

# 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.** We routinely use online maps on a phone or computer, where  
 2     we usually browse a huge map. The online maps have efficient pan and  
 3     zoom operations, and at each view we are not overwhelmed with the  
 4     amount of information. One of the main technologies allowing it is tiling:  
 5     the tiles are carefully designed in advance, and split into levels. They are  
 6     loaded on demand from the Internet, or from the device.  
 7     In contrast, browsing a large graph in this style is still a challenge, al-  
 8     though some progress has been achieved. The main difficulty here is the  
 9     tiling calculation.  
 10    In this paper we describe a few novel approaches to tiling for a large  
 11    graph that we developed in XJS.  
 12    Our first contribution is a new edge routing algorithm, where the edges  
 13    are routed around the nodes. The algorithm produces edge paths which  
 14    are visually appealing and shortest in their homotopy class. This algo-  
 15    rithm is used to calculate the edge routes on the most detailed tile level,  
 16    and then it is reused to simplify routes for lower levels.  
 17    The second contribution is a tiling scheme with a new method of subdivi-  
 18    ding an edge into segments for the sub-tiles.  
 19    XJS is open-source JavaScript software. With XJS one can browse a  
 20    large, but not a huge, graph on a phone or computer, similar to browsing  
 21    an online map.

## 22   1   Introduction

23   Our open-source software, XJS, is written in TypeScript and is consumed as a  
 24   set of NPM packages. It runs on client computers or phones and renders graphs  
 25   in web browsers. We target large but not huge graphs.

26   The paper consists of two main sections. The first section describes our edge  
 27   routing algorithm that creates paths that are shortest in their homotopy class.  
 28   We measure the performance of our new edge routing algorithm on public bench-  
 29   mark sets and compare it with the previous state-of-the-art routing algorithm.

30   The second section describes our new tiling method. The method allows  
 31   loading graphs in a browser with up to 10K nodes and 40K edges and interacting  
 32   with them in the style of online maps.

## 33 2 Related work

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

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

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

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

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

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

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

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

## 69 Edge routing

70 The user study of Xu et al [13] shows that straight edges improve the user  
71 comprehension of a graph drawing. From the other hand, Holten et al [14] show  
72 that strongly curved edges do not perform well in this regard. In our routing, we  
73 try to keep the edges as straight as possible, and to curve them just enough to

74 avoid the nodes. Our motivation for developing the routing described here was  
 75 to improve the quality of edges and the algorithm speed.

86 The edge routing starts, as in [15], by building a spanner graph, an approxi-  
 87 mation of the full visibility graph, and then finding the edge routes as shortest  
 88 paths on the spanner. The spanner, see Fig 2, is built on a variation of a Yao  
 89 graph, which was introduced independently by Flinchbaugh, and Jones [16], and  
 90 Yao [17]. This graph is built with a help of a set of cones with the apices at  
 91 the vertices. Each cone of the set has the same angle, usually in the form of  $\frac{2\pi}{k}$ ,  
 92 where  $k$  is a natural number,  $k = 12$  in our settings. The family of cones with  
 93 the apex at a specific vertex partition the plane, as illustrated in Fig. 1. For each  
 94 cone at most one edge is created connecting the cone apex with a vertex inside  
 95 the cone. This way the spanner has at most  $kn$  edges, where  $n$  is the number of  
 96 the vertices. We cover each node by a polygon with a relatively small number of  
 97 corners, at most 8. Polygon corners play role of the vertices of the spanner. As a  
 98 result, the spanner has  $O(N)$  edges, where  $N$  is the number of the graph nodes.

103 The approach of [15] applies local optimizations to shorten an edge path.  
 104 Namely, it tries to shortcut one vertex at a time from the path, as illustrated in  
 105 Fig 3. To smoothen a path, it fits Bezier segments into the polyline corners by  
 106 using a binary search to find a large fitting segment, see Fig 4.  
 107 We noticed that when the shortcutting of polyline corners fails, the resulting  
 108 path might remain not visually appealing, as shown in Fig. 3. We replace the  
 109 shortcutting with a more precise, but still efficient optimization described below:  
 110 that is one of the main contributions of our work.

