Minimizing Edge Length Ratio in Planar Graphs on the Grid

Bachelor Thesis of

Luca Buchholz

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	J.-Prof. Thomas Bläsius
Advisors:	Paul Jungeblut

Time Period: 16th May 2022 – 16th September 2022

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, October 18, 2022

Abstract

The *edge length ratio* of a graph drawing is defined as the ratio obtained by dividing the length of the longest edge and the shortest Euclidean distance between any two adjacent vertices. Optimizing this ratio for planar graphs drawn on a grid is the topic of the live challenge at the 30th International Symposium on Graph Drawing and Network Visualization. It is already known that deciding whether a graph admits a drawing with an *edge length ratio* of exactly one is $\exists\mathbb{R}$-complete.

For this thesis, an algorithm optimizing the *edge length ratio* in drawings of planar graphs on grids is implemented. The first step is finding an initial drawing, for which several planar embedders and graph layout algorithms are compared. Using these, one can optimize the *edge length ratio* by performing a local search on these initial drawings. To assess the quality of the drawings, several new measure functions need to be introduced, each taking a drawing as its input and outputting a numerical value. Developing the algorithm and measures introduces several parameters, which are subsequently optimized in the experiments.

Deutsche Zusammenfassung

Die *Edge Length Ratio* einer Graphenzeichnung ist das Verhältnis zwischen der Länge der längsten Kante und der kürzesten Entfernung zweier adjazenter Knoten eines Graphen. Die Optimierung dieses Werts ist das Thema der Live-Challenge auf dem dreißigsten “Symposium on Graph Drawing and Network Visualization”. Es ist bereits bekannt, dass das Entscheidungsproblem ob eine Zeichnung mit einer *Edge Length Ratio* von exakt Eins existiert $\exists\mathbb{R}$-vollständig ist

Der in dieser Arbeit eingeführte Algorithmus zur Minimierung der *Edge Length Ratio* planarer Graphen auf Gittern arbeitet in zwei Schritten. Zuerst werden die bekannten Algorithmen zur Einbettung und zum Zeichnen von Graphen auf Gittern eingeführt. Auf das Ergebniss dieser Graphzeichnungen wird dann ein Algorithmus angewendet, der eine lokale Suche ausführt um die *Edge Length Ratio* der Zeichnung zu reduzieren. Dafür werden Maße definiert, die einer Graphenzeichnung einen Wert zuordnen welcher mit der *Edge Length Ratio* zusammenhängt.. Abschließend wird die Leistung des Algorithmus auf den Instanzen verschiedener Datensätze analysiert und die Parameter des Algorithmus optimiert.

Contents

1. Introduction

The goal of graph drawing is mapping vertices and edges of graphs to points and curves on a surface, usually the 2-dimensional plane. According to Battista et al. [Bat+98, p.11], three essential characteristics define graph drawings: drawing conventions, aesthetics, and constraints.

Drawing conventions give basic rules drawings have to satisfy to be admissible ([Bat+98, p.12]). One example is that each edge in the drawing must be a straight line. *Aesthetics* are values dependent on the specific drawing an algorithm tries to optimize. Often finding the optimum for a given aesthetic is computationally hard. As the name suggests, most aesthetics are associated with the graph drawing being more aesthetically pleasing to the human eye. An example of an aesthetic is the total edge length of a drawing. *Constraints* give rules for more localized subgraphs, for instance, placing a given subgraph near a point in the drawing space. More examples of common conventions, aesthetics, and constraints are given in Battista et al. [Bat+98, pp. 12-17].

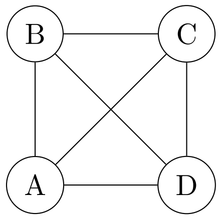
Problem

The target of the drawing algorithm proposed by this work is optimizing the planar edge length ratio of a graph on the grid. The conventions the algorithm adheres to are planarity and creating a grid drawing. A drawing is considered planar if no two edges cross each other. A graph admitting such a drawing is also called planar. For an example, compare Figures 1.1a and 1.1b.

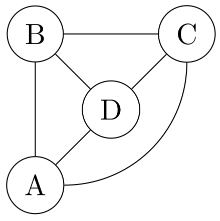
In a grid drawing, every vertex must have integer coordinates. Additionally, all the curves connecting two vertices in the drawing must be *polylines*. A *polyline* is a line consisting of several straight line segments. The bends of these polylines have to be on integer coordinates as well. An illustration for a polyline grid drawing is given in Figure 1.1c.

A drawing adhering to these conventions is also called a *planar polyline drawing*. The aesthetic to be optimized is the *edge length ratio (ELR)*, defined as the ratio between the Euclidean length of the longest edge and the shortest edge in a graph drawing.

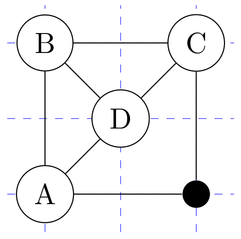
We will use a slightly different definition of this ratio, substituting the length of the shortest edge with the shortest Euclidean distance between any two connected vertices of the graph. This change is due to the altered definition of ELR in the graph drawing contest at the 30th International Symposium on Graph Drawing and Network Visualization [Gra22], in which the resulting algorithm shall participate. If we have to differentiate between classic ELR and the ELR as defined for the contest, we use ELR^{GD} for the contest definition.



(a) An embedding without restrictions.

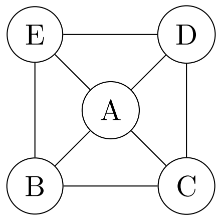


(b) A planar embedding.

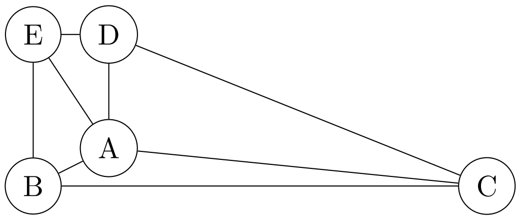


(c) A planar embedding on a grid.

Figure 1.1.: Three different embeddings of the complete graph K_4.



(a) An embedding with small ELR.



(b) An embedding with big ELR.

Figure 1.2.: Two embeddings of the same graph.

Motivation

An ELR close to 1 results in a drawing with equally spaced vertices in which densely connected subgraphs are close to each other. According to Bennett et al. [Ben+07, p. 59], these are two of the most important aesthetics when placing vertices. They also emphasize minimizing edge crossings as “the most agreed-upon edge placement heuristic” [Ben+07, p. 59] and mention uniform edge lengths as a desirable heuristic to have greater regularity in the resulting drawing. An example of how the ELR can change the readability of a graph is given in Figures 1.2a and 1.2b.

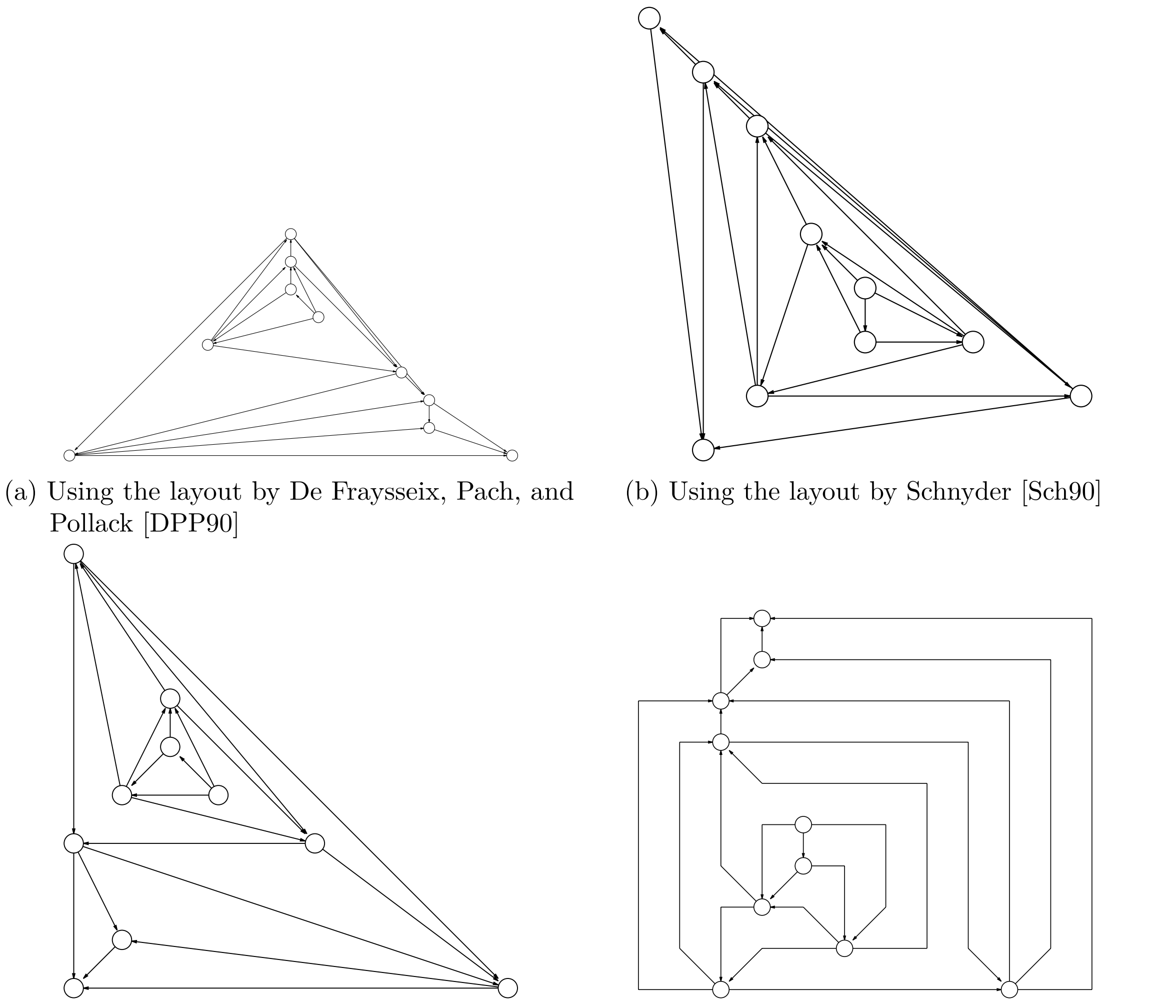
1.1. Related Work

The developed algorithm is located at the intersection of two research areas. On one side, we want to draw planar graphs on a grid, a well-researched area with several known algorithms. Conversely, we minimize ELR, a topic that only recently emerged. Most related work regarding ELR has been on bounds for the ELR of different graph classes.

1.1.1. Planar Grid Drawing Algorithms

Drawing planar graphs on a grid has already been subject to extensive research, with the first algorithm being published in 1988 by Fraysseix, Pach, and Pollack [FPP88]. It embeds any planar graph on a grid of size $(n-2) \times (2n-4)$. An example of a layout generated by their approach is given in Figure 1.3a. Shortly after, this bound was improved by Schnyder [Sch90] to a grid size of $(n-2) \times (n-2)$, an example is given in Figure 1.3b. This bound on drawings of *general* planar graphs has not been broken to date. For some subclasses of planar graphs, better bounds were found.

Soon after these results, Kant [Kan96] developed an algorithm generating convex drawings of triconnected planar graphs on a grid of size $(2n-4) \times (n-2)$. A graph drawsing is *convex* if every region bound by the edges in the plane is a convex polygon. A connected graph is considered *triconnected* if the deletion of any two vertices in the graph results in a graph that is still connected. Chrobak and Kant [CK97] then improved this algorithm and



(a) Using the layout by De Fraysseix, Pach, and Pollack [DPP90]

(b) Using the layout by Schnyder [Sch90]

(c) Using the layout by Chrobak and Kant [CK97]

(d) Using the layout by Gutwenger and Mutzel [GM98]

Figure 1.3.: Three embeddings of the contest graph `auto21_3.json` of last years graph drawing contest at GD2021 [Gra21], generated using the Open Graph Drawing Framework (Chimani et al. [Chi+13]).

reduced the grid area needed to $(n - 2) \times (n - 2)$. A layout generated by their improved approach is given in Figure 1.3c. All previous algorithms generate straight-line drawings, meaning all edges in the resulting drawings have no bends.

The *angular resolution* of a planar polyline drawing is the smallest angle formed by any two neighboring edges incident to the same vertex. According to Battista et al. [Bat+98], angular resolution is an essential aesthetic for a graph's readability. In Figure 1.2, the graph with high ELR also has a low angular resolution due to vertex C. An algorithm developed by Gutwenger and Mutzel [GM98] gives grid drawings with good angular resolution on a grid of size $(2n-5) \times (3/2n-7/2)$. Additionally, the drawings generated by their algorithm are almost orthogonal, meaning that every edge is either horizontal or vertical. The only exceptions are the polyline segments that are directly incident to vertices, as these have to be non-orthogonal for vertices of degree greater than four. A layout generated by their approach can be seen in Figure 1.3d.

1.1.2. Edge Length Ratio

Research regarding the edge length ratio of graph drawings has emerged relatively recently. The following sources relax the convention of the drawing being on a grid and only assume it is a straight-line drawing.

The first mention of ELR was by Lazard, Lenhart, and Liotta [LLL19], showing that all outerplanar graphs not drawn on a grid allow drawings with ELR less than or equal to two. A graph is considered *outerplanar* if there is a planar embedding in which all vertices are on the outer face. The trivial upper bound of $O(n)$ for the ELR of general planar graphs was consolidated into $\Theta(n)$ by Borrazzo and Frati [BF19] when it was shown that for some graphs, the best achievable ELR is in $\Omega(n)$. Blažej, Fiala, and Liotta [BFL20] showed that for 2-trees the ELR is within $\Omega(log(n))$ and $O(n^{0.695})$. The graph family of the *k-trees* is generated by starting with the complete Graph K_{k+1} and repeatedly adding a new vertex which forms a clique with k other vertices in the original graph.

A topic closely related to ELR is graph drawing with specified edge lengths. This was first mentioned by Eades and Wormald [EW90], who showed that drawing graphs with unit edge lengths is NP-hard. In 2016, Abel et al. [Abe+16] strengthened this result by showing that the problem is not only NP-hard but $\exists\mathbb{R}$-complete. This result also indirectly shows the $\exists\mathbb{R}$-completeness of the decision problem whether a given graph admits a drawing with ELR 1, as this would be a drawing with unit edge lenghts.

Theorem 1.1. *Deciding whether a given graph admits a drawing with an edge length ratio exactly one is $\exists\mathbb{R}$-complete.*

The computational complexity of the decision problem for ELR > 1 is still unknown.

Cabello, Demaine, and Rote [CDR04] refined the results by Eades and Wormald [EW90]. They show that deciding whether a planar 3-connected graph admits a drawing using specified edge lengths can be done in linear time when only the outer face is not a triangle. On the other hand, they also show that “Even for planar 3-connected graphs, deciding planar embeddability with unit edge lengths is NP-hard” (Cabello, Demaine, and Rote [CDR04, p. 2]).

Another approach similar to absolutely specified edge lengths is relative edge length specifications, as proposed by Aichholzer et al. [Aic+14]. They define a *relative edge length specification* as a partial order of the edge lengths of a given graph. A graph is considered *length universal* if it admits a planar drawing respecting any given relative edge length specification. They show that some graph families like outerplanar graphs, 2-trees, or the complete graph K_4 are not length universal. In opposition, they show that all bipartite graphs $K_{2,m}$ are length universal. The *bipartite graph $K_{2,m}$* is a graph where all vertices can be partitioned into two subsets of size two and m. Every vertex is only connected with all vertices of the other subset.

We start by introducing some definitions and preliminaries in Chapter 2.

In Chapter 3, we then take an in-depth look at the embedders and layout algorithms suitable for drawing planar graphs on the grid.

Chapter 4 introduces the main result of this work, the optimization algorithm. Additionally, we give insights on which optimizations we applied to reduce runtimes significantly.

Because Chapters 3 and 4 introduce many options and parameters, we run experiments to tune them and evaluate the performance of our approach. A summary and analysis of these experiments are given in Chapter 5.

In Chapter 6, we then give a conclusion on our work and some ideas for future work on optimizing ELR^{GD}.

2. Preliminaries

An undirected graph G is a tuple (V, E) of vertices with $E \subseteq V \times V$. Two vertices v_1 and v_2 in G are *adjacent* when $\{v_1, v_2\} \in E$. A graph is called *connected* if we can find a path between any two vertices in the graph and k-connected if we have to remove at least n vertices for the graph to become disconnected. 2-connected graphs are often called *biconnected*, 3-connected graphs *triconnected*. A particular subset of graphs is *planar* graphs. A graph is planar if it admits a drawing in the 2-dimensional plane without edge crossings.

We have to differentiate between graph embeddings and graph drawings in graph drawing. Both terms are closely connected and can easily be confounded.

Definition 2.1 (Combinatorial Embedding)**.** *A* combinatorial embedding *is given by the clockwise order of all incident edges for each vertex of a graph.*

Combinatorial embeddings do not give any information on specific placements of vertices or edges in the plane. For planar graphs, they do additionally specify how the edges separate the plane into different regions. Each of these connected regions of the plane is called a *face*.

If we add information about the placement of vertices and edges in the plane, we receive a *graph drawing*. In a graph drawing, all except one face of the graph are bounded regions in the plane. The single unbounded face is the region on the outside of the drawing, which we call the *outer face*. Most graph drawing algorithms take a specific embedding of a graph and the face, which shall be the outer face, and then calculate a graph drawing based on this information. A planar drawing of a graph always induces a planar embedding of the drawn graph.

We also need the so-called *planar triangulations* for the Schnyder-Layout in Section 3.2.2 and some of the experiments in Chapter 5. These are planar graphs in which adding any edge would result in a non-planar graph. Triangulations are the densest existing planar graphs.

To introduce the concept of graph depth used in Section 3.1, we have to define the *block-cutvertex-tree* (BC-Tree) and the *dual graph* of graph G. The BC-tree is a representation of a graphs *blocks* and *cut-vertices*. Each maximal biconnected subgraph of a graph is called a block, and cut-vertices separate blocks. In other words, cut-vertices are all vertices that would increase the number of connected components when removed.

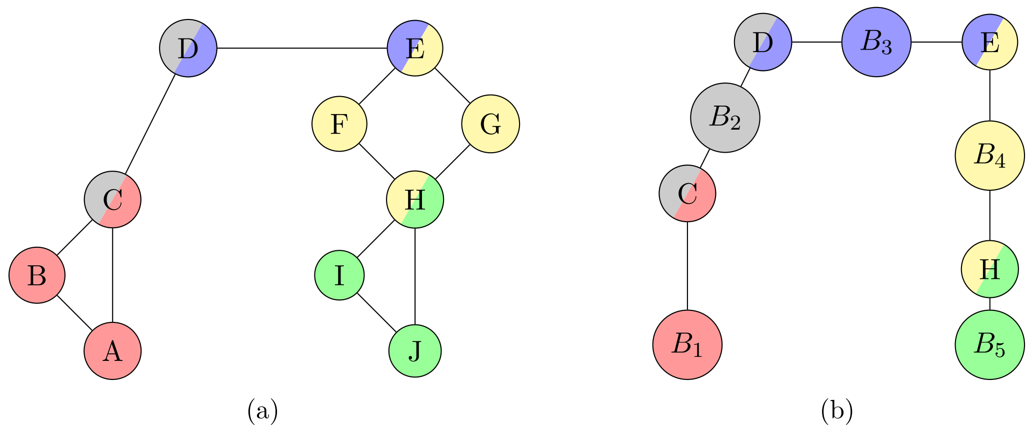


Figure 2.1.: In Figure 2.1a we give a graph with its blocks marked in different colors. The multicolored vertices are cut-vertices, as each cut-vertex belongs to several blocks. In Figure 2.1b we see the BC-tree belonging to this graph.

