

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. There has been progress in visualization of large graphs recently. Tools appeared that can render a huge graph in seconds. However, if we request that the node labels are readable, and the edges are routed around the nodes, then the problem remains difficult. Interacting with a large graph in an Internet browser with the same ease as browsing an online map is still a challenging task. In this paper we describe a few novel approaches to large graph visualization that we developed in open-source JavaScript software.

We give a new efficient 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.

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

The edge routing algorithm mentioned above is reused at the tiling stage to simplify the paths on the lower levels.

Introduction

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

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

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

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

29 Related work

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

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

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

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

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

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

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

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

65 Edge routing in XJS

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

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

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

95
 99 The approach of [15] applies local optimizations to shorten an edge path.
 100 Namely, it tries to shortcut one vertex at a time from the path, as illustrated in
 101 Fig 3. To smoothen a path, it fits Bezier segments into the polyline corners by
 102 using a binary search to find a large fitting segment, see Fig 4.
 103 We noticed that when the shortcutting of polyline corners fails, the resulting
 104 path might remain not visually appealing, as shown in Fig. 3. We replace the
 105 shortcutting with a more precise, but still efficient optimization described below:
 106 that is one of the main contributions of our work.

107 Path optimization

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

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

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

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

126 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

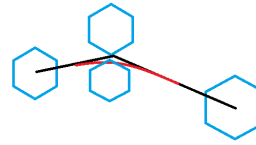
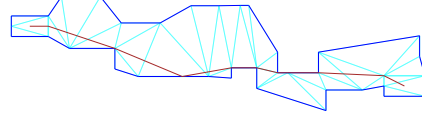


Fig. 4. Fitting a Bezier segment into a polyline corner



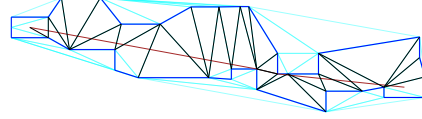
127 **Fig. 5.** Path \mathcal{L} with \mathcal{T} , a fragment.



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



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



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

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

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

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

148 The discussion [21] of the algorithm helped us in the implementation of the
149 funnel algorithm.

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} .

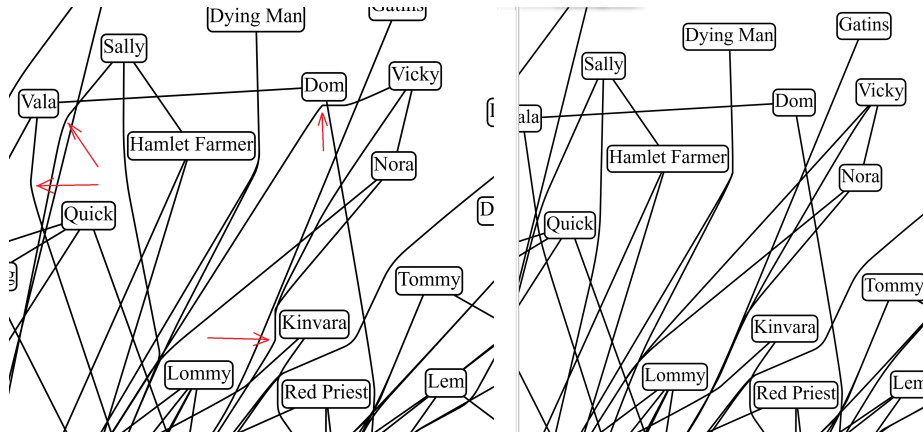
153 Performance and quality comparison

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 know
159 that these paths are the shortest in their 'channels'. Arguably, the new method
160 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.



132 **Fig. 9.** \mathcal{P} is not simple. The dotted path is shorter than the dashed one that
 133 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.

164 We prefer the new method regardless of the graph size because it provides better
 165 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.