## 111 2.1 Path optimization

112 We finalize edge routes by a slight modification of the “funnel” algorithm [3, 4],  
 113 routing a path inside a 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 Graphviz used the ‘funnel’ algorithm [18], 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 they build the visibility graph which is very expensive for a large  
 121 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  
 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 Triangula-  
 129 tion [19] on  $\mathcal{O}$ , following [20]. Let us call this triangulation  $\mathcal{T}$ .

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



**Fig. 1.** Yao graph



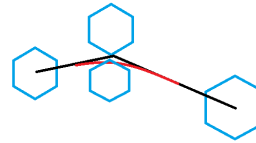
**Fig. 2.** Spanner graph is built using the idea of Yao graphs. The dashed curves are the original node boundaries. Each original curve is surrounded by a polygon with some offset to allow the polyline paths smoothing without intersecting the former.

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



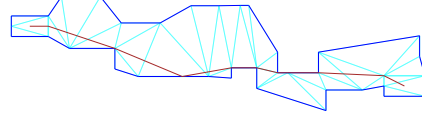
**Fig. 3.** Unsuccessful shortcut



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



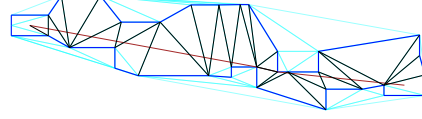
131 **Fig. 5.** Path  $\mathcal{L}$  with  $\mathcal{T}$ , a fragment.



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.

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

144 To create the sleeve [3, 4], 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 Triangulation of  $\mathcal{P}$ , where the set of constrained edges is the boundary of  $\mathcal{P}$ , see  
148 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 [21] of the algorithm helped us in the implementation of the  
153 funnel algorithm.

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

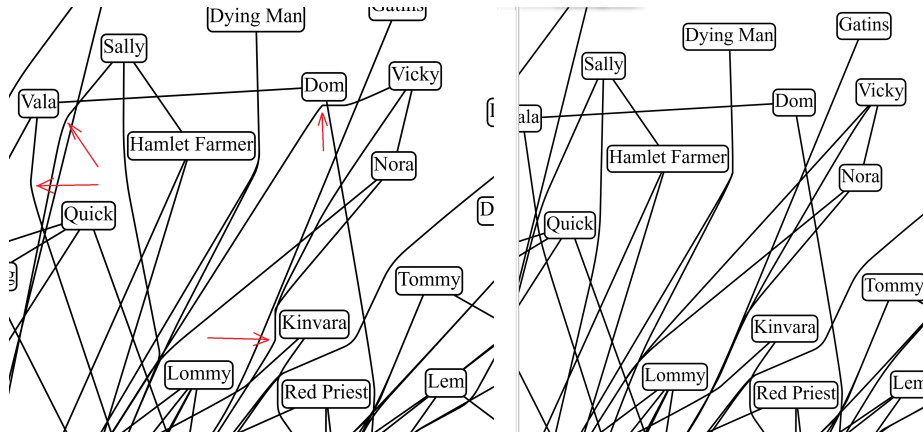
## 157 Performance and quality comparison

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

165 Our performance experiments are summarized in Table. 1. We see that the  
166 older approach outperforms the new one on the smaller graphs; those with the  
167 number of nodes under 2000. The new method is faster on the rest of the graphs.



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



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

168 We prefer the new method regardless of the graph size because it provides better  
 169 path quality and the slowdown is insignificant.

graph	nodes	edges	old method's time	new time
social network [22]	407	2639	1.0	1.4
b103 [23]	944	2438	1.6	2.0
b100 [24]	1463	5806	5.6	5.785
composers [25]	3405	13832	510.5	20.3
p2p-Gnutella04 [26]	10876	39994	375.4	304.2
facebook_combined [27]	4039	88234	132.2	123.7
lastfm_asia_edges [28]	7626	27807	43.3	54.7
deezer_europe_edges [28]	28283	92753	1596.9	1402.6
ca-HepPh [29]	12008	237010	521.2	495.0

180 **Table 1.** Performance comparison with time in seconds.