Definition 2.2 (BC-Tree). *The* BC-tree *of graph G is the graph in which every cut-vertex and block of G is a vertex. Two vertices in the BC-tree are adjacent if one corresponds to a cut-vertex v in G, one corresponds to a block B in G, and v lies within block B.*

Definition 2.3 (Dual Graph). *The dual graph belonging to a planar embedding of a graph G is the graph where each face of G is a vertex. Two dual vertices are adjacent if the corresponding faces share an edge.*

An example for a graph and its BC-tree is given in Figures 2.1a and 2.1b.

For the Schnyder-Layout, explained in Section 3.2.2, *barycentric coordinates* are necessary. Barycentric coordinates are used to specify coordinates relative to the corners of a triangle $\triangle_{ABC}$. They also work for the n-dimensional extension of the triangle, which is also called *simplex*. For planar graph drawing, the 2-dimensional case suffices.

Definition 2.4 (Barycentric Coordinates). *The barycentric coordinates of a point p relative to a triangle $\triangle_{ABC}$ are all tuples (v_a, v_b, v_c) with $p = v_A \cdot A + v_B \cdot B + v_C \cdot C$. Barycentric coordinates are not unique.*

One useful property of these coordinates is that for each point within the triangle $\triangle_{ABC}$, there is exactly one linear combination with $v_A + v_B + v_C = 1$. This additional restriction also makes barycentric coordinates become unique within $\triangle_{ABC}$. When we have positive barycentric coordinates with $v_A + v_B + v_C = 1$, we also know that the point specified by these coordinates must be within $\triangle_{ABC}$.

3. Initial Layouts and Preprocessing

In Section 1.1.1, four algorithms for drawing planar graphs on a grid were introduced. We use the implementations provided by the Open Graph Drawing Framework (OGDF, Chimani et al. [Chi+13]) for all algorithms. We did not consider other algorithms as we had to rely on a readily available implementation of the algorithms.

Before computing layouts, we must choose a planar embedding for the drawing algorithm to work on. A comparison of different embedders is given in Section 3.1. When we find a suitable embedding, we pass it to a layout algorithm, which then computes a layout of our graph embedding. The different algorithms and a comparison of them can be found in Section 3.2.

3.1. Planar Embedders

The initial planar embedding passed to the layout algorithm and optimizer is the first crucial choice for the algorithm. Because the layout algorithm does not change the embedding, and our algorithm only makes minor changes, finding a suitable initial embedding is vital to the success of our algorithm.

For our optimization approach, we prefer embeddings that maximize the number of vertices in the outer face of the graph. This is preferred because the furthest distances in a drawing usually have to be covered in the outer face, as it has to span around all inner faces. If we have more vertices in the outer face, the average length of the edges between these vertices can be smaller, giving the algorithm better chances at optimizing the ELR^{GD} of the drawing.

Additionally, more vertices in the outer face give the algorithm more possibilities for the shape of the outer face. If only three vertices are located in the outer face, it can only be a triangle, wasting valuable grid space.

We also expect embeddings with small *depth*, a concept introduced by Pizzonia and Tamassia [PT00], to work better in our algorithm.

Definition 3.1 (Graph Depth)**.** *The depth of a graph G is the depth of the BC-tree of the dual graph of G, rooted at the cut vertex belonging to the outer face. If there are no cutvertices in the outer face, we add a dummy vertex as the root.*

Table 3.1.: A summary on the different embedders implemented in the OGDF and the optimizations they try to make. If an embedder supports multiple optimizations it enforces them from left to right (e.g. `EmbedderMinDepthMaxFace` tries to find the min-depth embedding out of all embeddings with a maximal outer face).

Embedder	Maximize outer face	Minimize graph depth	Optimize layers
`SimpleEmbedder`	Yes	No	No
`EmbedderOptimalFlexDraw`	No	No	No
`EmbedderMinDepth`	No	Yes	No
`EmbedderMinDepthPiTa`	No	Yes	No
`EmbedderMaxFace`	Yes	No	No
`EmbedderMaxFaceLayers`	Yes	No	Yes
`EmbedderMinDepthMaxFace`	Yes	Yes	No
`EmbedderMinDepthMaxFaceLayers`	Yes	Yes	Yes

Those embeddings are preferred because moving a vertex in the deeper faces of the graph necessitates moving all vertices around. If no space is available, we have to move some vertices around it, which is more complex if the vertex is deeper in the graph with more vertices and edges surrounding it.

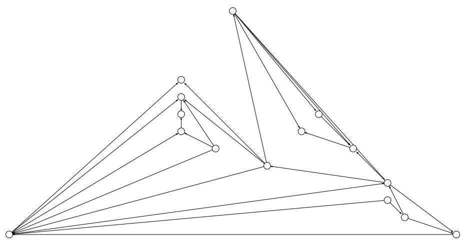
Most algorithms used to embed graphs internally use BC-trees of the dual graphs. A question raised by Kerkhof [Ker07] is how to embed blocks that do not have their embedding fixed by being embedded on the outside of the graph or the outside of inner blocks. For example, a block that is only directly connected to one cut-vertex can be embedded into any face adjacent to that cut-vertex. One variant he proposed is to put such free blocks in the face closest to the external face. This should also be beneficial for our goals, as more vertices near the outer face imply that the local search algorithm has to move even fewer vertices, as we pointed out when addressing graph depth. This approach is called “layer optimization.”

The OGDF implements several embedders, each providing some of the abovementioned features. They are summarized in Table 3.1. Notably, there are two embedders for only minimizing the graph depth. The `EmbedderMinDepthPiTa` is the one proposed by Pizzonia and Tamassia [PT00]. It takes a planar embedding of the biconnected components as an input and uses it to calculate a minimum depth embedding. The second `EmbedderMinDepth` was published by Gutwenger and Mutzel [GM03] and finds embeddings of any planar graph without needing external information. Examples for all embedders implemented in the OGDF are given in Figures 3.1a to 3.1g. For `EmbedderOptimalFlexDraw` we cannot give an example, as the version currently implemented in the OGDF does not run properly within our algorithm.

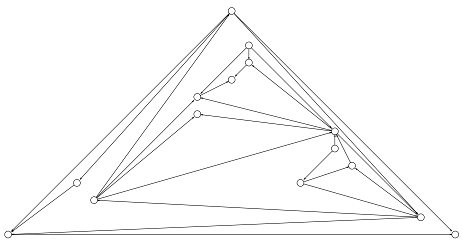
3.2. Drawing Algorithms

After calculating a suitable initial embedding, we pass it on to a drawing algorithm. We can compare these algorithms on several aspects.

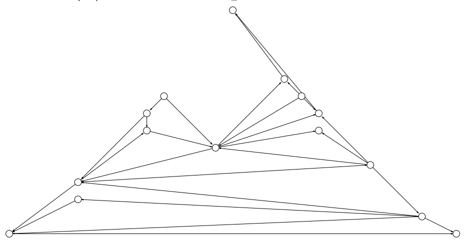
First of all, local search profits from an even vertex spacing. Space is necessary to move vertices around on the grid. A local search cannot operate properly if the graph is clustered, meaning that the vertices are partitioned into several sets that are closely placed to each other. Naturally, the grid already has a first spacing in unit lengths, but we also need empty grid spaces on which the local search algorithm can move vertices.



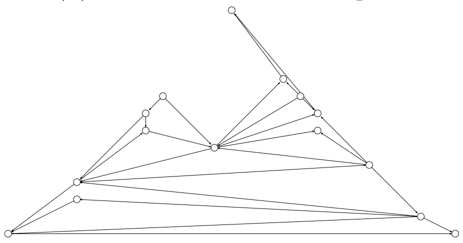
(a) Using `SimpleEmbedder`.



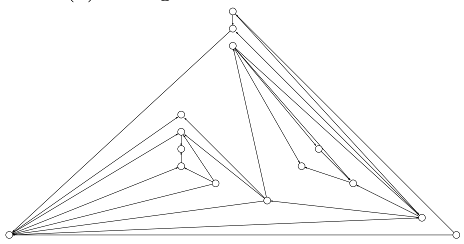
(b) Using `EmbedderMinDepth`.



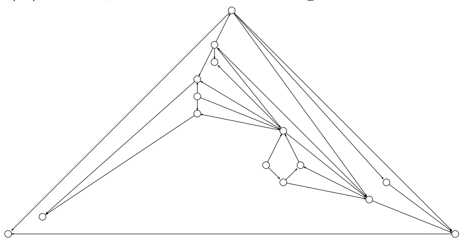
(c) Using `EmbedderMaxFace`.



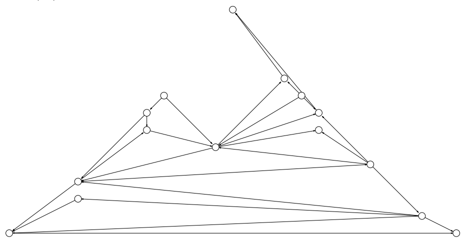
(d) Using `EmbedderMinDepthMaxFace`.



(e) Using `EmbedderMinDepthPiTa`.



(f) Using `EmbedderMaxFaceLayers`.



(g) Using `EmbedderMinDepthMaxFaceLayers`.

Figure 3.1.: Seven drawings of the contest graph `auto21_2.json` of last years graph drawing contest at GD2021 [Gra21], using different embedders for each drawing. For small graphs different embedders can generate the same embeddings, which has happened in Figures 3.1c and 3.1g. All drawings were generated using the Open Graph Drawing Framework (Chimani et al. [Chi+13]) and the drawing algorithm by De Fraysseix, Pach, and Pollack [DPP90].

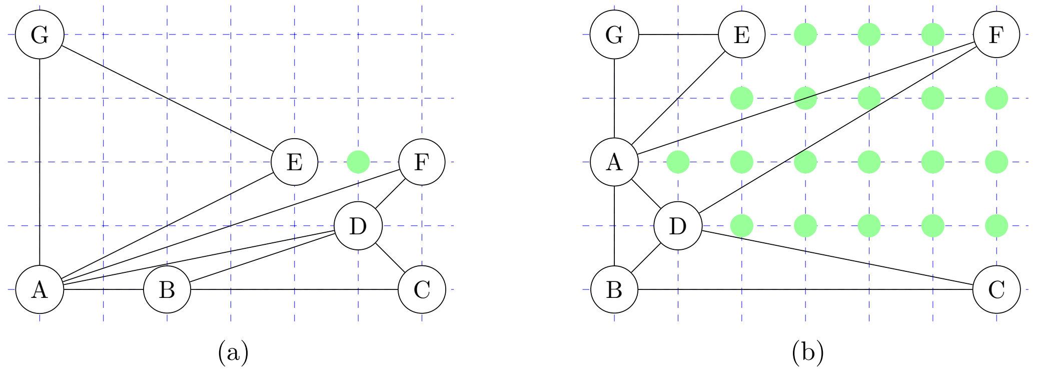


Figure 3.2.: Two drawings of the same graph. Figure 3.2a has bad angular resolution and distorted faces, resulting in few possible positions to move the vertex F without breaking planarity (marked by points). Figure 3.2b has better angular resolution and more evenly shaped faces, resulting in much better options to move vertex F.

Also helpful is a good angular resolution. It is important not only for the readability of the drawing but also for the algorithm. Small moves can quickly introduce crossings in a graph with bad angular resolution, due to the limited area available for the moved vertex. An example for this problem is given in Figure 3.2.

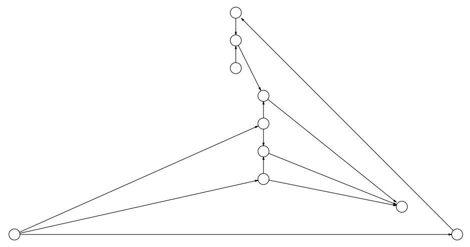
Another place where crossings can quickly occur is in non-convex faces. A drawn face is considered convex if its edges form a convex polygon. In a convex face, we can move a vertex to any grid point in the face without introducing any crossings between edges bordering the face. We can still have crossings with edges incident to the vertex that are not part of the face. However, eliminating the risk of having any crossings between edges of the face still results in more places to move the vertex to.

All algorithms specify the grid space they need in the worst case. Realistically, most algorithms can not use all of the available grid space by design because, they often draw the graphs with a triangular outer face. Generally, drawings using less grid space are preferable because, in small drawings, the longest edge cannot be overly long, resulting in a better ELR^{GD} compared to a large drawing.

3.2.1. De Fraysseix, Pach, and Pollack Layout (FPP-Layout)

The first layout presented in the introduction is the layout proposed by De Fraysseix, Pach, and Pollack [DPP90]. In the OGDF, it is known as `FPPLayout` and is based on the so-called *shift method.* It works by computing a *canonical ordering* of all vertices of a given graph embedding. Then it inserts the vertices on an empty plane respecting this order, shifting vertices left and right when necessary. Two examples of a small and medium-sized graph drawn using the FPP-Layout are given in Figures 3.3a and 3.3b.

In general, the layouts of this algorithm are clustered. Even in the simple examples in Figures 3.3a and 3.3b one can see that there are large spaces with almost no vertices and other spaces where several vertices are positioned close together. The shifting is the cause of this layout's problems with angular resolution. Parts of the graph get shifted to ensure that the outer face edges in the drawing always have a slope of 0 or ±1. Through this, new points are always placed on integer coordinates. Additionally, this is also used in the proof for the size of the drawing When such a shift is done, the edge that formerly was on the outer face can have a slope very close to ±1, forming a low angle with the newly introduced edge.


(a) Graph `auto21_1.json`.

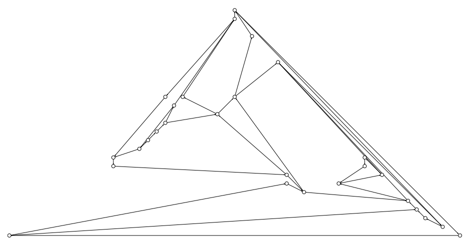

(b) Graph `auto21_5.json`.

Figure 3.3.: Two graphs layouted with the FPP-layout using `EmbedderMinDepthMaxFace-Layers` as implemented in OGDF (Chimani et al. [Chi+13].)

Because the outer face edges always have a slope of zero or ±1, the drawings generated by this algorithm are almost triangular and can only use at most half of the available rectangular space. During the experiments in Chapter 5, we also found that the FPP-Layout generally needs the largest grid out of all drawing algorithms we evaluated.

3.2.2. Schnyder-Layout

This layout is based on the paper by Schnyder [Sch90]. In the OGDF, it is known as `SchnyderLayout` and uses an approach called *realizer method.* The algorithm only works for triangulated graphs. Due to this we have to augment the graph to make it triangulated. *Augmenting* a graph means adding edges until it satisfies a certain condition.

Schnyder then calculates a so-called *realizer* of the inner edges of the graph. A realizer is a specific partition of the inner edges into three sets of oriented edges. Each set forms a tree rooted at one of the three vertices in the outer face of the graph. For each vertex, these trees imply a partitioning of the faces of the given embedding into three sets. The cardinality of these three sets then gives us barycentric coordinates with respect to the three points $A(2n-5,0,0)$, $B(0,2n-5,0)$, and $C(0,0,2n-5)$.

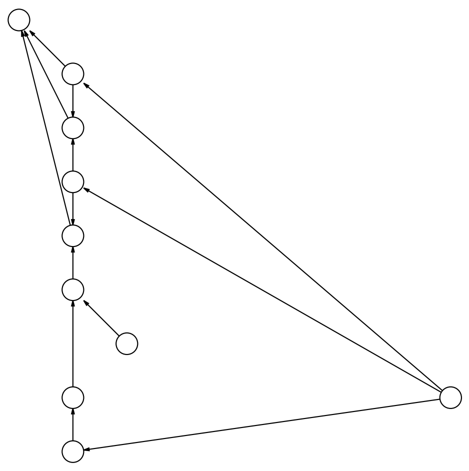
The approach described above only generates layouts with a maximum size $(2n-5)\times(2n-5)$. We can achieve a better bound of $(n-2)\times(n-2)$ by counting the number of vertices in each partition instead of the number of faces. We then subtract the depth of the path to vertex v in the corresponding tree and get barycentric coordinates respecting the points $A(n-2,0,0)$, $B(0,n-2,0)$, and $C(0,0,n-2)$. Two examples for graph drawings generated using this approach are given in Figures 3.4a and 3.4b.

The Schnyder-Layout has the same problems as the FPP-Layout regarding clustered vertices but fewer problems with angular resolution. Most shallow angles appear near the outer face of the drawing, where the longest edges are located. These edges often span almost a whole side and can form shallow angles when there is more than one of them.

The Schnyder-Layout also can use at most half of the space on the grid because all coordinates are given barycentric with respect to the triangle $\triangle_{ABC}$ with $A(n-2,0,0)$, $B(0,n-2,0)$, and $C(0,0,n-2)$.

3.2.3. Planar Straight Layout (PSL) and Planar Draw Layout (PDL)

The Planar Straight Layout, called `PlanarStraightLayout` in OGDF, is an implementation of the algorithm proposed by Kant [Kan96]. Similar to the algorithm by De Fraysseix, Pach, and Pollack [DPP90], he uses a canonical ordering but refines it to become a leftmost canonical ordering. Using this ordering, he then reinserts the vertices in a way that, for 3-connected graphs, ensures that all faces will be convex. For these reinsertion moves, he uses the shift method proposed by De Fraysseix, Pach, and Pollack [DPP90].


(a) Graph `auto21_1.json`.

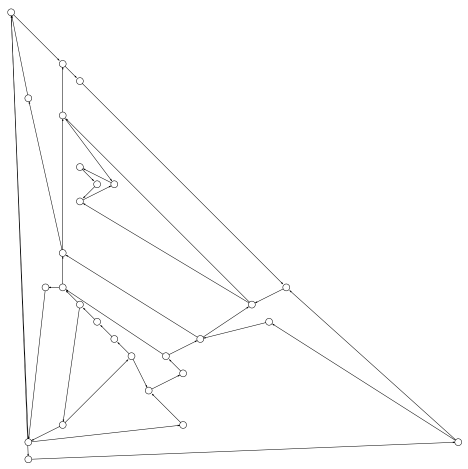
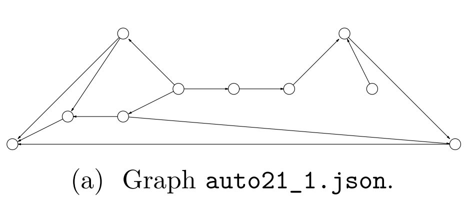

(b) Graph `auto21_5.json`.

Figure 3.4.: Two graphs drawn with the Schnyder-Layout using `SimpleEmbedder` and counting the vertices minus the depth as implemented in the OGDF (Chimani et al. [Chi+13].)


(a) Graph `auto21_1.json`.

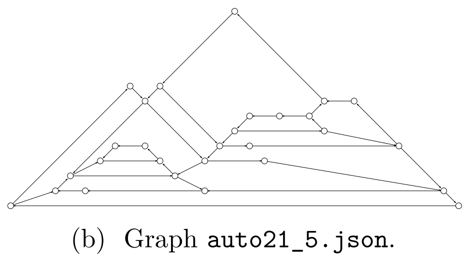
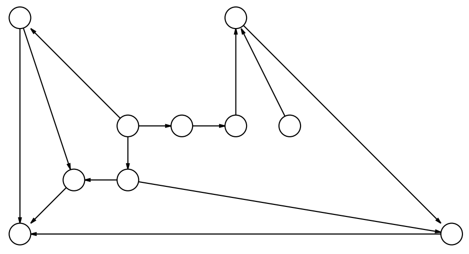

(b) Graph `auto21_5.json`.

Figure 3.5.: Two graphs drawn with the PS-Layout using `SimpleEmbedder` as implemented in the OGDF (Chimani et al. [Chi+13].)


(a) Graph `auto21_1.json`.

