

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>

Abstract. We routinely use online maps on a phone or computer, where we usually browse a huge map. The online maps have efficient pan and zoom operations, and at each view we are not overwhelmed with the amount of information. One of the main technologies allowing it is tiling: the tiles are carefully designed in advance, and split into levels. They are loaded on demand from the server, or from the device.

In contrast, browsing a large graph in this style is still a challenge, although some progress has been achieved. The main difficulty here is the tiling calculation.

In this paper we describe a few novel approaches to tiling for a large graph that we developed in XJS.

Our first contribution is a new edge routing algorithm, where the edges are routed around the nodes. The algorithm produces edge paths which are visually appealing and shortest in their homotopy class. This algorithm is used to calculate the edge routes on the most detailed tile level, and then it is reused to create simplified routes for lower levels.

The second contribution is a tiling scheme with a new method of subdividing an edge into segments for the sub-tiles.

XJS is open-source JavaScript software. With XJS one can browse a large, but not a huge, graph on a phone or computer, similar to browsing an online map.

1 Introduction

Our software is open source written in TypeScript, it is consumed as a set of NPM packages. It runs on the client desktop or on a phone, and renders the graph in a web browser. We target large but not huge graphs. The maximum number of vertices of the graphs we applied our tool to was 28k, and the maximum number of the edges was 237k.

The methods and algorithms described below are implemented in XJS.

The rest of the paper is divided into sections, including Related Work, Edge routing, Tiling, Conclusion, and Future work.

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

2 Related work

A popular graph drawing tool Graphviz [1] applies method Scalable Force-Directed Placement [2] for large graphs, with no support for tiling. The edge routing for this method builds the whole visibility graph and routes edges on it. 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 [3, 4], 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 [5] 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 [6] 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" [7] 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 [8] visualizes large graphs in a circular layout. It does not support tiles.

Cosmograph [9] 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.

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

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

Edge routing

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

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

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

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

110 2.1 Path optimization

111 We finalize edge routes by a slight modification of the “funnel” algorithm [3, 4],
 112 routing a path inside a simple polygon, that is a polygon without holes.

113 An application of the ‘path in a simple polygon’ optimization to edge routing
 114 is not a new idea: the novelty of our work is in how we find the polygon and
 115 how we use it. The authors of Graphviz used the ‘funnel’ algorithm [18], but
 116 only for hierarchical layouts, where a simple polygon, \mathcal{P} , containing the path is
 117 available. They write: “If \mathcal{P} does not contain holes ... we can apply a standard
 118 “funnel” algorithm ... for finding Euclidean shortest paths in a simple polygon”.
 119 In general case they build the visibility graph which is very expensive for a large
 120 graph.

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

126 We call obstacles, \mathcal{O} , the set of polygons covering the original nodes, see
 127 Fig. 2. Before routing edges, we calculate a Constrained Delaunay Triangula-
 128 tion [19] on \mathcal{O} , following [20]. Let us call this triangulation \mathcal{T} .

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



Fig. 1. Yao graph

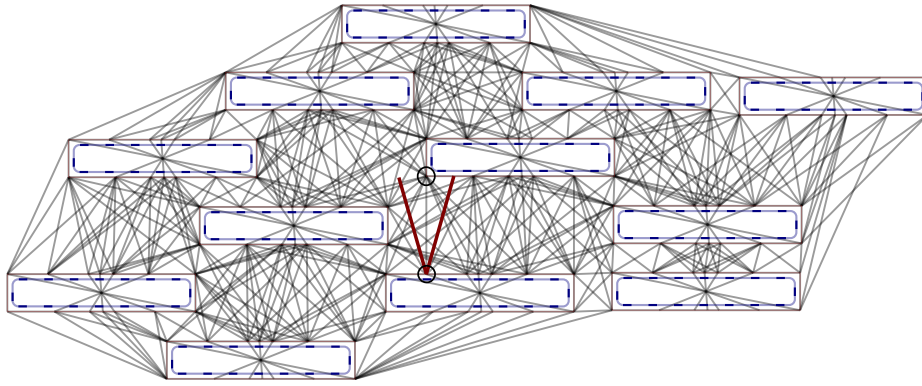


Fig. 2. Spanner graph is built using the idea of Yao graphs. The dashed curves are the original node boundaries. Each original curve is surrounded by a polygon with some offset to allow the polyline paths smoothing without intersecting the former. The edge marked by the circles is created because the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

