Implementing Planar L-Drawings of Bimodal Triangulated Graphs

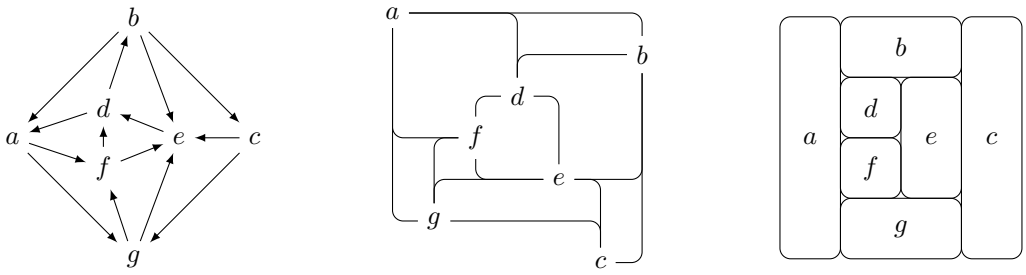
July 1, 2021

Abstract

In their 2020 paper, Angelini et al. [2] present a linear-time algorithm for constructing planar L-drawings of bimodal graphs. We implement and evaluate their algorithm. To this end, we also implement an algorithm for drawing rectangular duals [3] and for sampling plane triangulations [10]. Our implementation realizes the theoretical linear running times. Furthermore, we develop an easy-to-implement algorithm that decomposes a graph into its 4-connected components. Finally, we point out flaws in the papers on planar L-drawings and rectangular duals.

1 Introduction

L-drawings for directed graphs were first introduced by Angelini et al. [1] for their increased readability of vertices with high degree. In an L-drawing, vertices are drawn on distinct x- and y-coordinates, and a directed edge from vertex v to vertex w is drawn as a 1-bend polyline consisting of a vertical segment incident to v and a horizontal segment incident to w. An example of an L-drawing can be seen in Figure 1b. Chaplick et al. [6] focused on upward-planar L-drawings and gave a linear-time algorithm for finding such a drawing if the embedding is fixed and admits an upward-planar L-drawing. Angelini et al. [2] finally showed that every bimodal graph without 2-cycles admits a planar L-drawing, however not always an upward-planar one. Their approach is based on rectangular duals. An example of a rectangular dual can be seen in Figure 1c. They leave it as an open problem whether all bimodal graphs *with* 2-cycles admit a planar L-drawing.



(a) Straight-line (b) Planar L-drawing (c) Rectangular dual

Figure 1: Various drawings of a graph

In this report, we implement the algorithm for planar L-drawings of bimodal graphs, as well as the algorithms it uses as subroutines, and evaluate it on sampled [10] input graphs. The algorithm for planar L-drawings involves decomposing a graph into its 4-connected components. To this end we describe an algorithm, which is comparably easy to implement and yields the desired output directly. Our method follows a similar approach as the left-right planarity test, as described by Brandes [5].

The remainder of this work is organized as follows: Section 2 lays out all necessary definitions. The algorithms used are described in Section 3. We describe the experimental setup in Section 4 and give the experimental results in Section 5. In Section 6, we correct some minor faults we found in the used algorithms. We conclude with open problems in Section 7.

2 Preliminaries

2.1 Graphs

An *undirected graph* $G = (V, E)$ consists of a *vertex* set V and a set of *undirected edges* $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$. A *directed graph* $G = (V, E)$ consists of a *vertex* set V and a set of *directed edges* $E \subseteq \{(v, w) \mid v, w \in V, v \neq w\}$. We leave out *directed* and *undirected* whenever it is clear from context. Given an edge $e = (v, w)$, we say v and w are the end points of e, v is the *tail* of e, w is the *head* of e. e is an *outgoing* edge of v, e is an *incoming* edge of w. v and w are *neighbors* or *adjacent*. An edge is *incident* to its endpoints, and its endpoints are *incident* to the edge. The degree $deg(v)$ of a vertex v is the number of edges incident to v.

2.2 Connectivity

A *directed* (v_0, v_l)*-path* of length l in a directed graph G is a sequence $\langle v_0, e_1, v_1, e_2, \ldots, e_l, v_l\rangle$ of vertices v_i and edges e_i, such that $e_i = (v_{i-1}, v_i)$ for $i \in \{1, \ldots, l\}$. An *undirected* (v_0, v_l)*-path* of length l in a directed graph G is a sequence $\langle v_0, e_1, v_1, e_2, \ldots, e_l, v_l\rangle$ of vertices v_i and edges e_i, such that $e_i = (v_{i-1}, v_i)$ or $e_i = (v_i, v_{i-1})$ for $i \in \{1, \ldots, l\}$. A directed (resp. undirected) (v_0, v_0)-path of length l is called a directed (resp. undirected) l*-cycle*. We identify a path with the sequence of its vertices or edges. G is *connected* if for each pair of vertices $v, w \in V$ there is an undirected (v, w)-path in G. $G \setminus S$ is the graph G with all vertices from S along with their incident edges removed. G is k*-connected* if there is no set $S \subseteq V$ of $k-1$ vertices such that $G \setminus S$ is disconnected. If there is such a set S, it is called a separating $(k-1)$*-tuple*. A separating 3-tuple is also referred to as *separating triple* or, if there are edges between its members, *separating triangle*. A *tree* is a connected undirected graph that contains no cycle. In a tree vertices of degree 1 are called *leaves*.