### 181 3 Tiling

182 We had two goals when working on tiling. The first goal was to make exploring  
 183 the graph in our tool similar to using online maps. The second goal was efficiency.  
 184 The algorithm works in three phases. The first phase builds the levels starting  
 185 from the lowest level and proceeding to higher and more detailed levels, with  
 186 smaller tiles, until no more tile subdivision is required. The second phase filters  
 187 out the entities from the layers to satisfy the capacity quota, as in [10]. Finally,  
 188 the third phase simplifies the edge routes to utilize the space freed by the filtered  
 189 out entities.

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

195 The initial tile, the only tile on level 0, is represented by pair  $(0, 0)$ . For  
 196  $z = 1$ , there are four tiles:  $(0, 0)$ ,  $(0, 1)$ ,  $(1, 0)$ , and  $(1, 1)$ . Each tile  $(i, j)$  can be  
 197 subdivided into four sub-tiles for level  $z + 1$ :  $(2i, 2j)$ ,  $(2i, 2j + 1)$ ,  $(2i + 1, 2j)$ , and  
 198  $(2i + 1, 2j + 1)$ .

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

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

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

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

### 208   **3.1   First phase of tiling**

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

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

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

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

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

235   Two parameters control the algorithm: tile capacity,  $\mathcal{C}$ , and the minimal size  
 236 of a tile:  $(\mathcal{W}, \mathcal{H})$ . If for each  $(i, j)$  the number of elements in  $L_z(i, j).tiledata$   
 237 is not greater than  $\mathcal{C}$ , and if  $w \leq \mathcal{W}$  and  $h \leq \mathcal{H}$ , where  $(w, h)$  is the current  
 238 tile size, then we try to build the next level  $L_{z+1}$ . Otherwise, the second phase  
 239 starts.

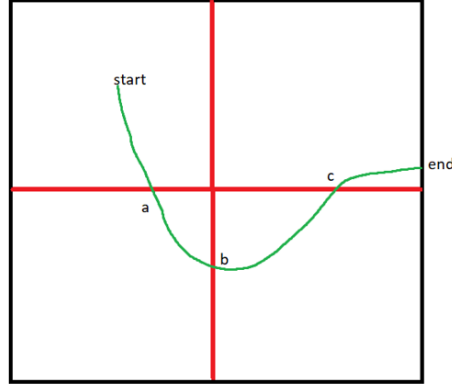
240   In our setting  $\mathcal{C} = 500$ , and  $(\mathcal{W}, \mathcal{H}) = 3(w, h)$ , where  $w$  is the average width  
 241 and  $h$  is the average height of the nodes of the graph.

242   For efficiency, we do not create a new curve in an edge clip but keep two  
 243 parameters indicating the clipped segment start and end. A possible optimization  
 244 here is to find the repeated segments in the edge curves that naturally appear  
 245 while routing through the same graph with the same algorithm, and reuse the  
 246 repeated segment to save memory and to avoid the same calculation in edge  
 247 clipping.

### 248   **3.2   Second phase of tiling**

249   In this phase we filter out some entities from the lower levels. We do not change  
 250 the highest, the most detailed level. We sort the nodes of the graph into array  $N$





227 **Fig. 11.** Intersect curve  $[start, end]$  with the midlines. Sort the intersections pa-  
 228 rameters, together with start and end, into array  $u = [start, a, b, c, end]$ . Split the  
 229 curve to sub-curves  $[start, a]$ ,  $[a, b]$ ,  $[b, c]$ , and  $[c, end]$ . Each sub-curve is confined  
 230 to a single sub-tile.