177 1 Tiling

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

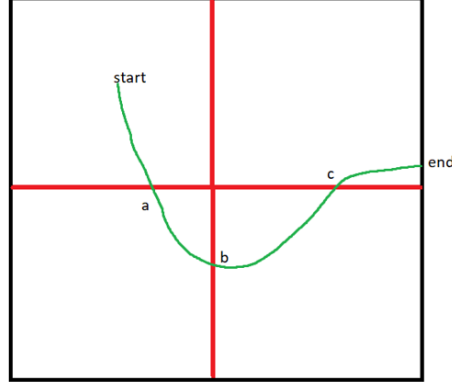
186 A tile, in our settings, is a pair $(rect, tiledata)$, where $rect$ is the rectangle of
 187 the tile and $tiledata$ is a set of *tile elements* visible in $rect$. A *tile element* could
 188 be a node, an edge label, an edge arrowhead, or an *edge clip*. An edge clip is a
 189 pair (e, p) , where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

191 The initial tile, the only tile on level 0, is represented by pair $(0, 0)$. For
 192 $z = 1$, there are four tiles: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. Each tile (i, j) can be
 193 subdivided into four sub-tiles for level $z + 1$: $(2i, 2j)$, $(2i, 2j + 1)$, $(2i + 1, 2j)$, and
 194 $(2i + 1, 2j + 1)$.

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

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

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



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

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

204 First phase of tiling

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

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

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

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

226 Because, by the induction assumption property \mathcal{F} is true on L_z , and by
 227 construction, each new edge clip can cross the boundary of the sub-tile only at
 228 the clip endpoints. We also cover all the intersections of p with the sub-tiles with
 229 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} , and if $w \leq \mathcal{W}$ and $h \leq \mathcal{H}$, where (w, h) is the current tile size, then we try to build the next level L_{z+1} . Otherwise, 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.