2.3 Planarity

A *drawing* of G is a mapping of the vertices and edges of G into the Euclidean plane; often vertices are drawn as points and edges as lines. We identify vertices and edges with their drawings. If the cyclic order of edges around each vertex is given, we call it an *embedding* of G. A drawing (with points and lines) is *planar* if edges only intersect in common end points. An embedding of G is *planar* if there is a planar drawing of G respecting that embedding. A graph is *planar* if it has a planar embedding. A planar drawing divides the plane into regions; these regions are called *faces* of the corresponding plane graph. The unbounded face is called the *outer face*. All other faces are called *inner faces*. A graph together with a planar embedding and a fixed outer face is called a *plane* graph. The *degree* of a face is the number of half-edges that form its boundary. In 2-connected graphs this is equal to the number of vertices that lie on its boundary. A graph is *triangulated* if every face has degree 3. A graph is *internally triangulated* if every inner face has degree 3. A vertex v in a plane directed graph is k*-modal* ($mod(v) = k$) if in the embedding around v there are exactly k pairs of consecutive edges that are neither both incoming nor both outgoing. A plane directed graph is k*-modal* if $mod(v) \leq k$ for each of its vertices v. A 2-modal graph is also referred to as *bimodal*.

2.4 L-Drawings

An *L-drawing* of a directed graph $G = (V, E)$ is a mapping of the vertices of G to points into the plane such that no two vertices are assigned the same x- or y-coordinates. In other words an L-drawing is a pair of injective functions $x, y : V \to \mathbb{N}$. A directed edge (v, w) is then drawn

as the polyline $(x(v), y(v)) - (x(v), y(w)) - (x(w), y(w))$, that is a vertical segment incident to v, followed by a horizontal segment incident to w. For each vertex, we consider four *ports*, North, East, South, and West. In an L-drawing, an edge is assigned to the North (resp. South) port of its tail if it leaves it to the top (resp. bottom), and to the East (resp. West) port of its head if it enters it from the right (resp. left). An L-drawing is *planar* if no two edges cross, however they may share a subsegment at a common endpoint.

2.5 Rectangular Duals

An *irreducible trianglulation* is a 4-connected internally triangulated plane graph with an outer face of degree 4. A *rectangular tiling* is a partition of a rectangle into smaller rectangles such that no four rectangles touch in the same point. A rectangular tiling T is called a *rectangular dual* of a graph G if there is a one-to-one correspondence between the rectangles in T and vertices of G such that there exists an edge between two vertices iff the corresponding rectangles touch.

3 Algorithms

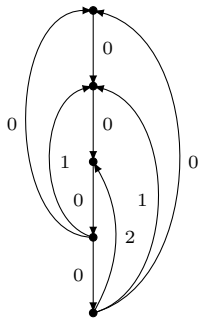
3.1 4-Connected Components

The first task is to find the 4-connected components of the triangulated input graph. To this end, Angelini et al. [2] cite Kant [9], who described a method for this problem. The main idea there is to list all triangles, as described by Chiba and Nishizeki [7], select the separating ones among them, and then to order them, either from the inside out or from the outside in, such that one can split the graph along each separating triangle in constant time while keeping track of which edge is incident to which copy of the original vertex. This splitting step is not well documented and rather involved; therefore we will propose a different method of ordering the separating triangles, namely only from the inside out. This way the splitting need only be done in linear time to accomplish the decomposing in linear time. The input graph is considered to be undirected for this step, and any direction defined on the edges is defined during a traversal of the graph. For examples see Figure 2.

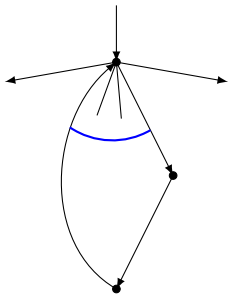
Listing separating triples. Here, we make use of the fact that in a plane triangulation all separating triples are separating triangles. Therefore, it suffices to list all triangles [7] and then select those triangles u, v, w where v and w do not appear consecutively in the adjacency list of u.