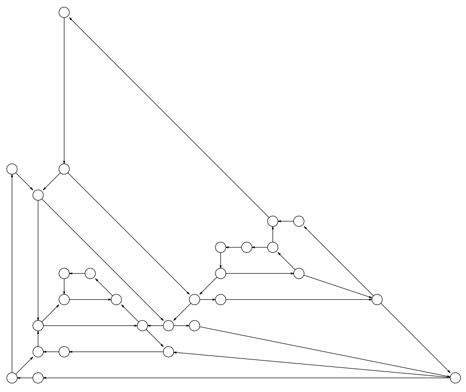

(b) Graph `auto21_5.json`.

Figure 3.6.: Two graphs drawn with the PD-Layout using `SimpleEmbedder` as implemented in the OGDF (Chimani et al. [Chi+13].)

The `PlanarDrawLayout` is the successor to PSL and was proposed by Chrobak and Kant [CK97]. It achieves a smaller drawing than the PSL by allowing to place edges at slopes different from zero or ± 1.

Examples for graph drawings using PSL and PDL can be seen in Figures 3.5a, 3.5b, 3.6a and 3.6b. By comparing them, one can easily see the similarity in the layouts. The difference is that the PDL leans the leftmost edges towards the left border of the drawing rectangle. We can verify this when looking at the maximal slope on the outer face of the drawings. While for PSL it is ± 1, the maximal slope for PDL is ∞.

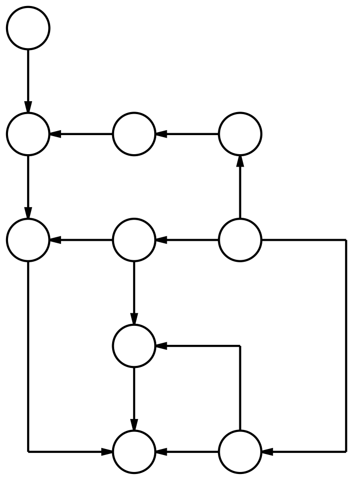
As mentioned above, both layouts guarantee 3-connected graphs to be drawn convex. For non-3-connected graphs, this does not hold. The faces can become concave when removing the edges added while augmenting the graph to be 3-connected. Still, most faces of the drawings generated by the PDL and PSL are nearly convex because, in many cases, the faces stay convex even after removing the augmenting edges.

Both layouts inherit drawbacks from using a shift approach very similar to the FPP-Layout. They both do not have good vertex spacing or good angular resolution and do not use part of the available space. Regarding space usage, the PDL is slightly better, as the left-leaning nature of this layout makes better use of the upper left corner of the grid. This can be seen when comparing Figures 3.5a and 3.6a. We also confirmed this in Chapter 5, where the PDL uses the least space out of all straight-line drawing algorithms.

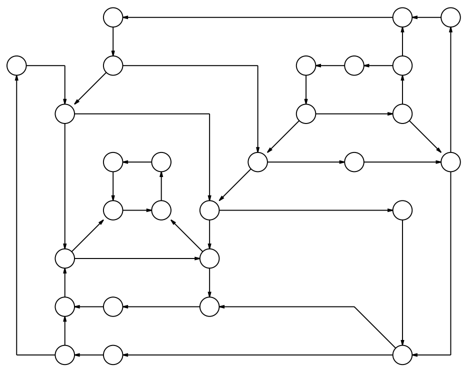
3.2.4. Mixed Model Layout (MML)

This layout, known as `MixedModelLayout` in the OGDF, was presented by Gutwenger and Mutzel [GM98]. The main difference is that it layouts the graph in 2 phases and uses bends to increase angular resolution. The first phase computes a bounding-box for each vertex depending on how many edges are incident to that vertex. In this phase, it tries to leave enough space so that the edges can be evenly distributed around the vertex, resulting in good angular resolution. It then layouts these bounding boxes by computing a canonical ordering of the graph's vertices and then calculating the coordinates when reinserting. Some examples for layouts generated by this algorithm are given in Figures 3.7a and 3.7b.

The layouts generated by MML have a low angular resolution by design. They are also almost rectangular and therefore do not waste any space on the rectangular grid. One problem with the layout is that the grid it needs in the worst case is bigger than for some of the other layouts, most notably the PDL and Schnyder-Layouts. Both layouts only use a grid of size $(n-1) \times (n-1)$, while the MML needs a grid of size $(2n-6) \times (3/2n-7/2)$.



(a) Graph `auto21_1.json`.



(b) Graph `auto21_5.json`.

Figure 3.7.: Two graphs drawn with the MM-Layout using `SimpleEmbedder` as implemented in the OGDF (Chimani et al. [Chi+13].)

However, in the experiments in Chapter 5, we could not confirm this worst case. The MML consistently produced the drawings needing the least grid space, often by a considerable margin.

4. Local Search

After calculating a suitable base layout, as shown in Chapter 3, we want to optimize layout's edge length ratio (ELR^{GD}) by performing a local search to find optima regarding certain quality measures. Each measure function takes a graph drawing as an input and returns a numerical value measuring how "good" the drawing is regarding this measure.

For this chapter, we consider a *move* to be a translation of a particular vertex in one of the four directions possible on the grid. Moves are the atomic component of this algorithm. Each iteration of the algorithm ends with a move.

The algorithm's input is a list of sets of quality measures M, an initial drawing of the given graph, and a list of iterations I specifying the number of iterations per run. We run the algorithm given in the next paragraph for each set M' of measures in M and each iteration count I' in I. The result of the first sub-run is used as an input for the second sub-run and so forth. The output of the algorithm is the result of the last sub-run.

Each sub-run of the algorithm receives one of the sets of measures M', a drawing, and a number of iterations I' as its input. An iteration starts by calculating the value of the measure functions for each possible move. This results in a list of lists of measures L, containing one sublist for the measures of each possible move. We then compare all these sublists using one of the comparators described in Figure 4.2 to find the minimal list L' and do the move associated with L'. This process is iterated until the number of iterations reaches the desired number I' specified in the input.

This general algorithm is summarized in Algorithm 4.1.

The measure functions implemented by us are described in detail in Section 4.1. In Section 4.2, we explain how we can combine all measures to decide which move to do. The unoptimized version of Algorithm 4.1 is slow, especially because of the number of intersection tests necessary to filter out moves that would violate planarity. To overcome this and other deficiencies, we employ several optimizations to the algorithm. The optimizations are explained in Section 4.4. We then present the finalized algorithm and show some examples for runs of the algorithm in Section 4.5

4.1. Quality Measures

4.1.1. Planarity

The first and most important "measure" is the planarity of the drawing after a move. As introduced in Chapter 1, this is not a real measure but a drawing convention. Consequen-

Algorithm 4.1: The basic optimizer algorithm

```
Data: A drawing of graph $G$
Data: A list of sets of measure-functions $M$
Data: A list of iteration counts $I$
Result: A drawing of graph $G$ with better ELR
for each set of measure-functions $M'$ in $M$ and each iteration count $I'$ in $I$ do
    $It \leftarrow 0$;
    while $It < I'$ do
        $L \leftarrow$ new list of list of doubles;
        for vertex $v$ in $G$ do
            for direction $d$ do
                $L' \leftarrow$ new list of doubles;
                for measure-function $m$ in $M'$ do
                    D.push_back($m(v, d)$);
                end
                L.push_back($L'$);
            end
        end
        find minimum $L'$ in $L$ according to quality measures;
        move in vertex and direction associated with $L'$;
        $It \leftarrow It + 1$;
    end
end
```

tially planarity can only have two states, either the graph is planar, or it is not. We *always* have to adhere to this drawing convention and have to ignore any move resulting in a non-planar drawing when deciding whether to move a vertex.

To decide if a graph drawing is planar, we have to do an intersection test between all edges in the drawing. The naive approach, testing all the edges pairwise against each other, has a runtime in $O(n^2)$. Testing each possible move then results in a runtime in $O(n^3)$, which is not feasible. The approaches taken to reduce the time needed for intersection tests are given in Section 4.4.

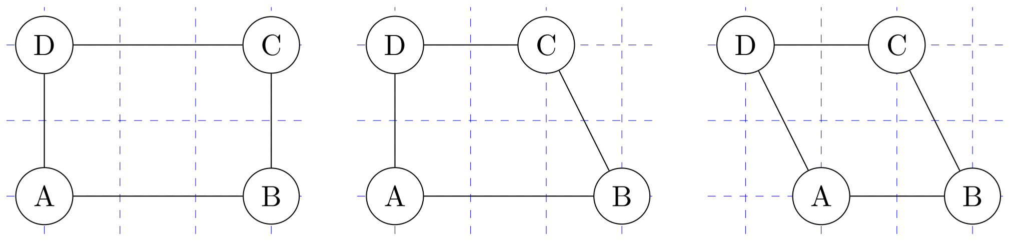
4.1.2. Edge Length Ratio (ELR^{GD} and ELR)

An obvious measure of how good a drawing is regarding its ELR^{GD} is ELR^{GD} itself. Only using ELR^{GD} to decide on the next move has one disadvantage: In most cases, it is not enough to move a single edge to improve it. An example of a graph that admits a drawing with ELR^{GD} 1 but would need two vertices to be moved is given in Figure 4.1a. The following lemma gives a short explanation of where this problem originates from.

Lemma 4.1. *If the ELR can be improved by moving a vertex, all longest or all shortest edges are incident to this vertex.*

Proof. If the ELR can be improved, all longest edges must be shortened, or all shortest edges must be lengthened. Both can only be done if all these longest or shortest edges are incident to a single vertex. Otherwise more vertices would have to be moved to achieve a decrease in ELR.

This proof also holds for ELR^{GD}, when the shortest edge is substituted by the smallest Euclidean distance between any two connected vertices in the graph. □



(a) $d_{mm} = 4$, ELR $= 3/2$ (b) $d_{mm} = 2$, ELR $= 3/2$ (c) $d_{mm} = 4$, ELR $= \sqrt{5}/2$

Figure 4.1.: A graph with no possibility for direct improvement of the ELR by moving any vertex is given in Figure 4.1a. How a reduction in d_{mm} consecutively helps with reducing the ELR is then shown in Figures 4.1b and 4.1c.

To counteract this problem of ELR^{GD} it usually is not used as the only measure but paired with other measures in the ways described in Section 4.2. This helps overcome this behavior.

Calculating ELR^{GD} for a drawing needs linear time. First, we have to calculate the length of all edges and the Euclidean distance of all connected vertices, which is easily done by two passes over all edges. Afterward, finding the minimum and maximum of these values can be done by one iteration over the lengths.

We can also use the definition of edge length ratio used for the graph drawing contest at [Gra21]. The lemma, drawbacks, and runtime estimations of ELR^{GD} also hold for ELR. The advantage of ELR is that we can introduce unnecessary bends to short edges, artificially lengthening them. This results in lower ELR when successful. The decision on where to add such bends becomes very complex and goes beyond the scope of this thesis, as it aims to optimize ELR^{GD}.

4.1.3. Min/Max Count

In Lemma 4.1, we saw that only optimizing for ELR^{GD} can have problems finding an optimum if more than one longest or shortest edge exists.

Given a graph $G = (V, E)$ with a drawing, we define *max* as the number of longest edges and *min* as the number of shortest edges. We can then set $d_{mm} = max + min$ as the Min/Max count of the drawing. Reducing the Min/Max count of a drawing potentially shortens long edges and lengthens short edges. For example, in Figure 4.1a, moving any single one of the vertices inwards would result in a reduction of d_{mm}, which is then given in Figure 4.1b. The obvious move is then done in Figure 4.1c, reducing ELR^{GD}.

One problem to consider when using d_{mm} is that it incentivizes creating a new longest or shortest edge. Creating a new longest or shortest edge sets the count of longest or shortest edges to one but will result in a worse ELR^{GD}. In most cases we can avoid this behavior by pairing Min/Max count with another metric that gives a penalty on very long or short edges. One example would be ELR^{GD}.

Calculating Min/Max count can easily be done in linear time. We first have to find the longest and shortest edge lengths. In a second pass over the edge lengths, we count the occurrences of the longest and shortest edge lengths to receive the Min/Max count of the drawing.

4.1.4. Average Edge Length Difference (AELD)

For the edges of a graph with average edge length e_{avg} we define the average edge length distance (AELD) as:

$$\sum_{\forall e \in E} |length(e) - e_{avg}| \tag{4.1}$$

If all edge lengths were to be exactly the average, we would have a drawing with ELR^{GD} 1 and AELD 0.

The advantage of using AELD over ELR^{GD} is that it does not depend on having at most one longest or shortest edge to find an improvement. Most possible moves will result in an increase or decrease of AELD. Only if the resulting edge lengths are the same or a permutation of the original edge lengths does the measure not change. This is a desired property for a measure used for a local search algorithm because it avoids plateaus in the measures that otherwise obstruct the successful search for the next step.

A drawback of this approach is that it also incentivizes shortening long edges. This shrinks the available space for moves and is something we want to avoid at the beginning of the optimization process. For countering this, the algorithm is given a preference for lengthening short edges by taking the n-th power of the differences to the average edge length for all edges shorter. Through this change, lengthening a short edge becomes more beneficial the shorter the edge.

Another possibility to influence the performance is adding an offset o to the calculated average edge length. When we then sum the differences of the edge lengths to this offset average edge length, the penalty then changes in favor of edges that are o units longer than the average length. Simply put, changing o changes this measure's "target" edge length. This is desired for the same reason as introducing an exponent.

With exponent n and offset o, we can extend Equation (4.1) to:

$$\sum_{\forall e \in E} ((|length(e) - e_{avg}|) + o)^n \tag{4.2}$$

The optimal choice of n and o is complicated because both parameters influence each other and depend on the input graph. The optimization of the parameters is done in Section 5.3.2.

Calculating the AELD of a graph drawing can easily be done in linear time. We have to do one pass over all edges to calculate the average edge length and one pass to calculate and sum the differences.

4.1.5. Total Edge Length

In Section 4.1.4, we mentioned that, in general, it is advantageous to have more grid space to draw our graph. Sometimes this can lead to drawings getting too large during the optimization process. Reducing the total edge length helps with this problem by shrinking the drawing.

One problem with this measure is that it can negate the progress done by previous measures. One simple solution is always pairing the measure with another measure like ELR^{GD} to incentivize the algorithm to keep the ELR^{GD} high.

4.1.6. Random Measure

An interesting measure for experimental purposes is the random measure, which ignores its input drawing and returns a randomly generated number. Its primary purpose is to randomize the local search result and to provide a baseline measure for comparison. One possible usage is to run the algorithm evolutionarily.

This would be possible by starting many optimizer runs, all using the random measure in their measure set. Due to the randomness added to the algorithm by the random measure, some of the runs might diverge into different local optima. We can then choose the best result of these runs and use it as the input for another round of our local search, repeating this pattern until arriving at an acceptable result.

We can also use the randomizing aspect of this measure for optimizing small graphs. If ELR^{GD} arrives at a local optimum, adding a random measure to the measure set can help the algorithm to find a way out of the local optimum toward an even better optimum. An evaluation of this idea is given in Section 5.3.4.

4.2. Combining the Measures

Each iteration ends with a list of measures for each vertex and direction. We then must compare these measure lists to select the best possible move. For this, two methods to compare lists with each other are introduced.

The first comparator is the lexicographical ordering. It takes a list of lists of values, numerical values in our case, and compares all sublists by comparing their first element. If there is a tie on the first element, it compares all sublists that tie each other on their second element. It continues to do so until it either runs out of elements to compare, resulting in a non-strict ordering or until it achieves a strict ordering of the lists.

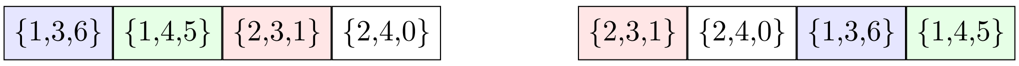
This comparator is very simple and is the default comparator implemented for `std::vector` and `std::list` in C++. In the comparison process, it never compares a value from one measure to a value from another measure. Due to this, we do not have to scale the measures to become comparable. For example, the AELD will usually be much higher than a graph's Min/Max Count. If we compared both directly, we would need to norm both measures.

Another comparator is a weighted ordering. It takes a list of lists of values, which we will call L, and a list of weights, called W. It then calculates the linear combination of L and W to reduce L to a list of numerical values, which we can easily order numerically.

This approach is more flexible than the lexicographical ordering. Suppose bounds on the highest possible values for the different measures are already known. In that case we can normalize them to a range within $[0, 1]$ and then set the weights to $10^0, 10^1, \ldots, 10^{n-1}$ to emulate a lexicographical ordering.

The main problem of the weighted ordering is that we have to find sensible values for the weights, resulting in more parameters to tune. Also, as already mentioned, not all measures are easily comparable. To ensure comparability, the weights must be normalized into a range (a standard range would be $[0, 1]$).

An example of how both combinators behave with specific inputs L and W is given in Figure 4.2. For our experimental evaluation and the participation in the graph drawing contest at [Gra22] we chose not to use the weighted ordering, as choosing good weights is a rather difficult process. Experiments on different measure sets using the lexicographical ordering can be found in Section 5.3.3.



(a) Lexicographical ordering (b) Weighted ordering

Figure 4.2.: An example on how the lexicographical (Figure 4.2a) and weighted (Figure 4.2b) ordering would order the list $L = (\{1, 3, 6\}, \{1, 4, 5\}, \{2, 3, 1\}, \{2, 4, 0\})$ with weights $W = (2, 3, 1)$.

4.3. Iterations

The last part of the algorithm is to decide when to stop iterating. An intuitive break condition would be to detect when we arrive at a specific layout for a second time. As long as we do not use the random measure, we found a deterministic circle in the layouts visited by the algorithm. The problem with this approach is that it requires comparing our current layout against all previous layouts. Each layout comparison would take at least linear time. This would result in an increasingly long runtime overhead for introducing a break condition.

An additional problem arises when we add the random measure to our measure set. In this case, there cannot be a deterministic cycle within the algorithm execution.

Due to these problems, we decided that the number of iterations should be a hyperparameter set by the user. Section 5.3.5 gives examples of the runtimes needed for convergence for different graph sizes.

4.4. Optimizations

When analyzing Algorithm 4.1 we can see that each algorithm iteration is rather complex. In every iteration, we calculate all measures for all vertices and all four directions. With a naive intersection test, this has a runtime within $O(n^2)$ for each vertex and direction. Everything combined, we then have a runtime in $4 \cdot O(n) \cdot O(n^2) = 4 \cdot O(n^3)$ just for the recalculation of all measures in one iteration. Especially for large graphs, this can lead to problems, as each iteration has to be as fast as possible. Otherwise, optimizing large graphs would not be feasible.

4.4.1. Intersector

To decide if a vertex can be moved in a direction, we have to perform an intersection test to ensure that the drawing does not violate the planarity convention. The first naive approach tests all line segments pairwise for intersections, resulting in $O(n^2)$ runtime.

The intersection test implemented in the OGDF is an upgrade to this approach. It tests all edges against each other using the sweep-line algorithm proposed by Bentley and Ottmann [BO79]. For calculating all crossings of n edges, it has a runtime in $O((n + k)log(n))$, with k being the number of crossings found. Because a planar graph has an average degree smaller than six, moving one vertex will not introduce many new crossings, keeping k small.

An obvious approach to reducing the time needed to check the graph for crossings is reducing the number of edges we have to check. In each step we, only move one vertex. Therefore only the edges incident to this vertex can move. We can then improve the naive approach by ignoring crossing tests between any two edges that were not incident to the moved vertex. Since planar graphs have an average vertex degree smaller than six, the expected average number of edges moved by a single vertex is small for most cases. In Figure 4.3, we give an example of which edges have to be intersected with all other edges to ensure planarity after moving a single vertex. One can easily imagine that this approach

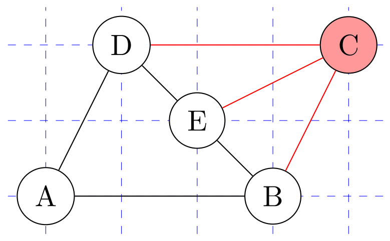


Figure 4.3.: An illustration which vertices and edges get checked against each other in our improved intersection test.

