Aesthetic Value of Graph Layouts

Investigation of Statistical Syndromes for Automatic Quantification

Master's Thesis of

Moritz Klammler

Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dr. Tamara D. Mchedlidze
	Dr. Alexey Pak

Time Period: September, 15 2017 – March, 15 2018

*

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, March 15, 2018

Abstract

Visualizing relational data as drawing of graphs is a technique in very wide-spread use across many fields and professions. While many graph drawing algorithms have been proposed to automatically generate a supposedly high-quality picture from an abstract mathematical data structure, the graph drawing community is still searching for a way to quantify the aesthetic value of any given solution in a way that allows one to compare graph layouts created by different algorithms for the same graph (presumably to automatically choose the better one). We believe that one promising path towards this goal could be enabled by combining data analysis techniques that have proven useful in other scientific disciplines that are dealing with large structures such as astronomy, crystallography or thermodynamics. In this work we present an initial investigation of some statistical properties of graph layouts that we believe could provide viable syndromes for the aesthetic value. As a proof of concept, we used machine learning techniques to train a neural network with the results of our data analysis and thereby built a model that is able to discriminate between better and worse layouts with an accuracy of 95 %. A rudimentary evaluation of the model was performed and is presented. This work primarily provides an infrastructure to enable further experimentation on the topic and will be made available to the public as Free Software at the place mentioned in the appendix of this document.

Deutsche Zusammenfassung

Relationale Zusammenhänge in Graphen zeichnerisch darzustellen ist eine sehr weit verbreitete Technik in vielen Disziplinen und Anwendungsfeldern. Während zahlreiche Algorithmen vorgeschlagen wurden, die vollautomatisch vergleichsweise gute Layouts für eine gegebene Beschreibung eines Graphen generieren können, besteht bisweilen kein Konsens darüber, wie der ästhetische Wert einer bildlichen Darstellungen eines Graphen unabhängig von einem konkreten Layout-Algorithmus bewertet und verglichen werden kann. Eine solche Anwendung wäre etwa wünschenswert, um unter den Ausgaben zweier Layout-Algorithmen automatisch das bessere Ergebnis auszuwählen. Wir meinen, dass ein möglicher Weg hin zu einer automatischen Quantifizierung des gestalterischen Werts einer Graphzeichnung darin bestehen könnte, Methoden der Datenanalyse, wie sie sich in anderen Disziplinen, die sich mit großen Strukturen befassen – etwa der Astronomie, Kristallographie oder Thermodynamik – bewährt haben, miteinander zu verbinden, und auf Graphzeichnungen anzuwenden. Mit der vorliegenden Arbeit stellen wir eine erste Untersuchung vor, in der wir einige Eigenschaften betrachten, von denen wir meinen, dass sie das Potential haben könnten, als zuverlässige Syndrome für den gestalterischen Gehalt einer Graphzeichnung zu fungieren. Als eine erste Demonstration präsentieren wir ein neuronales Netz, das mithilfe der auf diese Weise gewonnenen und aufbereiteten Daten trainiert wurde, und in der Lage ist, bessere von schlechteren Layouts mit mehr als 95 % Zuverlässigkeit unterscheiden zu können. Ferner liefern wir ein rudimentäre Analyse der Eigenschaften dieses Diskriminators. Der Schwerpunkt dieser Arbeit liegt jedoch auf der Schaffung einer Infrastruktur, die in Zukunft weitere Experimente einfach ermöglichen soll und der Öffentlichkeit als Freie Software an der im Anhang beschriebenen Stelle zugänglich gemacht werden soll.

Contents

List of Algorithms, Figures and Tables

List of Definitions and Theorems

Abbreviations

§	section
a.d.	also known as (*alias dictus*)
algo.	algorithm
ASCII	American Standard Code for Information Interchange
cf.	compare, consult (*confer*)
def.	definition
ed.	edition, editor, edited
e.g.	for example (*exempli gratia*)
eq.	equation
et al.	and others (*et alii*)
fig.	figure
FMMM	Fast Multipole Multi-Level Method
GraphML	Graph Markup Language
HTML	Hypertext Markup Language
ID	identity, unique identifier
i.e.	that is (*id est*)
I/O	input & output
JSON	JavaScript Object Notation
MSE	mean squared error
NIST	National Institute of Standards and Technology
OGDF	Open Graph Drawing Framework
PC	personal computer
PCA	principal component analysis
POSIX	Portable Operating System Interface
pp.	pages
RAM	random access memory
RDF	radial distribution function
ReLU	rectified linear unit
RMS	root mean squared
SGD	stochastic gradient descent
SQL	Structured Query Language
s.t.	such that
tab.	table
TAR	tape archive
thm.	theorem
UI	user interface
URL	uniform resource locator
w.r.t.	with respect / regard to
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Notation

We write $\mathbb{Z}$ for the set of integers (positive, negative and zero alike), $\mathbb{N} = \{1, 2, \ldots\}$ for the set of positive (*a.d.* natural) integers and $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$ for the set of non-negative integers (including zero). The set of real numbers is written as $\mathbb{R}$ with an optional subscript to limit the range. For example, $\mathbb{R}_{>0}$ refers to the positive and $\mathbb{R}_{\geq 0}$ to the non-negative real numbers.

For a set X and integer $k \in \mathbb{N}$ the expression X^k denotes the k-fold Cartesian product $X \times \cdots \times X$ or, in other words, all k-tuples $(x_1, \ldots, x_k)$ with $x_i \in X$ for $i \in \{1, \ldots, k\}$. The expression X^+ denotes the union over all X^n for positive $n \in \mathbb{N}$ and X^* the union over all X^n for non-negative $n \in \mathbb{N}_0$, which includes the empty tuple.

We use the notation $\{\ldots\}$ for ordinary sets and $[\ldots]$ for multisets. (Unlike an ordinary set, a multiset may contain duplicate elements.) For a finite (multi-)set S we write $|S|$ to denote the number of elements in S. The symbol "$\emptyset$" refers to the empty (multi-)set.

We use the infix operators "$\cap$" and "$\cup$" for the conjunction (intersection) and disjunction (union) of sets and likewise "$\wedge$" and "$\vee$" for the conjunction (AND) and disjunction (OR) of logical values respectively.[1] The infix operator "$\setminus$" is used for differences of sets, therefore the expression $A \setminus B$ refers to all elements that are in set A but are not in set B. Logical negation is written using the prefix operator "$\neg$". The expression $\neg(x = y)$ is a convoluted way to write $x \neq y$. Finally, we use the symbol "$\Rightarrow$" to denote implication (in either direction) and "$\Leftrightarrow$" to denote equivalence. $P \Rightarrow Q$ means "P is a sufficient condition for Q" and is equivalent to $\neg P \vee Q$ whereas $P \Leftrightarrow Q$ means "P is a necessary *and* sufficient condition for Q" and is equivalent to $(P \Rightarrow Q) \wedge (P \Leftarrow Q)$.

The quantors "$\forall$" and "$\exists$" are to be read as "for all" and "there exists" respectively. When applied to the empty set, the former is always true while the latter is always false. We use a colon to mean "such that". For example, the expression $\{n \in \mathbb{N}_0 : \exists m \in \mathbb{N}_0 : n = m^2\}$ is a somewhat overly complicated way to define the set of square integers[2] $\{0, 1, 4, 9, 16, 25, \ldots\}$.

The expression $\arg\max_{x \in \mathbb{R}}\{f(x)\}$ is to be understood as that $x \in \mathbb{R}$ for which $f(x)$ is a maximum. $\arg\min$ is defined analogously. For example, $\arg\min_{0 \leq x \leq 2\pi}\{\cos(x)\} = \pi$ because $\cos(\pi) = \min_{0 \leq x \leq 2\pi}\{\cos(x)\} = -1$.

[1] Both uses are Boolean algebra but people seem to prefer different symbols anyway.

[2] N. J. A. Sloane *et al.* "The squares: $a(n) = n^2$". In: *The On-Line Encyclopedia of Integer Sequences.* URL: `https://oeis.org/A000290` (visited on 2018-02-18).

For $x \in \mathbb{R}$ we write $|x|$ for the magnitude of x. The expressions $\lfloor x \rfloor$ (floor) and $\lceil x \rceil$ (ceiling) denote the largest / smallest integer that is not greater / less than x respectively. The expression $\lfloor x \rceil$ denotes the *nearest* integer to x which is commonly defined as "round to even"[3]. The function sign (signum) is defined as $\operatorname{sign}(x) = +1$ if $x > 0$ or $\operatorname{sign}(x) = -1$ if $x < 0$ or $\operatorname{sign}(x) = 0$ if $x = 0$.

Occasionally, we will write informal expressions like $0 < \epsilon \ll 1$ which are to be read as "ϵ is greater than zero but *much less* than one" where it is intentionally left unspecified what "much less" means exactly.

We use bra/ket-notation [10] for vector products. For a Hilbert space V and vector $v \in V$ the expression $\langle v|$ denotes v itself while $|v\rangle$ is the (conjugate) transpose of v. Therefore, the expression $\langle u|v\rangle$ refers to the inner (scalar) and the expression $|u\rangle \langle v|$ to the outer (tensor) product of the vectors u and v. Finally, we write $\|v\|$ to denote the vector norm $\sqrt{\langle v|v\rangle}$ of v.

For an undirected graph $G = (V, E)$ and $v \in V$ the expression $\deg(v)$ refers to the *degree* (number of incident edges) of vertex v.

We use the symbol "$\perp$" to denote missing or undefined values.

We write $x \leftarrow f(42)$ to denote the assignment of the value of the expression "$f(42)$" to the variable x in algorithm listings while trying to avoid variabe reassignment as much as possible.

To describe the asymptotic complexity of algorithms, we use the common "$\mathcal{O}$" notation. For function $f : \mathbb{N} \to \mathbb{R}$ the Landau symbol $\mathcal{O}(f)$ refers to the set of all functions $g : \mathbb{N} \to \mathbb{R}$ for which there exist constants $n_0 \in \mathbb{N}$ and $c \in \mathbb{R}$ such that $|g(n)/c| \leq f(n)$ for all $n \geq n_0$. That is to say, g is *asymptotically dominated* by f. Like most authors, we usually cannot be bothered to write the "argument" of the Landau symbol as a function as in $\mathcal{O}(x \mapsto x^2)$ as it would be correct and will simply write $\mathcal{O}(x^2)$ instead.

[3] E. W. Weisstein. "Nearest Integer Function". In: *MathWorld – A Wolfram Web Resource*. URL: `http://mathworld.wolfram.com/NearestIntegerFunction.html` (visited on 2018-03-12).

1 Introduction

1.1 Motivation

Graphs as data structures and mathematical models of finite binary relations are an ubiquitous concept found in practically every discipline of the arts in one form or another. For human interaction, graphical representations of vertices and edges as connected symbols is by far the most preferred form of presentation and much more approachable, – especially by a non-technical audience – than, say, a representation in matrix form. A good drawing of a graph should not only give a good understanding of the graph's essential structural properties, it should also provide an aesthetically pleasing experience to the beholder. The process of deriving a (usually two-dimensional) graphical representation of an abstract mathematical definition of a graph is called *graph drawing*. While this surely can – and will [35] – be understood as an artistic process, the vast majority of everyday applications must rely on automated procedures for the quick and economic unattended creation of graph drawings with good – or, at least, acceptable – quality.

There is no shortage on graph drawing algorithms[1] today. See for example Tollis *et al.* [56] or Tamassia [52] to mention just two popular books on the topic. However, these algorithms usually tend to "solve their own definition of the problem" which is of course fine except that it may lead to the situation where one can have two (or more) drawings of the same graph which are each "optimal" according to the definition of the algorithm that produced them and human intervention is required in order to settle for a favorite. At the time of this writing, no widely accepted and generally applicable automatically computable quality function is available to the best of our knowledge.

We believe that a reliable way to compare the aesthetic value of graph layouts could be very useful for many applications. An obvious use case would be to run multiple layout algorithms – or the same probabilistic algorithm with different random seeds – in parallel and then pick the result with the highest ranking. Especially for algorithms that produce good quality output in general but fail under certain pathological circumstances, this could prove very helpful. Other possible use cases could be in choosing or optimizing a domain-specific algorithm in a semi-automatic fashion.

An inherent problem of the task outlined so far is of course that the human perception of beauty is not easily expressed as a mathematical quantity. It therefore occurs to us

[1]We will continue to refer to (a subset of) them as *layout algorithms* once we have defined the scope of this work.

that a viable approach towards the problem might be to use elementary concepts that have already proven useful in other disciplines such as astronomy or crystallography. (Which, maybe not purely coincidentally, study objects which many people perceive as aesthetically appealing.) Combining as many of these ideas as possible and analyzing the resulting data might give insights that would not be gained by a purely analytical approach.

We have investigated several properties of graph layouts and found evidence for the hypothesis that they may be reliable syndromes of a drawing's aesthetic value in so far as their statistical analysis lead us to a relatively small number (58) of observables that allowed us to build a model to which we applied machine learning techniques in order to create a discriminator that outputs a preference in favor of either of two layouts. Within our rather limited experiments, we achieved success rates exceeding 95 % reproducibly.

Our methodology involved the automatic generation of a large corpus of labeled data without requiring human intervention. For this purpose, we devised probabilistic graph generators, built a small collection of known-good and *really bad* layout algorithms and crafted data augmentation techniques that allow us to produce even more labeled data. A number of characteristics was computed for all thusly obtained layouts and finally processed into a form that could be used to train a Siamese neural network and used it as a black-box discriminator.

A major part of our contribution as we perceive it was the creation of an open collection of useful command-line tools and a setup that allows for easy experimentation with different sources of graphs and layouts as well as with their statistical syndromes. Last but not least, we developed a web front-end for the convenient inspection of the data. Hopefully, these facilities will be found useful by other researchers, too. Instructions on how to obtain and use the software are given in the appendix.

We would also like to point out that all of our work is completely reproducible to the largest extent we were able to ensure this. We would like to encourage others to actually repeat running our experiments – possibly with variations to some of the parameters – and compare the results. Doing so on a POSIX system is merely a matter of typing a few simple commands (and then waiting very long). As a matter of fact, even this document can be typeset automatically from the sources we provide and will update itself with the current experimental results.

Before we're going to dwell further into the matter, we'll introduce a few definitions that will prove useful for the remainder of this work and also become clear on the scope of this discussion.

1.2 Preliminaries

When speaking about a *graph* in this work, we always mean a *simple graph*, unless explicitly mentioned otherwise. A simple graph is an undirected graph without multiple edges or loops. Furthermore, there are no labels, weights or other attributes associated with the vertices or edges. We will use the terms *vertex* and *node* interchangeably.

In order to produce a two-dimensional graphical representation of a given graph, a necessary step is to assign coordinates to each vertex. This leads us to the following definition.

Definition 1.1 (Vertex Layout): *Given a graph $G = (V, E)$, a vertex layout of G is a function $\Gamma : V \to \mathbb{R}^2$ that assigns a point in two-dimensional Euclidean space to each vertex of G.*

There exist an unlimited number of vertex layouts for any given graph. This is why we will be interested in comparing the aesthetic value associated with each of them. Before that, however, we will have to specify how a vertex layout is presented in a form approachable by the human sense of aesthetics.

Given a graph $G = (V, E)$ together with a vertex layout Γ, a *drawing* of $\Gamma(V)$ is a two-dimensional graphical representation of G obtained by drawing each vertex $v \in V$ as a symbol (such as a circle or square) at position $\Gamma(v)$ and connecting adjacent vertices with straight lines.

Figure 1.1 shows some examples of graph drawings that illustrate the limits of the understanding of a graph drawing within this work's scope. Because this simple understanding of a graph drawing is – apart from global parameters such as the choice of the symbol used for representing a vertex, its color and size as well as the line width and color of lines representing edges – already fully specified by a vertex layout, we will usually omit the "vertex" part and only speak of "layouts". Furthermore, we hope that if a "reasonable" choice is made for the remaining degrees of freedom mentioned, the particular choice won't have a significant influence on the relative aesthetic value of the drawings associated with two vertex layouts as long as the same choices are made for both. That is, we assume that if $\Gamma_1(G)$ looks better than $\Gamma_2(G)$ when both are drawn with a red pen and with circles for vertices, $\Gamma_1(G)$ will still look better than $\Gamma_2(G)$ even if a green pen is used and vertices are drawn as diamonds. Therefore, we will from now on exclusively speak about the aesthetic value of layouts rather than drawings.