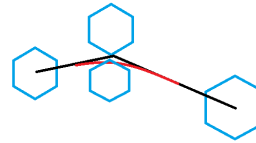
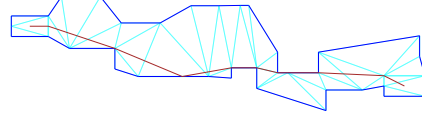


Fig. 4. Fitting a Bezier segment into a polyline corner



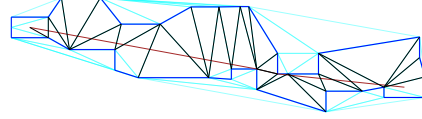
130 **Fig. 5.** Path \mathcal{L} with \mathcal{T} , a fragment.



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



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



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

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

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

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

151 The discussion [21] of the algorithm helped us in the implementation of the
152 funnel algorithm.

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 \mathcal{P} .

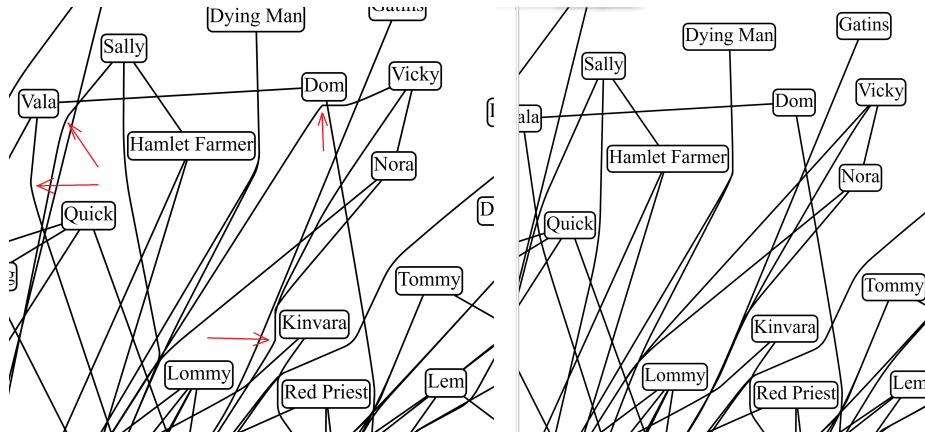
156 Performance and quality comparison

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

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



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



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

167 We prefer the new method regardless of the graph size because it provides better
 168 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

Table 1. Performance comparison with time in seconds.

180 3 Tiling

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

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

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

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

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

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

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

207 3.1 First phase of tiling

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

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

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

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

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

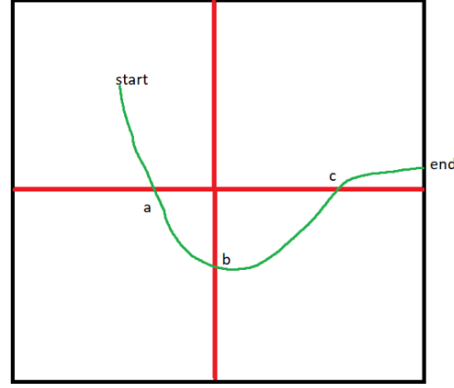
234 Two parameters control the algorithm: tile capacity, \mathcal{C} , and the minimal size
 235 of a tile: $(\mathcal{W}, \mathcal{H})$. If for each (i, j) the number of elements in $L_z(i, j).tiledata$
 236 is not greater than \mathcal{C} , and if $w \leq \mathcal{W}$ and $h \leq \mathcal{H}$, where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

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

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

247 3.2 Second phase of tiling

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



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

250 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

```

251
 252 Here $\text{removeEntities}(L)$ empties all the tiles of level L , and returns map r al-
 253 lowing to restore the tiles. Map r maps each graph element to an array of tile
 254 elements representing it in L . Function $\text{addNodeToLevel}(n)$ tries to add node n
 255 to L , it also tries to add the tile elements for self edges of n , and the tile elements
 256 for the edges connecting n with the nodes ranked at least as high as n . These
 257 nodes are the nodes already added to L .

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

259 3.3 Third phase of tiling

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

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

266 4 Conclusion

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

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

274 The algorithm is fast and creates visually pleasing results.

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

282 5 Future work

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

296 References

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

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

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