Ordering separating triangles. This step is heavily inspired by the left-right planarity test [5]. First, we perform a depth-first search on the undirected input graph, starting from one of the outer vertices r. During this process we take note of the following:

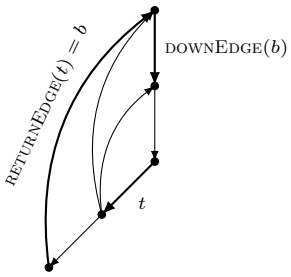
- the direction that each edge was traversed in
- for each edge e whether it is a tree edge (its head was newly discovered when traversing e) or a back edge (its head had already been discovered)
- $\textsc{depth}(v)$ of each vertex v, the length of the (r, v)-path along tree edges
- $\textsc{parentEdge}(v)$ of each vertex v, the edge along which v was first discovered. For r, this is a virtual edge between the edges to its two neighbors on the outer face.
- for each back edge e:
  - $\textsc{lowpoint}(e) = \textsc{depth}(e.head)$
  - $\textsc{downEdge}(e) = (p, c)$ if $p = e.head, t = e.tail$ and $\langle p, c, \ldots, t\rangle$ is the unique (p, t)-path along tree edges.
  - $\textsc{returnSide}(e) \in \{\text{Left}, \text{Right}\}$ is the side on which e enters its head p: Left if the clockwise order of edges around p is $\textsc{parentEdge}(p)$, $\textsc{downEdge}(e)$, e; Right if it is $\textsc{parentEdge}(p)$, e, $\textsc{downEdge}(e)$.



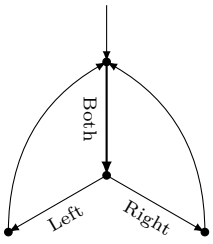
(a) Edges with their LOWPOINTS



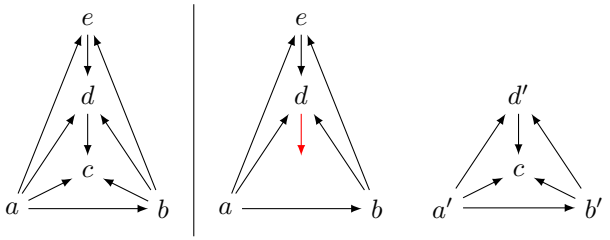
(b) A back edge with RETURNSIDE = Left and ANGULARDISTANCE = 2



(c) A tree edge, its RETURNEDGE and a DOWNEDGE



(d) A tree edge with RETURNSIDE = Both



(e) Before and after splitting along the separating triangle with virtual edge

Figure 2: Examples illustrating the decomposition step

  - angularDistance(e) is the number of edges that appear in the adjacency list of $e.head$ in clockwise order...
    * if returnSide(e) = Left: ... between downEdge(e) and e.
    * if returnSide(e) = Right: ... between e and downEdge(e).

- for each tree edge e:
  - returnEdge(e): Consider directed paths $\langle t_1, \dots, t_l, b \rangle$ where t_i are tree edges, b is a back edge and $t_1 = e$. Select those with minimal lowpoint(b) among them. Then select those with maximal angularDistance(b) among them. Finally, select any path from these and let returnEdge(e) = b of that path.
  - lowpoint(e) = lowpoint(returnEdge(e))
  - returnSide(e): If, when finding returnEdge(e), not all paths $\langle t_1, \dots, t_l, b \rangle$ with minimal lowpoint(b) had the same returnSide(b), then let returnSide(e) = Both. Otherwise let returnSide(e) = returnSide(returnEdge(e)).
  - angularDistance(e) = angularDistance(returnEdge(e))

With this information, we define for each vertex v a partial ordering $\prec$ of its outgoing edges:

- $e_1 \prec e_2$ if lowpoint(e_1) > lowpoint(e_2)
- $e_1 \prec e_2$ if lowpoint(e_1) = lowpoint(e_2), returnSide(e_1) = returnSide(e_2) and angularDistance(e_1) < angularDistance(e_2)
- $e_1 \prec e_2$ if lowpoint(e_1) = lowpoint(e_2), returnSide(e_1) $\neq$ returnSide(e_2), e_1 is a back edge and e_2 is a tree edge
- $e_1 \prec e_2$ if returnSide(e_2) = Both. Note that there is at most one such edge e_2.

After sorting the outgoing edges of each vertex according to this ordering $\prec$, we perform a second depth-first search of the graph, following the order of edges. During this second traversal of the graph, we take note of the order in which the separating triangles are discovered. For this purpose, we consider a separating triangle to be discovered once all three of its edges have been traversed.