Let us also introduce another definition (or notation) here that we will need later.

Definition 1.2 (Graph Distance): *For a graph $G = (V, E)$, the distance between two vertices $\operatorname{dist}(v_i, v_j)$ for $v_i, v_j \in V$ is defined as the length (number of edges) of the shortest path from v_i to v_j if such a path exists or else infinity. It is a non-negative integer or infinity.*

Graph distance is defined on the abstract mathematical concept of a graph. We can define a related property for a layout as follows.

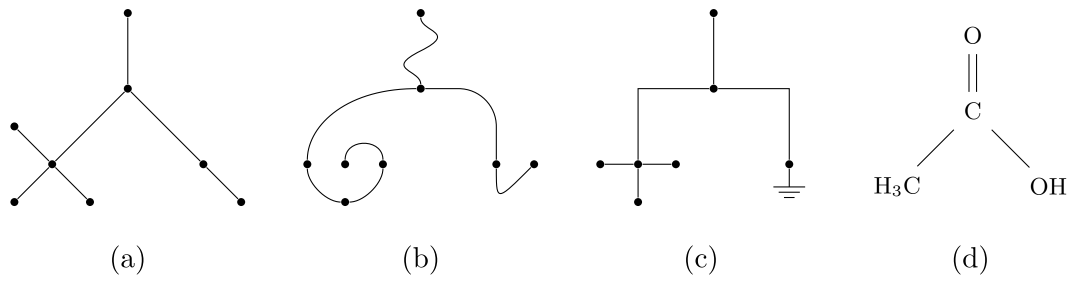


Figure 1.1: The picture in (a) shows an example of a graph drawing as we consider it in this work. Picture (b) is out of scope for this work because the edges have bends. So is picture (c) because the edges have kinks and one node is a ground symbol. Picture (d) is off limits too because the nodes are labeled and the graph has multiple edges in the form of the C=O carbonyl double-bond.

Definition 1.3 (Node Distance): *Given a layout* Γ *for a graph* $G = (V, E)$*, the distance between vertices* $\operatorname{dist}_\Gamma(v_i, v_j)$ *in* $\Gamma(V)$ *for* $v_i, v_j \in V$ *is defined as* $\operatorname{dist}_\Gamma(v_i, v_j) = \|\Gamma(v_i) - \Gamma(v_j)\|$ *and is a non-negative real number.*

Node distance is defined between any two vertices. We can restrict this definition to pairs of adjacent vertices which leads us to the following definition.

Definition 1.4 (Edge Length): *Given a layout* Γ *for a graph* $G = (V, E)$*, the length of an edge* $\operatorname{length}_\Gamma(e)$ *in* Γ *for* $e = \{v_i, v_j\} \in E$ *is defined as* $\operatorname{length}_\Gamma(\{v_i, v_j\}) = \operatorname{dist}_\Gamma(v_i, v_j)$*.*

It is now time to “define away” some uninteresting cases of layouts. Therefore, let us first introduce the following definition.

Definition 1.5 (Degenerated Layout): *A layout* Γ *of a graph* $G = (V, E)$ *is degenerated if and only if* $\Gamma(v) = 0$ *for each* $v \in V$*.*

For this work, we are only interested in layouts that are not degenerated.[2]

There are still several degrees of freedom in a vertex layout according to definition 1.1 that we wish to remove, namely the choice of the center and scale of the coordinate system. It is clear that these have no influence on the aesthetic value whatsoever. Therefore, we introduce the following definition.

Definition 1.6 (Normalized Layout): *A layout* Γ *of a graph* $G = (V, E)$ *with* $V = \{v_1, \ldots, v_n\}$ *and* $E = \{e_1, \ldots, e_m\}$ *is normalized if and only if*

[2] Our implementation also uses the term “degenerated layout” if the layout Γ turns out to be only a partial function, that is, $\Gamma(v) = \pm\infty$ or $\Gamma(v) = \bot$ for one or more vertices v which is already prohibited by definition 1.1 but unavoidable in practice when approximating real numbers with finite-precision floating-point numbers. Some layout algorithms also tend to produce such values under pathological circumstances.

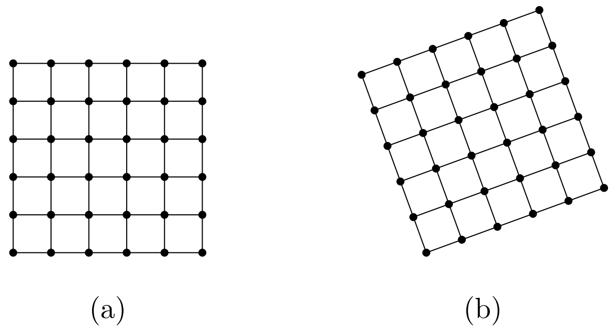


FIGURE 1.2: Example of the visual effect of rotating a layout. The pictures in (a) and (b) show the same layout of the same graph except that (b) is rotated by 20°. It is expected that most people will perceive (a) as more aesthetically pleasing than (b).

- *it is non-degenerate,*
- *the center of gravity* $\frac{1}{n}\sum_{i=1}^{n}\Gamma(v_i) = 0$ *is at the origin and*
- *the average edge length* $\frac{1}{m}\sum_{i=1}^{m}\mathrm{length}_\Gamma(e_i)$ *or – in the case that the graph has no edges at all – the average node distance* $\frac{2}{n(n-1)}\sum_{1\leq i<j\leq n}\mathrm{dist}(v_i, v_j)$ *has the arbitrary value of* 100.

Any given layout can easily be transformed into a normalized layout by subtracting from each vertex position the center of gravity (translation) and applying the appropriate scaling in order to obtain an average edge length (or node distance) of 100.

From now on, we shall assume that all layouts are normalized according to this definition. Note that this precludes layouts of graphs with only zero or one vertices, which is fine with us as there's not much of interest that could be said about those layouts anyway.

A point could be made that definition 1.6 still leaves too many degrees of freedom. For example, it could be argued that layouts should also be normalized according to certain linear transformations such as rotation. In case of doubt, we prefer to be conservative, though, and only apply those normalizations that we are certain cannot have any influence on the aesthetic value. As for rotating a layout, there is even some evidence suggesting that it actually affects the human perception of symmetry.[3] Rather than to nullify these effects by an overly restrictive definition, we wish to cover them in our analysis and maintain the ability to evaluate their influence. It is always safe to err on the side of more degrees of freedom, save for more work to be done during data analysis.

Now we have defined the subject of our study, we can introduce related algorithms. Since most algorithms described in this work are not fully-deterministic, we first need a generalized definition of an algorithm.

[3]See Giannouli [14] and references therein reporting that humans are most likely to detect inflectional symmetry along the vertical axis. Also see figure 1.2 as an illustration.

Definition 1.7 (Probabilistic Algorithm): *A probabilistic algorithm is an algorithm that formally receives an additional implicit input in the form of an unlimited stream of* `0` *and* `1` *bits with an independent probability of* 1/2*.*

In practice, the unlimited stream of random bits is substituted by a fixed number of more or less random bits that are then used to *seed* a so-called *pseudo random generator* which itself is a deterministic algorithm.

Note that the concept of a probabilistic algorithm is a generalization rather than a refinement of a deterministic algorithm. A probabilistic algorithm *may* make random decisions while a deterministic algorithm *must not*. Therefore, each deterministic algorithm is a probabilistic algorithm in the trivial sense that it makes use of zero random bits.

Equipped with this definition, we can define algorithms that lay out graphs.

Definition 1.8 (Layout Algorithm): *A layout algorithm is a probabilistic algorithm that receives as inputs a graph G and outputs a (normalized) vertex layout Γ for G.*

The definition is presented here because it brings together the concepts introduced so far. We will revisit specific layout algorithms in section 5.2 when we discuss our choice of layout algorithms for this work.

1.3 Quantifying the Aesthetic Value of Graph Layouts

It would be foolhardy to attempt a formal definition of the human perception of beauty. Let us therefore define our expectations on an automatic quantifier instead. The most ambitious dream might be to find a function f that can be given a layout Γ and will output a bounded value $0 \leq f(\Gamma) \leq 1$ which measures the layout's aesthetic value in an absolute sense. Comparing two layouts Γ_{A} and Γ_{B} would then just be a matter of comparing the scalar values $f(\Gamma_{\mathrm{A}})$ and $f(\Gamma_{\mathrm{B}})$. We strongly believe that such a function does not exist and searching for it likely a waste of time, even if one should not require that the function's image is confined to a closed interval.

Instead of looking for an absolute measure of aesthetic value, we'd be more than happy if we could find an algorithm that computes a relative order of the aesthetic value. But even there the question remains whether the human perception of beauty follows properties that are mathematically required for an ordering relation. Most notably, transitivity.

The scope we're going to set out for this work is to find an approximation of a partial order. A *partial order* is a binary relation that is reflexive, antisymmetric and transitive. While an ordering relation is defined to be only either true or false, we consider it more practical and useful to have a function that outputs a *certainty* which allows for a continuous approximation. We can always take the signum of such a function if we want a binary decision.

Formally, if G is a graph and Γ_A and Γ_B are two layouts for G then we want a function f such that $-1 \leq f(\Gamma_\mathrm{A}, \Gamma_\mathrm{B}) \leq +1$ with the interpretation that $f(\Gamma_\mathrm{A}, \Gamma_\mathrm{B}) < 0$ indicates a preference in favor of Γ_A while $f(\Gamma_\mathrm{A}, \Gamma_\mathrm{B}) > 0$ indicates a preference in favor of Γ_B and $f(\Gamma_\mathrm{A}, \Gamma_\mathrm{B}) \approx 0$ means that the answer is unclear. We shall use this interpretation consistently throughout this work. It is obvious that any linear mapping to another interval is trivial and can be done freely at will and convenience.

1.4 Overview of our Contribution

The remainder of this work is organized as follows.

- Chapter 1 motivated the work, defined its scope and introduced some basic concepts and provided a short overview.
- Chapter 2 gives a quick overview over existing research on the topic.
- Chapter 3 discusses our methods in more detail.
- Chapter 4 introduces the properties we have investigated and provides reasoning why we believe that they can be reliable syndromes of aesthetic value of a graph layout. At this point, a *property* is a multiset of scalar values with an unbounded number of elements.
- Chapter 5 provides a detailed discussion how we obtained our test data (graphs and layouts).
- Chapter 6 explains in detail how we applied data augmentation techniques to our data set in order to gain even more samples.
- Chapter 7 is devoted to the exfiltration of a small fixed-sized feature vector from the collection of properties and mentions some of the problems we encountered along the way.
- Chapter 8 describes the structure of our discriminator model and why it was chosen that way. It also addresses the issues of training and testing the model.
- Chapter 9 presents the experimental results of our (limited) empirical evaluation.
- Chapter 10 draws some final conclusions and mentions a lot of ideas that we were unable to peruse within the scope of this work but consider worthwhile nonetheless.
- The appendix provides information how to obtain, compile and use our software. Either for the purpose of verifying our experiments or conducting new ones.

2 Related Work

2.1 Quality Measures

Purchase [41] was probably the first to suggest seven simple measures for the aesthetic value of a layout that include factors like the number of crossings or orthogonal angles.

Koren and Çivril [31] proposed a modification of the stress function originally introduced by Kamada and Kawai [25] and coined the term “binary stress” for it.

Huang *et al.* [22] addressed a very similar problem than we do, namely that “many automatic graph drawing algorithms implement only one or two aesthetic criteria since most aesthetics conflict with each other” and conclude that their “study indicates that aesthetics should not be considered separately” and rather try “improving multiple aesthetics at the same time, even to small extents”. The measures they consider are those suggested by Purchase. Their major contribution is a force-directed layout algorithm that has an energy function crafted specifically to give consideration to all metrics they mention.

Klapaukh [28] has suggested a more complicated measure based on the identification of axes of symmetry.

Huang, Huang, and Lin [21] in 2016 suggested a very simple aesthetic relation that can be applied to any given set of layouts of the same graph by combining a few scalar properties – for which they suggest basically those proposed by Purchase – via subtracting the mean and dividing by the standard deviation of this quantity for all layouts in the set and then forming the sum. The resulting number is only meaningful with respect to the other layouts in the same set.

2.2 Machine Learning Techniques

Masui [36] applied machine learning techniques to the field of graph drawing as early as 1994. They designed an interactive system in which a user first labels a set of layouts and this information is then used to parameterize a model that will in turn employ an evolutionary algorithm in order to improve an existing or derive a new layout. Their paper does not mention how large their largest graphs were but it seems like they consisted only of very few (probably on the order of a dozen) vertices. Otherwise, a training set of $N = 22$ as they used it could barely be sufficient. Their model is formulated using several high-level constraints which incorporate some assumptions about good layouts that are most likely universally true. The paper mentioned that the approach was computationally expensive.

More than 20 Years later, a 2015 review article by Santos Vieira, Nascimento, and Silva [46] records that “Surprisingly, only a few pieces of research can be found about this subject.” This paper cites a few other references of works where direct user interaction was used in order to adjust a parameterized fitness function in real-time. The review also mentions other approaches towards unattended application of machine learning, although in a different sense than we use it, employing neural networks to approximate hard optimization problems [20] and using these results again to compute a layout via an explicit algorithm (in this case minimization of edge crossings via planar embeddings [8]).

Meyer [38] presented a very interesting approach in which they did not “use an external network structure ‘to learn the graph’, instead the graph itself will be turned into a learning network.” By constructing a neural network with the same topology as the graph to be laid out, they achieve that the neuron weights derived during training can be used to derive vertex positions. Given that evaluating a neural network is very fast but training it takes time, this might not be as cool as it looks at first sight.

Bach *et al.* [2] designed an “interactive random graph generation with evolutionary algorithms” that looks like it could have been useful for us too, had we discovered it earlier.

A very recent work of Kwon, Crnovrsanin, and Ma [32] is closely related to ours even if quite different. Here the authors used machine learning techniques not to – as we do – process information about a given layout, but rather *infer* what the properties of the layout *would be* if it were to be computed using a given algorithm. Their main interest seems to lie in the number of edge crossings. This approach allows the authors to consider many more layouts in the same time than it would be possible if the layouts actually were to be computed. They can then use this information to decide what layouts they actually want to compute.

Apart from this last paper, we were not able to find any relevant publications on the topic that are not already mentioned in the review article by Santos Vieira *et al.* during our, admittedly superficial, literature survey.

2.3 User Studies

Purchase [42] conducted a user study that evaluated people’s ability to *understand* a graph – which is not necessarily the same as liking it – and concluded that “reducing the number of edge crosses is by far the most important aesthetic”. Other studies [57] have come to different conclusions.

Welch and Kobourov [58] conducted a study to compare the metrics suggested by Purchase, Klapaukh and Koren *et al.* and concluded that the last one (binary stress) is the most realistic measure available today.

3 Methodology

We wish to provide evidence for the correlation of some statistical properties of graph layouts and their aesthetic value. These syndromes should be easy to compute and be as generic as possible, such as to avoid encoding implicit assumptions into the feature vector as much as possible and instead derive syndromes for aesthetic value from first principles that are applicable to a large variety of graphs and layouts.

To this end, our intention is to collect a large corpus of graphs and several "labeled" pairs of layouts for each of them. A labeled pair of layouts for a graph G is a triple $(\Gamma_{\mathrm{L}}, \Gamma_{\mathrm{R}}, t)$ where the number $-1 \leq t \leq +1$ indicates which layout is the better one and to what degree so. If $t = -1$ we are most certain that Γ_{L} is better, if $t = +1$ we are most certain that Γ_{R} is better and if $t \approx \pm 0$ we have no preference.

For each layout, we compute *properties* that we believe are viable syndromes of the layout's aesthetic value (*cf.* § 4 for a detailed discussion of these). All properties are unordered collections of events (such as the sequence of edge lengths). In order to analyze their distributions, a few statistical parameters (such as arithmetic mean, root mean squared and, in particular, entropy) are computed. This gives us a feature vector of fixed small size regardless of the graph's size. The feature extraction process is explained in detail in chapter 7.

