

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

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

94
95
96 The approach of [13] applies local optimizations to shorten an edge path.
97 Namely, it tries to shortcut one vertex at a time from the path, as illustrated in
98 Fig 3. To smoothen a path, it fits Bezier segments into the polyline corners by
99 using a binary search to find a large fitting segment, see Fig 4.
100 We noticed that when the shortcutting of polyline corners fails, the resulting
101 path might remain not visually appealing, as shown in Fig. 3.

102 We replace the shortcutting with a more precise, but still efficient optimiza-
103 tion described below.
104
105

106 Path optimization

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

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

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

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

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

126 We route a path, called \mathcal{L} , on the spanner, as illustrated by Fig. 5. Let \mathcal{S} and
127 \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

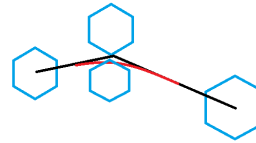
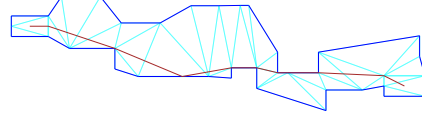


Fig. 4. Fitting a Bezier segment into a polyline corner



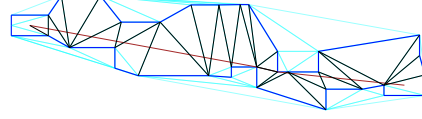
126 **Fig. 5.** Path \mathcal{L} with \mathcal{T} , a fragment.



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



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



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

135 \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
136 intersects \mathcal{L} and is not inside of any obstacle in $\mathcal{O} \setminus \{S, E\}$. The union of \mathcal{U} gives
137 us \mathcal{P} . The boundary of \mathcal{P} comprizes all sides e of the triangles from \mathcal{U} such that
138 e belongs to exactly one triangle from \mathcal{U} , see Fig. 6.

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

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

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

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

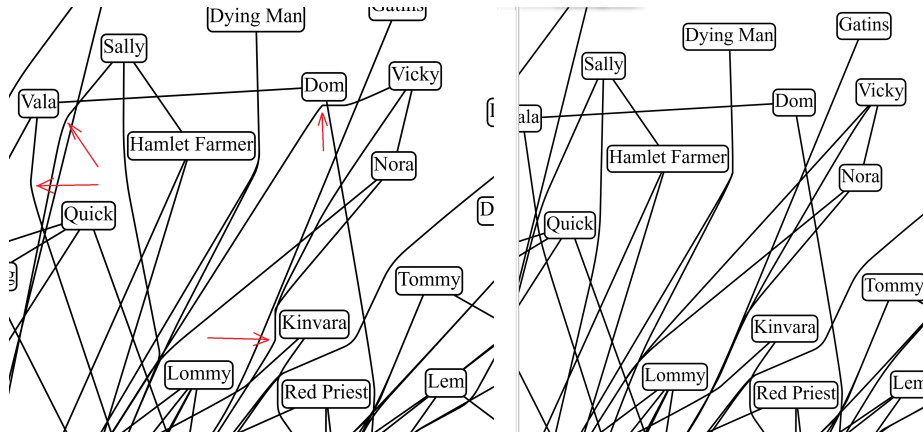
151 Performance and quality comparison

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

159 Our performance experiments are summarized in Table. 1. We see that the
160 older approach outperforms the new one on the smaller graphs; those with the
161 number of nodes under 2000. The new method is faster on the rest of the graphs.
162 We prefer the new method regardless of the graph size because it provides better
163 path quality and the slowdown is insignificant.



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



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

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

Table 1. Performance comparison with time in seconds.

1 Tiling

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

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

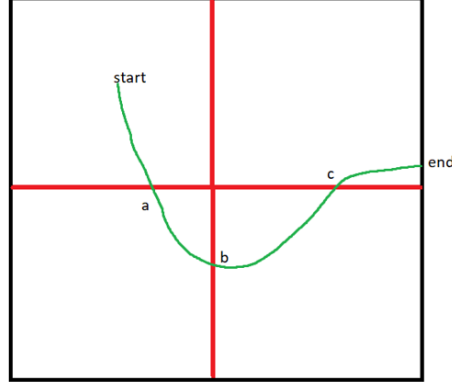
The initial tile, the only tile on level 0, is represented by pair $(0, 0)$. For $z = 1$, there are four tiles: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. Each tile (i, j) can be subdivided into four sub-tiles for level $z + 1$: $(2i, 2j)$, $(2i, 2j + 1)$, $(2i + 1, 2j)$, and $(2i + 1, 2j + 1)$.

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

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

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

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



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

202 First phase of tiling

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

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

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

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

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

228 Two parameters control the algorithm: tile capacity, \mathcal{C} , and the minimal size
 229 of a tile: $(\mathcal{W}, \mathcal{H})$. If for each (i, j) the number of elements in $L_z(i, j).tiledata$
 230 is not greater than \mathcal{C} , and if $w \leq \mathcal{W}$ and $h \leq \mathcal{H}$, where (w, h) is the current

231 tile size, then we try to build the next level L_{z+1} . Otherwise, the second phase
 232 starts.

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

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

241 Second phase of tiling

242 The second phase of tiling filters out the entities from the lower levels. We do not
 243 change the highest, the most detailed level. We sort the nodes of the graph into
 244 array N by PageRank [27]. 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

```

245 Here $\text{removeEntities}(L)$ empties all the tiles of level L , and returns map r allow-
 246 ing to restore the tiles. Function $\text{addNodeToLevel}(n)$ tries to add the node to L ,
 247 it also tries to add the tile elements for self edges of n , and the tile elements for
 248 the edges connecting n with the nodes ranked at least as high as n . These nodes
 249 are the nodes already added to L .
 250

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

252 Third phase of tiling

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

259 2 Future work

- 260 – Find a tiling method that guarantees that each tile has no more than \mathcal{C} ele-
261 ments on every level. One approach could be to use a more aggressive, and
262 regular edge bundling to reduce the number of edge clips in the tiles.
- 263 – Our tile calculation is memory intensive and take a long time for larger
264 graphs. The largest graph from the Table 1 that we were able to load with
265 Chrome, and Edge using the tiling procedure was p2p-Gnutella04 [23]. One
266 of the reasons was the memory limit on a process in those browsers, another
267 was the long running-time of the tiling procedure. A possible measure would
268 be saving the tiles to the disk and loading them on demand.
- 269 – For the user convenience we would like to run the layout, routing, and tiling
270 in a worker thread to avoid blocking the main thread.
- 271 – Addressing node labels visibility is an important task. We would like to en-
272 large the most important nodes of the view so that their labels are readable.

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. A. Beveridge and M. Chemers, "The game of game of thrones: Networked con-
311 cordances and fractal dramaturgy," in *Reading Contemporary Serial Television*
312 *Universes*, pp. 201–225, Routledge, 2018.
- 313 20. "b103." [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot)
314 [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).
- 315 21. "b100." [https://github.com/microsoft/automatic-graph-](https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot)
316 [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).
- 317 22. "Skewed." <http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml>.
- 318 23. "p2p-gnutella04." <https://snap.stanford.edu/data/p2p-Gnutella04.html>.
- 319 24. "facebookcombined." https://snap.stanford.edu/data/facebook_combined.txt.gz.
- 320 25. B. Rozemberczki and R. Sarkar, "Characteristic Functions on Graphs: Birds of a
321 Feather, from Statistical Descriptors to Parametric Models," in *Proceedings of the*
322 *29th ACM International Conference on Information and Knowledge Management*
323 *(CIKM '20)*, p. 1325–1334, ACM, 2020.
- 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.