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

Second phase of tiling

In this phase, some entities from the lower levels are filtered out. We do not change the highest, the most detailed level. We sort the nodes of the graph into array N by PageRank [30]. For each level 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 ! $addNodeToLevel(n, r, N)$  then break
5:   end if
6: end for
7: end procedure

```

Here $removeEntities(L)$ empties all the tiles of level L , and returns map r allowing to restore the tiles. Map r maps each graph element to an array of tile elements representing it in L . Function $addNodeToLevel(n)$ tries to add node n to L , it also tries to add the tile elements for self edges of n , and the tile elements for the edges connecting n with the nodes ranked at least as high as n . These nodes are the nodes already added to L .

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

Third phase of tiling

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

262 2 Conclusion

263 The first contribution is an efficient edge routing algorithm. In XJS the algorithm
264 starts with routing on the spanner, but instead any routing with polylines outside
265 the nodes can be used as the first step for our method. The algorithm replaces
266 each polyline with the shortest polyline that does not intersect the nodes and
267 stays in the polyline homotopy class, the channel, and then produces the smooth
268 composite curve. The algorithm is fast and produces visually pleasing results.

269 The second contribution is a tiling method that enables large graphs to be
270 visualized in a web browser in a way that is similar to online maps. The method
271 provides an overview of the most important nodes and edges of the graph by
272 making them visible on the top levels while keeping the number of visible el-
273 ements under a given limit. The novelty of the method is that it subdivides
274 a smooth curve using tiles, and incorporates the edge routing algorithm, the
275 'funnel' in channel heuristic, to simplify the edges on the upper levels.

276 3 Future work

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

290 References

- 291 1. “Graphviz.” <http://www.graphviz.org/>.
- 292 2. “sfdp.” <https://graphviz.org/docs/layouts/sfdp/>.
- 293 3. B. Chazelle, “A theorem on polygon cutting with applications,” in *23rd Annual*
294 *Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE,
295 1982.
- 296 4. J. Hershberger and J. Snoeyink, “Computing minimum length paths of a given
297 homotopy class,” *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
- 298 5. “yworks.” <https://yworks.com/products/yed>.
- 299 6. “Regraph.” <https://cambridge-intelligence.com/regraph/>.
- 300 7. “Skewed.” <https://graph-tool.skewed.de>.
- 301 8. “Circos.” <http://circos.ca/>.

- 302 9. “Cosmograph.” <https://cosmograph.app>.
- 303 10. L. Nachmanson, R. Prutkin, B. Lee, N. H. Riche, A. E. Holroyd, and X. Chen,
304 “Graphmaps: Browsing large graphs as interactive maps,” in *Graph Drawing and*
305 *Network Visualization: 23rd International Symposium, GD 2015, Los Angeles, CA,*
306 *USA, September 24-26, 2015, Revised Selected Papers 23*, pp. 3–15, Springer, 2015.
- 307 11. A. Perrot and D. Auber, “Cornac: Tackling huge graph visualization with big data
308 infrastructure,” *IEEE Transactions on Big Data*, vol. 6, no. 1, pp. 80–92, 2018.
- 309 12. C. Hurter, O. Ersoy, and A. Telea, “Graph bundling by kernel density estimation,”
310 in *Computer graphics forum*, vol. 31, pp. 865–874, Wiley Online Library, 2012.
- 311 13. K. Xu, C. Rooney, P. Passmore, D.-H. Ham, and P. H. Nguyen, “A user study
312 on curved edges in graph visualization,” *IEEE transactions on visualization and*
313 *computer graphics*, vol. 18, no. 12, pp. 2449–2456, 2012.
- 314 14. D. Holten and J. J. Van Wijk, “A user study on visualizing directed edges in
315 graphs,” in *Proceedings of the SIGCHI conference on human factors in computing*
316 *systems*, pp. 2299–2308, 2009.
- 317 15. T. Dwyer and L. Nachmanson, “Fast edge-routing for large graphs,” in *Graph*
318 *Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September*
319 *22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.
- 320 16. B. Flinchbaugh and L. Jones, “Strong connectivity in directional nearest-neighbor
321 graphs,” *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463,
322 1981.
- 323 17. A. C.-C. Yao, “On constructing minimum spanning trees in k-dimensional spaces
324 and related problems,” *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736,
325 1982.
- 326 18. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, “Implementing a
327 general-purpose edge router,” in *Graph Drawing: 5th International Symposium,*
328 *GD’97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer,
329 1997.
- 330 19. B. Delaunay *et al.*, “Sur la sphere vide,” *Izv. Akad. Nauk SSSR, Otdelenie Matem-*
331 *aticheskii i Estestvennyka Nauk*, vol. 7, no. 1, pp. 793–800, 1934.
- 332 20. V. Domiter and B. Žalik, “Sweep-line algorithm for constrained delaunay triangulation,”
333 *International Journal of Geographical Information Science*, vol. 22, no. 4,
334 pp. 449–462, 2008.
- 335 21. “Funnel algorithm.” <https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf>.
- 336 22. A. Beveridge and M. Chemers, “The game of game of thrones: Networked concordances and fractal dramaturgy,” in *Reading Contemporary Serial Television*
337 *Universes*, pp. 201–225, Routledge, 2018.
- 338 23. “b103.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot)
339 [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).
- 340 24. “b100.” [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot)
341 [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).
- 342 25. “Skewed.” <http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml>.
- 343 26. “p2p-gnutella04.” <https://snap.stanford.edu/data/p2p-Gnutella04.html>.
- 344 27. “facebookcombined.” <https://snap.stanford.edu/data/facebook.combined.txt.gz>.
- 345 28. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a
346 Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the*
347 *29th ACM International Conference on Information and Knowledge Management*
348 *(CIKM ’20)*, p. 1325–1334, ACM, 2020.
- 349 29. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification
350 and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data*
351 *(TKDD)*, vol. 1, no. 1, pp. 2–es, 2007.
- 352

- 353 30. L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking:
354 Bringing order to the web," *Stanford InfoLab*, vol. 249, no. 373, pp. 1–17, 1999.