Armed with this data, we seek to find interesting correlations between feature vectors and expected layout quality. Since no single investigated feature seemed to be usable as a reliable metric in isolation, we decided to apply machine learning techniques in order to build a discriminator model that would receive a pair of layouts as inputs and outputs a prediction p for the value t. We count a prediction as a success if $\mathrm{sign}(p) = \mathrm{sign}(t)$.

Ideally, we would independently generate many layouts for each graph and then query an eternal source of truth in order to label each pairwise combination. In the absence of such a source, human labeling might be applied, for example by conducting a large-scale user study where random pairs of layouts are shown to participants which are then asked to express their preference for one layout or the other. Finally, p could be estimated by subtracting the votes in favor of the first layout from those in favor of the second and dividing the resulting number by the total number of votes. Unfortunately, such a study was beyond the resources and time-frame available for this work.

Hence, we used a technique called *data augmentation* to obtain labeled layout pairs from a priori knowledge about the way a layout was created. Unfortunately, we don't know to rate pairs of layouts if both were produced by a state-of-the-art layout algorithm or one is a native layout (see below). The only thing we know is that such layouts are probably "good" to some extent. We shall refer to those as *proper layouts* from now on.

As a first step towards labeled pairs, we generate additional layouts in the dumbest conceivable manner, basically laying out nodes randomly (the exact procedure is explained and discussed in sections 5.2.4 and 5.2.5). Those layouts we will refer to as *garbage layouts*. We assume that any combination of a proper and a garbage layout is 100 % in favor of the proper layout. Two garbage layouts would be considered of equal quality but we found it sufficient to compute only a single garbage layout per graph anyway.

The introduction of garbage layouts provides a convenient "zero point" of an example of the worst possible layouts. However, it would not be very representative to compare only extremes. Therefore, let us introduce the following concept.

Definition 3.1 (Unary Layout Transformation): *A unary layout transformation is a probabilistic algorithm that receives as inputs a graph G together with a layout Γ of G and a parameter $0 \leq r \leq 1$ and outputs a Layout Γ' for G. The interpretation of the parameter r is up to the algorithm, however $r = 0$ shall imply $\Gamma' = \Gamma$.*

For *layout worsening*, we define a number of unary layout transformations that distort a given layout to a configurable degree (controlled by the parameter r). Applying such transformation to a proper layout Γ (referred to as the *parent layout*) with parameters $r \in \{r_1, \ldots, r_n\}$ yields $(n+1)^2$ triples $(\Gamma_i, \Gamma_j, t_{ij})$ for $i, j \in \{0, \ldots, n\}$ using $\Gamma_0 = \Gamma$. For these we assume that $\operatorname{sign}(t_{ij}) = \operatorname{sign}(r_i - r_j)$. Applying worsening to a garbage layout is not assumed to have any effect on the quality of the layout in either direction.

A second data augmentation strategy involves computing new layouts with an anticipated quality from two existing layouts. Let us provide the following definition first.

Definition 3.2 (Binary Layout Transformation): *A binary layout transformation is a probabilistic algorithm that receives as inputs a graph G together with an ordered tuple of two layouts Γ_A and Γ_B of G and a parameter $0 \leq r \leq 1$ and outputs a Layout Γ' for G. The interpretation of the parameter r is up to the algorithm, however $r = 0$ shall imply $\Gamma' = \Gamma_\mathrm{A}$ and $r = 1$ shall imply $\Gamma' = \Gamma_\mathrm{B}$.*

For *layout interpolation*, we define binary layout transformations that interpolate between two layouts. The similarity of the interpolated layout to either layout is controlled by the parameter r. Applying such interpolation to a pair of layouts Γ_A and Γ_B (referred to as the *parent layouts*) with parameters $r \in \{r_1, \ldots, r_n\}$ yields $(n+2)^2$ triples $(\Gamma_i, \Gamma_j, t_{ij})$ for $i, j \in \{0, \ldots, n+1\}$ using $\Gamma_0 = \Gamma_\mathrm{A}$ and $\Gamma_{n+1} = \Gamma_\mathrm{B}$. For these we assume that $t_{ij} \approx (r_j - r_i) t_\mathrm{AB}$ where t_AB is the estimated result of comparing the parent layouts.

This is not of much help if we don't know how to rank Γ_A and Γ_B in the first place (*i.e.* $t_\mathrm{AB} = \perp$). However, if one layout is a proper layout and the other is a garbage layout, interpolation provides us with a fine-grained series of intermediate steps in addition to the blunt comparison of extremes.

The design space for the approach outlined so far is vast. There are countless ways to generate graphs and layouts as well as unary and binary transformations that could be applied to them for the sake of worsening and interpolation. Not to mention the number of properties that could be computed for populating the feature vector. In practice, graphs often have a certain structure specific to the application domain. For example, social graphs will possess a different structure than network graphs in computer communication networks. It would be overly ambitious to hope that our investigation will provide optimal answers for each of these domains. Rather than aiming for this unrealistic and unverifiable goal, we take care to keep our setup flexible enough to make it easy to adapt to real-world applications if domain-specific knowledge is available. The classes of graphs we have investigated are by no means comprehensive but merely an example data collection without any specific practical application. By designing our framework as a collaborative ensemble of small independent tools, it is easy to add or remove graph generators, layouts, layout transformations and feature extractors. The remainder of this work describes the implementation of these components used for our study. Since the experiment is fully automated, replicating it with different components is simply a matter of adding or removing individual components and re-running the experiment. The individual tools communicate with each other through simple command-line interfaces which does not tie them to a particular technology. An overview of the data-flow pipeline is given in figure 3.1 and will be described briefly in the remainder of this section.

Graph generators, layout algorithms and transformations and feature extractors are implemented as small independent programs written in C++ for performance using the *Open Graph Drawing Framework (OGDF)* [7] as a library for graph data structures and algorithms, in particular, layout algorithms. The neural network is built with Keras [26] using the TensorFlow [53] library as back-end. The individual components are connected together and orchestrated by a driver script written in the Python programming language. The driver has some knowledge of each tool, such as the command-line options it has to be passed. This means that addition of a new tools requires a small modification to the driver script, which is as simple as adding a new enumeration constant and specifying an executable name and any additional command-line parameters, if any.[1] Data is stored in and retrieved from a relational database (provided via SQLite) by the driver script. Graphs, layouts and data series are stored as individual text files on the file system and referenced from the database. For graphs and layouts the *Graph Markup Language (GraphML)* [5, 54] format[2] is used. Data inspection and queries to the neural network

[1]It is usually not necessary to use specific command-line arguments as the tools ought to adhere to some common conventions, such as to accept the files to be read as final arguments and allow specifying the output file via the `--output=`*`FILE`* option. However, the additional flexibility provided by supporting tool-specific command-line parameters allows implementing several tools in a single executable. For example, the `MOSAIC1` and `MOSAIC2` generator (*cf.* § 5.1.5) are implemented in the same executable and selected via passing or not passing the `--symmetric` option.

[2]Which is an XML (text). Our tools additionally support transparent compression in order to preserve disk space.

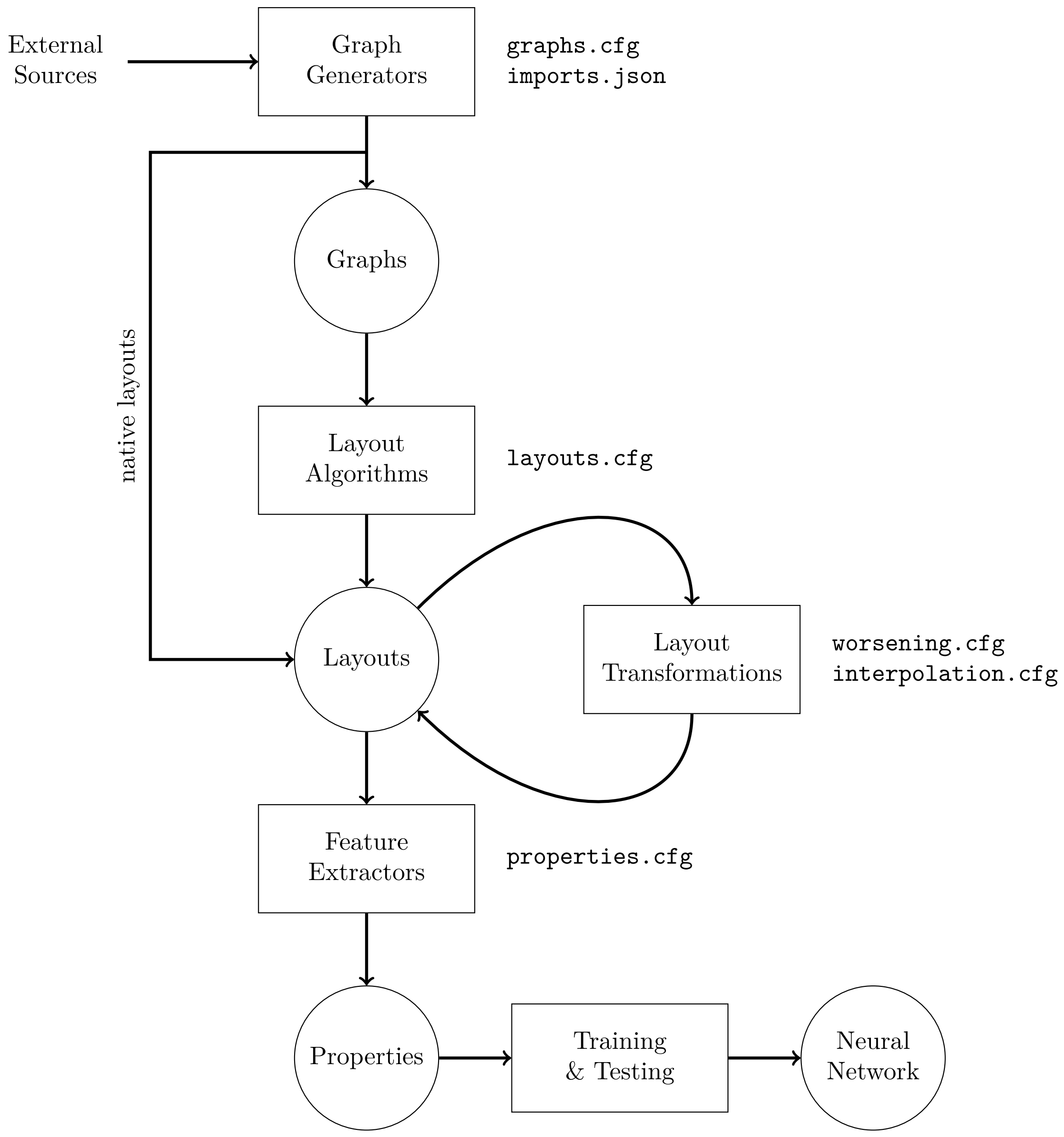


FIGURE 3.1: Overview of the pipeline used. Circles represent data assets while rectangular boxes represent collections of tools. Arrows indicate data flow (as managed by the driver script). File names in monospace font refer to the configuration files that can be used to control the selection of tools to be used without modifying the driver script.

are provided via a web interface that is written in a mixture of Python, XSLT JavaScript and HTML.

Besides using the driver script and web interface, the individual tools we've written may also be used in isolation; either alone or as I/O filters for quick experimentation. For example, the shell commands

```
$ mosaic --symmetric --nodes=1000 --output=sample.xml.bz2
$ picture --output=sample.svg sample.xml.bz2
```

create a random symmetric "mosaic" graph (*cf.* § 5.1.5) with approximately 1 000 nodes and save it as GraphML file `sample.xml.bz2` with `bzip2` (Burrows-Wheeler) compression applied. The file is then read again on the second line and a graphical rendition of the layout is saved as file `sample.svg`.

The shell pipeline

```
$ wget -q -O - ftp://math.nist.gov/.../bcspwr01.mtx.gz                \
    | import --format=matrix-market --simplify --meta=2 STDIO:gzip   \
    | force --algorithm=fmmm                                         \
    | edge-length --kernel=boxed --output=histogram.txt
{ "nodes": 42, "edges": 85, "native": false, ... }
```

downloads[3] (using the standard command-line utility `wget`) a graph from NIST's "Matrix Market" [3] as `gzip` (Lempel-Ziv) compressed file, simplifies[4] the graph and converts it to the preferred GraphML format, then computes a force-directed layout for the graph and finally analyzes the distribution of edge lengths in that layout, saving a histogram as text file `histogram.txt`. The command given the `--meta=2` option will print additional information to standard error output (selected by the POSIX file descriptor 2) in JSON format which is partially shown in the above snippet after the command prompt. (It cannot be printed to standard output which is already used for the pipeline or it would be invisible and clobber the graph data.) The histogram file could be plotted using a tool like `gnuplot`. The last program could (and probably should) also be instructed to output additional information like the mean or entropy in JSON format using the `--meta` option again which was omitted in the example for the sake of brevity.

We hope that this collection of command-line tools will prove useful to other researchers as well. A more detailed documentation of their usage is given in the appendix. All command-line tools also support the `--help` option to quickly print a short usage summary.

[3]The URL had to be shortened in the snippet to make it fit on a line: `ftp://math.nist.gov/pub/MatrixMarket2/Harwell-Boeing/bcspwr/bcspwr01.mtx.gz` (tested on 2018-02-22)

[4]This operation makes edges undirected and deletes loops and fuses multiple edges into one.

4 Statistical Syndromes of Graph Layouts

Aesthetic value is not a mathematical or physical quantity that can be measured directly. Hence, we're out on a survey for things that *can* be measured quantitatively and ideally tell us something about the aesthetic value of a graph layout. Unlike with previous approaches to the topic, our investigations are not primarily driven by the desire to find scalar quantities that are supposed to be mini- or maximized in order to obtain good layouts. Rather, we look for *symptoms* of aesthetic value. A *syndrome* is a collection of symptoms that might not all be present at the same time in the same intensity. By including as many syndromes as possible in our toolbox, we hope to get a more comprehensive view of the problem and therefore, ideally, will eventually become able to make decisions that are general and "robust" in the sense that they are not subjected to *a priori* assumptions that might not hold in all cases.

In this chapter, we will introduce several *properties* of graph layouts that we investigated and believe to be viable candidates for reliable syndromes of aesthetic value. Mathematically, each property is a multiset of scalar values that can be computed for a graph layout. We will refer to the elements of those multisets as *events*.

We will introduce the mathematical definitions of the properties in the present chapter and discuss how they can be computed efficiently. Furthermore, we will present examples of different graphs and layouts and demonstrate how said properties behave for them.

In order to visualize properties, we will use density plots which show the event density as a function of the event magnitude. We will omit the scale and units of the ordinate in our plots because the concrete numeric values are not really important for a general understanding. The plots were obtained by computing sliding averages using a Gaussian kernel. (The width of which will be indicated in the diagrams.) We will discuss this process further in section 7.3. For the purpose of the discussion in this chapter, it is not important how the diagrams are obtained but rather what they visualize. We will revisit the acquisition of meaningful diagrams in chapter 7 when we shall discuss how the properties presented in this chapter can be turned into inputs for a machine learning algorithm.

4.1 Principal Components (`PRINCOMP1ST` and `PRINCOMP2ND`)

The first property we want to discuss is the distribution of vertices on the drawing pane. Most likely, this is an elementary property of a vertex layout. It is to be expected that the coordinate distribution of a regular drawing exposes distinctive peaks at certain intervals while a random arrangement of nodes features a more or less smooth distribution.

In order to investigate the coordinate distribution, we perform a *principal component analysis (PCA)*. See Abdi and Williams [1] for a brief or Jolliffe [24] for a thorough introduction to the topic.

Definition 4.1 (Principal Component): *Let $X \subset \mathbb{R}^n$ for $n \in \mathbb{N}$ be a finite point cloud (multiset) with center*

$$\bar{x} = \frac{1}{|X|} \sum_{x \in X} x \ . \tag{4.1}$$

The unit vector

$$p^{(1)} = \operatorname*{arg\,max}_{v \in \mathbb{R}^n \ s.t. \ \|v\|=1} \left\{ \sum_{x \in X} \langle v | x - \bar{x} \rangle^2 \right\} \tag{4.2}$$

is the first principal component of X. Let $p^{(1)}, \ldots, p^{(k)}$ for $k < n$ be the first k principal components of X. Then the next principal component is the unit vector $p^{(k+1)} \in \mathbb{R}^n$ satisfying equation 4.2 with the additional requirement that it is linear independent of $p^{(1)}, \ldots, p^{(k)}$.

