics forum*, vol. 31, pp. 865–874, Wiley Online Library, 2012.
- 294 13. T. Dwyer and L. Nachmanson, "Fast edge-routing for large graphs," in *Graph*
 295 *Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September*
 296 *22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.
- 297 14. B. Flinchbaugh and L. Jones, "Strong connectivity in directional nearest-neighbor
 298 graphs," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463,
 299 1981.
- 300 15. A. C.-C. Yao, "On constructing minimum spanning trees in k-dimensional spaces
 301 and related problems," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736,
 302 1982.
- 303 16. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, "Implementing a
 304 general-purpose edge router," in *Graph Drawing: 5th International Symposium,*
 305 *GD'97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer,
 306 1997.
- 307 17. B. Delaunay, "Sur la sphere vide, bull. acad. science ussr vii: Class," *Sci. Mat. Nat*,
 308 pp. 793–800, 1934.

- 309 18. “Funnel algorithm.” <https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf>.
- 310 19. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a
311 Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the*
312 *29th ACM International Conference on Information and Knowledge Management*
313 *(CIKM '20)*, p. 1325–1334, ACM, 2020.
- 314 20. A. Beveridge and M. Chemers, “The game of game of thrones: Networked con-
315 cordances and fractal dramaturgy,” in *Reading Contemporary Serial Television*
316 *Universes*, pp. 201–225, Routledge, 2018.
- 317 21. “b103.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot)
318 [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).
- 319 22. “b100.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot)
320 [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).
- 321 23. “Skewed.” <http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml>.
- 322 24. “p2p-gnutella04.” <https://snap.stanford.edu/data/p2p-Gnutella04.html>.
- 323 25. “facebookcombined.” https://snap.stanford.edu/data/facebook_combined.txt.gz.
- 324 26. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification
325 and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data*
326 *(TKDD)*, vol. 1, no. 1, pp. 2–es, 2007.
- 327 27. L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking:
328 Bringing order to the web,” *Stanford InfoLab*, vol. 249, no. 373, pp. 1–17, 1999.