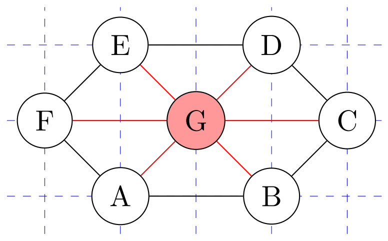


Figure 4.4.: The 7-wheel, a canonical example out of a graph family that behaves badly with our custom intersection approach.

cuts off many unnecessary intersection tests, especially if the graph becomes large and the changes to the graph become very localized.

For some graph families and vertices, this approach does not work. The most basic example of graphs with this behavior are wheel graphs. An example is given in Figure 4.4. When moving the central hub in a wheel graph, calculating intersections with our custom approach has a runtime in $O(n^2)$ because $n/2$ edges are incident to this vertex.

When analyzing this approach we can also see that the runtime to recalculate planarity for all moves is within $O(n^2)$. This is due to each edge being incident to two vertices, and each of these vertices being tested for four directions.

We could further speed up the intersection tests between edges by using spatial data structures like *quadtrees*. Such a data structure sorts the edges in a tree of rectangular bounding boxes. We can quickly rule out edges that are not relevant by first testing our moved edges against these bounding boxes. Then we do a regular intersection test between all remaining edges and the edge against which we want to check. The advantage of this approach grows with the size of the graph because quadtrees help us to easily rule out irrelevant edges with low overhead. Because the planarity test introduced above is already quick enough for the graphs we encountered during our test runs for the graph drawing contest, we did not implement quadtrees for edge storage.

We also have to check whether the moved vertex overlaps with any other vertex. The naive approach implemented by the OGDF would test all vertices against each other. In our case, this is unnecessary. We only have to check the position of the moved vertex against all other vertices because we only move one vertex at a time.

Another point that can be optimized is the intersection test itself. We are not interested in the intersection point of the edges. We only want to know whether two segments intersect or not. The algorithm used is given in Algorithm 4.2 and was inspired by [Ie22]. We added an early-out in case the bounding boxes of the segments do not overlap. This early-out alone makes most intersection tests a simple comparison of 4 integral values.

In case the early-out gives an overlap of the bounding boxes, we have to continue with the regular intersection test. To decide whether the two segments are collinear, it needs two cross-products. The cross product used for the 2D points is realized by setting the z-value to zero. The return value then becomes a scalar, which determines how the two segments are oriented relative to each other.

If the segments are not collinear, we have to check whether the endpoints a and b of the first line segment are on different sides of line segment $\overline{cd}$ and inversely. For each of these tests, we need two additional cross-products.

Algorithm 4.2: The intersection algorithm inspired by [Ie22].

```
Data: Four points $a$, $b$, $c$, $d$ on the integer grid
Result: True if the line segments $\overline{ab}$ and $\overline{cd}$ intersect, False otherwise
1 if bounding boxes of $\overline{ab}$ and $\overline{cd}$ do not overlap then
2     return false;
3 end
4 if $\overline{ab}$ and $\overline{cd}$ are collinear then
5     return true;
6 end
  // Test if $a$ and $b$ are on the same side of $\overline{cd}$ and if $c$ and $d$ are on
     the same side of $\overline{ab}$
7 $ab \leftarrow sgn(a.cross(b, c)) \neq sgn(a.cross(b, d))$;
8 $cd \leftarrow sgn(c.cross(d, a)) \neq sgn(c.cross(d, b))$;
9 return $ab \wedge cd$;
```

This test also operates *only on integers*, making it numerically stable and more reliable, especially for large coordinates. To calculate the intersection point, we additionally need to calculate several determinants and norm our line segments. This makes it necessary to use floating point arithmetics, resulting in a loss of accuracy.

4.4.2. Lazy Recalculation

One iteration of the main loop of Algorithm 4.1 only does one move, having very local effect on the graph. Most notably the planarity condition does not change for most possible moves.

We can exploit this by calculating the measures and planarity condition for all moves lazily. Before the first iteration of the algorithm, the planarity and measures are calculated for all possible moves and the results are saved in a data structure for later use.

For each iteration, the minimal move regarding the chosen comparator from this data structure is extracted. Then the planarity and all measures for the chosen move are recalculated. If the move is still viable (i.e., it respects planarity), it is done. If the best move does violate planarity, it is ignored. In both cases, the move does not get reinserted into the data structure.

Before making the move we also check whether the measures after the move are strictly worse than the current measures. If this is the case, the measures in the data structure are assumed to be outdated, and all information is deleted and recalculated.

We have to choose this approach because, especially for AELD, the measures can quickly become outdated. We catch this problem by fully recalculating when the metrics of the current optimal move can not compete with the metrics of the current drawing. The approach is also summarized from Line 8 to Line 23 in Algorithm 4.3.

4.4.3. Data Structures for Measures

When using lazy recalculation as presented in Section 4.4.2, we can also speed up the iterations by using a data structure that enables us to repeatedly extract the minimal element faster than in linear time. The natural choice for this task is a heap. For our implementation we use the C++ Standard Template Library (STL) functions `std::make_-heap`, `std::pop_heap` and `std::push_heap`, which give us heap access to most C++ STL container classes.

Building a heap of n elements needs $O(n)$ time. Removing the minimal element from a heap or adding a new element has a runtime in $O(logn)$. This implies that after just some extractions, we achieve a runtime advantage over repeatedly scanning the list for the minimal element. Especially for large graphs, this outweighs the performance overhead. Depending on the size of the graph, several hundred lazy moves are possible. Further experiments on this are available in Section 5.4.

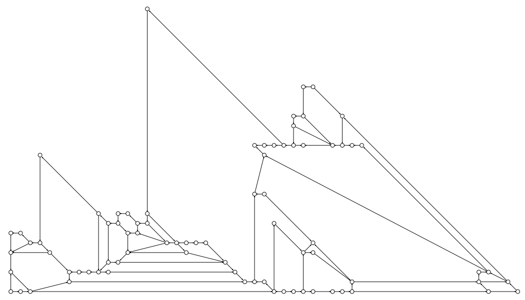
4.4.4. Edge Length Caching

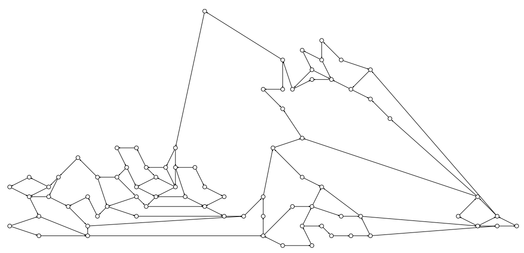
For most of the measures, the edge lengths of the drawing need to be calculated. Because the algorithm only moves one vertex at a time, we can save many unnecessary computations by caching the edge lengths and only recalculating the lengths for the moved edges. Even though this introduces some overhead for managing the edge length cache, it still greatly improves runtime.

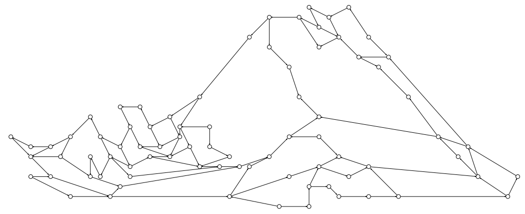
This is especially useful if the drawing the algorithm is working on has bends. For simplicities sake, we handle the bends of a drawing as *dummy* vertices. This enables us to move them using the same code as the vertices. It also introduces chains of dummy vertices, each chain representing one former edge with bends. To calculate the length of these edges, we have to walk along the chain to collect the edge length of the original edge. By edge length caching, this walk along the chain can be ignored. We only have to do it again when we update the cache.

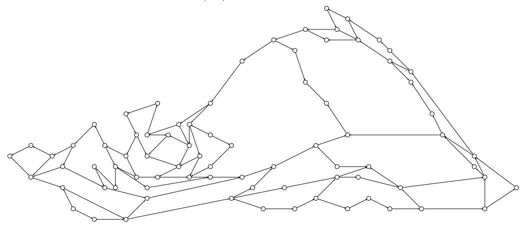
4.5. The final algorithm

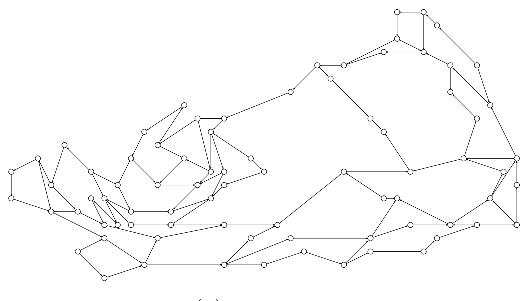
Finally, we can combine all optimizations and improvements for Algorithm 4.1 and get an optimized version of our algorithm in Algorithm 4.3. A sequence of images showing how the algorithm progresses on a graph small enough to be visualized properly is given in Figure 4.5. The experiments in Chapter 5 were done using a C++ implementation of this algorithm.

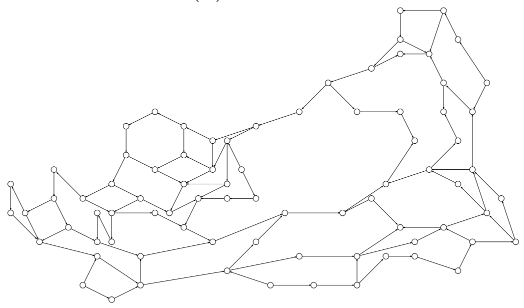

(a) Initial Layout

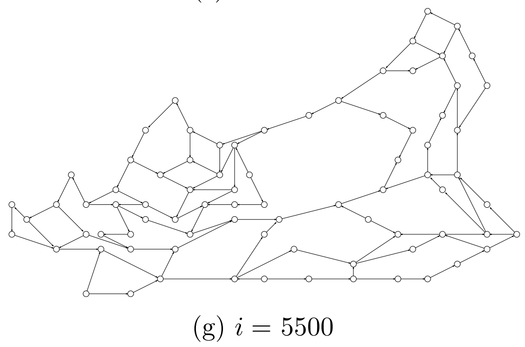

(b) $i = 300$

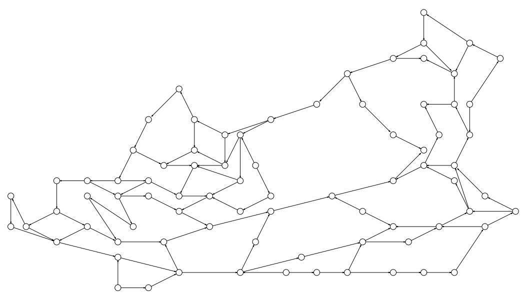

(c) $i = 600$

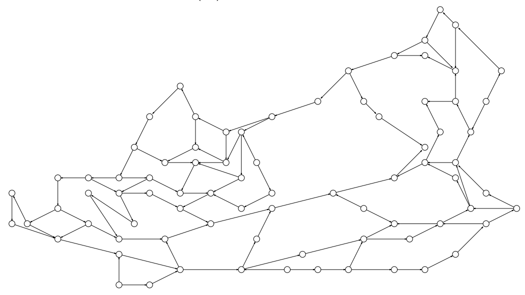

(d) $i = 1000$


(e) $i = 2500$


(f) $i = 4000$


(g) $i = 5500$


(h) $i = 7000$


(i) $i = 15\,000$

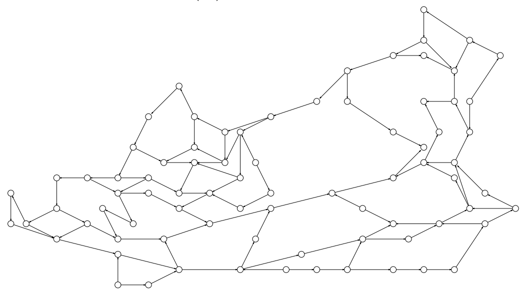

(j) $i = 20\,000$

Figure 4.5.: A sequence of images showing the algorithm running on the input file `auto21_-6`. The drawing uses the PD-Layout and `EmbedderMinDepthMaxFaceLayers`. The optimizer set is $\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}$ with exponent 15, offset 5, and lazy evaluation enabled. The number of iterations is given by i.

Algorithm 4.3: The optimized algorithm

```
Data: A measure heap $H$
Data: A drawing of graph $G$
Data: A list of sets of measure-functions $M$
Data: A list of iteration counts $I$
Result: A drawing of graph $G$ with better ELR
initialize edge length cache;
for each set of measure-functions $M'$ in $M$ do
    for each direction $d$ and vertex $v \in V$ do
        for each measure $m$ in $M'$ do
            $H$.insert(m(v,d));
        end
    end
    while specified number of iterations not met do
        $v, d \leftarrow H$.pop_heap() ;
        recalculate planarity for $v, d$;
        if $v, d$ is still planar then
            move vertex $v$ in direction $d$;
            update edge length cache;
        end
        if the minimum in $H$ ise worse than the measures of the current drawing
         then
            H.clear();
            for each direction $d$ and vertex $v \in V$ do
                for each measure $m$ in $M'$ do
                    $H$.insert(m(v,d));
                end
            end
        end
    end
end
```

5. Experiments

Using the embedders, layouts, quality measures, and algorithm given in Chapters 3 and 4 we perform several experiments to evaluate the performance of the different approaches and tune the parameters of the algorithm.

In Section 5.1, we give an overview of the data sets we used and summarize the runtime environment of our experimental setup. In Section 5.2, the effects of different embedders and layouters on the initial ELR^{GD} and the optimizer are evaluated. The most important part of this chapter is Section 5.3. In this section, we compare the performance of different combinations of quality measures for the local search algorithm. We also evaluate multi-run approaches, in which the output of one run of the local search is used as an input for another run. In Section 5.4, the runtime of the algorithm and the runtime improvements possible by using lazy recalculation are analyzed.

5.1. Methodology

Runtime Environment

We implemented the algorithms in C++ and compiled the binaries using GCC 11.3 with all optimizations enabled. The experiments were conducted on compute nodes using quad AMD Opteron 6172 CPUs with 48 cores at 2.10 GHz. The nodes have 256GB of ECC DDR3 RAM clocked at 1.33 GHz.

The algorithm implementation is available on GitLab. The specific algorithm version the experiments were conducted with is Git commit `7147fcfa`.

Testing Datasets

We tested our approach on several planar graph datasets.

First, we have the challenge graphs of the graph drawing challenge at [Gra21]. We hope these graphs are comparable to the challenge graphs of the graph drawing challenge at [Gra22]. We can also use the results of last year's participants as a baseline to compare our solutions against. One key difference is that last year's challenge optimized ELR, while the target of our algorithm is optimizing ELR^{GD}, as defined for this year's graph drawing contest. Also, on the graphs reserved for the "automatic" category, only three teams could compute results on two of the seven provided graphs.

Table 5.1.: A summary on the different datasets we tested our algorithm on with some embedding- and layout-independent statistics.

Dataset	Graph Count	Vertex Count			Edge Count			Max Degree		
		min	avg	max	min	avg	max	min	avg	max
Rome	3281	10	25.87	105	9	30.18	103	2	5.42	10
North	854	10	28.96	100	9	32.69	120	2	9.02	71
GD21’	17	4	459.77	4135	5	730.24	5781	3	5.65	15
GD21	16	4	230.06	1525	5	424.56	3152	3	5.75	15
PlanTri	2788	4	36.61	64	6	103.85	249	3	17.95	29

For our evaluations, we removed the graph `auto21_10` from this dataset. Even though the performance on this graph would be interesting, as it is by far the largest one, the runtime needed to optimize the graph exceeded the compute time we deemed sensible. To differentiate both datasets, we will call the one with this graph GD21’ and the one without GD21.

A common dataset for testing graph algorithms is a test suite of directed and undirected graphs from the GDToolkit project. GDToolkit is a project aiming to provide a framework for drawing graphs, similar to the OGDF, but is not actively maintained anymore. This set of graphs is also known as the “Rome”-dataset, the Roma Tre University maintained GDToolki. We use the undirected planar graphs of the Rome-dataset as supplied on the website of the graph drawing contest [Con22].

Another standard dataset is the “North”-graphs from AT&T. We use the directed planar graphs of the North-dataset, which were also supplied on the website of the graph drawing contest [Con22].

We also test our approach on a set of self-generated graphs from the tool PlanTri by Brinkmann and McKay [BM22]. PlanTri generates planar triangulations, which are planar graphs in which adding any edge would result in a nonplanar graph. Triangulations are the densest existing planar graphs, and embeddings of planar triangulations with many vertices often only admit embeddings with high graph depth. This case is interesting for our optimizer because we expect it to struggle with dense and high-depth graphs.

We sampled the dataset “PlanTri” by first calculating graphs with four to 64 vertices using `plantri`. Graphs with more vertices are not supported by the tool. Due to the immense amount of graphs generated by PlanTri, we let the tool run for 0.1 s for each vertex count. This runtime is enough to generate about 9000 graphs with ten vertices or 300 graphs with 64 vertices. We then randomly choose up to 49 graphs of each vertex count.

Statistics on the datasets mentioned above are summarized in Table 5.1.

5.2. Initial Layouts and Embeddings

First, we have to choose an initial embedding and layout algorithm. Different choices for these parameters can result in differences in the initial ELR^{GD}. As already mentioned in Chapter 3, we also expect some combinations of embedders and layout algorithms to result in drawings that are easier to optimize for our approach.

Because only six embedders and five layout algorithms exist, we can run all possible combinations of embedders and layout algorithms with a fixed measure set. The results of these test runs are summarized in Tables 5.2 to 5.4.

Table 5.2.: The average initial ELR^{GD} on the dataset GD21 for all combinations of embedders EMB and layouting algorithms LAY.

EMB / LAY	MD	MDMF	MDMFL	MDPTA	MF	MFL	Simple
FPP-Layout	363.58	363.58	363.58	363.78	363.31	363.31	363.81
Schnyder-Layout	229.50	241.50	240.64	229.49	226.87	228.19	236.75
PDL	153.73	94.03	94.03	155.73	94.03	151.72	98.02
PSL	225.75	209.98	209.98	218.55	209.98	219.52	219.69
MML	92.50	66.58	66.58	92.17	66.58	95.40	67.14

Table 5.3.: The average relative ELR^{GD} decrease on the dataset GD21, defined as ELR$^{GD}_{opt}$/ELR$^{GD}_{start}$, for all combinations of embedders EMB and layouting algorithms LY.

EMB / LAY	MD	MDMF	MDMFL	MDPTA	MF	MFL	Simple
FPP-Layout	0.29	0.25	0.22	0.28	0.24	0.24	0.21
Schnyder-Layout	0.40	0.38	0.43	0.43	0.37	0.37	0.39
PDL	0.55	0.50	0.50	0.55	0.50	0.54	0.49
PSL	0.45	0.35	0.36	0.50	0.35	0.48	0.33
MML	0.89	0.84	0.82	0.82	0.84	0.88	0.82

The optimized values in these experiments were calculated by running the optimizer with the measure set $\{\text{AELD}, \text{ELR}^{GD}\}$. The exponent of AELD, as defined in Section 4.1.4, is set to 17 and the offset to 10 for 20 000 iterations. The first run is followed by a run with measure set $\{\text{ELR}^{GD}, \text{RAND}\}$ for 2000 iterations. For both runs, lazy recalculation was activated.

Layout Algorithms

In Section 3.2, we gave examples of drawings generated by the different layout algorithms. As can be seen in Figures 3.3a, 3.3b, 3.4a, 3.4b, 3.5a and 3.5b the FPP-, Schnyder- and PS-layout draw the graph in a large triangle with the longest edge at the bottom. The drawings for the FPP-layout are normally the largest, followed by the Schnyder- and PS-layout.

The PD-layout shortens this longest edge by pushing the vertices toward the left margin of the drawing. This often results in an approximately rectangular drawing as given in

Table 5.4.: The average optimal ELR^{GD} on the dataset GD21, for all combinations of embedders EMB and layouting algorithms LY.