The principal components form an orthonormal basis with the property that the moment of inertia is largest along the first axis, second largest along the second axes and so forth. PCA is therefore usually used in order to reduce the dimensionality of empirical data. In our case, however, we have only two-dimensional data to begin with so we merely use PCA to find the direction of the principal components which might tell us something about the layout. In particular, we are interested in the *distribution* along the principal components. PCA is often employed under the assumption that the data follows a normal distribution. In our case, however, we know that this will most certainly not be the case[1] so we hope to gain valuable insights from analyzing the distribution.

Because of the geometric application, we will often speak of principal *axes* instead of principal *components* where we will use the terms *major* and *minor* axis when referring to the first and second principal component respectively. For visualization, we multiply the unit vectors with the standard deviation of the data set along their direction.

The `PRINCOMP1ST` property analyzes the distribution of vertex coordinates along the major and the `PRINCOMP2ND` property along the minor principal axis. More precisely, given a layout Γ for graph $G = (V, E)$, where $p^{(1)}$ and $p^{(2)}$ are the first and second

[1]Unless the layout was generated by `RANDOM_NORMAL` (§ 5.2.4) of course.

principal component of $\Gamma(V)$ respectively, the two properties consider the following multisets.

$$\texttt{PRINCOMP1ST} = \left[\left\langle p^{(1)} \middle| \Gamma(v) \right\rangle : v \in V \right] \tag{4.3}$$

$$\texttt{PRINCOMP2ND} = \left[\left\langle p^{(2)} \middle| \Gamma(v) \right\rangle : v \in V \right] \tag{4.4}$$

The significance of these distributions is best understood by an example which can be found in figure 4.1.

In order to actually compute the principal components, we use power iteration and then use Gram-Schmidt orthonormalization[2] in order to find the second component. Repeated application of the Gram-Schmidt process is numerically unstable [15] but since we only apply it at most once, this should not be a problem for us. The algorithm – which is not particularly clever but was easiest to implement – is shown in algorithm 4.2. The computational cost for this property is basically linear in the number of nodes.

4.2 Angles Between Incident Edges (ANGULAR)

Another simple property is the distribution of angles between the edges incident to a vertex. Figure 4.3 illustrates how these angles are defined. Let Γ be a layout for graph $G = (V, E)$ and let ϕ_Γ be a function that assigns to each vertex its (possibly empty) multiset of angles as defined below and in figure 4.3. Then the property at hand can be defined as

$$\texttt{ANGULAR} = \bigcup_{v \in V} \phi_\Gamma(v) \ . \tag{4.5}$$

One would expect again that a high-quality layout features less variation in the angular distribution. Ideally, the only angles that appear should be those fractions $2\pi/n$ where there is a vertex with degree n in the graph.

In order to determine the angles, the polar angles for each incident edge is computed first. These numbers are then sorted and formed to a cyclic list. The adjacent differences between the list elements define the angles between the incident edges around the node. As a special rule, a vertex with degree 1 contributes an angle of 2π rather than 0 so it can be distinguished from the case where two incident edges have the exact same polar angle (which can only happen if the adjacent vertices coincide). Another corner case arises if an edge has length zero in which case its polar angle is undefined. There are three possible ways to deal with this case.

- Abort the computation and report an error.
- Use a special “not a number” value.

[2] E. W. Weisstein. “Gram-Schmidt Orthonormalization”. In: *MathWorld – A Wolfram Web Resource*. URL: `http://mathworld.wolfram.com/Gram-SchmidtOrthonormalization.html` (visited on 2018-03-06).

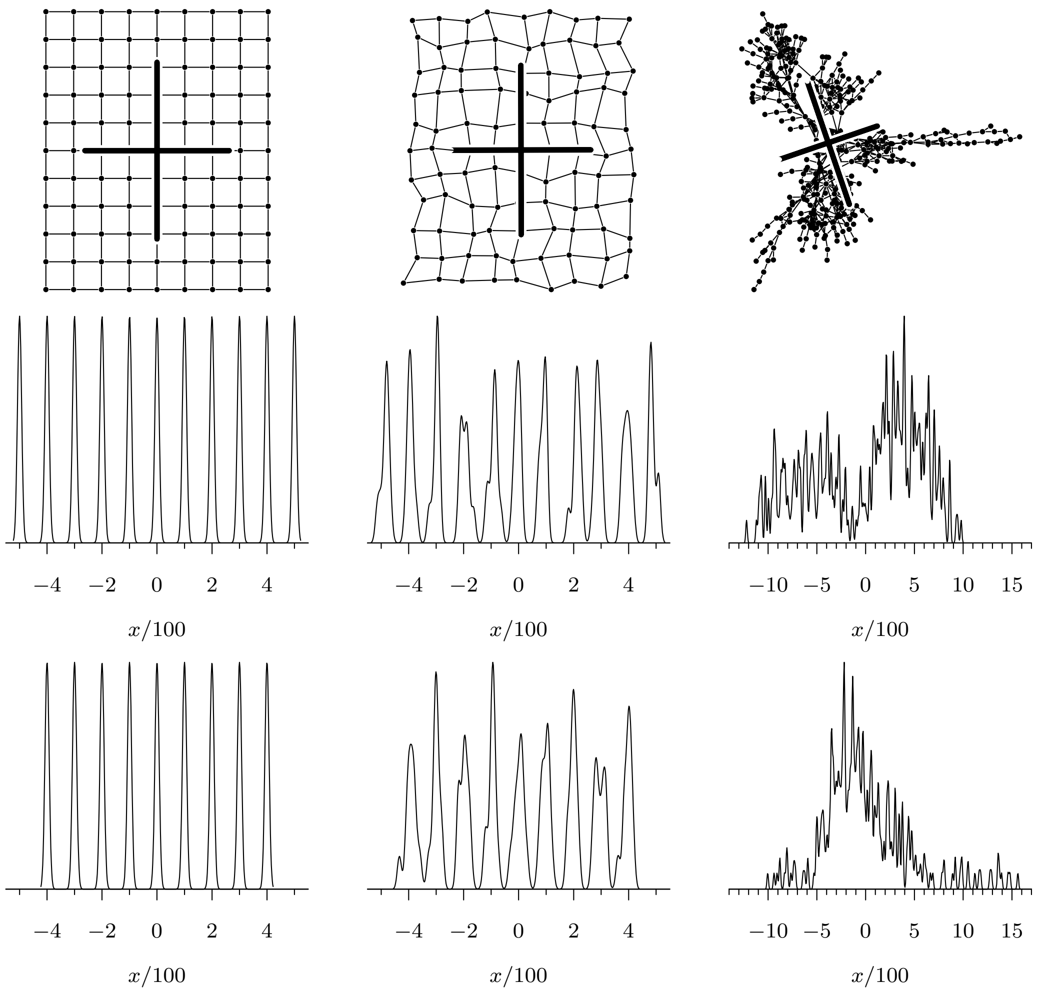


FIGURE 4.1: Distributions of node density analyzed via PRINCOMP1ST and PRINCOMP2ND for three layouts illustrated. The top row shows the graph drawing with superimposed principal components (multiplied with the respective standard deviation). The second and third row show the density function for the PRINCOMP1ST and PRINCOMP2ND property respectively. The left column corresponds to a perfectly regular grid. Its distribution function is a comb of very sharp peaks at unit distance. In theory, the spikes would be Dirac delta functions but the diagram was created by moving a Gaussian filter over the data so the peak width is finite. The middle column shows a layout for the same graph but this time less regular. The distribution still shows peaks roughly at the average node distances but they are broader (even though the same filter width was used for all diagrams) and less pronounced. The right column shows a force-directed layout of a power-network pattern taken from the BCSPWR set of the Harwell-Boeing Collection in NIST's Matrix Market [3]. One can identify the two larger clusters along the major axis and a relatively compact non-symmetric distribution along the minor axis. Especially towards the ends of the spectrum, small peaks at fairly regular intervals corresponding to the nodes in the "strand" that extends to the right are visible.

```
INPUT: Point cloud $X = \{x_1, \dots, x_m\} \subset \mathbb{R}^n$ and integer $k \leq n$.
OUTPUT: The first $k$ principal components $p^{(1)}, \dots, p^{(k)}$.
CONSTANTS: Tolerance $0 < \delta \ll 1$.
ROUTINE
    $\bar{x} \leftarrow \frac{1}{m} \sum_{j=1}^{m} x_j$
    $X^{(0)} \leftarrow \{x - \bar{x} : x \in X\}$
    FOR $i \leftarrow 1$ TO $k$ DO
        *Power Iteration:*
        Choose random unit vector $r_0 \in \mathbb{R}^n$
        FOR $l \leftarrow 1$ TO *some reasonable limit* DO
            $r_l \leftarrow \frac{1}{m} \sum_{j=1}^{m} x \langle x | r_{l-1} \rangle$ normalized such that $\|r_l\| = 1$
            IF $\|r_l - r_{l-1}\| \leq \delta$ THEN
                $p^{(i)} \leftarrow r_l$
                BREAK
            END
        END
        *Gram-Schmidt:*
        $X^{(i)} \leftarrow \{x - p^{(i)} \langle p^{(i)} | x \rangle : x \in X^{(i-1)}\}$
    END
END
```

ALGORITHM 4.2: Principal component analysis using power iteration and Gram-Schmidt orthonormalization. We used $\delta = \sqrt{\epsilon_{\text{float}}}$ which turned out to work very well.

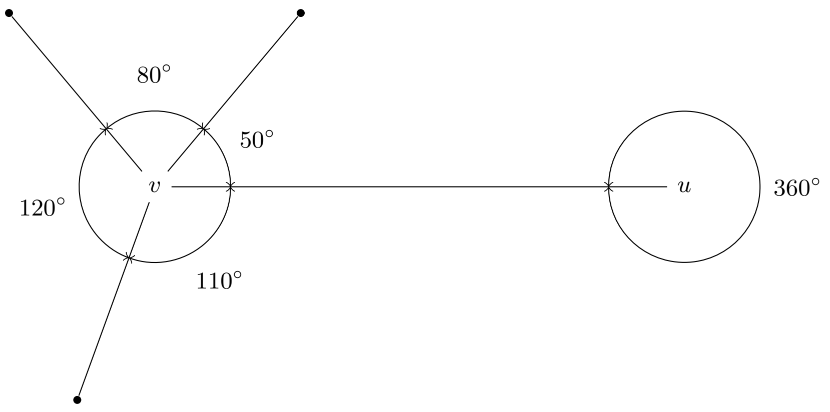


FIGURE 4.3: Our definition of angles between incident edges as used in the `ANGULAR` property illustrated.

- Ignore the edge and move on.

We've decided to simply ignore such edges when computing this property. Aborting the computation is too harsh as coinciding vertices do occasionally occur even in high-quality layouts. Inserting special values might seem like the best approach at first but turns out to merely postpone the problem because no meaningful analysis can be performed with a data set containing such values.

The computational effort for the `ANGULAR` property is linear in the number of edges in the graph.

4.3 Edge Lengths (`EDGE_LENGTH`)

The next property we wish to introduce is the distribution of edge lengths. For layout Γ of graph $G = (V, E)$ we look at the distribution of values in the multiset

$$\texttt{EDGE\_LENGTH} = [\mathrm{length}_\Gamma(e) : e \in E] \quad . \tag{4.6}$$

For an "ideal" layout, edges would all be of the same length. Of course, this can only work for very boring graphs. But even for graphs with a more complicated structure, one would expect to see peaks in the edge length distribution of a good layout. For example, inter- and intra-cluster edges should be visible. Figure 4.5 shows edge length distributions for thee sample layouts.

The `EDGE_LENGTH` property is straight-forward to compute with computational expense linear in the graph's number of edges.

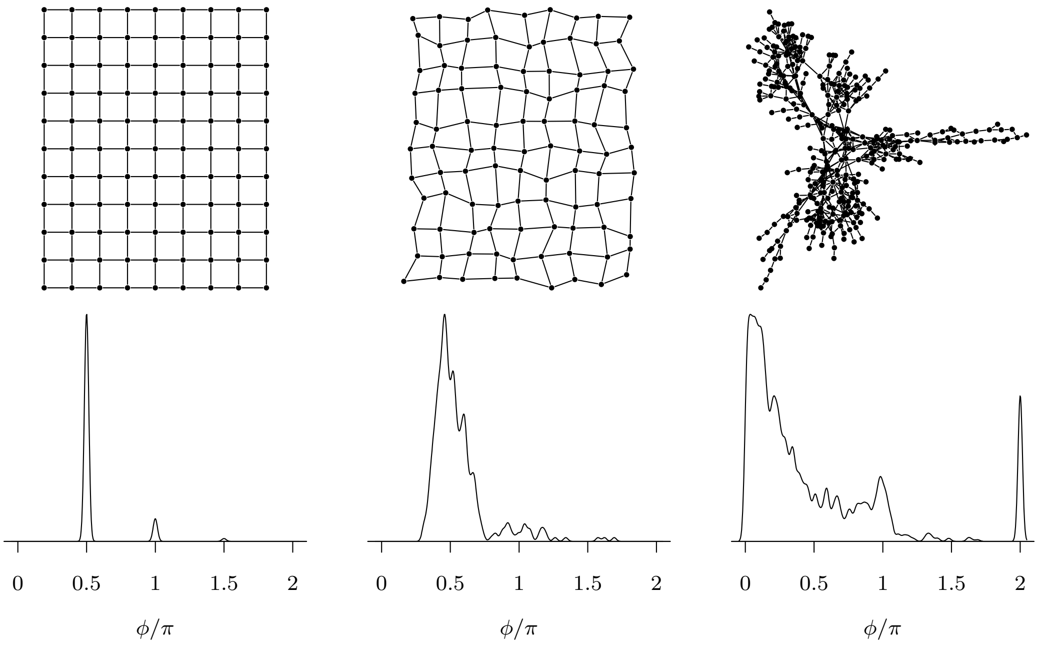


FIGURE 4.4: Distributions for the `ANGULAR` property shown for the three example layouts. The regular grid again features very sharp peaks. The dominating peak at $\pi/2$ is the angle between the incident edges of nodes with degree 4. A smaller peak at π corresponds to the angles between the edges along the margin of the grid. The barely visible signal at $\pi 3/2$ is caused by the four corner vertices. The distorted grid shows again a similar distribution but with wider and additional peaks. The layout for the graph on the right has a maximum near 0 due to the many vertices with a high degree. Nevertheless, there are two additional pronounced peaks around π and 2π courtesy of the vertices with degree 2 and 1 respectively.

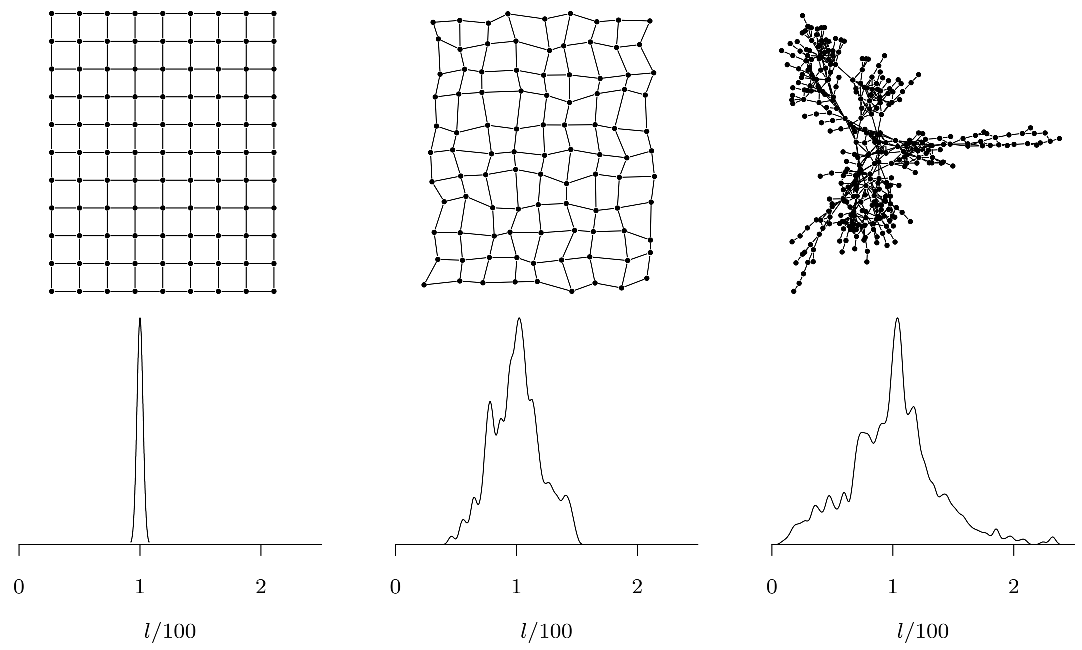


FIGURE 4.5: Distribution of the `EDGE_LENGTH` property for three layouts. The distribution for the regular grid on the left shows only a single sharp peak which would again be a Dirac delta function if it were not for the finite filter width. The distorted gird in the middle has a broad signal with a few unidentifiable buckles forming an overall Gaussian shape. The force-directed layout on the right exposes a clear peak at the target edge length with two roughly symmetric shoulders and low but non-zero signal towards either end of the spectrum.

