Lines: 148

Browsing large graphs with MSAGLJS, a graph draph drawing tool in JavaScript

Lev Nachmanson and Xiaoji Chen

Microsoft Research, US,
levnach@hotmail.com, cxiaoji@gmail.com,
Msagljs github home page: https://github.com/microsoft/msagljs

Abstract. There has been progress in visualization of large graphs recently. Still, interacting with a large graph in the browser, with the same ease as browsing an online map, inspecting the high level structure and zooming to the lower details, is still an unsolved problem. In this paper we describe novel approaches to two aspects of this problem.
Firstly, we give a new algorithm for edge routing, where the edges do not overlap the nodes. The algorithm does not necesserely creates the optimal paths, but is efficient and creates visually appealing routes.
Secondly, to facilitate graph vizualization with DeckGL, we propose a new simple and fast tiling method. The method guarantees that in every view the number of visible entities is not larger than a predefined bound.

Introduction

We discuss large but not huge graphs. The maximum number of vertices of graphs we looked at was 28283, and the maximum number of edges was 237010. There are many algorithms that calculate a node layout for such graphs in a few seconds [1, 2], and we do not discuss them.

In the first part of the paper we address edge routing where an edge only intersects the nodes it is adjacent to. Our approach works for any node layout, as long as the nodes do not overlap each other. The approach builds on [3] and improves it.

Related work

[4]

[5]

[6]

[7]

[8]

machine learing approach [9]

[10]

[11]

Edge routing

The edge routing starts, as in [3], by building a spanner graph, an approximation of the full visibility graph. The spanner, see Fig. 2, is built on a variation of a Yao graph, which was introduced independently by Flinchbaugh and Jones [12] and Yao [13]. This kind of graph is defined by the set of cones with the apices at the vertices. The cones have the same angle, usually in the form of $\frac{2\pi}{n}$, where n is a natural number, and. The family of cones with the apex at a specific vertex partition the plane as illistrated in Fig. 1. For each cone at most one edge is created connecting the cone apex with a vertex inside of the cone, so the graph has $O(n)$ edges where n is the number of vertices.

The approach of [3] first builds a polyline path through the spanner, then applies some local modifications to shorten and smoothen the path. It tries to shortcut a vertex iteratively, as illustrated in Fig 3. To smoothen it fits Bezier segments into the polyline corners, using the binary search to find the larger fitting segments,see Fig 4. While anylyzing performance of edge routing in MSAGLJS, we noticed that for a graph with more than 100O nodes these heuristics sometimes create a performance bottleneck in spite of using R-Trees[14]. In addition, when the naive shortcutting of polyline corners fails, the resulting path is not visually appealing, as shown in Fig. 3.

We replace these heuristics with a more precize optimization.

Path optimization

Remember that a simple polygon is a polygon without holes.

An application of the 'path in a simple polygon' optimization is not a new approach. The authors of [15] used it, but only for hierarchical layouts, where a simple polygon, $\mathcal{P}$, containing the path is available. They write: "If $\mathcal{P}$ does not contain holes ... we can apply a standard "funnel" algorithm [16, 17] for finding Euclidean shortest paths in a simple polygon". In general case, for a non-layered layout, they build the visibility graph which is very expensive.

Here we show how to build polygon $\mathcal{P}$, and create a better path, for any layout. Let us describe our method.

We call obstacles $\mathcal{O}$ the set of polygons covering the original nodes, see Fig. 2. Before routing edges we calculate a Constrained Delaunay Triangulation [18] on $\mathcal{O}$ and call it $\mathcal{T}$. Then for each edge of the graph we proceed with the following steps.

We route a path, called $\mathcal{L}$, on the spanner, as illistrated by Fig. 5. Let $\mathcal{S}$ and $\mathcal{E}$ be the obstacles containing correspondengly $\mathcal{L}$'s start and end point. To obtain $\mathcal{P}$, 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 intersects $\mathcal{L}$ and is not inside of any obstacle in $O \setminus \{S, E\}$. The union of $\mathcal{U}$ gives us $\mathcal{P}$. The boundary of $\mathcal{P}$ comprizes all edges e of the triangles from $\mathcal{U}$ such that e is adjacent to exactly one triangle from $\mathcal{U}$, see Fig. 6.