EMB / LAY	MD	MDMF	MDMFL	MDPTA	MF	MFL	Simple
FPP-Layout	105.08	91.56	63.41	87.76	76.61	71.30	77.91
Schnyder-Layout	82.76	91.58	118.66	90.78	73.58	67.14	95.52
PDL	92.68	47.56	47.56	92.59	47.56	90.68	54.05
PSL	88.99	66.13	66.15	78.30	66.13	84.24	75.37
MML	86.61	61.08	61.08	85.66	61.08	89.42	58.14

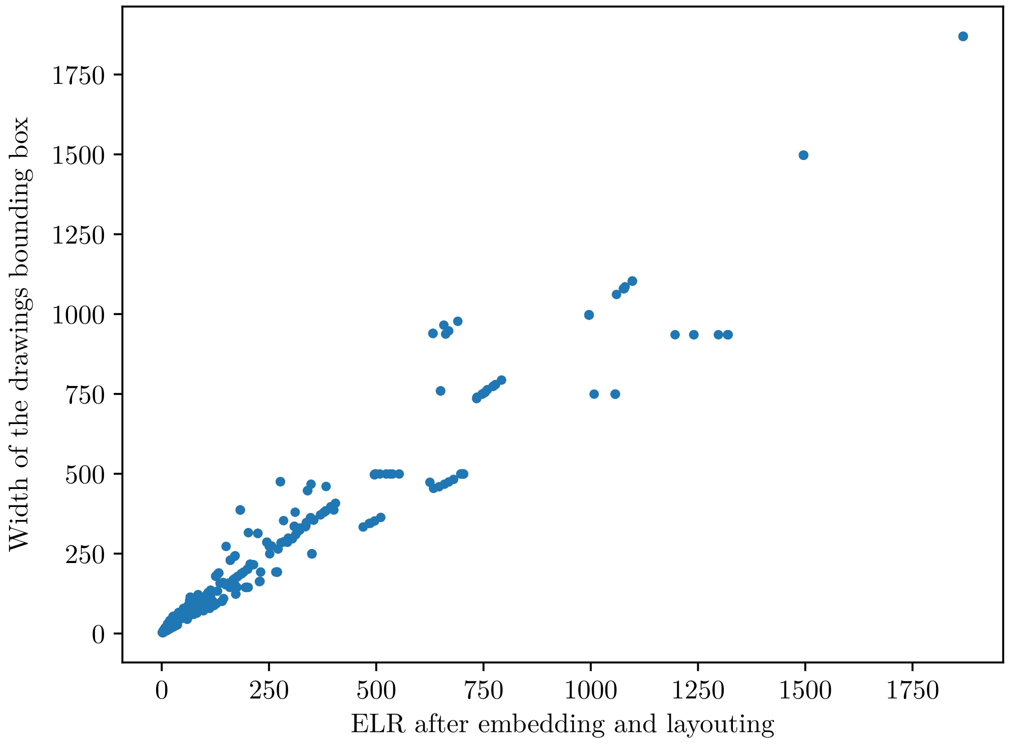


Figure 5.1.: The ELR^{GD} of graph drawings after embedding and layout, plotted against the width of the drawing for the dataset GD21.

Figures 3.6a and 3.6b. The MM-layout uses a completely different approach, packing the vertices closer to each other than in the previous four layouts. In theory, the MM-layout has the biggest space requirement of all layout algorithms. When we tried to verify this in practice, we discovered that for any graph in the datasets GD21, Rome, and North there was at least one algorithm needing more space than the MM-layout.

All layouts have in common that the length of the longest edge is similar to the width of the drawing. Combined with the fact that the shortest edge in the drawing almost always has a length of 1 or $\sqrt{2}$, we can assume a linear connection between the width of a drawing and its ELR^{GD}. This is confirmed by Figure 5.1. In Figure 5.2, the scatter points are additionally colored by their layout algorithm, for instance `auto21_12`. This visualizes how the different layout algorithms produce similar layouts, only slightly depending on the embedding used.

The relative ELR^{GD} decrease for the different drawing algorithms given in Table 5.3 is also closely connected to the initial ELR^{GD}. While the average relative ELR^{GD} decrease for the FPP-layout is 0.25, the Schnyder- and PS-layout only manage a decrease of 0.40, and the PD-layout only achieves a decrease of 0.52.

The decrease is especially bad for the MM-layout, having an average relative decrease of around 0.85. The compact layout cannot be the sole reason for such bad behavior. One problem is that the bends give the faces of the drawing an irregular shape, observable in Figures 3.7a and 3.7b. As demonstrated in Figure 3.2, this reduces the optimizers's options for moving vertices. Another problem with the MM-Layout is that almost all edges are drawn on grid lines, due to it being mostly orthogonal. The only exception to this is short edge segments directly adjacent to vertices. A line that runs orthogonal on a grid line

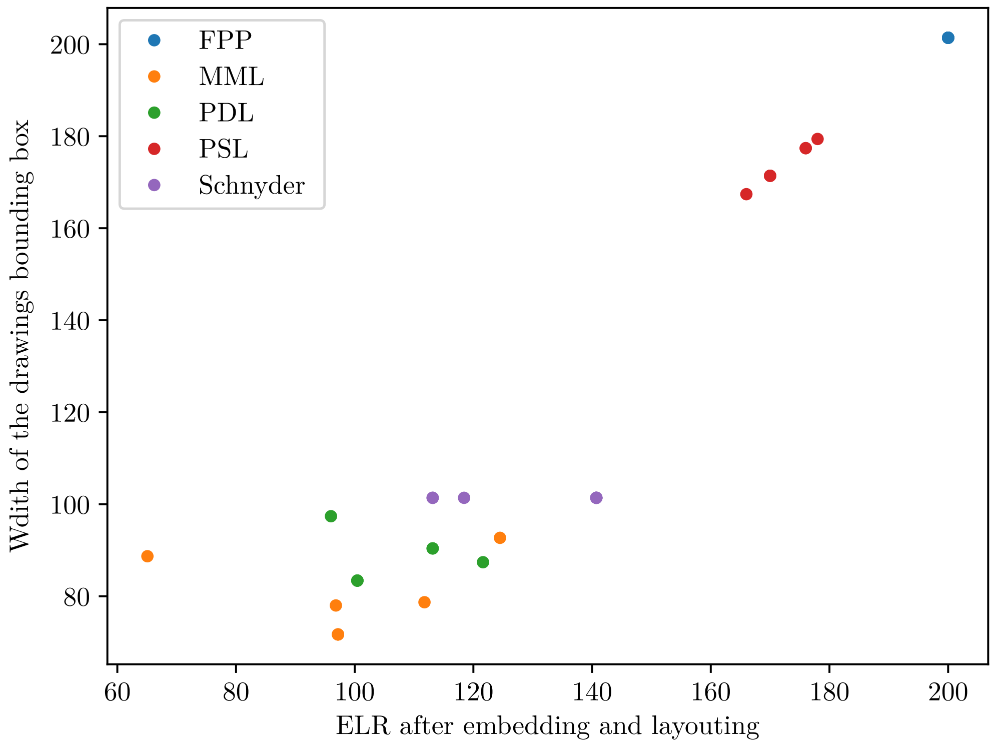


Figure 5.2.: The ELR^{GD} of graph drawings of graph `auto21_12.json` from dataset GD21 after embedding and layout, plotted against the width of the drawing. The differently colored scatter points showcase the different layouts. For some layouts, not all points are visible because different embedders sometimes result in the same drawings and thus overlapping points.

Table 5.5.: The average ELR^{GD} before and after the optimization and drawing size after optimization for all embedders split by graph size. The columns labeled "small" only consider graphs with less than or equal to 100 vertices from GD21, the columns labeled "large" all other graphs from GD21.

Layouter	Small			Large		
	ELR$^{GD}_{Start}$	ELR$^{GD}_{Opt}$	Area$_{Opt}$	ELR$^{GD}_{Start}$	ELR$^{GD}_{Opt}$	Area$_{Opt}$
FPP-Layout	45.94	<u>6.15</u>	1359	840.00	195.65	380 694
Schnyder-Layout	27.29	6.93	723	542.27	211.04	207 145
PDL	16.04	7.13	326	276.40	<u>158.11</u>	52 719
PSL	23.68	11.51	603	505.00	170.35	126 901
MML	<u>10.25</u>	6.88	<u>230</u>	<u>179.96</u>	169.35	<u>13 178</u>

blocks all grid points on its way for the optimizer. This greatly reduces the number of options the optimizer has and limits its possibilities even further.

Another measure of the suitability of the drawing generated by the layout algorithms is the grid size they need before and after optimization. The algorithm should be as space-efficient as possible since outside hyperparameters often limit grid size. Additionally, the grid becoming too fine contradicts the idea of having a grid. In Table 5.5 we summarize this information alongside the ELR^{GD} before and after optimization.

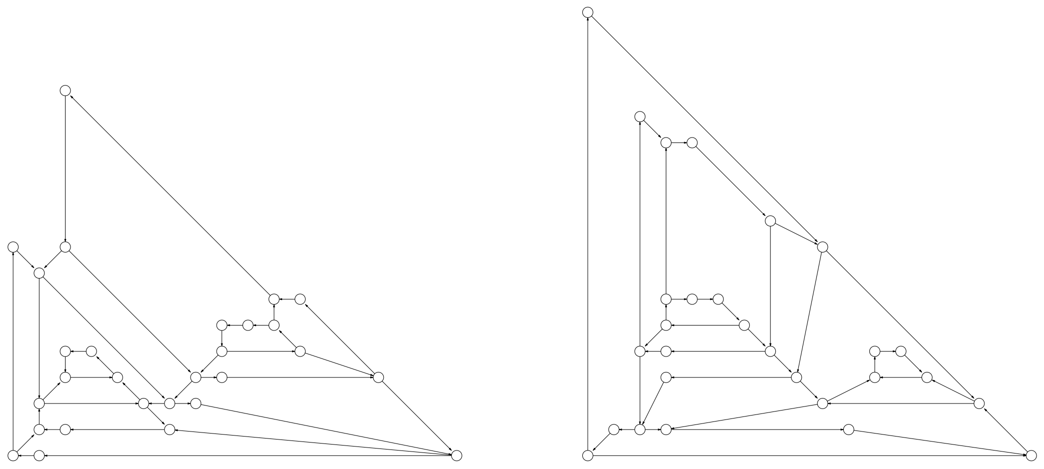
This table also differentiates the two cases we have to handle. For small graphs, the chosen drawing algorithm is almost irrelevant for the optimal ELR^{GD}. In this case, we can focus on an optimal drawing size, resulting in the MM-Layout for graph drawings with bends, and the PD-Layout for straight-line graph drawings. The differences in optimal ELR^{GD} are more prominent in large graphs. While the MM-Layout produces significantly smaller drawings than the PD-Layout, the ELR^{GD} of the drawings generated using the PD-Layout is slightly better.

Embedders

The second part for calculating a suitable initial layout is the embedder. In Table 5.2, we can see that the influence of the embedder on the initial ELR^{GD} was minimal for the FPP-, Schnyder-, and PD-Layout. In contrast, for the PD- and MM-layout the embedders `SimpleEmbedder` (Simple), `EmbedderMaxFace` (MFL), `EmbedderMinDepthMaxFace` (MDMF), and `EmbedderMinDepthMaxFaceLayers` (MDMFL) resulted in significantly better initial ELR^{GD}.

The aspect these four embedders have in common is that they generate a drawing with a maximal outer face. This led us to the question of why the `EmbedderMaxFaceLayers` (MFL) performed worse than the other four embedders, even though it still claims to maximize the outer face. After checking two layouts from the GD21 dataset embedded by Simple and MFL (given in Figures 5.3a and 5.3b), we could confirm that, indeed, MFL does *not* always generate embeddings with a maximal outer face. This can also be reproduced for other graphs in dataset GD21.

This influence of the embedder on the initial ELR^{GD} also carries on into the optimized ELR^{GD}. In Table 5.4, the optimized ELR^{GD} is significantly better when using any of the four embedders mentioned above. This only holds for the PD- and MM-Layout drawing algorithms, but due to their superior optimized ELR^{GD} and drawing area, we will not use the other algorithms anyway.



(a) Using the `SimpleEmbedder` with ten vertices along the outer face.

(b) Using the `EmbedderMaxFaceLayers` with 5 vertices along the outer face.

Figure 5.3.: Two drawings of the same graph using the layouting algorithm PDL showcasing how `EmbedderMaxFaceLayers` does not generate drawings with a maximal outer face.

Conclusion

Using this information, we can formulate a strategy for which layout and embedder to choose. If we are bound on space and are allowed to use at most three bends, we choose the MM-Layout using any embedder maximizing the outer face. If space is not limited or the number of bends is limited to two or less, we can divert to using the PD-Layout with any embedder maximizing the outer face for large graphs.

5.3. Quality Measures

The most significant influence on the optimization result is the choice of measures for the runs of our algorithm and the comparator we choose. Each run receives a set of quality measures as its configuration.

We first evaluate each measure alone in Section 5.3.1. As a measure that does not perform well independently will usually not perform well as the leading measure for an algorithm run using more than one measure, this will already give us a general idea of which measures are suitable to use with a high priority or weight. Some measures also have hyperparameters that have to be optimized. One example would be the exponent for the AELD.

Because we cannot test all possible combinations and orderings of measures, we have to reduce the number of feasible measure sets. We use the insights from the single-measure runs and the ideas the measures are based on in Section 4.1 to formulate possible measure combinations, which we then test in Section 5.3.3.

The last point to optimize is the number of iterations. In general, we want to cap the number of iterations as close as possible to the moment we arrive at a local optimum. In Section 4.3, we showed why we could not exactly know when we reached this point.

5.3.1. Single Measure Runs

We first compare the possible measures running alone to get a general overview of the different measures. The results of these runs are given in Table 5.6. Each measure was run for 15 000 iterations, using the embedder MDMFL, the PDL, and lazy recalculation.

Table 5.6.: The average relative ELR^{GD} decrease after embedding with MDMFL, layouting with PDL, and optimizing with a single measure for 15000 iterations in lazy mode. The columns labeled “small” only consider graphs with less than 100 vertices from GD21, the columns labeled “large” considers all other graphs from GD21.

Optimizer	All	Small	Large
ELR^{GD}	0.78	0.67	0.96
Total Edge Length	0.79	0.72	0.88
AELD (Exponent 1.0, Offset 4)	0.96	1.05	0.81
AELD (Exponent 2.0, Offset 4)	0.60	0.48	0.78
AELD (Exponent 5.0, Offset 4)	0.66	0.62	0.72
AELD (Exponent 15.0, Offset4)	0.64	0.60	0.69
Min/Max count	40.75	66.97	1.42
Random	1.51	1.84	1.02

The worst measure for optimizing ELR^{GD} is the Min/Max count, which is even worse than the random measure we included as a baseline comparison. It behaves bad because, as already mentioned in Section 4.1.3, we only introduced it as a way to make the ELR^{GD} better at finding a possible next step. By itself, it does not have any incentive to shorten long edges or lengthen short edges, which would result in better ELR^{GD}.

ELR^{GD} as the sole measure is very dependent on the size of the input graph. It achieves a considerable decrease in ELR^{GD} for small graphs, but for large graphs the performance is rather bad at an average decrease of 0.96. This behavior is induced by Lemma 4.1, as we can only achieve a decrease in ELR^{GD} when all longest or shorted edges are incident to one vertex. In large graphs this is unlikely, resulting in bad performance.

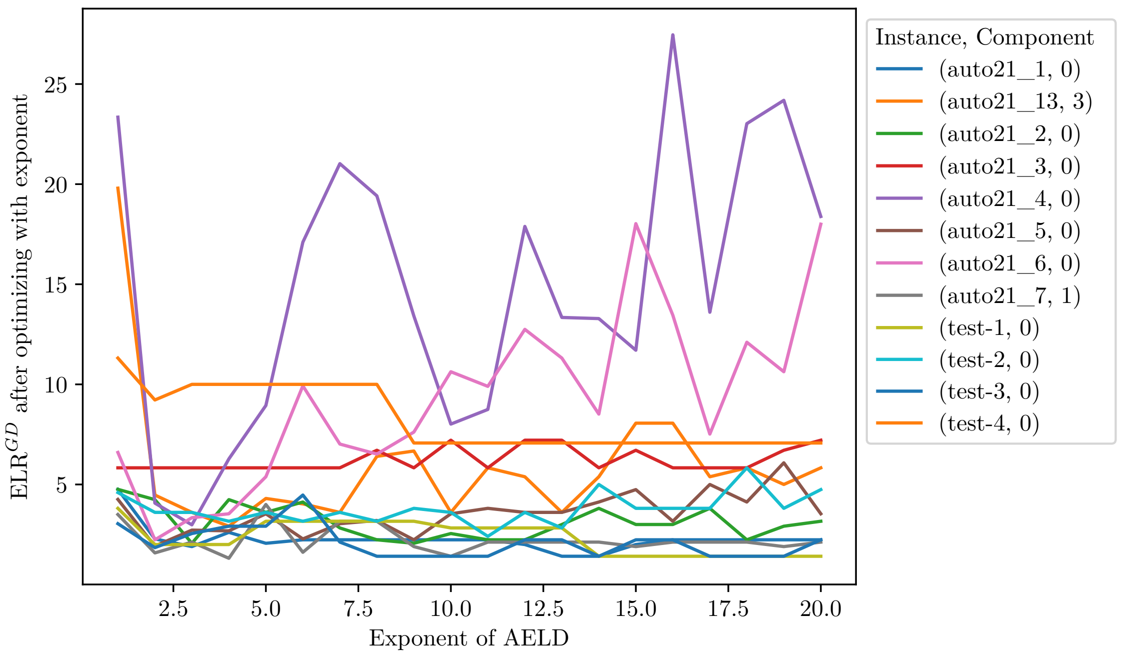
A success was running the total edge length alone. We only introduced it in Section 4.1.5 to have a measure that can keep the overall size of the drawing low. It is successful because decreasing total edge length sometimes decreases the length of the current longest edge. Additionally, in most drawings, the initial shortest edge has a length of one. Therefore the measure can not accidentally increase the ELR^{GD} of the drawing.

AELD achieved the best results. As the performance of AELD is strongly dependent on the chosen exponent and offset, we have to tune these two parameters for the best results. We take a detailed look at these parameters in the following subsection.

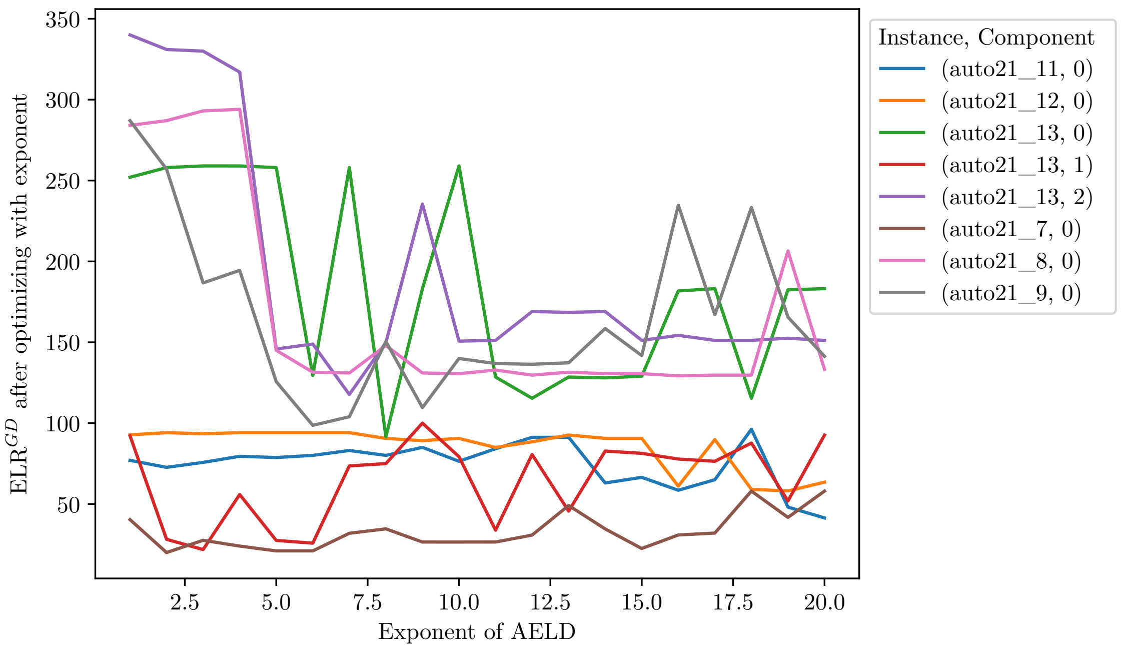
5.3.2. Parameter Tuning for AELD

The AELD is the only measure that takes additional parameters as input. As introduced in Section 4.1.4, it needs an exponent, which is applied to the differences for edges shorter than the average. It also accepts an offset to the average edge length. The intended influence of the exponent is giving the algorithm an incentive to prefer lengthening short edges. This is advantageous because it keeps the drawing size high and gives the optimizer a preference for lengthening short edges. The offset changes our target edge length, having the same effect as the exponent. Additionally, it adds a “tension” into the drawing, making it impossible to reach an AELD of zero.