4.4 Pairwise Distances (`RDF_GLOBAL` and `RDF_LOCAL`)

The notion of a *radial distribution function (RDF)* is a concept borrowed from statistical thermodynamics and crystallography. [11] It considers the distribution of pairwise distances between particles, which in our case are the nodes in the layout.

Definition 4.2 (radial distribution function): *For an ensemble of particles in space $\mathbb{R}^n$ with $n \in \mathbb{N}$, the radial distribution function is a function $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ where $g(r)$ for $r \in \mathbb{R}_{\geq 0}$ is the average number of other particles found in the infinitesimal spherical shell $\{x \in \mathbb{R}^n : r \leq \|x - p\| \leq r + \mathrm{d}r\}$ around a particle at position $p \in \mathbb{R}^n$ divided by the volume of the shell.*

For a finite ensemble, the RDF may be computed via the pairwise distances between particles.

RDF seems to be an interesting concept in the field of graph drawing, too. Ideally, one would expect $g(r) = 0$ for all values of r below a certain threshold. That is, nodes are never drawn closer to each other than a minimum distance which should be at least the size of the symbol used to visualize a vertex. In the context of physical chemistry, this is commonly referred to as a “hard sphere” model, meaning that nodes are thought of as impenetrable solid bodies.

Furthermore, one would expect to see distinctive features in the distribution function corresponding to certain characteristic distances in the drawing of a graph with a regular structure. On the other hand, the RDF $g(r)$ for a random placement of nodes (akin to an ideal gas) would be a constant independent of the radius r. One has to be aware, however, that graph drawings usually feature structures with a number of vertices that is considerably smaller than Avogadro’s number[3] so the margins cannot be ignored which makes distributions look different than those that would be expected from physics.

Equipped with this definition and motivation, we can define the property

$$\texttt{RDF\_GLOBAL} = [\mathrm{dist}_\Gamma(v_1, v_2) : v_1, v_2 \in V] \tag{4.7}$$

for a layout Γ of graph $G = (V, E)$. Figure 4.6 shows the `RDF_GLOBAL` property for the familiar example layouts used in this section before.

In order to compute the property `RDF_GLOBAL` the algorithm has to loop over all (unordered) pairs of vertices in the graph which means quadratic expense. Another practical problem arises from the fact that even for a relatively moderately-sized graph of, say, $n = 50\,000$ vertices, the memory requirement to store the $n^2/2$ pairwise distances as individual 8 byte IEEE 754 floating-point numbers would be close to 10 GiB. Even on a machine that has that much RAM available, excessive cache misses would make the computation very slow. Therefore, we avoid storing the entire data set in memory at

[3]The Avogadro constant N_A is the number of carbon atoms found in 12 g of the carbon isotope $^{12}\mathrm{C}$ and is currently estimated [55] to be $N_\mathrm{A} \approx 6.022\,140\,857\,(74) \times 10^{23}$.

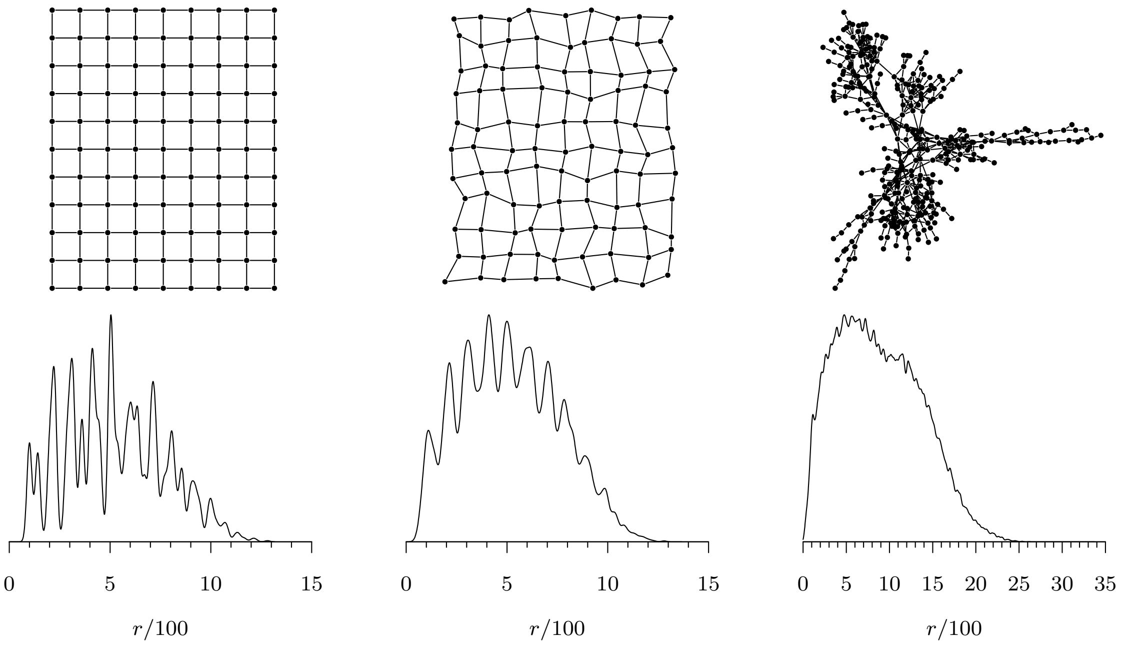


FIGURE 4.6: Distributions for the `RDF_GLOBAL` property shown for the three example layouts. The RDF for the regular grid shows that $g(r) = 0$ for $r < 100$ with a distinguished peak at $r = 100$ corresponding to the closest distance in the lattice. The next peak at $r = 100 \times \sqrt{2}$ corresponds to the next larger (diagonal) distance. Large double peaks around $r = 200$ and $r = 300$ are caused by the even more frequently occurring distances to vertices two or three cells away respectively as well as the distances of $r \approx 224$ and $r \approx 283$ which correspond to the diagonals of 2×1 and 2×2 cells respectively. As the radius increases, more and more peaks become visible until the distribution starts to decay again due to the finite graph size before a continuum is reached. The RDF of the distorted grid resembles the basic structure of that for the regular grid but features "fingers" superimposed on a large base signal instead of distinctive sharp peaks. The peaks at the radii discussed before can still be clearly identified, though. The RDF for the force-directed layout of the larger graph at the right makes it hard to identify any significant features except for the clearly visible peak at $r = 100$ caused by the many single-edge distances along the more or less regular strands of nodes.

once and instead compute the numbers on-the-fly as they are needed for the subsequent analysis. In C++ this can be done straight-forwardly and without additional overhead by using iterators.

Looking at distributions of RDF_GLOBAL such as in figure 4.6, it seems that the signal-to-noise ratio is not very good. Especially the long tails towards larger radii contain little information while the details near $r = 0$ are obscured by the overwhelming amount of data. Another reason to question the applicability of the unmodified concept of RDF to graph drawing is that unlike for the analysis of gasses and liquids where each particle is alike, the vertices in a graph have very special connections, namely, edges. Therefore, it might be more insightful to study pairwise distances only of those pairs of vertices that have a graph-theoretical distance below a given threshold. This leads to the following parameterized property

$$\texttt{RDF\_LOCAL}(d) = [\mathrm{dist}_\Gamma(v_1, v_2) : \mathrm{dist}(v_1, v_2) \leq d : v_1, v_2 \in V] \tag{4.8}$$

for a layout Γ of graph $G = (V, E)$ and $d \in \mathbb{N}$. This property turns out to be an interesting intermediate between two properties we've already introduced as the following limits show.

$$\texttt{EDGE\_LENGTH} = \texttt{RDF\_LOCAL}(1) \tag{4.9}$$

$$\texttt{RDF\_GLOBAL} = \lim_{d\to\infty} \texttt{RDF\_LOCAL}(d) \tag{4.10}$$

For connected graphs, the limit in equation 4.10 is an exaggeration. The property won't change for values of d larger than the graph's diameter, that is $d > \max\{\mathrm{dist}(v_1, v_2) : v_1, v_2 \in V\}$. Furthermore, we found it to be sufficient to compute only RDF_LOCAL(2^i) for $i \in \mathbb{N}_0$. Examples for RDF_LOCAL are shown in figure 4.7.

In order to compute the RDF_LOCAL property, an all-pairs shortest path matrix is computed first, using Dijkstra's algorithm (*cf.* Mehlhorn and Sanders [37]). This matrix must be kept in memory. We have tried to avoid this by computing the shortest single-source shortest path vector instead on-demand but this turned out to be prohibitively slow. On the other hand, the almost cubic operation of finding the matrix is so expensive that memory constraints will rarely be the limiting factor. Once the matrix is computed, the same approach as for RDF_GLOBAL is used to do the actual analysis except that the iterator which computes pairwise distances on-the-fly is modified to skip over pairs of vertices if their graph distance exceeds the locality parameter. The same shortest-paths matrix can be reused for computing RDF_LOCAL(2^i) for $i = 0$ up until 2^i exceeds the graph's diameter.[4]

4.5 Tension (TENSION)

The last property we considered is the ratio of graph distance and layout distance. This property is strongly motivated by the stress function introduced by Kamada and

[4]If the graph is disconnected, we use the longest shortest path in any connected component instead.

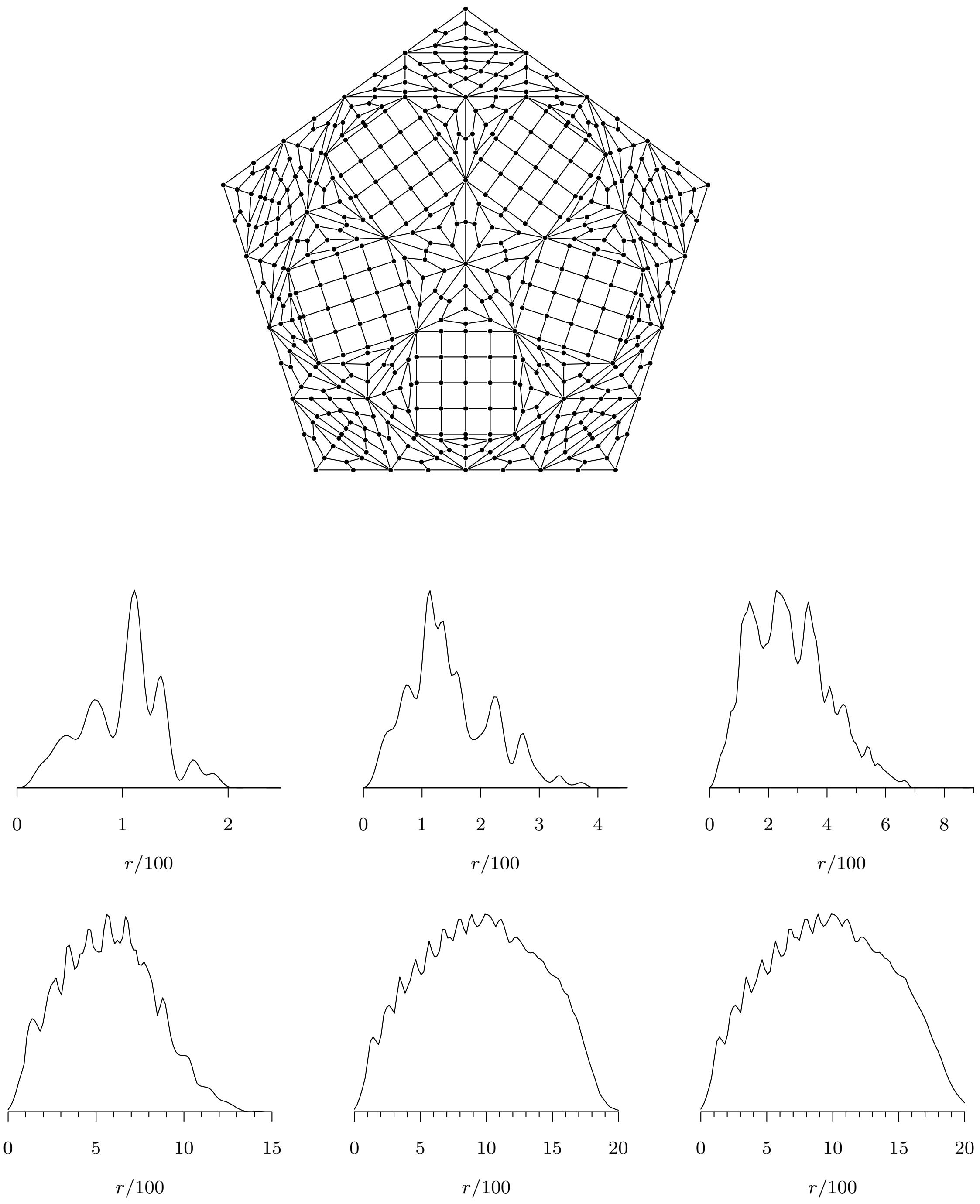


FIGURE 4.7: Distributions for the property `RDF_LOCAL`(2^i) shown for increasing values of $i \in \{0, \ldots, 5\}$. The example shows a native layout of a graph generated by the `MOSAIC2` generator (§ 5.1.5). The graph has 521 vertices, 975 edges and a diameter of 20.