To apply the funnel algorithm [16, 17], we need to have a triangulation of $\mathcal{P}$ such that every edge of the triangulation is either a boundary edge of $\mathcal{P}$, or a diagonal

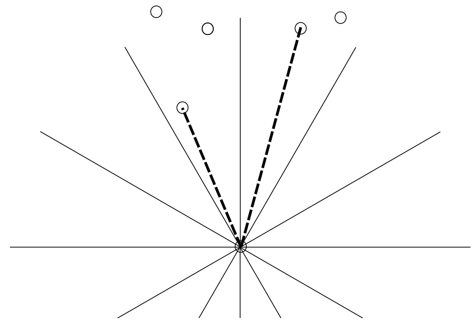

Fig. 1. Yao graph

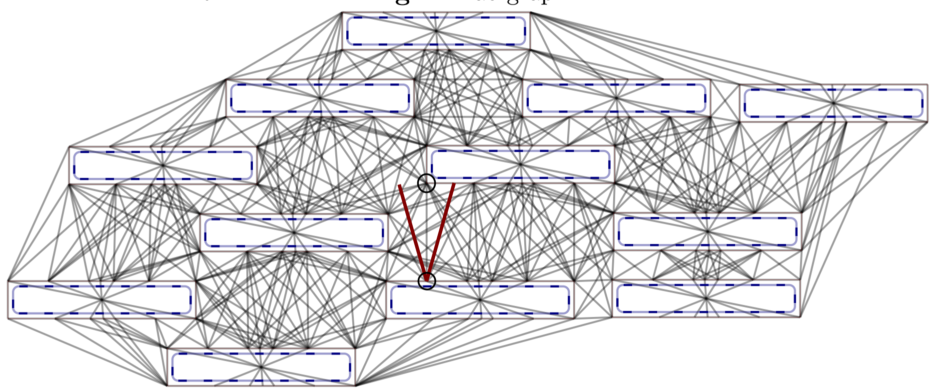

Fig. 2. Spanner graph is built using the idea of Yao graphs. The dashed curves are the original node boundaries. Each original curve is surrounded by a polygon with some offset to allow the polyline paths smoothing without intersecting the former. The edge marked by the circles is created because the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

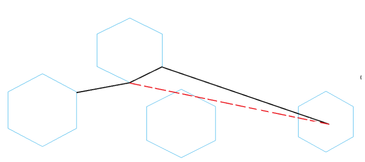

Fig. 3. Unsuccessful shortcut

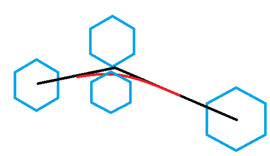

Fig. 4. Fitting a Bezier segment into a polyline corner

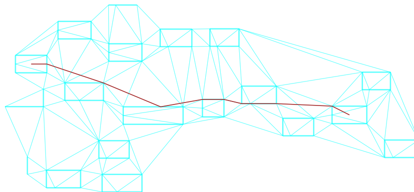

Fig. 5. Path $\mathcal{L}$ with $\mathcal{T}$, a fragment.

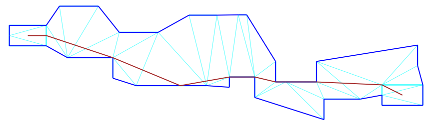

Fig. 6. Polygon $\mathcal{P}$ containing $\mathcal{L}$.

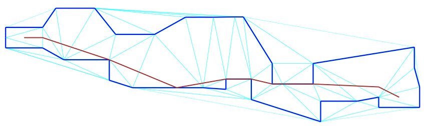

Fig. 7. New triangulation of $\mathcal{P}$.

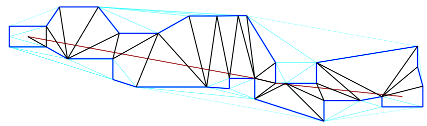

Fig. 8. The optimized path together with the sleeve diagonals.

of $\mathcal{P}$. In our setup $\mathcal{U}$ might not have this property, as in Fig. 6. We create a new Constrained Delaunay Triangulation of $\mathcal{P}$, where the set of constrained edges is the boundary of $\mathcal{P}$, see Fig. 7.
Finally, we apply the funnel algorithm with the new triangulation and obtain the path which is the shortest in the homotopy class of $\mathcal{L}$, as in Fig. 8.
The discussion [19] of the algorithm helped us in the implementation.
Polygon $\mathcal{P}$ is not neccesserely simple, as shown in Fig. 9. In this example the path that we calculate with the funnel algorithm is not the shortest path inside of $\mathcal{P}$.