The experiments in this section were all run using 20 000 iterations, the layout algorithm PD-Layout, the embedder MDMFL, and lazy recalculation.



(a) On all graphs in the dataset GD21 with less than 100 vertices.



(b) On all graphs in the dataset GD21 with at least 100 vertices.

Figure 5.4.: The resulting ELR^{GD} after optimizing GD21 using AELD with different exponents and offset four for 20 000 iterations.

Exponent

The results for the exponent are given in Figures 5.4a and 5.4b, for the offset in Figures 5.5a and 5.5b. Overall we can see that the algorithm reacts strongly to changes in the exponent.

For the small graphs in Figure 5.4a, the results are unstable. We can observe that for most graphs, the optimum is at an exponent of two or three. Additionally, the results deteriorate with rising exponent. For small graphs, we suggest running the optimizer twice. Once with an exponent of two, once with an exponent of three, choosing the best result from both runs.

In the large graphs in Figure 5.4b the exact opposite behaviour is visible. Exponents smaller than five give subpar results. Depending on the specific instance, an exponent of up to 15 is needed to see the exponent's influence on the quality of the drawing. The big problem with optimizing the exponent for these larger graphs is that the resulting ELR^{GD} is very unstable. As a result, we can not choose one specific exponent and have to run the algorithm with several exponents, only keeping the best output. This can be rather slow when done sequentially, so in time-limited environments like the graph drawing contest at Graph Drawing [Gra22], we ease this problem by starting several runs of the algorithm in parallel.

Offset

The results for the offset are given in Figures 5.5a and 5.5b. They are more uniform and stable over the different offsets than the results for the exponent.

Using an offset works well for most graphs. Overall, an offset of around ten seems optimal for most instances. For some of the smaller graphs, a lower offset of five is also working.

One problem we have to catch is that an overly high offset leads to deteriorating results for large graphs. When investigating the drawings generated with a very high offset, one can see that a high offset results in irregularly shaped faces (e.g., very long but thin faces, non-convex faces). These faces then block possible moves and place our algorithm at a disadvantage (as already illustrated in Figure 3.2). The difference between a drawing with a low offset and an overly high offset is shown in Figures 5.6a and 5.6b for two drawings of instance `auto21_8`.

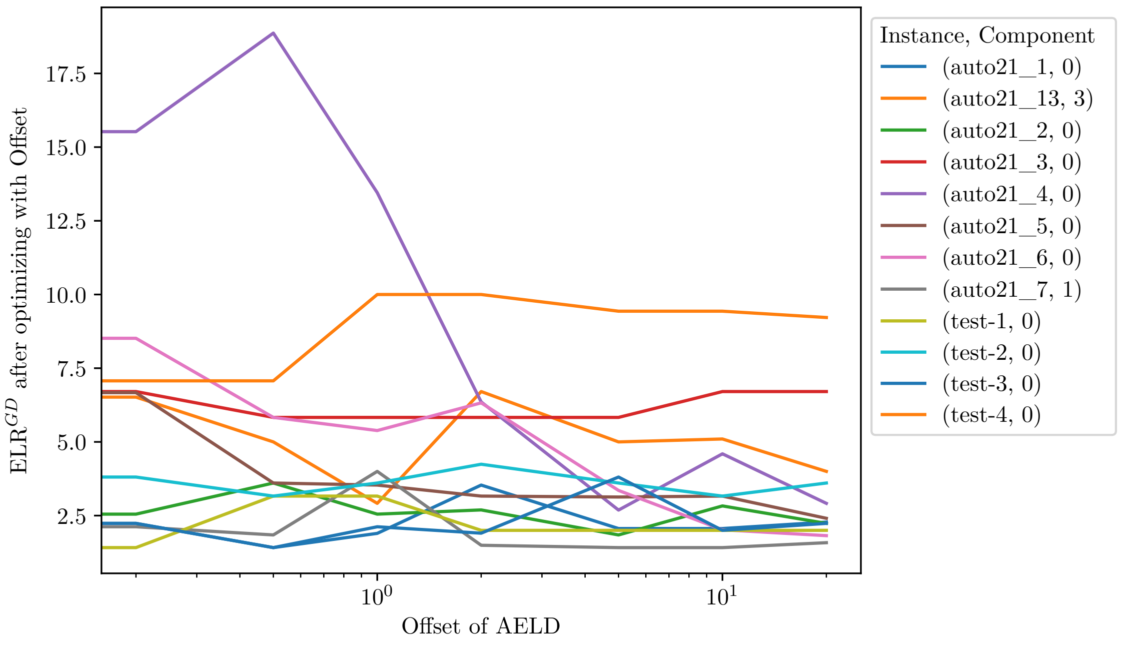
Combining and Conclusion

The last step is combining the optimization of the exponent and offset. We choose some sensible values for both parameters and run the optimizer on the dataset GD21, collecting the results in Tables 5.7 and 5.8. We once again can split the results of the optimization for two cases.

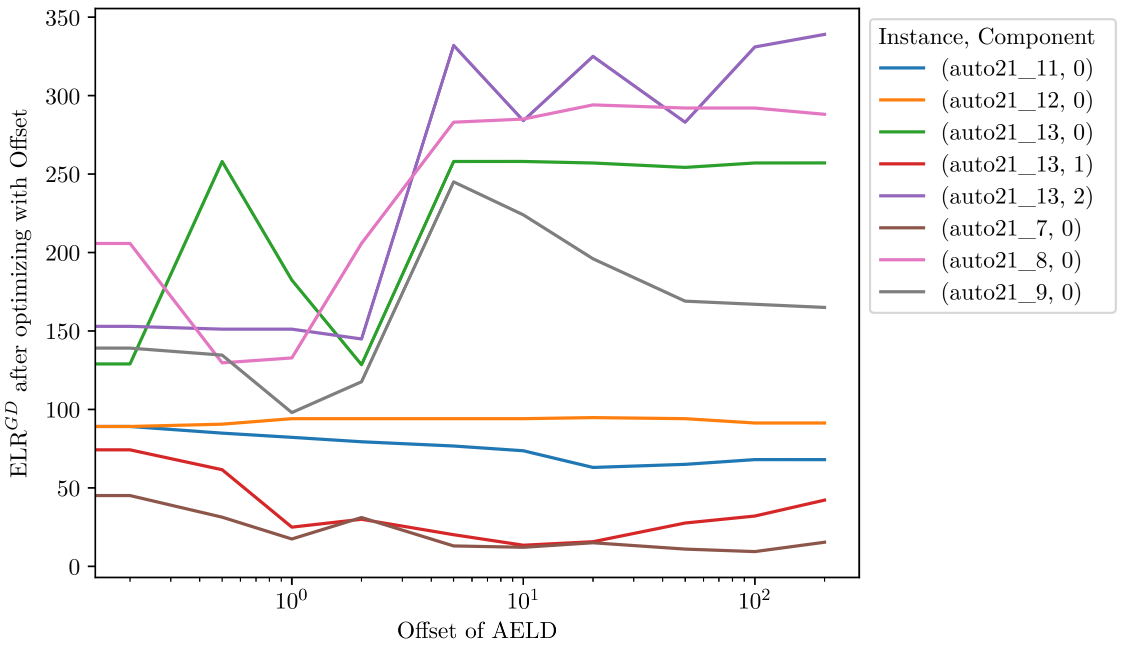
First, we have the graphs with less than 100 vertices in Table 5.7. For these graphs, no strong connection between the different exponents and optimization result is visible. Only for small offsets, there is a slight influence, with smaller exponents producing better results. The bigger impact on the results of the optimization is the offset. Higher offsets, even up to 5000, give better results.

For graphs with more than 100 vertices, given in Table 5.8, the results are quite the opposite, even though they generally are a lot noisier and more difficult to analyze. The best results use a low offset, between five and ten, and high exponents. The problem here is that the results are relatively unstable. Therefore, we must try several exponents and choose the best-performing run.

A high offset has one effect on both graph sizes, "stabilizing" the results of the algorithm. This is already visible at offsets of around 100, where the results for different exponents

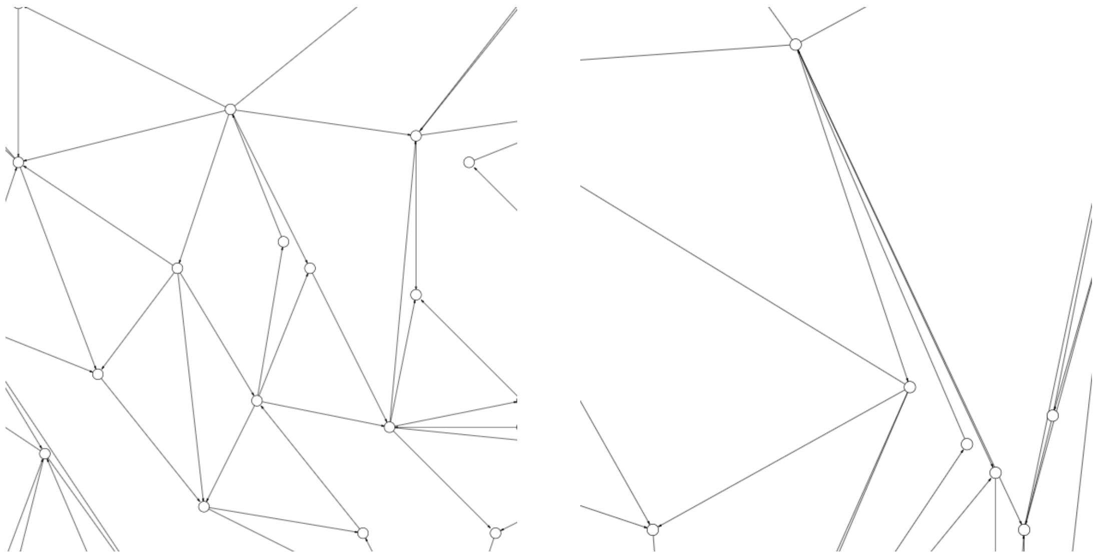


(a) On all graphs in the dataset GD21 with less than 100 vertices.



(b) On all graphs in the dataset GD21 with at least 100 vertices.

Figure 5.5.: The resulting ELR^{GD} after optimizing GD21 using AELD with different offsets and exponent ten for 20 000 iterations.



(a) With an offset of 10.

(b) With an offset of 2000

Figure 5.6.: An illustration on how a too high offset can result in drawings with irregularly shaped faces using instance `auto21_8`.

Table 5.7.: The ELR^{GD} improvement of different exponents and offsets combined for all graphs in dataset GD21 with less than 100 vertices.

Exponent \ Offset	0	5	10	100	1000	5000	Mean
3	0.48	0.52	0.41	0.38	0.40	0.36	0.42
5	0.45	0.42	0.43	0.40	0.39	0.38	0.40
15	0.63	0.51	0.48	0.38	0.40	0.38	0.45
25	0.62	0.46	0.47	0.38	0.40	0.40	0.43
Mean	0.54	0.48	0.45	0.39	0.40	0.38	

Table 5.8.: The ELR^{GD} improvement of different exponents and offsets combined for all graphs in dataset GD21 with at least 100 vertices.

Exponent \ Offset	0	5	10	100	1000	5000	Mean
3	0.71	0.73	0.68	0.65	0.63	0.63	0.67
5	0.61	0.53	0.70	0.63	0.65	0.64	0.64
15	0.60	0.50	0.49	0.65	0.66	0.64	0.60
25	0.64	0.56	0.51	0.63	0.66	0.63	0.62
Mean	0.64	0.58	0.60	0.64	0.65	0.64	

have a significantly lower variance in Tables 5.7 and 5.8. We can use this effect to our advantage by starting an additional run using a high offset to get a baseline comparison subsequent runs.

We can use the insights of the last few paragraphs to develop a system for choosing the parameters of AELD. First we do a run with a high offset and a low exponent. This run solves two problems: It acts as a baseline comparison for large graphs and as the suspected optimal run for small graphs. For large graphs, we then do additional runs to try higher exponents with a low offset. We can always compare the results of these runs with our baseline to decide whether another run is necessary or not.

Regarding the hyperparameters of AELD, we will use the exponent/offset pairs $(3, 500)$, $(10, 5)$, $(15, 5)$, $(20, 5)$, and $(25, 5)$ from now on, always keeping the best result of these attempts for further analysis.

5.3.3. Lexicographical Ordering

In Section 5.3.1, we saw that not all different measures perform well alone. This is why our algorithm can use multiple measures for a single run. We then order the possible moves in the lexicographical ordering presented in Figure 4.2a.

We have to limit our experiments to several interesting combinations for our evaluation of measure combinations in this section. We summarize the chosen combinations and the results of these experiments in Figures 5.7a and 5.7b, once again split into large and small instances.

The algorithm was run using the PD-Layout, embedder MDMFL, and lazy recalculation. Each measure set was run for 20 000 iterations.

For all graphs, both large and small, we can observe that the measure set $\{\text{ELR}^{GD}, min\_max\}$ does not perform well. In Table 5.6 ELR^{GD} it achieves an average decrease of 0.93 for large graphs and 0.56 for smaller graphs. With the addition of *min_max* the decrease is slightly better at 0.84 for large graphs and 0.44 for small graphs, but in comparison to most other run combinations, this performance is still bad. For most instances, the measure set $\{\text{ELR}^{GD}, \text{TEL}\}$ performs comparably.

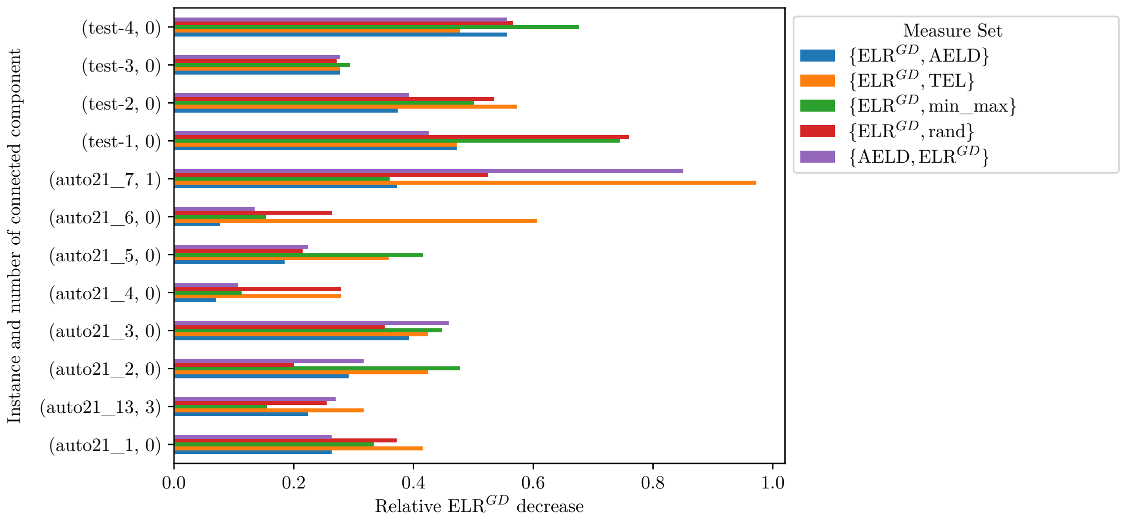
For small graphs, a good performing approach is combining ELR^{GD} with the randomized measure. As this approach stops working for large graphs, we conclude that the efficiency of the random measure is directly linked to the size of the graphs. Randomizing helps the algorithm to move out of saddle points in ELR^{GD} and escape local optima.

The last measure with ELR^{GD} as the first measure is $\{\text{ELR}^{GD}, \text{AELD}\}$. It performs considerably well across all graphs in GD21, from small to large. It is one of the best-performing measure sets for almost every graph, with a mean decrease in ELR^{GD} of 0.36 for graphs with at least 100 and 0.30 for all other graphs. This is also considerably better than running any of the two measures, be it ELR^{GD} or AELD, alone.

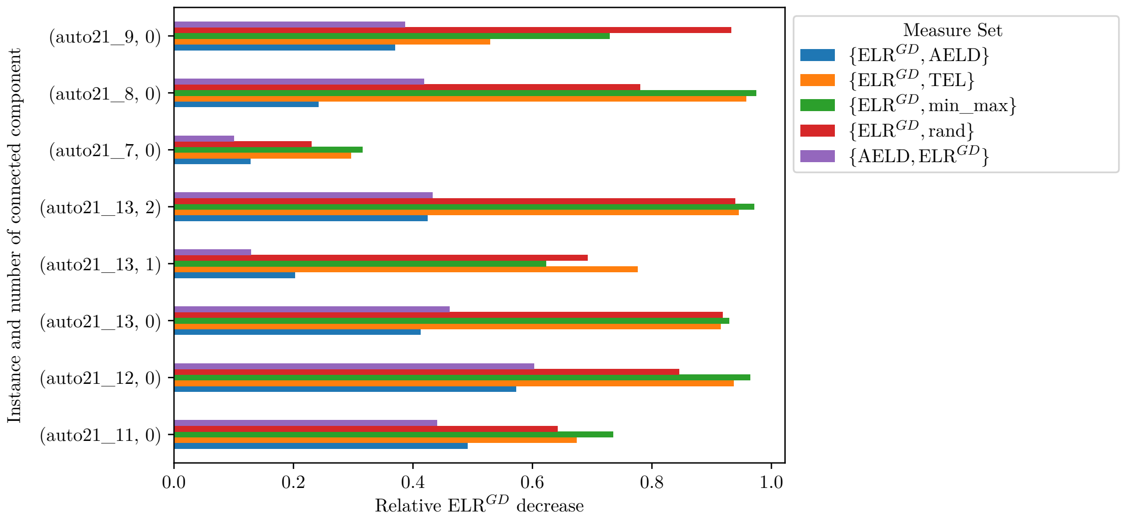
We additionally tested the opposing set $\{\text{AELD}, \text{ELR}^{GD}\}$. In most cases, the results of both sets are almost equal, with only slight differences in some measures. If they differ, usually the measure set $\{\text{AELD}, \text{ELR}^{GD}\}$ performs slightly worse.

5.3.4. Multi-Run Setups

The goal of this section is to test whether doing multiple optimization runs with different optimizers subsequently benefits the result of the optimization. We only tried two-run setups because anything more would have resulted in too many potential combinations.



(a) For all instances with up to 99 vertices.



(b) For all instances with 100 vertices or more.

Figure 5.7.: The average ELR^{GD} decrease for the different instances of dataset GD21 after optimizing with the given measure sets for 20 000 iterations.

Table 5.9.: The average relative ELR^{GD} improvement after embedding with MDMFL, layouting with PDL and optimizing with a single measure for 5000 iterations. The columns labeled “small” only consider graphs with less than 100 vertices, the columns labeled “large” all other graphs from GD21.