If several separating triangles $T_1, \dots, T_n$ are discovered at the same time, they share an edge $\{v, w\}$ and have distinct third vertices $u_1, \dots, u_n$. For each triangle T_i consider the vertex x on it that has the lowest depth; parentEdge(x) is on the outside of T_i. Now sort $T_1, \dots, T_n$ according to the number of neighbors v (or w) has on the inside of the respective triangle. The triangle with the least neighbors of v (or w) on its inside is the innermost. We consider $T_1, \dots, T_n$ to be discovered in this sorted order, from the innermost to the outermost.

The order in which the separating triangles are discovered here is the order in which they will be processed in the next step.

Splitting along separating triangles. For each separating triangle with vertices u, v, w, make copies u', v', w' of its three vertices and add the three edges $\{u', v'\}, \{v', w'\}, \{w', u'\}$. Transfer to u' the edges incident to u on the inside of the triangle; analogously for v and w.

If the edges $\{u, v\}$ and $\{u, w\}$ are both incoming (resp. outgoing) in the original graph and u' has now an outgoing (resp. incoming) edge, add a virtual outgoing (resp. incoming) edge to u between $\{u, v\}$ and $\{u, w\}$; analogously for v and w. This is done to preserve the bimodality of u in the outer 4-connected component for the port assignment.

Identifying 4-connected components. Finally, identify the connected components of the resulting graph, for example using breadth-first search. These are the 4-connected components of the original graph since the 4-connected components have been turned into connected components during the splitting step.

If we take for each separating triangle u, v, w the connected component with the outer face u', v', w' in the reverse splitting order, it holds that the component containing u, v, w will appear before the component with the outer face u', v', w'.

Linear running time. During the first depth-first search, do not list the paths $\langle t_1, \ldots, t_l, b\rangle$ explicitly, but instead keep track of the back edge b with minimal LOWPOINT and maximal ANGULARDISTANCE in the subtree that is currently being traversed. When popping a tree edge t from the depth-first search stack, we will have RETURNEDGE$(t) = b$. To find RETURNSIDE(t), take note if a back edge with LOWPOINT and ANGULARDISTANCE equal to b is found in the subtree.

Do not find the path $\langle p, c, \ldots, t\rangle$ explicitly after traversing the subtree rooted at c, but instead save the edge (p, c) as ACTIVECHILDEDGE(p) while the subtree rooted at c is being traversed.

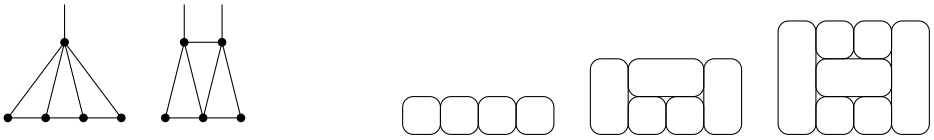
Use bucket sort to sort the outgoing edges of each vertex before the second depth-first traversal.

3.2 Rectangular Duals

The next task is to construct rectangular duals of the 4-connected components. For this, we use the algorithm presented in Biedl and Derka [3], which takes as input an irreducible trianglulation.

Finding a $(3, 1)$-ordering. The first step is to find an ordering of paths in the graph, such that each of these paths has at least 3 predecessors in the ordering (neighboring vertices that appear before it) and at least one successor. Each of these paths may take one of the two forms shown in Figure 3a.

Constructing the rectangular dual. Each of the paths is then placed to the right of its leftmost predecessor, to the left of its rightmost predecessor and above all other predecessors, as can be seen in Figure 3b.



(a) Singleton and fan paths in the $(3, 1)$-ordering

(b) Incremental drawing of the dual

Figure 3: The algorithm for drawing rectangular duals

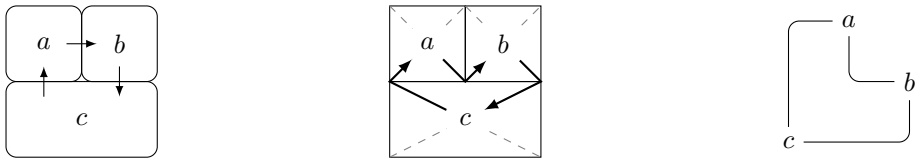
3.3 Port Assignment

The port assignment algorithm [2] takes as input a 4-connected bimodal triangulation and an L-drawing of its outer face. First, split one of the edges on the outer face by inserting a dummy vertex into it. Now the outer face has degree 4; therefore the graph is now an irreducible triangulation, for which a rectangular dual can be constructed.

After drawing the rectangular dual, we place each vertex in the center of its rectangle (Figure 4a). The North port of a vertex is then to its top left, the East port to its top right, the South port to its bottom right, and the West port to its bottom left. The edges are routed parallel to the diagonals of the rectangles (Figure 4b). An outgoing edge that leaves the rectangle of a vertex to the top will be assigned to its North port if possible, because that port points to the top (left), unlike the South port, which points to the bottom (right). Recall that outgoing edges may only be assigned to the North or South port. We handle incoming edges and edges on other sides of the rectangle analogously.