251 by PageRank [30]. For each level  $L$ , 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 ! $\text{addNodeToLevel}(n, r, N)$  then break
5:   end if
6: end for
7: end procedure

```

---

252  
 253 Here  $\text{removeEntities}(L)$  empties all the tiles of level  $L$ , and returns map  $r$  al-  
 254 lowing to restore the tiles. Map  $r$  maps each graph element to an array of tile  
 255 elements representing it in  $L$ . Function  $\text{addNodeToLevel}(n)$  tries to add node  $n$   
 256 to  $L$ , it also tries to add the tile elements for self edges of  $n$ , and the tile elements  
 257 for the edges connecting  $n$  with the nodes ranked at least as high as  $n$ . These  
 258 nodes are the nodes already added to  $L$ .

259 This procedure guarantees that each tile of  $L$  has no more than  $\mathcal{C}$  elements.

### 260 3.3 Third phase of tiling

261 In the third phase we use the fact that some nodes are not present on the  
 262 level. For all levels, except of the highest, we reroute the edges but only around  
 263 the nodes that are present in the level. We do not calculate edge routes from  
 264 scratch, but use the existing routes and only apply the "funnel" heuristic in

265 larger channels. This gives us simpler edge routes but still has the visual stability  
266 during the level change while browsing.

## 267 4 Conclusion

268 The first contribution is an efficient edge routing algorithm. The algorithm re-  
269 places each polyline path with the shortest path that does not intersect the  
270 nodes and stays in the polyline homotopy class, the channel, and then produces  
271 the smooth composite curve. It uses a modification of the 'funnel' in the simple  
272 polygon algorithm, where we drop the requirement that the polygon is simple.

273 We described the scenario where the graph spanner is used. Instead, our  
274 approach can start with any routing with polylines outside the nodes.

275 The algorithm is fast and creates visually pleasing results.

276 The tiling method is the second contribution of this study. It allows large  
277 graphs to be visualized in a web browser, similar to online maps. The method  
278 provides an overview of the most important nodes and edges of the graph by  
279 making them visible on the top levels while keeping the number of visible el-  
280 ements under a given limit. The novelty of the method is that it efficiently  
281 subdivides the edges by using the tiles, and simplifies the edges on the upper  
282 levels with the edge routing algorithm mentioned above.

## 283 5 Future work

- 284 – Find a tiling method that guarantees that each tile has no more than  $\mathcal{C}$  el-  
285 ements on every level. One approach is to use more aggressive, and regular  
286 edge bundling to reduce the number of edge clips in the tiles.
- 287 – Our tile calculation is memory intensive and takes a longer time for larger  
288 graphs. The largest graph from the Table 1 that we were able to load with  
289 Chrome, and Edge using the tiling procedure was p2p-Gnutella04 [26]. One  
290 of the reasons was the memory limit on a process in those browsers, another  
291 was the long running-time of the tiling procedure. A possible measure would  
292 be saving the tiles to the disk and loading them on demand.
- 293 – For the user convenience we would like to run the layout, routing, and tiling,  
294 in a worker thread to avoid blocking the main thread.
- 295 – Addressing node labels visibility is an important task. We would like to en-  
296 large the most important nodes of the view so that their labels are readable.

## 297 References

- 298 1. “Graphviz.” <http://www.graphviz.org/>.
- 299 2. “sfdp.” <https://graphviz.org/docs/layouts/sfdp/>.
- 300 3. B. Chazelle, “A theorem on polygon cutting with applications,” in *23rd Annual*  
301 *Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE,  
302 1982.

- 303 4. J. Hershberger and J. Snoeyink, "Computing minimum length paths of a given  
304 homotopy class," *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