Optimizer	All	Small	Large
$[\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}\{\mathrm{ELR}^{GD}, rand\}]$	0.38	0.29	0.52
$[\{\mathrm{AELD}\}\{\mathrm{ELR}^{GD}, rand\}]$	0.38	0.28	0.53
$[\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}\{\mathrm{ELR}^{GD}, tel\}]$	0.42	0.36	0.52

For this section, the setups tried are combinations of the most successful lexicographic optimizer sets.

All graphs were first drawn using the PD-Layout and embedder MDMFL. We always run the first measure set for 20 000 iterations and the second set for 2000 iterations using lazy recalculation. The number of iterations is lower for the second run because the algorithm has already optimized most of the graph, bringing it closer to a local optimum. The results of these runs are summarized in Table 5.9.

The first evaluated combination is the setup $[\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}\{\mathrm{ELR}^{GD}, rand\}]$. The basic idea of this approach is that the first set of measures will lead to a local optimum, while the second set of measures tries to find a better optimum nearby, with the help of the random measure to avoid getting stuck on plateaus. This could help the random measure to become more effective in large graphs. Overall, it performs worse than running $\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}$ alone. Even though the performance is comparable for small instances, the performance on large instances drops from an average ELR^{GD} decrease of 0.36 for the single run to 0.52 for the multi-run. The same results were achieved when only $\{\mathrm{ELR}^{GD}\}$ was used for the first run.

A success is the usage of $[\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}, \{\mathrm{ELR}^{GD}, \mathrm{TEL}\}]$. While the average performance is worse than the other run setups on both instance sizes, the second measure set achieves a significant reduction in drawing size for small graphs. On average, the resulting grid has a size of 159.39 grid points. The next-best measure set, $\{\mathrm{ELR}^{GD}, \mathrm{TEL}\}$ alone, only achieves drawings with an average size of 201.71 grid points and additionally has a significantly worse ELR^{GD} reduction at 0.47 for small graphs. This advantage shrinks for graphs with at least 100 vertices, with an average grid size of 30 556.79 against an average grid size of 31 475.51 for the measure set $\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}$ run alone. Additionally, the ELR^{GD} improvement for the latter measure set is significantly better at 0.36.

5.3.5. Number of Iterations

A conflict of interest binds the number of iterations the algorithm is run for. Few iterations avoid unnecessary calculations, which results in lower runtimes. Because there is usually no limit regarding the algorithm’s runtime, overestimating this runtime is usually preferred. If a runtime limit is present, the algorithm can also be run for this specific time. Since a valid output is computed after each iteration, this intermediary result can then be used. The algorithm can also be restarted on the output of a previous run, but in general, this approach is not desired. Usually, non-interactive algorithms that do not need user input during computation are preferred.

For the plots in this chapter, we ran the optimizer algorithm for 31 000 iterations using lazy recalculation. The first 30 000 iterations use the optimizer set $\{\mathrm{ELR}^{GD}, \mathrm{AELD}\}$, while the last 1000 iterations use $\{\mathrm{ELR}^{GD}, \mathrm{RAND}\}$. The initial layout was calculated using the PD-Layout and embedder MDMFL.

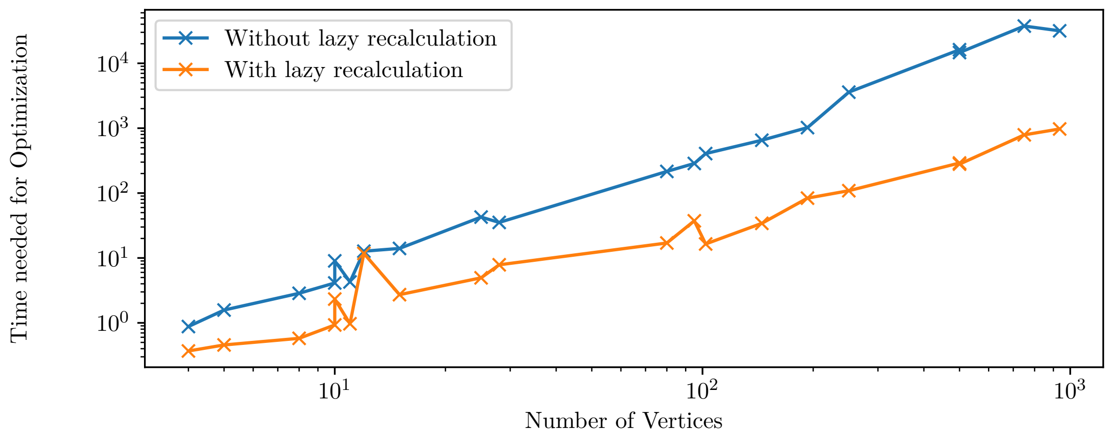


Figure 5.8.: The time needed to optimize the graphs from dataset GD21 for 31 000 iterations with and without lazy recalculation.

In Figures A.1 and A.2 the influence of the initial ELR^{GD} is clearly visible. Depending on the exact instance, the number of iterations can range from less than 100 to 17 000 iterations.

An interesting feature in these two plots is the two different kinds of decrease. Small decreases are usually shortenings of the longest edge. These do not have an overly strong effect on ELR^{GD} because decreasing an already large numerator by a number smaller or equal to one results in a small overall decrease. On the other side, there are some significant jumps in the plots. These big jumps are the result of the optimizer being able to increase the length of the last shortest edge. Increasing a small denominator by a number smaller or equal to one results in a strong downtick in ELR^{GD}. If the previous last shortest edge had length one and is increased to length $\sqrt{2}$, ELR^{GD} is reduced by $\frac{1}{\sqrt{2}}$.

We can also see in Figures A.1 and A.2, that the ELR^{GD} often is not very stable. In Section 5.4, we will see that this behavior is induced by the lazy evaluation used for these runs. We counter this behaviour by not saving the end result of the computation, but the overall best result encountered during the optimization process.

This also makes the results from the second run in Figures A.3 and A.4 hard to analyze. Even though there are new minima found for most of the input graphs, the overall randomness from the lazy evaluation makes the plots very hard to read.

5.4. Runtime and Lazy Recalculation

One last important statistic of the algorithm is the runtime for graphs of different sizes, with lazy recalculation enabled and disabled. Overall, we expect the runtime for one iteration to be within $O(n^2)$, with n being the number of vertices in the graph if we fix the measures. This runtime expectation stems from the fact that each iteration tests all measures, each having a runtime in $O(n)$ as described in Section 4.1 for all vertices.

The runtime needed for optimizing the graphs from the dataset GD21 is visualized in Figure 5.8. Because the plot is a log-log plot and the axes are not uniformly spaced, it masks the quadratic behavior, but the average slope of 2 on the logarithmic scale implies a quadratic connection between the two variables.

It was necessary to plot the values in such a plot to visualize the speedup achieved by lazy recalculation. For most data points, using lazy recalculation is at least an order of magnitude faster, which is excellent. This speedup gets even better for larger graphs

Table 5.10.: The performance of our finalized algorithm on all available datasets.

Dataset	Vertices	Edges	ELR^{GD}_{pre}	ELR^{GD}_{opt}	ELR^{GD}_{improv}	Area_{opt}	Time
GD21	230.06	424.56	84.23	29.57	0.34	9976.32	52.35 s
North	28.96	32.69	14.83	1.95	0.19	420.49	0.74 s
Rome	25.87	30.18	12.78	1.93	0.21	258.38	0.51 s
PlanTri	36.61	103.85	46.61	26.27	0.57	1471.41	8.69 s

because the algorithm can do more moves using the lazily calculated information before refilling the heap. The complete recalculation of the heap is the only step in the algorithm that still needs superlinear time.

One disadvantage of the lazy recalculation is visible when comparing Figures A.1 and B.5, Figures A.2 and B.6, Figures A.3 and B.7, and Figures A.4 and B.8. In the non-lazy plots, the optimization is significantly less noisy, and the number of iterations needed for the algorithm to converge to a drawing is lower. However, the difference in iterations needed is nowhere near the point at which the time saved by doing fewer iterations gives a better runtime for the non-lazy variant.

In terms of results, no version is clearly better. Most of the results are very similar, only in some instances one or the other generates drawings with lower ELR^{GD}. Due to this, there is no real benefit in using the non-lazy recalculation, as it is significantly slower for the same results.

5.5. Evaluation on different Datasets

The final experiments we conducted were test runs of the finalized and optimized algorithm on all available datasets. For these tests the algorithm was set up to use the embedder MDMFL, layout algorithm PD- and MM-Layout and then 20 000 iterations of the optimizer with the measure set $\{\text{ELR}^{GD}, \text{AELD}\}$ with lazy recalculation enabled. We then chose the best run for each instance for our statistics. The results of these runs are summarized in Table 5.10.

Overall the algorithm did a good job of reducing the ELR^{GD} of the drawings. For the dataset GD21, we already know from our previous experiments that the reduction of 0.36 is about what to expect. The algorithm was able to calculate solutions for all graphs of the dataset. It also found valid solutions (i.e., within height and width bounds) for several graphs the participants in the previous year were not able to solve.

For the datasets North and Rome, the algorithm performs very well. On average, the ELR^{GD} of the optimized drawings is less than two for both datasets. The average runtime is also very low. Compared to the runtime of the PlanTri dataset, which has a roughly comparable average edge count, the runtimes on the North and Rome graphs are 16 times quicker. The same goes for the comparison with the runtimes of the graphs in GD21, even though the quadratic relationship between vertex count and computation time has to be kept in mind when comparing these runtimes.

In general, the runtime seems to be influenced by the number of vertices, and the ratio between vertices and edges. The first claim is evident. For more vertices we have to try more possible moves, resulting in longer runtime per iteration. The second claim stems from our custom intersection test that scales with the average degree of the vertex that got moved. If the average vertex degree in a graph is high, like in dataset PlanTri, we get significantly higher runtimes.

A last interesting datapoint in Table 5.10 is the ELR^{GD}-improvement for dataset PlanTri. We introduced the dataset because we suspected the optimizer to perform badly on the dense graphs generated by the tool. This assumption turns out to be true. The improvement of only 0.57 is far behind the improvements achieved on all other datasets.

6. Conclusion and future work

In this work, we explored a way to optimize the Edge Length Ratio (ELR^{GD}) for planar graphs on the grid. First, we reviewed existing planar embedders and algorithms for drawing planar graphs on the grid. We then evaluated them on their initial Edge Length Ratio and how suited they are to further optimize ELR^{GD}.

After the initial layout, we proposed a local search algorithm working on the drawings generated to optimize the ELR^{GD} by moving the vertices of the drawing. We introduced several measures directly or indirectly linked to the ELR^{GD} of the drawing, the most successful one being the Average Edge Length Distance (AELD). This measure is closely connected to force-directed graph drawing algorithms by treating edges like springs, giving each edge a penalty related to its distance from the average edge length.

The main limitation of this approach is its locality because it can only move one vertex at a time. If we have a very compact and deeply layered drawing, we need to move many vertices before we can achieve a decrease in ELR^{GD}. This does not work well with our concept of a local search because we mostly want to avoid increases in ELR^{GD} altogether.

Future Work

These limits leave options for future research of different kinds. One approach would be implementing a custom layout algorithm capable of generating graph drawings with low ELR^{GD}. This algorithm could also be modified to generate drawings suitable for our local optimizer.

An extension of the optimizer using AELD to a complete force-directed graph drawing algorithm working on a grid is a similar option. This would require adding a repulsive force between vertices and potentially between vertices and edges to avoid drawings with bad angular resolutions. An impulse for this could be the algorithm `ImPrEd` by Simonetto et al. [Sim+11], a force-directed planar graph drawing algorithm *not* working on a grid.

In Section 4.2, we introduced the concept of a weighted comparator for the measure sets. It compares lists of lists of measures by calculating a linear combination of the sublists with a weight list, reducing each sublist to a numerical value, which we then can easily compare. Because of the complexity of optimizing the parameters introduced by this ordering we did not evaluate it during our experiments. Tuning these parameters might enable the optimizer to achieve better results.

In Section 5.2 we extensively discussed why the MM-layout challenges our local search algorithm. This is especially bad because the MM-layout is a very good potential initial layout with an initial ELR^{GD} drastically lower than the ELR^{GD} of all other grid drawing algorithms we evaluated. Analyzing how the limitations of our algorithm regarding the MML can be overcome could result in new possibilities for further optimization of ELR^{GD}.

Improving the performance of our algorithm on drawings generated by the MML must also make use of the bends introduced in the layout. Researching the usage of bends for optimizing ELR^{GD} is also area where further optimizations could be made. Our approach does not make use of bends at all, if the input drawing does not have any bends. A well-placed bend can enable moves that were not possible before. The challenge regarding this is deciding where and when to place these bends.

Bibliography

[Abe+16] Zachary Abel, Erik D Demaine, Martin L Demaine, Sarah Eisenstat, Jayson Lynch, and Tao B Schardl. “Who needs crossings? Hardness of plane graph rigidity”. In: *32nd International Symposium on Computational Geometry (SoCG 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016. DOI: 10.4230/LIPIcs.SoCG.2016.3.

[Aic+14] Oswin Aichholzer, Michael Hoffmann, Marc J van Kreveld, and Günter Rote. “Graph Drawings with Relative Edge Length Specifications”. In: *CCCG*. 2014. DOI: 20.500.11850/96323.

[Bat+98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. 1st. 1998.

[Ben+07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. “The Aesthetics of Graph Visualization”. In: *Computational Aesthetics in Graphics, Visualization, and Imaging*. Ed. by Douglas W. Cunningham, Gary Meyer, and Laszlo Neumann. 2007. DOI: 10.2312/COMPAESTH/COMPAESTH07/057-064.

[BF19] Manuel Borrazzo and Fabrizio Frati. “On the Edge-Length Ratio of Planar Graphs”. In: *Graph Drawing and Network Visualization*. Ed. by Daniel Archambault and Csaba D. Tóth. 2019, pp. 165–178. DOI: 10.1007/978-3-030-35802-0_13.

[BFL20] Václav Blažej, Jiří Fiala, and Giuseppe Liotta. “On the Edge-Length Ratio of 2-Trees”. In: *Graph Drawing and Network Visualization*. Ed. by David Auber and Pavel Valtr. 2020, pp. 85–98. DOI: 10.1007/978-3-030-68766-3_7.

[BM22] Gunnar Brinkmann and Brendan McKay. *plantri and fullgen – Programs for Generation of certain types of Planar Graph*. 2022.

[BO79] Bentley and Ottmann. *Algorithms for Reporting and Counting Geometric Intersections*. 1979. DOI: 10.1109/TC.1979.1675432.

[CDR04] Sergio Cabello, Erik D. Demaine, and Günter Rote. “Planar Embeddings of Graphs with Specified Edge Lengths”. In: *Graph Drawing*. Ed. by Giuseppe Liotta. 2004, pp. 283–294. DOI: 10.1007/978-3-540-24595-7_26.

[Chi+13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. “The Open Graph Drawing Framework (OGDF)”. In: *Handbook of graph drawing and visualization* 2011 (2013), pp. 543–569. DOI: 10.1.1.231.6070.

[CK97] Marek Chrobak and Goos Kant. “Convex grid drawings of 3-connected planar graphs”. In: *International Journal of Computational Geometry & Applications* 7.03 (1997), pp. 211–223. DOI: 10.1142/S0218195997000144.

[Con22] International Graph Drawing Contest. *GD data*. 2022. URL: http://www.graphdrawing.org/data.html (visited on 07/30/2022).

[DPP90] Hubert De Fraysseix, János Pach, and Richard Pollack. "How to draw a planar graph on a grid". In: *Combinatorica* 10.1 (1990), pp. 41–51. DOI: `10.1007/BF02122694`.

[EW90] Peter Eades and Nicholas C. Wormald. "Fixed edge-length graph drawing is NP-hard". In: *Discrete Applied Mathematics* 28.2 (1990), pp. 111–134. DOI: `10.1016/0166-218X(90)90110-X`.

[FPP88] Hubert de Fraysseix, János Pach, and Richard Pollack. "Small sets supporting Fary embeddings of planar graphs". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing.* 1988, pp. 426–433. DOI: `10.1145/62212.62254`.

[GM03] Carsten Gutwenger and Petra Mutzel. "Graph Embedding with Minimum Depth and Maximum External Face". In: *International Symposium on Graph Drawing.* Springer. 2003, pp. 259–272. DOI: `10.1007/978-3-540-24595-7_24`.

[GM98] Carsten Gutwenger and Petra Mutzel. "Planar Polyline Drawings with Good Angular Resolution". In: *Graph Drawing.* Ed. by Sue H. Whitesides. 1998, pp. 167–182. DOI: `10.1007/3-540-37623-2_13`.

[Gra21] Graph Drawing. *The 29th International Symposium on Graph Drawing and Network Visualization.* 2021.

[Gra22] Graph Drawing. *The 30th International Symposium on Graph Drawing and Network Visualization.* 2022.

[Ie22] Max Ivanov and contributors to e-maxx-eng. *Check if two segments intersect - Algorithms for Competitive Programming.* 2022. URL: `https://cp-algorithms.com/geometry/check-segments-intersection.html` (visited on 08/10/2022).

[Kan96] Goos Kant. "Drawing planar graphs using the canonical ordering". In: *Algorithmica* 16.1 (1996), pp. 4–32. DOI: `10.1007/BF02086606`.

[Ker07] Thorsten Kerkhof. "Algorithmen zur Bestimmung von guten Graph-Einbettungen für orthogonale Zeichnungen". PhD thesis. Techn. Univ., Fak. für Informatik, 2007.

[LLL19] Sylvain Lazard, William J. Lenhart, and Giuseppe Liotta. "On the edge-length ratio of outerplanar graphs". In: *Theoretical Computer Science* 770 (2019), pp. 88–94. DOI: `10.1016/j.tcs.2018.10.002`.

[PT00] Maurizio Pizzonia and Roberto Tamassia. "Minimum depth graph embedding". In: *European Symposium on Algorithms.* Springer. 2000, pp. 356–367. DOI: `10.1007/3-540-45253-2_33`.

[Sch90] Walter Schnyder. "Embedding planar graphs on the grid". In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms.* 1990, pp. 138–148.

[Sim+11] Paolo Simonetto, Daniel Archambault, David Auber, and Romain Bourqui. "ImPrEd: An Improved Force-Directed Algorithm that Prevents Nodes from Crossing Edges". In: *Computer Graphics Forum.* Vol. 30. 3. Wiley Online Library. 2011, pp. 1071–1080. DOI: `10.1111/j.1467-8659.2011.01956.x`.

7. Appendix

A. Iteration Plots – Lazy Evaluation

Plots showing the decrease in ELR^{GD} over time, using 20 000 iterations of ELR^{GD} and AEL as their first run and 1000 iteratains of ELR^{GD} and the random measure as their second run. For these graphs, the algorithm was run using lazy evaluation to speed up computations.

B. Iteration Plots – Non-Lazy

Plots showing the decrease in ELR^{GD} over time, using 20 000 iterations of ELR^{GD} and AEL as their first run and 1000 iterations of ELR^{GD} and the random measure as their second run. For these graphs, the algorithm was run without using lazy evaluation.