After the port assignment in one 4-connected component C is complete, the same can be done for all other 4-connected components whose outer face is an inner face of C. This way, we execute a preorder traversal of the tree of 4-connected components. Note that in our implementation this tree is not built explicitly, but instead its nodes are given in preorder in the order of separating triangles (Section 3.1).

When all ports in the graph have been assigned, the edges imply an order of the vertices in x- and y- direction. For example, if an edge (v, w) has been assigned to the North port of v and to the West port of w, v must be drawn below and to the left of w. This order can be found through topological ordering.



(a) Rectangular dual (b) Port assignment (c) L-drawing

Figure 4: From rectangular dual to L-drawing

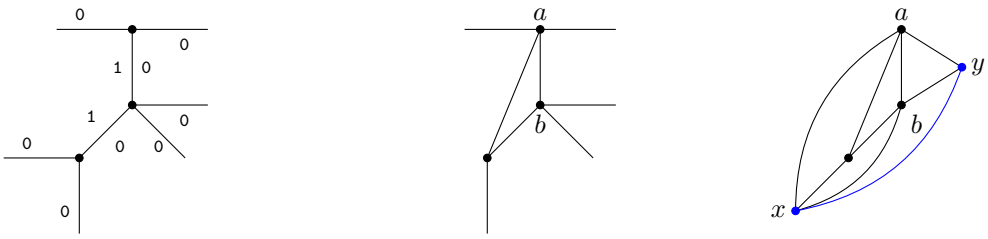
3.4 Sampling Triangulations

To test the above algorithms, we used a sampling algorithm that was developed by Poulalhon and Schaeffer [10]. It takes as input an integer n and outputs a plane triangulation with $n+2$ vertices and $3n$ edges. It randomly generates a bitstring of length $4n-2$ with $3n-1$ zeros and $n-1$ ones. Using the cycle lemma [8], it finds one of its cyclic shifts s that has the property that each prefix u of s satisfies $3|u|_1 - |u|_0 \geq -2$.

From this bitstring s a tree is then constructed. All inner (non-leaf) vertices in this tree are adjacent to exactly two leaves. The construction works as follows: The bitstring encodes a walk around the tree in counter-clockwise direction; a `1` stands for a downwards edge connecting to inner vertices; a `0` stands for a leaf if the current inner vertices does not have to adjacent leaves yet, otherwise it stands for an upwards edge. This way each inner edge appears twice in the encoding, each leaf edge only once. Such a tree constructed from a bitstring can be seen in Figure 5a.

Next, walk around the graph, searching for two consecutive inner edges followed by a leaf edge. Whenever this configuration is encountered, attach the leaf edge to the first of the three inner vertices that are incident to the three edges. The result of this operation is shown in Figure 5b.

When such a configuration cannot be found anymore, each vertex on the outer face will have exactly one incident leaf edge with the exception of two vertices a, b that still have two incident leaf edges. Now add two more vertices x, y in the outer face and connect all leaf edges between a and b to x and all leaf edges between b and a to y. Finally, triangulate the outer face by adding the edge $\{x, y\}$ in the outer face. Figure 5c shows the complete constructed graph.



(a) Tree with two leaves per inner vertex (b) After closing all triangles (c) Complete graph

Figure 5: Example of the sampling algorithm for $n = 3, s =$ `0110000000`

4 Experimental Setup

We used the algorithm described in Section 3.4 to sample plane undirected triangulations for different values of n. The edges of these graphs were then directed through an open ear decomposition [4]. This implies a bimodal embedding (upward-planar, even). The resulting directed graph was drawn by the algorithm of Angelini et al. [2].

To show the correctness of our implementation, we sampled 1000 graphs with $n = 1000$ and naively tested whether there are any crossings in the L-drawing computed by our implementation.

To test the running time, we timed our program for sampled input graphs with $n \in \{10^6, 2 \cdot 10^6, \ldots, 10^7\}$. All tests were executed on a 64-bit computer with 24 GB of DDR4-2133 RAM running Ubuntu 20.04.2 LTS, in a single thread on a single core of an Intel i7-7700K CPU @ 4.20 GHz. The implementation was written in C++17 using the Standard Template Library and compiled with g++ 9.3.0 set to the highest optimization level. The running time of the sampling algorithm was measured using the `time` builtin of zsh 5.8. The running times of the individual subroutines of the L-drawing algorithm were measured using `std::chrono::steady_clock`.

5 Test Results

No intersections were found in the correctness test.

The running times of the sampling algorithm, including the open ear decomposition can be seen in Figure 6. It is clear that the running time of our implementation is linear in n.