Performance and quality comparison

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

Our performance experiments are summarized in Table. 1. We see that the older approach outperforms the new one on the smaller graphs; those with the number of nodes under 2000. The new method is faster on the rest of the graphs. We still prefer to use the new method independently of the graph size since the total slowdown is insignificant, under a half second in our experiments, but the quality of the paths is better. On the larger graphs the new method runs faster and produces better paths, so it is an obvious choice.

1 Tiling

The algorithm works in two phases. The first phase builds more and more detailed levels with smaller tiles until no more tile subdivision is required. Then the phase going from the higher to lower levels starts to finalize the levels.

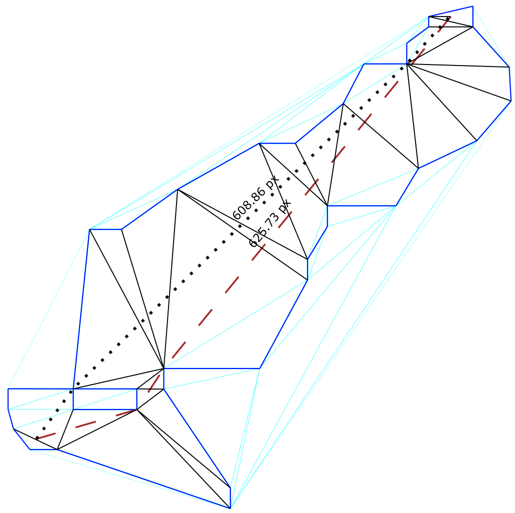

Fig. 9. $\mathcal{P}$ is not simple. The dotted path is shorter than the dashed one that was found by the routing.

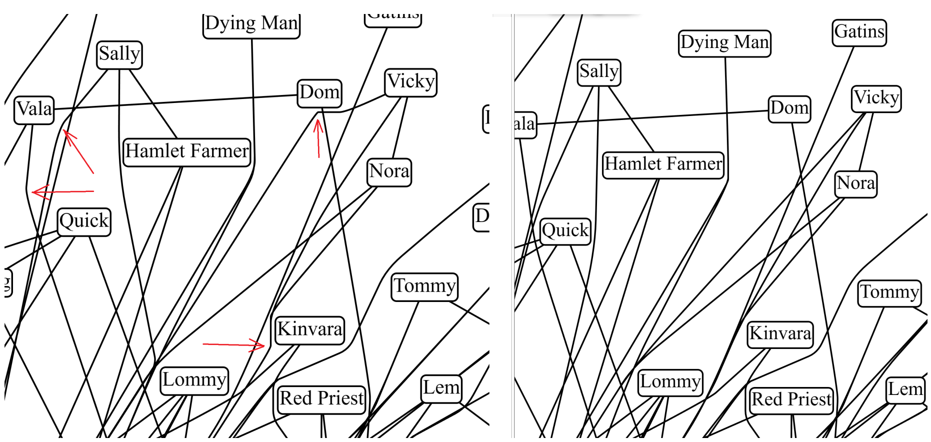

Fig. 10. The difference in the paths between the old, on the left, and the new, on the right, paths. The arrows on the left fragment point to the kinks that were removed by the new method.

graph	nodes	edges	old method's time	new time
social network [20]	407	2639	1.0	1.4
b103 [21]	944	2438	1.6	2.0
b100 [22]	1463	5806	5.6	5.785
composers [23]	3405	13832	510.5	17.5
p2p-Gnutella04 [24]	10876	39994	375.4	293.8
facebook_combined [25]	4039	88234	132.2	119.1
lastfm_asia_edges [26]	7626	27807	43.3	41.4
deezer_europe_edges [26]	28283	92753	1596.9	1209.3
ca-HepPh [27]	12008	237010	521.2	495.0

Table 1. Performance comparison with time in seconds.

A tile maybe thought of as a rectangle associated with some data. To form a tile hierarchy we code each tile by a triplet (i, j, z), where z is the level index and pair (i, j) indicates the rectangle inside of the level. The initial, the biggest tile on level 0 is represented by the triplet $(0, 0, 0)$. For $z = 1$ there are four tiles $(0, 0, 1), (0, 1, 1), (1, 0, 1)$ and $(1, 1, 1)$. Each tile (i, j, z) might be subdivided into four tiles of the same size one level higher: $(2i, 2j, z+1), (2i, 2j+1, z+1), (2i+1, 2j, z+1)$, and $(2i+1, 2j+1, z+1)$.