Kawai [25] (*cf.* § 5.2.3). Kamada and Kawai defined the stress in the layout Γ of graph $G = (V, E)$ with $|V = n|$ as proportional to

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{\operatorname{dist}_\Gamma(v_i, v_j) - l \operatorname{dist}(v_i, v_j)}{\operatorname{dist}(v_i, v_j)} \right)^2 \tag{4.11}$$

where l is the desired average edge length. Unfortunately, this definition depends in a non-linear fashion on the scale of the drawing. This problem was already pointed out by Welch and Kobourov [58] who circumvented it by using the minimal value the stress function will take on when the layout is scaled. In order to do so, they employed a binary search strategy. Since we'd rather avoid doing this, we decided to not use the quantity from equation 4.11 but instead modify it such that it behaves linear in response to scale. This gives rise to the following property

$$\mathtt{TENSION} = \frac{|E|}{\sum\limits_{e \in E} \operatorname{length}_\Gamma(e)} \left[\frac{\operatorname{dist}_\Gamma(v_1, v_2)}{\operatorname{dist}(v_1, v_2)} : v_1, v_2 \in V \right] \tag{4.12}$$

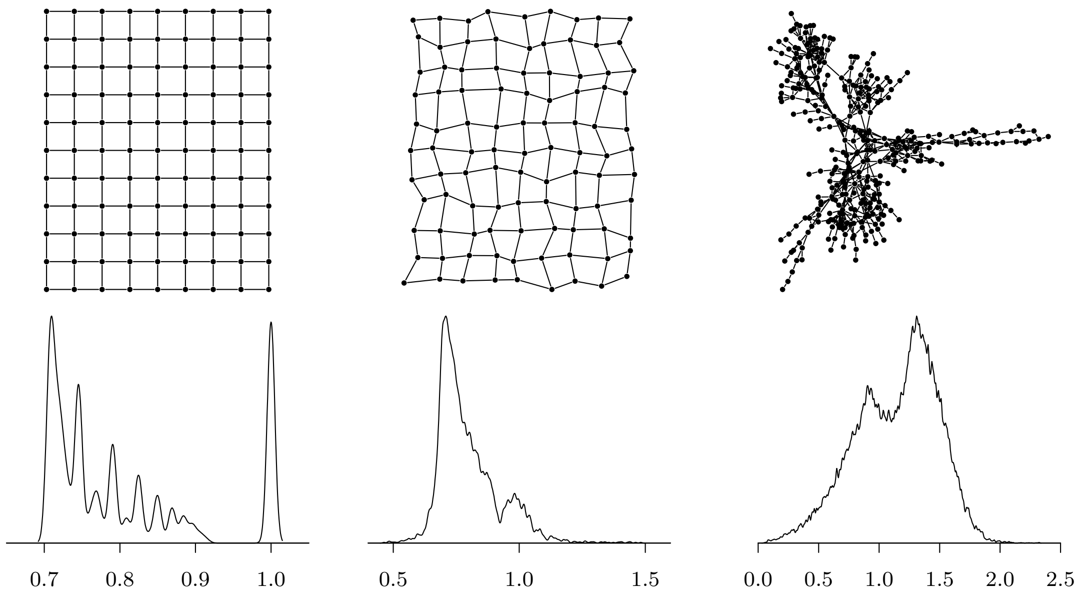
for which we've chosen the name "tension" for its similarity to "stress". `TENSION` is defined like this and not the reciprocal because the layout distance $\operatorname{dist}_\Gamma$ might be zero which would make the quantity undefined if it were to occur in the denominator while the graph distance dist can only take on values that are positive integers. The normalization factor could be left off without doing any harm except that it is nice to see the ideal value centered at 1 in diagrams. (Note that the factor is always 1/100 by definition 1.6 anyway.)

Our approach to tension differs from the usual understanding of stress in that we are not so much concerned about the absolute value (in fact, we never sum it up) but rather the *distribution* of the pairwise tension. This is illustrated for the familiar example graphs in figure 4.8.

In order to compute tension, an all-pairs shortest path matrix is again computed first using Dijkstra's algorithm. Once this is done – which might be as expensive as cubic in n – the remaining calculation has quadratic cost. As already discussed in the context of RDF, we avoid the upfront computation of all values and instead compute them lazily as needed.

4.6 Summary

We have presented seven statistical properties in this section that we believe can be representative syndromes of a graph layout's aesthetic value as illustrated with examples of layouts of different qualities. Each property is a multiset of real values. We have also described how those values can be computed efficiently. The discussed properties were (for a graph $G = (V, E)$ with $|V| = n$ and $|V| = m$):



Compare this to the stress-minimized layout of the same graph:

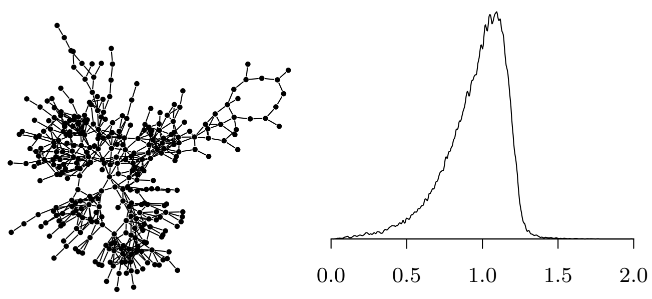


Figure 4.8: The two top rows show the distributions for the `TENSION` property for the three example layouts used throughout this section. The distribution for the regular layout of the grid features two distinctive peaks at 1 and at $\sqrt{2}/2$ corresponding to the direct neighbors and indirect neighbors that face each other diagonally respectively. The smaller peaks in between may be explained similarly. In the distorted grid, the peak at 100 is much smaller and the widened peak around $\sqrt{2}/2$ dominates the spectrum. The smaller peaks in between have almost disappeared. The distribution for the force-directed layout of the `BCSPWR` graph in the last column features two overlapping broad peaks. The smaller one is centered at somewhat less than 1 and corresponds to the nodes along the strands. Since those are the shortest edges in the layout and the layout and therefore shorter than the average edge length. The second larger peak around $\sqrt{2}$ is not so easily identified in the layout. The bottom row shows the stress-minimized layout of the `BCSPWR` graph and according `TENSION` distribution next to it. Compared to the alternative force-directed (`FMMM`) layout above, the double-peak got fused into a single asymmetric peak with a maximum at 1 that drops sharply to zero around $\sqrt{2}$ and has a much smoother decay towards the left end of the spectrum.

- `PRINCOMP1ST` — The distribution of vertex coordinates projected onto the major axis of the layout which can be found by means of a PCA. The property contains n values which can be computed with effort not significantly worse than $\mathcal{O}(n)$ (*cf.* algo. 4.2).
- `PRINCOMP1ST` — The distribution of vertex coordinates projected onto the minor axis of the layout. The property contains n values which can be computed like for `PRINCOMP1ST` except that a Gram-Schmidt orthonormalization and an additional power iteration step, both roughly $\mathcal{O}(n)$, have to be performed (*cf.* 4.2).
- `ANGULAR` — The distribution of angles between adjacent edges. The property has m values and can be computed with $\mathcal{O}(n+m)$ effort even if no specialized data structure is used.
- `EDGE_LENGTH` — The distribution of edge lengths. The property has m values and can be computed with $\mathcal{O}(m)$ effort.
- `RDF_GLOBAL` — The distribution of pairwise distances between all nodes. The property has $n(n-1)$ values and computation takes $\mathcal{O}(n^2)$.
- `RDF_LOCAL`(d) — The distribution of pairwise distances between vertices with graph-theoretical distance of no more than $d \in \mathbb{N}$. We found it sufficient to consider only values $d = 2^i$ for $i \in \mathbb{N}_0$. The property `RDF_LOCAL` provides an intermediate view between the extremes of the `EDGE_LENGTH` $(d = 1)$ and `RDF_GLOBAL` $(d \to \infty)$ properties. Consequently, it features between m and $n(n-1)$ values. Computation of `RDF_LOCAL` requires an all-pairs shortest path matrix to be computed which might take up to $\mathcal{O}(n^3)$ and dominates the subsequent $\mathcal{O}(n^2)$ accumulation step.
- `TENSION` — The distribution of quotients of node and graph distances. `TENSION` is motivated by and named after stress [25]. It yields $n(n-1)$ values but computing them requires knowledge of all pairwise shortest paths and can therefore be as expensive as $\mathcal{O}(n^3)$.

The reader might have noticed that we did not introduce any property that considers the *crossings* of edges. This obvious omission is left for future work to be done. One difficulty with it is that – unlike for all other properties we considered – the *number* of events[5] does not depend solely on the *graph* but actually on the *layout* which makes it a little challenging to integrate; especially the (desirable) case where there are no crossings in the drawing at all so no statistics can be done.

We did not explain yet how to feed the described properties into an algorithm that would make use of them. We will defer this discussion until chapter 7.

[5]The *values* of the events obviously depend on the layout or the measure would be no good for analyzing layouts.

5 Data Generation

5.1 Graphs

A *graph generator* is a probabilistic algorithm that might take an implementation-defined set of input parameters and outputs a graph, optionally alongside with a *native* layout (§ 5.2.1) that is assumed to be of good quality. Native layouts differ from layouts computed by layout algorithms in that they fall out directly as a by-product of the graph generation. For example, the `MOSAIC1` generator (§ 5.1.5) which derives graphs by recursively splitting facets of an initial simplex produces a naturally occurring layout alongside with the graph that, if nothing else, is guaranteed to be planar.

We have implemented several graph generators that produce a variety of graphs for which aesthetically pleasing layouts exist.

The driver for our experiments reads a configuration file `graphs.cfg` that specifies how many graphs of what size class are desired for which generator. It will then populate the database by repetitively running each generator (with appropriate inputs) until the database contains all desired graphs.

In order to avoid duplicates, a hash is computed of each graph. If a graph with the same hash already exists in the database, the graph is discarded and the generator called again, hoping it will produce another graph this time. Otherwise, the graph is added to the database and from now on referred to by using its hash as unique ID.

If the generator supports specifying the approximate graph size, the driver will make use of it. In any case, the actual graph size will be checked before the graph is added to the database. If the actual size diverges from the requested size, it is first checked whether a graph of that size is also still on the worklist and if so, the graph is added nonetheless. Otherwise, the graph has to be discarded.

5.1.1 Imported Graphs (`IMPORT`)

The `IMPORT` generator is special in that it does not genuinely generate graphs but merely imports existing graphs. Formally, its input is a collection of graphs (and maybe layouts) from which it randomly picks and returns one.

Our implementation allows to specify import sources in a configuration file `imports.json` that specifies the archive type (directories and TAR archives are supported as well as lists

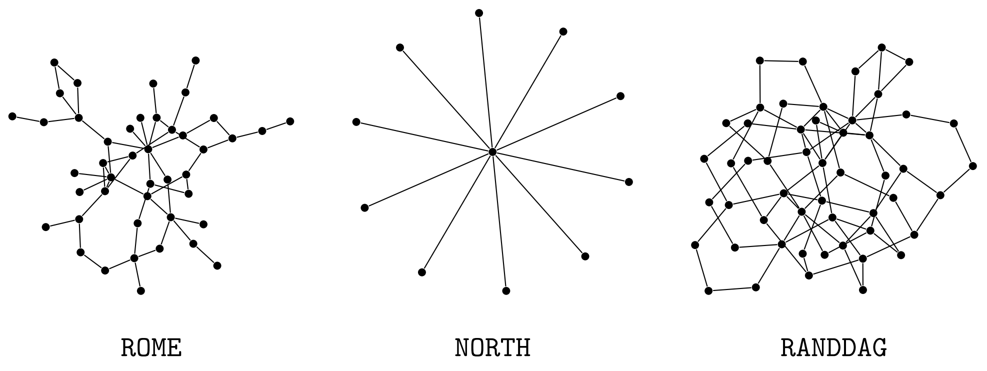


Figure 5.1: Examples of three randomly chosen graphs from the three considered archives (stress-minimized layouts).

of individual URLs), file format (all formats implemented in the OGDF are supported), file compression, whether the graphs have associated layout information and the URL where the archive can be found. An optional cryptographic checksum can be specified for TAR archives as well in order to validate the integrity of the archive prior to its use. The driver will then read this configuration file and feed the archives to the `IMPORT` generator. Caching is performed to avoid needless queries.

5.1.1.1 graphdrawing.org (`ROME`, `NORTH`, `RANDDAG`)

Three import sources were predefined for some graph collections found at graphdrawing.org [16] and referred to as the `ROME`[1], `NORTH`[2], and `RANDDAG`[3] generators which are really just aliases for the `IMPORT` generator. Each of these archives contains a number of graphs much larger than what would be practical to process with our setup so we only imported a small fraction. Unfortunately, the graphs in these collections have no associated known-good layouts. They are also all relatively small with the average number of nodes around 50. Examples of graphs from these archives are shown in figure 5.1.

5.1.1.2 Matrix Market (`SMTAPE`, `PSADMIT`, `GRENOBLE`, `BCSPWR`)

The "Matrix Market" [3] maintained by the NIST is an online collection of matrices from a wide variety of disciplines. Consequently, it contains many different matrices and not all of them are interesting examples for graph drawing. We have considered the `SMTAPE`[4],

[1] `http://graphdrawing.org/download/rome-graphml.tgz`
[2] `http://graphdrawing.org/download/north-graphml.tgz`
[3] `http://graphdrawing.org/download/random-dag-graphml.tgz`
[4] `https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/smtape/smtape.html`

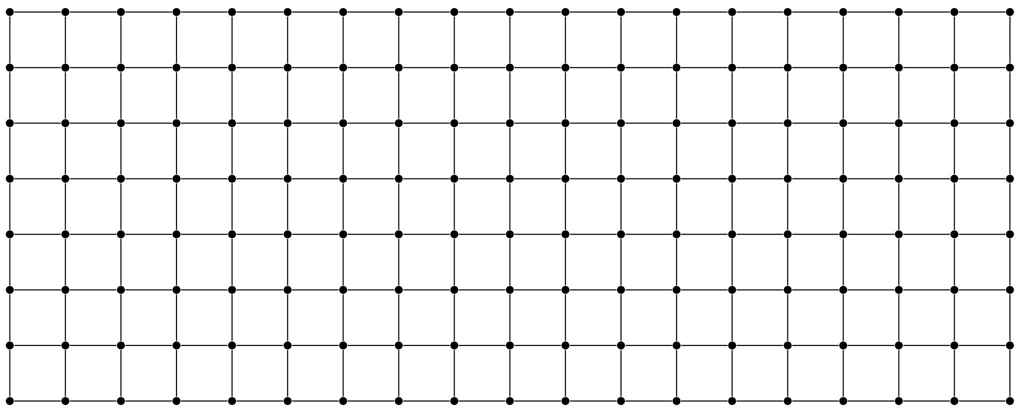


FIGURE 5.2: An example of a graph output by the `GRID` generator (native layout).

`PSADMIT`[5], `GRENOBLE`[6] and `BCSPWR`[7] sets (all from the Harwell-Boeing collection) where we found the latter to be most interesting. No examples are shown here because the graphs are pretty large (usually several thousand vertices) and it is hard to identify a representative example.

5.1.2 Regular Grids (`GRID`) and Tori (`TORUS`$\langle n \rangle$)

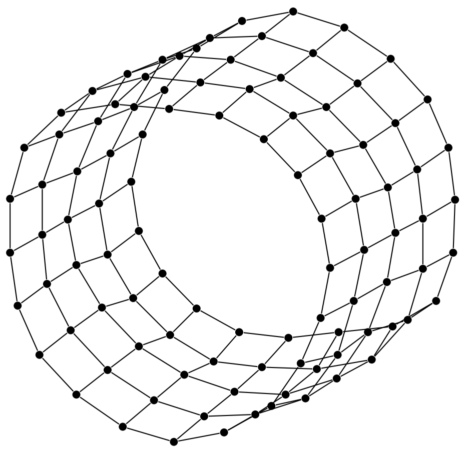
The `GRID` generator, when asked to produce a graph with $n \in \mathbb{N}$ vertices, chooses two random integers $1 \leq n_1, n_2 \leq 2\sqrt{n}$ independently according to a uniform distribution and outputs a graph that is a regular $n_1 \times n_2$ grid with the obvious native layout. An example of a `GRID` graph's native layout is shown in figure 5.2.

The `TORUS1` generator operates similarly except that it adds additional edges from the first to the last vertex in a "row" such as to obtain a 1-torus (a cylinder). The `TORUS2` generator additionally connects the first and last vertex of each "column" with an edge which yields a 2-torus (a doughnut). The `TORUS1` and `TORUS2` generators do not output native layouts because there is no obvious mapping of a n-torus to a 2-dimensional layout, although a projection of the 3-dimensional objects into two dimensions would be possible. Examples for graphs generated by `TORUS1` and `TORUS2` are shown in figure 5.3.

5.1.3 Stochastic L-Systems (`LINDENMAYER`)

A *Lindenmayer System* $L = (\Sigma, \omega, \Pi)$ is a text-rewriting system consisting of an *alphabet* Σ, an initial *axiom* $\omega \in \Sigma$ and a set of *production rules* Π. Each production $\pi \in \Pi$

[5]`https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/psadmit/psadmit.html`
[6]`https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/grenoble/grenoble.html`
[7]`https://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcspwr/bcspwr.html`



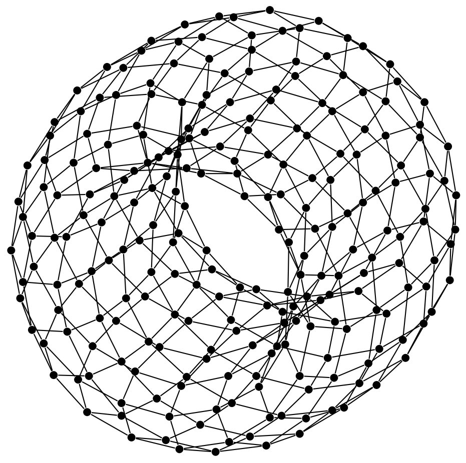


FIGURE 5.3: Examples of graphs output by the TORUS1 and TORUS2 generators (force-directed layouts).

is a two-tuple consisting of a *pattern* to be matched and a *replacement text* by which an occurrence of the pattern in the text may be replaced. A *stochastic L-system* is a probabilistic algorithm that derives a random word $w \in \Sigma^*$ by starting with the initial axiom ω and recursively applying productions from Π chosen randomly among those applicable. L-systems were first described by and named after Prusinkiewicz and Lindenmayer [40] who used them to describe the organic growth of plants in nature (see figure 5.4 for an inspiration).