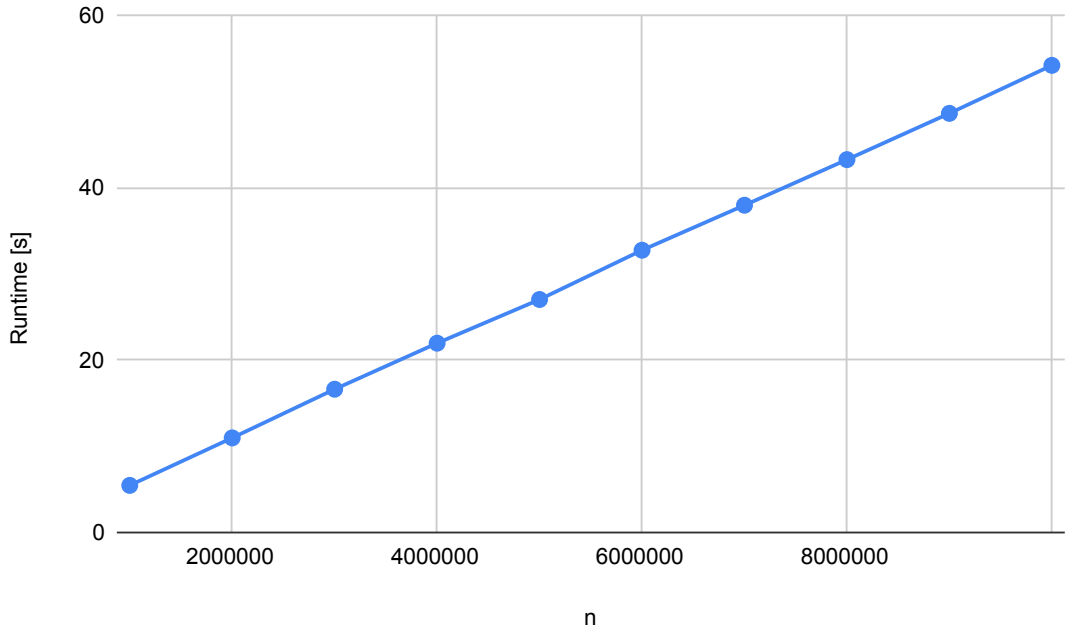


Figure 6: Running times for the sampling algorithm

Figure 7 shows the running times of our implementation of the algorithm of Angelini et al. [2]. The times for reading the input graph and writing the output drawing, for decomposing the graph into its 4-connected components, for drawing the rectangular dual of each 4-connected component, and for assigning each edge to the ports of its endpoints are given separately. The times are linear in n as well.

6 Other Results

6.1 Identifying Basic 2-Leg Centers for the (3,1)-ordering

The configuration on the right of Figure 3a is called a *fan*. A fan is a simple path of vertices $c_1, \ldots, c_k$ that are all adjacent to one common predecessor y, which is called *minimal 2-leg center*. Additionally, the leftmost vertex c_1 of the path is adjacent to a predecessor l one to the left from the minimal 2-leg center, and the rightmost vertex c_k is adjacent to a predecessor r one to the