- 305 5. "yworks." <https://yworks.com/products/yed>.
- 306 6. "Regraph." <https://cambridge-intelligence.com/regraph/>.
- 307 7. "Skewed." <https://graph-tool.skewed.de>.
- 308 8. "Circos." <http://circos.ca/>.
- 309 9. "Cosmograph." <https://cosmograph.app>.
- 310 10. L. Nachmanson, R. Prutkin, B. Lee, N. H. Riche, A. E. Holroyd, and X. Chen,  
311 "Graphmaps: Browsing large graphs as interactive maps," in *Graph Drawing and  
312 Network Visualization: 23rd International Symposium, GD 2015, Los Angeles, CA,  
313 USA, September 24-26, 2015, Revised Selected Papers 23*, pp. 3–15, Springer, 2015.
- 314 11. A. Perrot and D. Auber, "Cornac: Tackling huge graph visualization with big data  
315 infrastructure," *IEEE Transactions on Big Data*, vol. 6, no. 1, pp. 80–92, 2018.
- 316 12. C. Hurter, O. Ersoy, and A. Telea, "Graph bundling by kernel density estimation,"  
317 in *Computer graphics forum*, vol. 31, pp. 865–874, Wiley Online Library, 2012.
- 318 13. K. Xu, C. Rooney, P. Passmore, D.-H. Ham, and P. H. Nguyen, "A user study  
319 on curved edges in graph visualization," *IEEE transactions on visualization and  
320 computer graphics*, vol. 18, no. 12, pp. 2449–2456, 2012.
- 321 14. D. Holten and J. J. Van Wijk, "A user study on visualizing directed edges in  
322 graphs," in *Proceedings of the SIGCHI conference on human factors in computing  
323 systems*, pp. 2299–2308, 2009.
- 324 15. T. Dwyer and L. Nachmanson, "Fast edge-routing for large graphs," in *Graph  
325 Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September  
326 22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.
- 327 16. B. Flinchbaugh and L. Jones, "Strong connectivity in directional nearest-neighbor  
328 graphs," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463,  
329 1981.
- 330 17. A. C.-C. Yao, "On constructing minimum spanning trees in k-dimensional spaces  
331 and related problems," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736,  
332 1982.
- 333 18. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, "Implementing a  
334 general-purpose edge router," in *Graph Drawing: 5th International Symposium,  
335 GD'97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer,  
336 1997.
- 337 19. B. Delaunay *et al.*, "Sur la sphere vide," *Izv. Akad. Nauk SSSR, Otdelenie Matem-  
338 aticheskii i Estestvennyka Nauk*, vol. 7, no. 1, pp. 793–800, 1934.
- 339 20. V. Domiter and B. Žalik, "Sweep-line algorithm for constrained delaunay triangulation," *International Journal of Geographical Information Science*, vol. 22, no. 4,  
340 pp. 449–462, 2008.
- 341 21. "Funnel algorithm." <https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf>.
- 342 22. A. Beveridge and M. Chemers, "The game of game of thrones: Networked con-  
343 cordances and fractal dramaturgy," in *Reading Contemporary Serial Television  
344 Universes*, pp. 201–225, Routledge, 2018.
- 345 23. "b103." [https://github.com/microsoft/automatic-graph-  
346 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).
- 347 24. "b100." [https://github.com/microsoft/automatic-graph-  
348 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).
- 349 25. "Skewed." <http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml>.
- 350 26. "p2p-gnutella04." <https://snap.stanford.edu/data/p2p-Gnutella04.html>.
- 351 27. "facebookcombined." [https://snap.stanford.edu/data/facebook\\_combined.txt.gz](https://snap.stanford.edu/data/facebook_combined.txt.gz).
- 352

- 353 28. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a  
354 Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the*  
355 *29th ACM International Conference on Information and Knowledge Management*  
356 *(CIKM '20)*, p. 1325–1334, ACM, 2020.
- 357 29. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification  
358 and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data*  
359 *(TKDD)*, vol. 1, no. 1, pp. 2–es, 2007.
- 360 30. L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking:  
361 Bringing order to the web,” *Stanford InfoLab*, vol. 249, no. 373, pp. 1–17, 1999.