The LINDENMAYER generator generates graphs by recursively replacing nodes with symmetric sub-graphs. Its initial axiom is a singleton graph $G_\omega = (\{v\}, \emptyset)$. Productions are defined by the following sub-generators each of which replaces a single vertex by a more complicated sub-graph.

- L/SINGLETON is an identity transformation.
- L/STAR$_k$ for $k \in \mathbb{N}$ splits all edges incident to v by inserting new vertices and distributes $(k-1)\deg(v)$ additional vertices, each connected to v, radially between them. If $\deg(v) = 0$ then k vertices are distributed radially around v and connected to it.
- L/WHEEL$_k$ for $k \in \mathbb{N}$ is the same as L/STAR$_k$ except that the new vertices are additionally connected to a ring.
- L/RING$_k$ for $k \in \mathbb{N}$ is the same as L/WHEEL$_k$ except that vertex v is deleted.
- L/CLIQUE$_k$ for $k \in \mathbb{N}$ is the same as L/RING$_k$ except that all newly added vertices are connected to a clique.

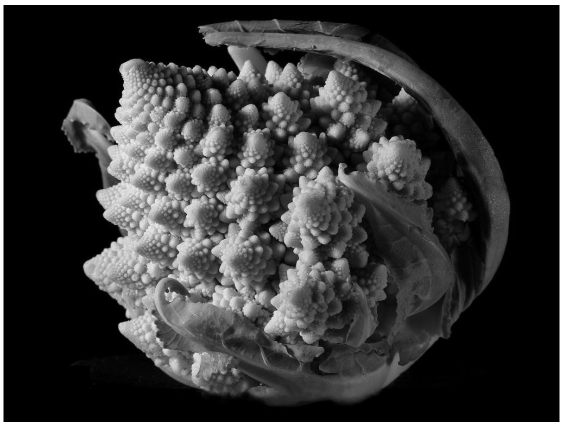


FIGURE 5.4: *Romanesco broccoli* is a particular spectacular example of a self-similar plant that may be modeled using a stochastic L-system. [Original photo by Jon Sullivan (2004), public domain.]

- `L/GRID`$_{n,m}$ for $n, m \in \mathbb{N}$ replaces a vertex v with $\deg(v) = 0$ by a regular $n \times m$ grid.

The operation of these sub-generators is illustrated in figure 5.5.

For a desired number of nodes n, the `LINDENMAYER` generator starts with a singleton graph $G_\omega = (\{v\}, \emptyset)$ and randomly picks one of the sub-generators, instantiates it with suitable parameters chosen probabilistically and applies it to v. This process is repeated recursively for each new node added until the graph has grown to the desired number of nodes. At each level of the recursion, the recursive application is done using the same random seed for each vertex which yields a symmetric graph. The process is illustrated in algorithm 5.6.

When a vertex v with $\deg(v) = d$ adjacent to vertices $u_1, \ldots, u_d$ is replaced by a sub-graph, the new vertices of the sub-graph are laid out in such a way that the edges that were incident to v do not change their direction. That is, new vertices are placed along the original edges connecting v and $u_1, \ldots, u_d$. Additional new vertices (if any) are distributed uniformly between those vertices already fixed. The scale of the sub-layout is chosen such that it doesn't overlap with any other existing structure. An example of such a graph and native layout is shown in figure 5.7.

5.1.4 Projections of Hyper-Lattices (`QUASI`$\langle n \rangle$`D`)

A *quasicrystal* is a chemical compound that has an ordered but aperiodic structure intermediate between an amorphous (unordered) and crystalline (periodic) structure. [44] As in a crystal, any sub-region of arbitrary size found in a quasicrystal may be brought

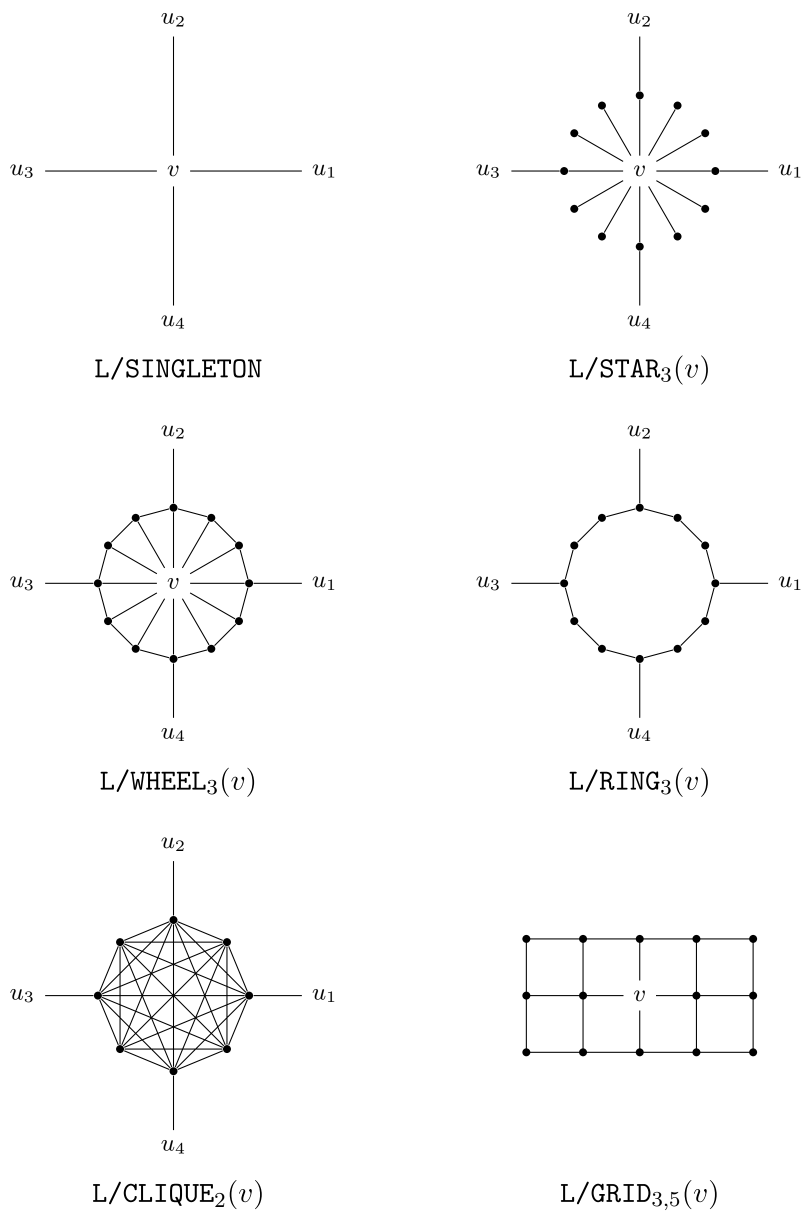


FIGURE 5.5: Operation of the LINDENMAYER sub-generators on a vertex v adjacent to four vertices $u_1, \ldots, u_4$ illustrated; except for the L/GRID$_{n,m}$ example where $\deg(v) = 0$ initially as required for its applicability. (Hence the grid is not connected to any other node after the operation has completed either.)

```
INPUT: Graph $G = (V, E)$, vertex $v \in V$, desired size $n \in \mathbb{N}$ and seed $s \in \{0, 1\}^*$.
OUTPUT: Graph with approximately $n$ vertices.
ROUTINE
    Replace $v$ by new vertices $v_1, \ldots, v_l$ to obtain $G^{(0)} = (V^{(0)}, E^{(0)})$        (1)
    $n' \leftarrow \lfloor (n - l)/l \rfloor$
    IF $n' \leq 0$ THEN
        RETURN $G^{(0)}$
    END
    Instantiate a pseudo random generator $\mathcal{R}$ with seed $s$
    Use $\mathcal{R}$ to choose and instantiate a sub-generator $\mathcal{G}$        (2)
    Use $\mathcal{R}$ to derive a new seed $s' \in \{0, 1\}^*$
    FOREACH $i \in \{1, \ldots, l\}$ DO
        $G^{(i)} \leftarrow \mathcal{G}(G^{(i-1)}, v_i, n', s')$;
    END
    RETURN $G^{(l)}$
END
```

ALGORITHM 5.6: Simplified conceptual operation of the `LINDENMAYER` graph sub-generators. The actual logic is a little bit more complicated because instead of choosing the same sub-generator $\mathcal{G}$, a more interesting but still symmetric graph can be obtained by choosing several sub-generators. For example, the `L/RING`$_k$ sub-generator uses two sub-generators $\mathcal{G}'$ and $\mathcal{G}''$. It then applies $\mathcal{G}'$ to each new vertex v_i with $i \equiv 0 \pmod{k}$ and $\mathcal{G}''$ to the other new vertices. The replacement at (1) is done according to the rules specific for the sub-generator and its parameters. For example, the `L/CLIQUE`$_k$ sub-generator would replace v by an l-clique (with $l = k \deg(v)$) and distribute the edges that were incident to v uniformly among the new vertices. The choice of sub-generator at (2) has to take into account that some sub-generators are not always applicable. For example, the `L/GRID` sub-generator requires that $\deg(v) = 0$. The parameters to instantiate the sub-generator should be chosen according to a random distribution that takes into account how many vertices the sub-generator should add at most. Also not shown is the logic to determine the coordinates for the new nodes as it is pretty straight-forward to imagine but rather tedious to implement.

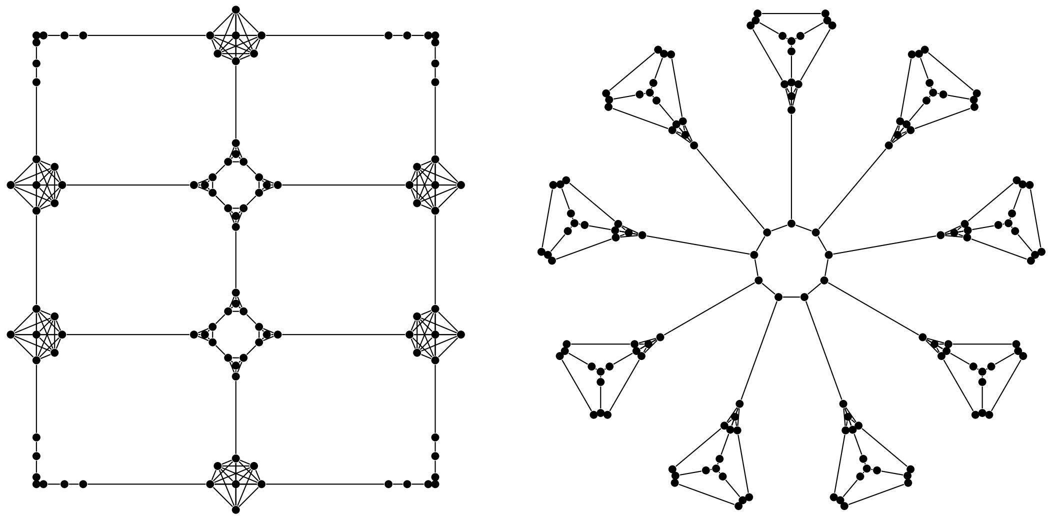


FIGURE 5.7: Examples of two graphs produced by the LINDENMAYER generator (native layouts).

to congruence with an infinite number of other sub-regions in the same structure by means of rotation and translation. However, unlike a crystal, the series defined by the required amounts of translation is not periodic. Quasicrystals can (mathematically) be obtained as the projection of a periodic structure in a higher-dimensional space onto a lower-dimensional space which intersects the higher-dimensional space at an irrational angle. If the angle of intersection is rational, a regular crystal (that is, a periodic pattern) will be obtained.

The QUASI$\langle n \rangle$D generators are inspired by this concept. They use a primitive cubic lattice in three- to six-dimensional space and project it onto a two-dimensional plane intersecting that space at a random angle. No precautions are taken to make the intersection angle an irrational number for three reasons. First, since we are not going to generate a graph of infinite size, periodicity is not really defined in the first place. Second, since our implementation uses fixed-precision floating-point numbers, irrational quantities cannot be expressed anyway. Third, it is totally acceptable for our purposes to occasionally obtain a structure that is actually periodic.

For a given hyper-space dimension $d \in \{3, \ldots, 6\}$ and desired number of vertices $n \in \mathbb{N}$, a random surface-normal $u \in \mathbb{R}^d$ is chosen which defines the plane P which (without loss of generality) is set to intersect at the coordinate origin. Two orthogonal unit vectors $e_1, e_2 \in P$ as well as a "thickness" $1/10 \leq t \leq 11/10$ are chosen at random. All points $p \in \mathbb{Z}^d$ with an orthogonal distance to P of no more than t are projected onto P. Among those, the points that fall into the square S spanned by linear combinations $\alpha_1 e_1 + \alpha_2 e_2$ of the unit vectors with $0 \leq \alpha_1, \alpha_2 \leq \sqrt{n}$ define the vertices of the graph. Two vertices v_1 and v_2 that stem from projecting points $p_1, p_2 \in \mathbb{Z}^d$ are connected by an edge if and only

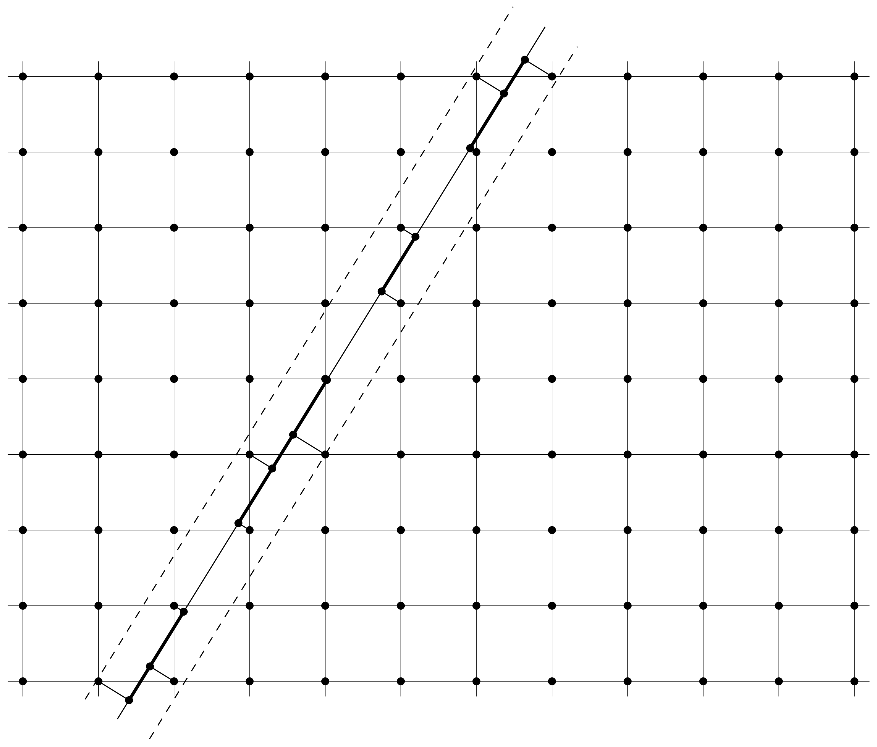


FIGURE 5.8: Concept of the `QUASI`$\langle n \rangle$`D` generator illustrated with a two-dimensional lattice projected onto a one-dimensional drawing "area". The generated graph in this example consists of four connected components: one pair, two triples and one quadruple of linearly connected vertices.

if $\|p_1 - p_2\| = 1$ (that is, p_1 and p_2 were neighbors in the d-dimensional hyper-lattice). The native layout follows immediately by interpreting S as the drawing area.