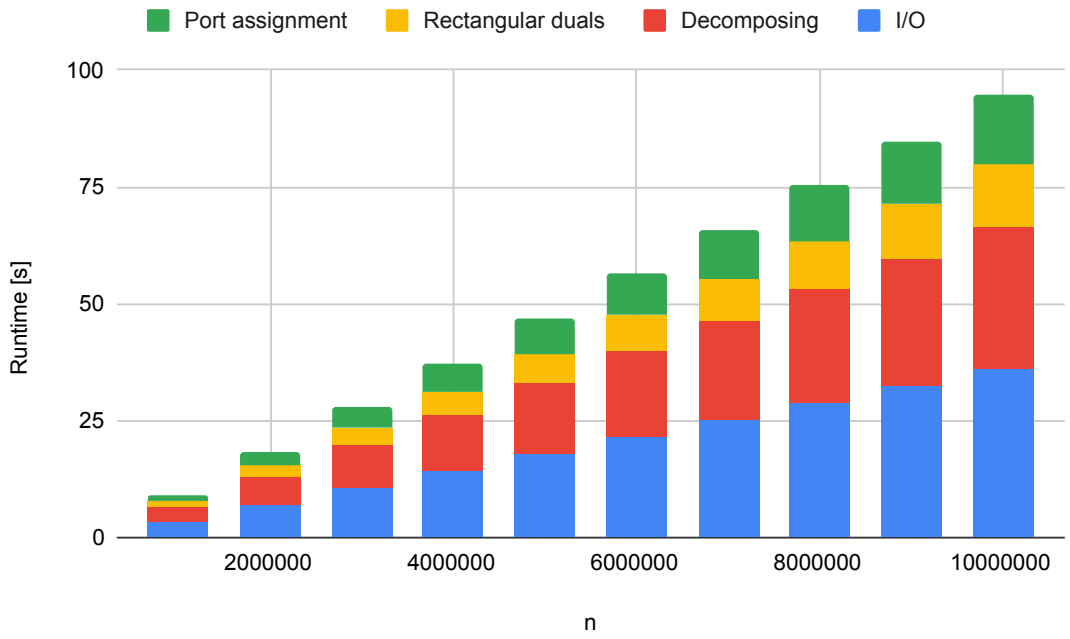


Figure 7: Running times for the L-drawing algorithm

right from y. A fan is picked whenever there is no singleton that can be picked. Biedl and Derka [3] define a *2-leg center* as an inner vertex that is adjacent to two non-adjacent vertices l and r on the outer face. A 2-leg center is called *basic* if the vertices between l and r on the outer face all have degree 3. They then claim that if a 2-leg center y is basic, y is the common neighbor of $c_1, \ldots, c_k$.

However, consider the situation depicted in Figure 8. According to the definition, both x and y are basic 2-leg centers, but the claim only holds true for y. By additionally requiring that $c_1, \ldots, c_k$ all have to be adjacent to x for x to be a basic 2-leg center, x is no longer basic and thus the claim is true.

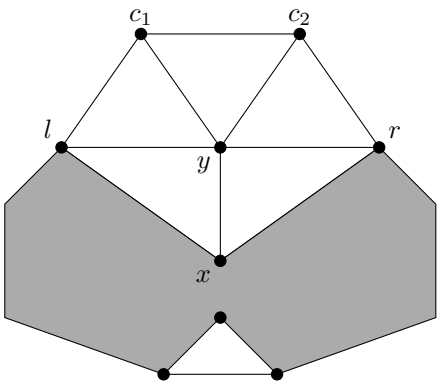


Figure 8: 2-leg center with fan above it

In order to test whether y is a basic 2-leg center, they compute $\textsc{outer}(y)$, the number of neighbors of x on the outer face, and $\textsc{outerDeg3}(y)$, the number of neighbors of x on the outer face with degree equal to 3. If $\textsc{outer}(y) = \textsc{outerDeg3}(y) + 2$, that is if y has exactly two neighbors on the outer face with degree greater than 3, then y is a basic 2-leg center.

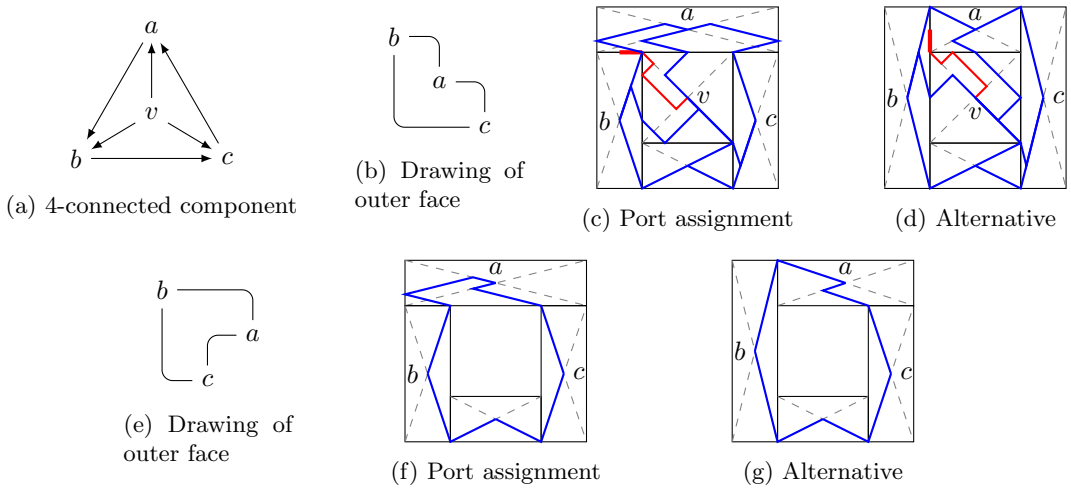
Once again, consider Figure 8. Both x and y have exactly two neighbors on the outer face with degree greater than 3, namely l and r. However, x is not a basic 2-leg center. If we additionally require $\text{OUTERDEG3}(x) > 0$ for x to be a a basic 2-leg center, x is no longer incorrectly identified as a basic 2-leg center, and the implementation works.

6.2 Drawing the Outer Face Correctly for Port Assignment

When performing the port assignment, the fixed drawing of the outer face determines which of the outer edges is subdivided and which rectangle occupies the corners of the rectangular dual. For example, consider the 4-connected component in Figure 9a with the drawing shown in Figure 9b. As presented by Angelini et al. [2], the edge (b, c) is subdivided and the rectangular dual is drawn as shown in Figure 9c.