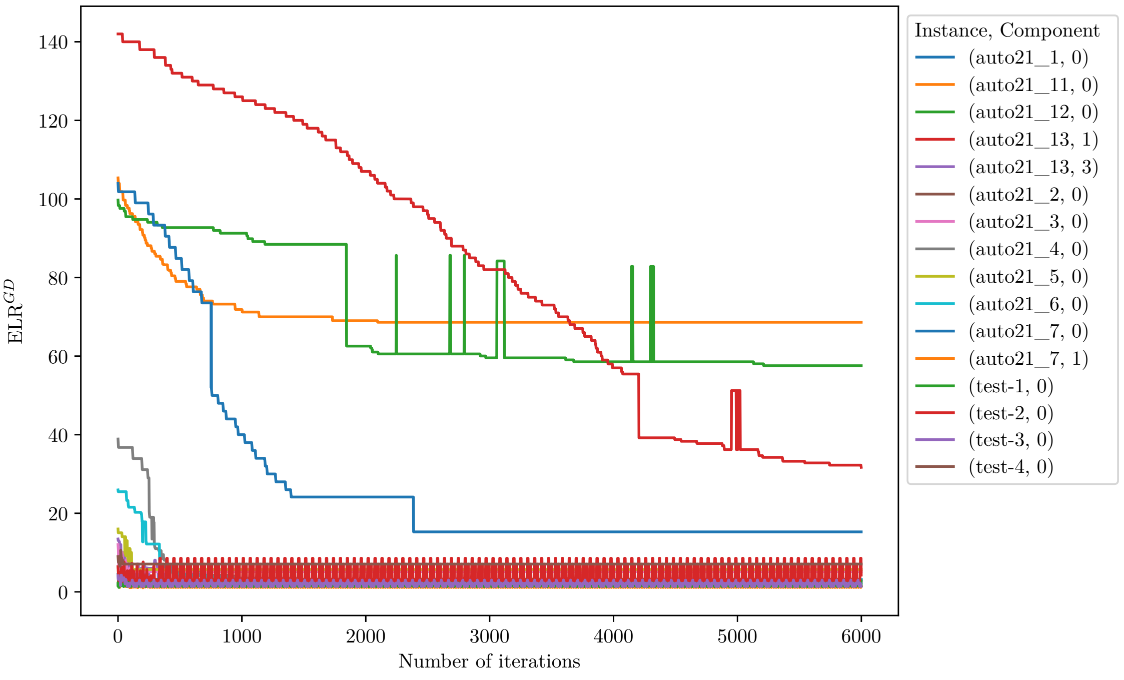


Figure A.1.: The first optimization run, for all graphs with an initial ELR^{GD} below 150.

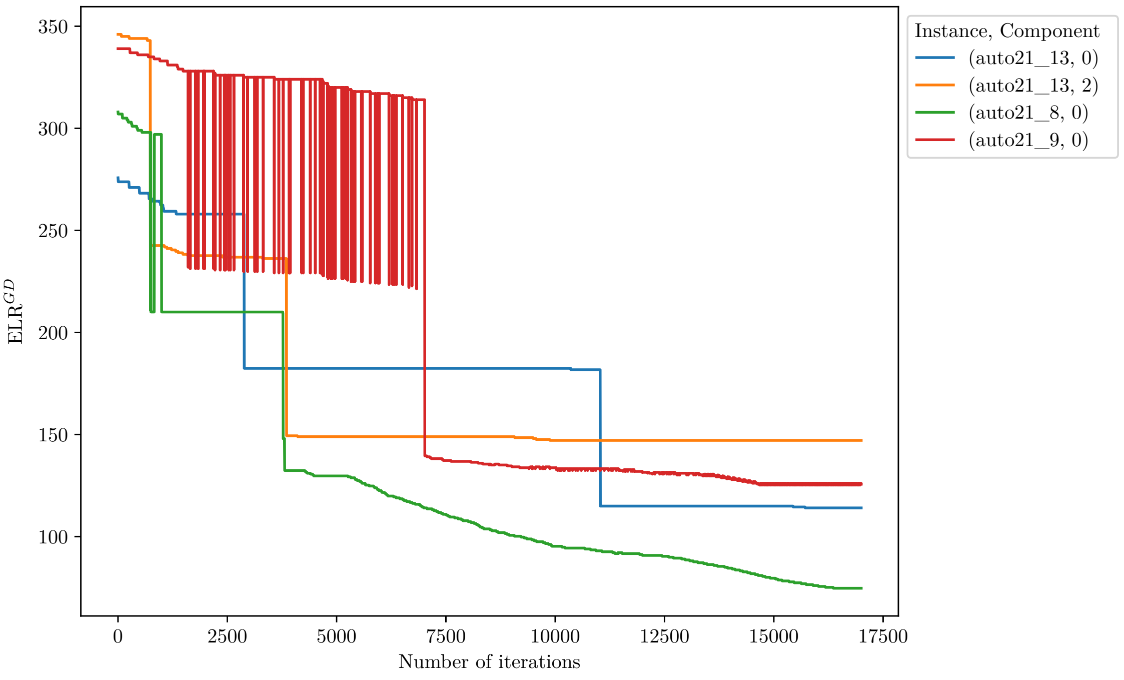


Figure A.2.: The first optimization run, for all graphs with an initial ELR^{GD} at least 150.

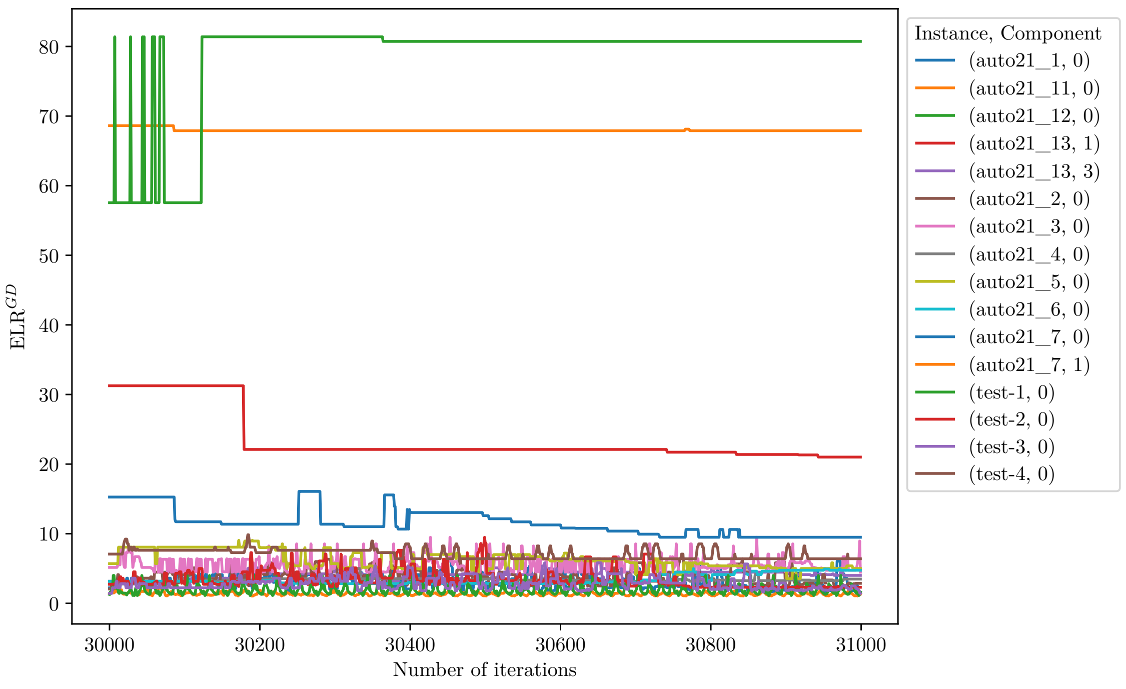


Figure A.3.: The second optimization run, for all graphs with an initial ELR^{GD} below 150.

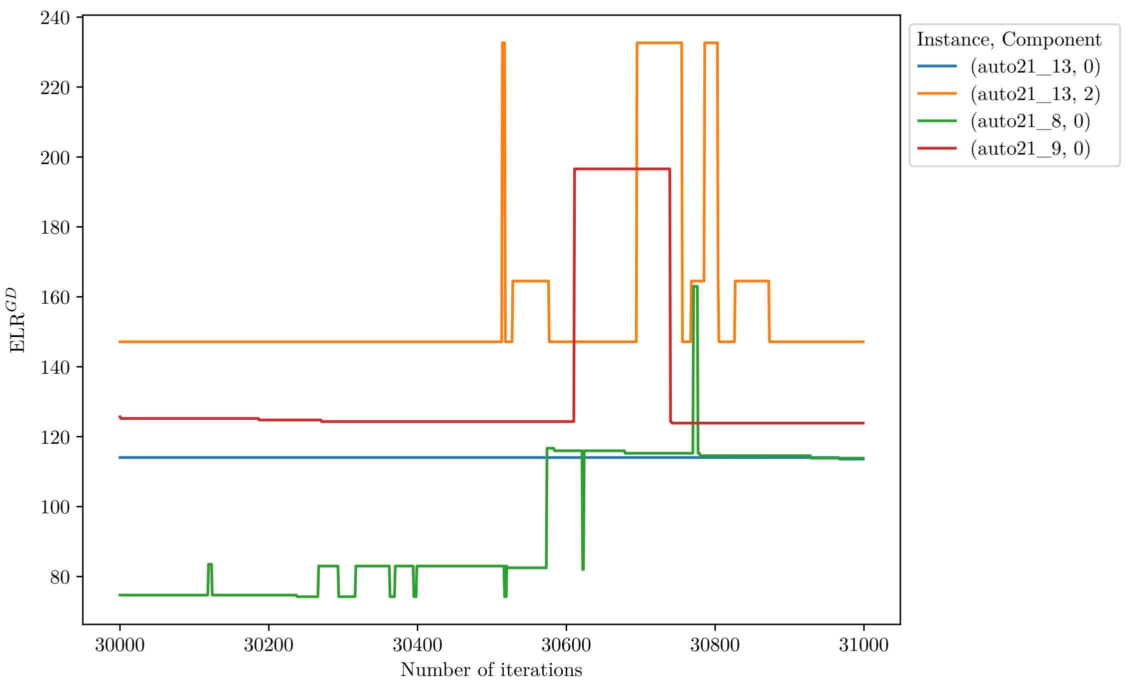


Figure A.4.: The second optimization run, for all graphs with an initial ELR^{GD} at least 150.

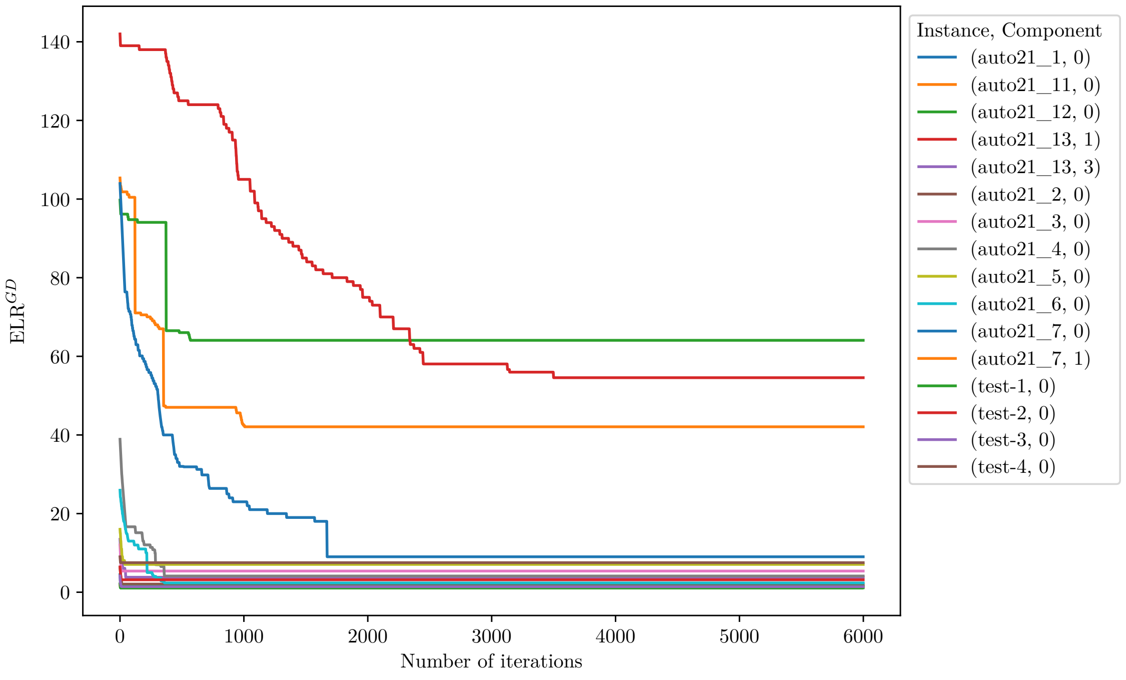


Figure B.5.: The first optimization run, for all graphs with an initial ELR^{GD} below 150.

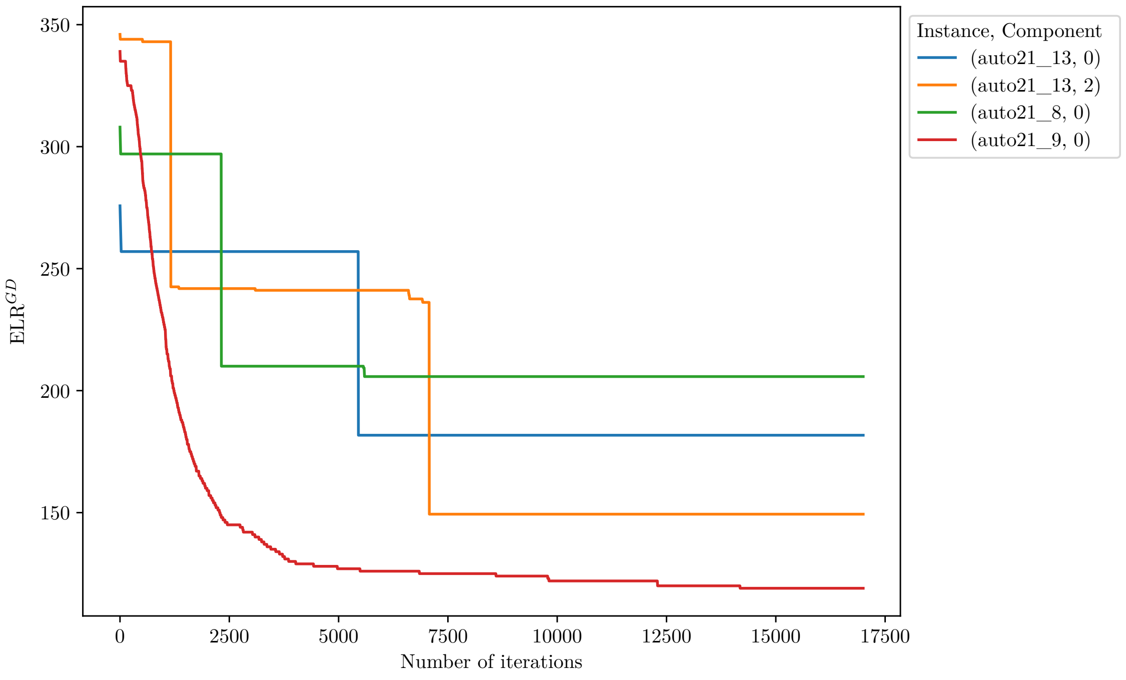


Figure B.6.: The first optimization run, for all graphs with an initial ELR^{GD} at least 150.

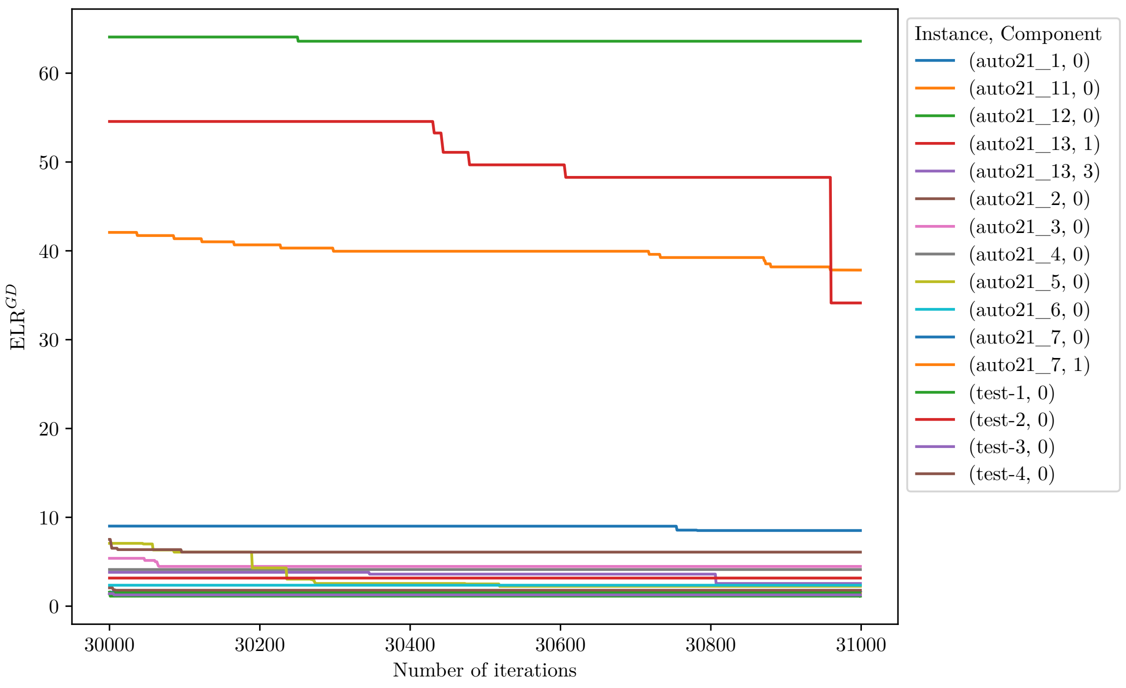


Figure B.7.: The second optimization run, for all graphs with an initial ELR^{GD} below 150.

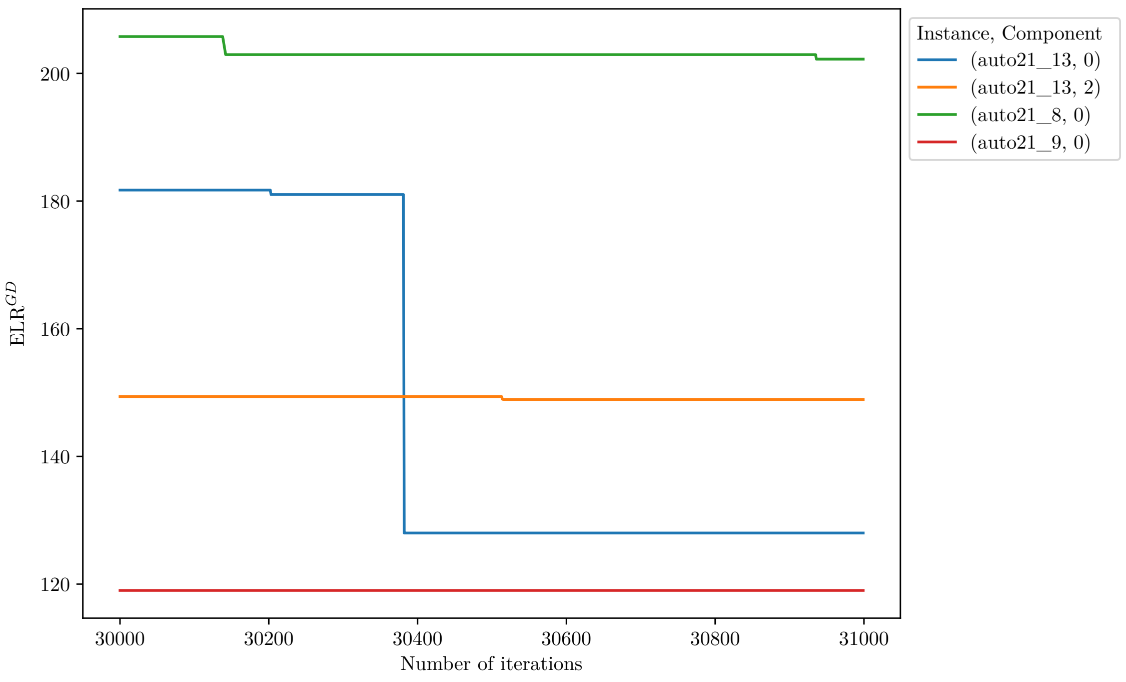


Figure B.8.: The second optimization run, for all graphs with an initial ELR^{GD} at least 150.