Each z-level is represented by a map $L(z)$, so $L(z)(i, j)$ gives us a specific tile. During the first phase we can discover some empty tiles which correspond to $L(z)(i, j)$ being not defined.

For the minimal size of the tile we take $(8 \times w, 8 \times h)$, where w is the average width and h is the average height of the nodes of the graph. The algorithm starts after the edge routing is done, so each edge has a curve, an optional label, and arrowheads associated with it. For each edge e with curve c we create a pair *curve clip*, (e, c). The algorithm keeps a map from tilesInitially, we create one top level tile and

One of the parameters controlling the algorithm is the tile capacity, $\mathcal{C}$, setting the upper limit on how many visual elements can be visible in one tile. The elements could be a curve clip, an arrowhead, a node, or a label. In our setting $\mathcal{C}$ is set by default to 10000.

References

1. Y. Hu and L. Shi, “Visualizing large graphs,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 2, pp. 115–136, 2015.
2. U. Brandes and C. Pich, “Eigensolver methods for progressive multidimensional scaling of large data,” in *Graph Drawing: 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers 14*, pp. 42–53, Springer, 2007.
3. T. Dwyer and L. Nachmanson, “Fast edge-routing for large graphs,” in *Graph Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September 22-25, 2009. Revised Papers 17*, pp. 147–158, Springer, 2010.

4. “Graphviz.” http://www.graphviz.org/.
5. “Regraph.” https://cambridge-intelligence.com/regraph/.
6. “Skewed.” https://graph-tool.skewed.de.
7. “Circos.” http://circos.ca/.
8. H. Gibson, J. Faith, and P. Vickers, “A survey of two-dimensional graph layout techniques for information visualisation,” *Information visualization*, vol. 12, no. 3-4, pp. 324–357, 2013.
9. O.-H. Kwon, T. Crnovrsanin, and K.-L. Ma, “What would a graph look like in this layout? a machine learning approach to large graph visualization,” *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 478–488, 2017.
10. Z. Lin, N. Cao, H. Tong, F. Wang, U. Kang, and D. H. Chau, “Interactive multi-resolution exploration of million node graphs,” in *IEEE VIS*, 2013.
11. “Cosmograph.” https://cosmograph.app.
12. B. Flinchbaugh and L. Jones, “Strong connectivity in directional nearest-neighbor graphs,” *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 4, pp. 461–463, 1981.
13. A. C.-C. Yao, “On constructing minimum spanning trees in k-dimensional spaces and related problems,” *SIAM Journal on Computing*, vol. 11, no. 4, pp. 721–736, 1982.
14. A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47–57, 1984.
15. D. P. Dobkin, E. R. Gansner, E. Koutsofios, and S. C. North, “Implementing a general-purpose edge router,” in *Graph Drawing: 5th International Symposium, GD’97 Rome, Italy, September 18–20, 1997 Proceedings 5*, pp. 262–271, Springer, 1997.
16. B. Chazelle, “A theorem on polygon cutting with applications,” in *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pp. 339–349, IEEE, 1982.
17. J. Hershberger and J. Snoeyink, “Computing minimum length paths of a given homotopy class,” *Computational geometry*, vol. 4, no. 2, pp. 63–97, 1994.
18. B. Delaunay, “Sur la sphere vide, bull. acad. science ussr vii: Class,” *Sci. Mat. Nat*, pp. 793–800, 1934.
19. “Funnel algorithm.” https://page.mi.fu-berlin.de/mulzer/notes/alggeo/polySP.pdf.
20. A. Beveridge and M. Chemers, “The game of game of thrones: Networked concordances and fractal dramaturgy,” in *Reading Contemporary Serial Television Universes*, pp. 201–225, Routledge, 2018.
21. “b103.” https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b103.dot.
22. “b100.” https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/Test/MSAGLTests/Resources/DotFiles/LevFiles/b100.dot.
23. “Skewed.” http://mozart.diei.unipg.it/gdcontest/contest2011/composers.xml.
24. “p2p-gnutella04.” https://snap.stanford.edu/data/p2p-Gnutella04.html.
25. “facebookcombined.” https://snap.stanford.edu/data/facebook_combined.txt.gz.
26. B. Rozemberczki and R. Sarkar, “Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models,” in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM ’20)*, p. 1325–1334, ACM, 2020.
27. J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graph evolution: Densification and shrinking diameters,” *ACM transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, pp. 2–es, 2007.