If there is now a virtual vertex (drawn in red) adjacent to v in the face bounded by v, a, b, this virtual vertex will be drawn as a rectangle with height 0 to the right of v. Because of this, the edge (v, b) will be assigned to the South port of v. Now b has to be drawn below v, v below a and a below b. This is impossible. Instead, we draw the rectangular dual as shown in Figure 9d. This way we avoid b being drawn below a, which would conflict with the assignment of the edge (a, b) to the North port of a.

Analogously, if the outer face is drawn as in Figure 9e, we do not draw the rectangular dual like in Figure 9f but instead like in Figure 9g.



(a) 4-connected component (b) Drawing of outer face (c) Port assignment (d) Alternative

(e) Drawing of outer face (f) Port assignment (g) Alternative

Figure 9: L-drawing of two particular 4-connected components

7 Discussion and Open Problems

We sampled undirected plane triangulations uniformly at random. However, the edge directions are not uniformly random. For example, the graphs we sampled are always cycle-free with a single source and a single sink on the outer face. A solution to this shortcoming may be to reverse some randomly chosen edges of the graph where bimodality permits it. This approach would add sources and sinks on the inside of the graph as well as cycles. It is probably still not an adequate method of sampling bimodal triangulations uniformly at random.

Angelini et al. [2] left as an open problem whether all bimodal graphs with 2-cycles admit a planar embedding. Our decomposition algorithm recognizes every triangle in the graph that has a 2-cycle as one of its sides as a separating triangle. Therefore it was impossible to do any extensive testing on graphs with 2-cycles. However, the port assignment algorithm without any

modifications certainly does not work correctly for graphs with 2-cycles, as one can see in Figure 10. There, the edge on the far left has three bends instead of the desired one bend.

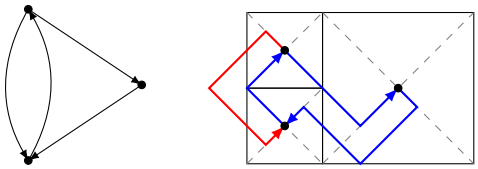


Figure 10: Graph with 2-cycle and an incorrect port assignment

In this particular case, the problem can be fixed by changing the embedding and swapping the two edges of the 2-cycle, but in more complex cases this may violate bimodality or even 4-modality.

References

[1] P. Angelini, G. Da Lozzo, M. Di Bartolomeo, V. Di Donato, M. Patrignani, V. Roselli, and I. G. Tollis. Algorithms and bounds for L-drawings of directed graphs. *International Journal of Foundations of Computer Science*, 29(04):461–480, 2018. URL `https://doi.org/10.1142/S0129054118410010`.

[2] P. Angelini, S. Chaplick, S. Cornelsen, and G. Da Lozzo. Planar L-drawings of bimodal graphs, 2020. URL `https://arxiv.org/abs/2008.07834v1`.

[3] T. Biedl and M. Derka. The (3,1)-ordering for 4-connected planar triangulations. *Journal of Graph Algorithms and Applications*, 20(2):347–362, 2016. URL `https://doi.org/10.7155/jgaa.00396`.

[4] U. Brandes. Eager st-ordering. In *Algorithms — ESA 2002*, pages 247–256. Springer Berlin Heidelberg, 2002. URL `https://doi.org/10.1007/3-540-45749-6_25`.

[5] U. Brandes. The left-right planarity test, 2009. URL `https://www.uni-konstanz.de/algo/publications/b-lrpt-sub.pdf`.

[6] S. Chaplick, M. Chimani, S. Cornelsen, G. Da Lozzo, M. Nöllenburg, M. Patrignani, I. G. Tollis, and A. Wolff. Planar L-drawings of directed graphs. In *International Symposium on Graph Drawing and Network Visualization*, pages 465–478, 2017. URL `https://doi.org/10.1007/978-3-319-73915-1_36`.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985. URL `https://doi.org/10.1137/0214017`.

[8] N. Dershowitz and S. Zaks. The cycle lemma and some applications. *European Journal of Combinatorics*, 11(1):35–40, 1990. URL `https://doi.org/10.1016/S0195-6698(13)80053-4`.

[9] G. Kant. A more compact visibility representation. *International Journal of Computational Geometry & Applications*, 07(03):197–210, 1997. URL `https://doi.org/10.1142/S0218195997000132`.

[10] D. Poulalhon and G. Schaeffer. Optimal coding and sampling of triangulations. *Algorithmica*, 46(3):505–527, 2006. URL `https://doi.org/10.1007/s00453-006-0114-8`.