A schematic illustration of this process is shown in figure 5.8 with the number of dimensions reduced for clarity. Examples of graphs generated by the `QUASI`$\langle n \rangle$`D` generators are shown in figure 5.9.

5.1.5 Mosaic Patterns (`MOSAIC1` and `MOSAIC2`)

The `MOSAIC1` generator starts with a regular polygon and randomly applies one of the following operations to a randomly chosen facet defined by corner vertices $u_1, \ldots, u_k$ of the graph layout.

- `M/STAR` Places a new vertex v in the center of the facet and adds an edge from v to each vertex $u_1, \ldots, u_k$.

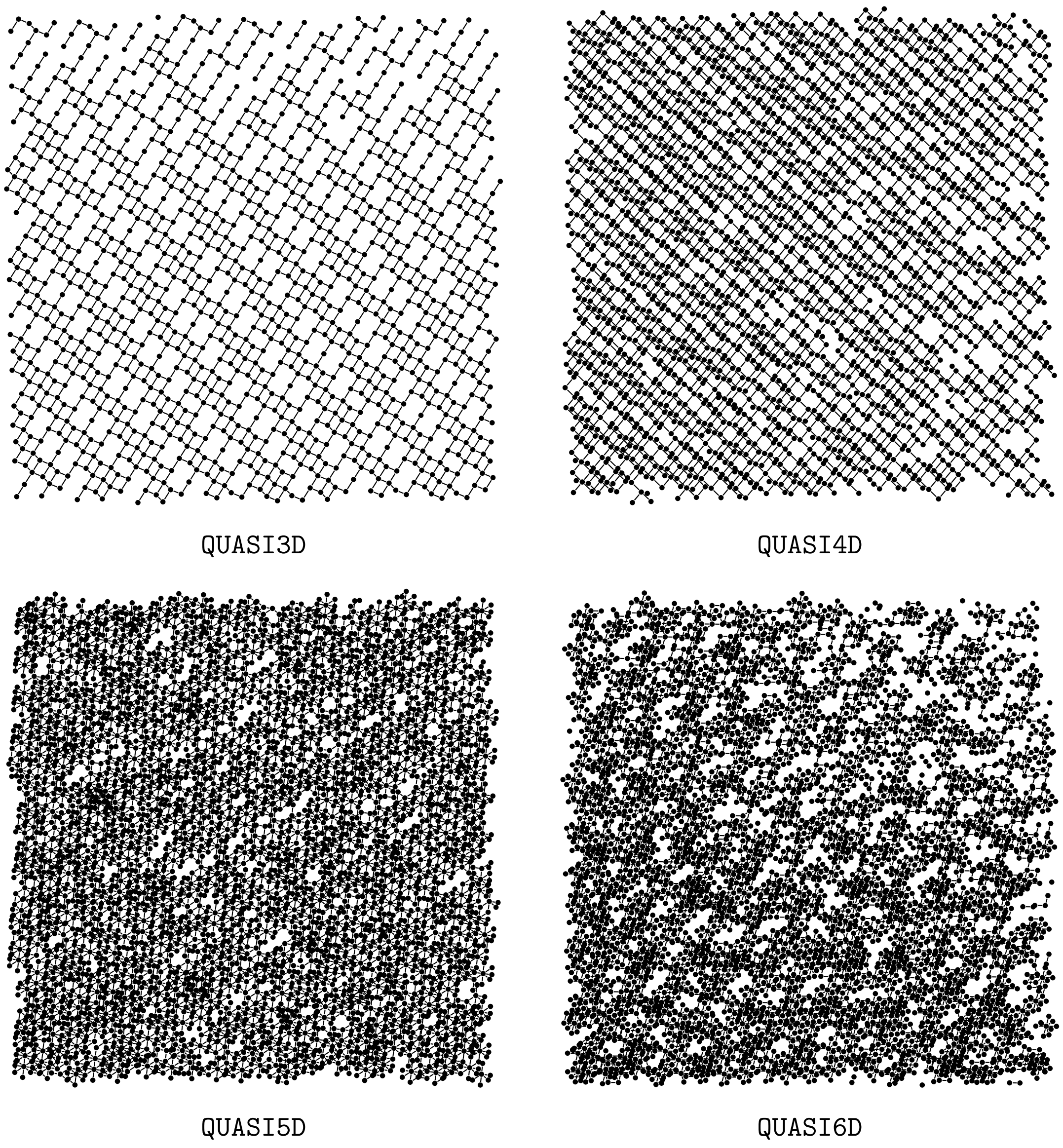


FIGURE 5.9: Examples of graphs produced by the QUASI⟨n⟩D generators via projecting a slice of a primitive cubic lattice in a n-dimensional hyper-space onto a two-dimensional plane that intersects that space at a random angle (native layouts).

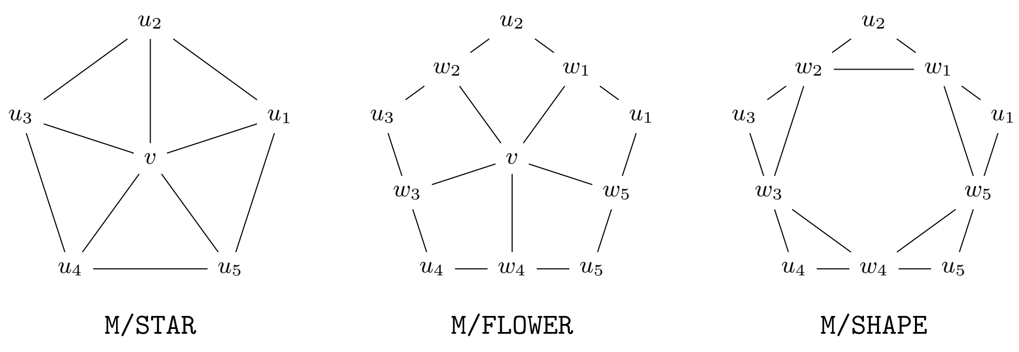


FIGURE 5.10: Operation of the MOSAIC sub-generators on a pentagonal facet $\{u_1, \dots, u_5\}$ illustrated.

- M/FLOWER Places a new vertex v in the center of the facet and splits each edge $\{u_i, u_j\}$ by inserting a new vertex w_i in the middle. Then, edges are added from v to each $w_1, \dots, w_k$.
- M/SHAPE Splits each edge $\{u_i, u_j\}$ by inserting a new vertex w_i in the middle and then connects the vertices $w_1, \dots, w_k$ to a ring.

When splitting edges, new nodes are only inserted if they do not already exist. Since $u_1, \dots, u_k$ are only the *corners* of the facet, it is possible that there are vertices between them. By construction of the graph layout, if two nodes fall on a straight line, they are always either connected directly by an edge or there is a node exactly in the middle between them. Figure 5.10 illustrates the operations described above.

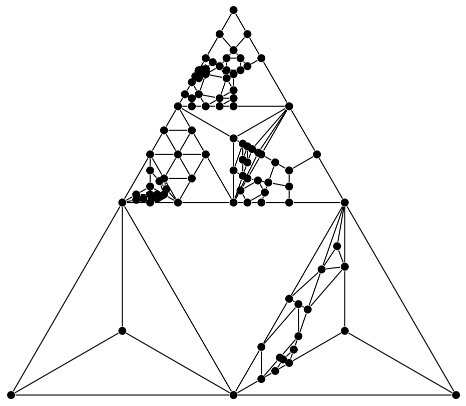
The MOSAIC2 generator is similar but instead of randomly choosing one facet at a time, it repetitively iterates over all facets in the graph layout and within each iteration, applies the same operation to each facet while yields a graph and layout with mush higher symmetry.

Examples of graphs generated by the two generators are shown in figure 5.11. By construction, these graphs are always planar and the native layouts never have edge crossings.

5.1.6 Meshes of Three-Dimensional Objects (BOTTLE)

The BOTTLE generator produces graphs that are meshes of the surface of a simple three-dimensional body and native layouts that are an axonometric projection thereof. It was named like this because the type of random objects it generates remotely looks like an awkwardly shaped bottle.

To generate the body and mesh, the generator proceeds as outlined in algorithm 5.12. The generated graph is the mesh and the native layout the axonometric projection of it



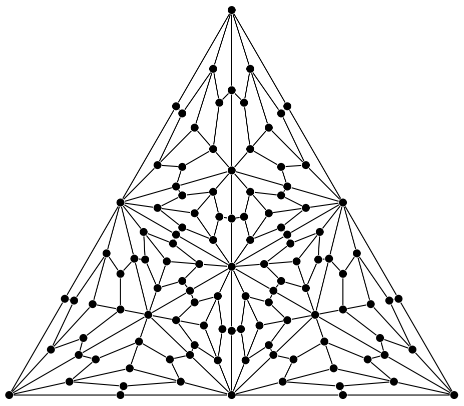


FIGURE 5.11: Examples of graphs produced by the `MOSAIC1` and `MOSAIC2` generators (native layouts).

onto a two-dimensional surface. An example of a possible output of the `BOTTLE` generator is shown in figure 5.13.

5.1.7 Summary

We have presented various strategies for obtaining graphs (often alongside with native layouts) and how we implemented them in the preceding sections. The discussed graph generators were:

- `IMPORT` — Imports a graph (and optionally an accompanying native layout) from a third-party source. We have used graphs from graphdrawing.org [16] (`ROME`, `NORTH`, `RANDDAG`) and NIST's *Matrix Market* [3] (`SMTAPE`, `PSADMIT`, `GRENOBLE`, `BCSPWR`) for our experiments. Care has to be taken when importing data from a third-party source that additional preprocessing might be required in order to make the graph and layout comply with our assumptions about normalized layouts of simple graphs.
- `GRID` — Produces a regular $n \times m$ grid where n and m are chosen randomly but with respect to the desired graph size. The graphs produced by this generator come along with the obvious native layout.
- `TORUS1` — Like `GRID` but the first and the last vertex in each "row" are connected by an additional edge such to form a 1-torus (a cylinder). No native layout is provided.
- `TORUS2` — Like `GRID` but the first and the last vertex in each "row" as well as the first and the last vertex in each "column" are connected by an additional edge such to form a 2-torus (a doughnut). No native layout is provided.

```
INPUT: Desired number of vertices $n \in \mathbb{N}$ and number of coefficients $k \in \mathbb{N}$.
OUTPUT: Surface mesh with approximately $n$ vertices.
ROUTINE
    Choose random radius $0 \leq r \leq \frac{1}{2}\sqrt{n}$
    Choose random length $2r \leq l \leq 2\sqrt{n}$
    Choose random coefficients $\alpha_1, \ldots, \alpha_k$ independently between 0 and $1/k$
    Define $g(z) \leftarrow \sqrt{r^2 - (r-d)^2}$ for $d \leftarrow \min\{z, l-z\}$ if $d \leq r$ or else $g(z) \leftarrow r$
    Define $f(z) \leftarrow g(z)\bigl(1 + \sum_{i=1}^{k} \alpha_i \sin(i\pi z/l)\bigr)$
    $S \leftarrow \{(f(z)\sin(\phi), f(z)\cos(\phi), z)\}$ for $0 \leq z \leq l$ and $0 \leq \phi \leq 2\pi$
    Lay a mesh over $S$ such that the edge length is as close to 1 as possible
END
```

ALGORITHM 5.12: Conceptual operation of the `BOTTLE` generator. The logic for determining the mesh and projecting it on a two-dimensional surface is omitted because it is tedious but not very enlightening.

- `LINDENMAYER` — Uses a stochastic L-system to derive a graph by performing randomly chosen "productions" to a graph, replacing individual vertices with more complicated substructures in each step (*cf.* fig. 5.5). A native layout is provided according to a non-trivial set of rules outlined in § 5.1.3.
- `QUASI`$\langle n \rangle$`D` for $d \in \{3, \ldots, 6\}$ — Projects a primitive cubic lattice in a d-dimensional hyperspace onto a 2-dimensional plane intersecting that space at a random angle. The native layout follows immediately from the construction. The patterns are regular but potentially aperiodic.
- `MOSAIC1` — Starts with a regular polygon and randomly divides facets according to a set of fairly simple rules (*cf.* fig. 5.10) until the desired graph size is reached. The native layout follows by-construction.
- `MOSAIC2` — Acts like `MOSAIC1` but uses less random bits in order to obtain more symmetric graphs.
- `BOTTLE` — Constructs a graph as a three-dimensional mesh over a random solid of revolution. The native layout is the axonometric projection thereof.

We have not discussed the computational complexity of these generators because they are not always straight-forward to quantify – some generators use nontrivial data structures and their exact efficiency depends on the concrete access pattern which is probabilistic. That said, the complexity of most generators is not considerably worse than linear and

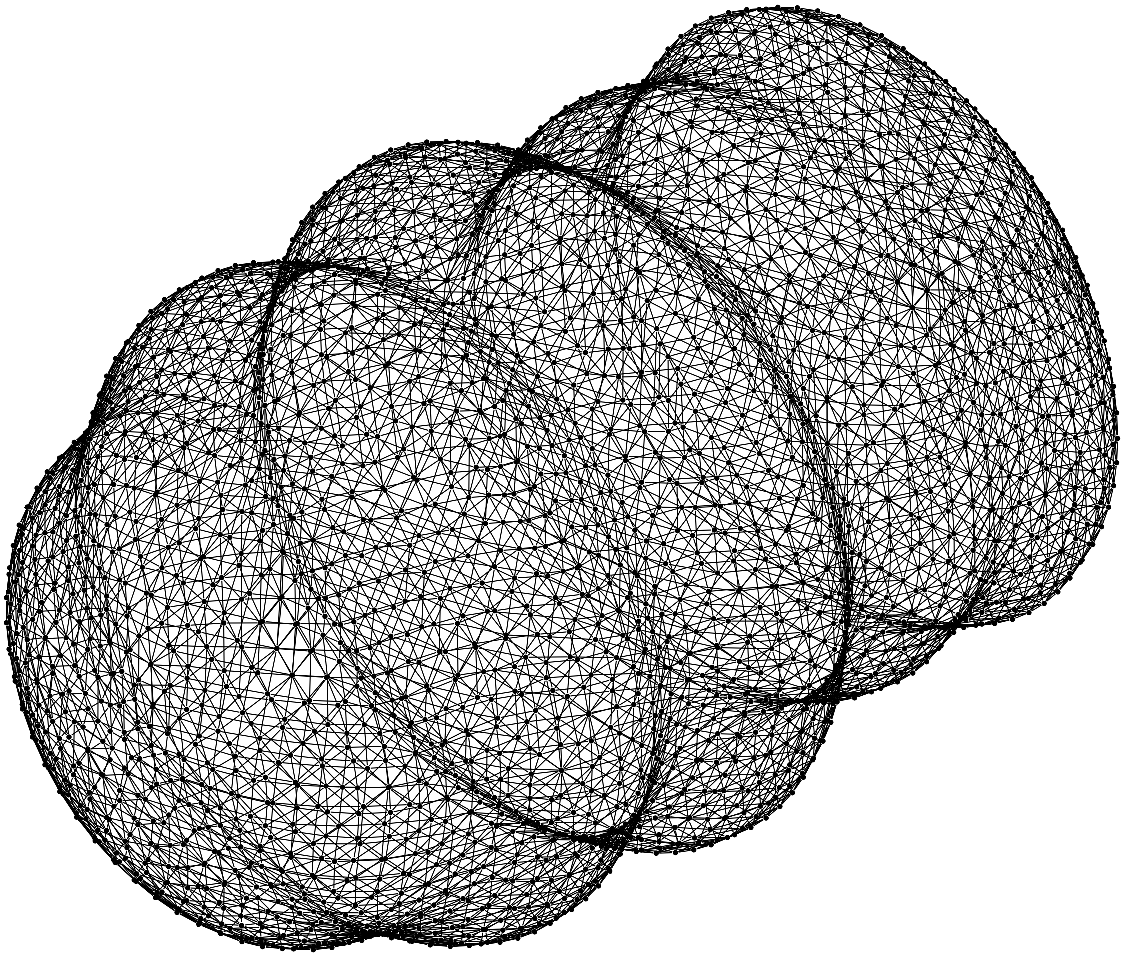


FIGURE 5.13: Example of a graph with 2 974 nodes and 11 877 edges output by the `BOTTLE` generator (native layout).

certainly negligible compared to the complexity of subsequent computation we carry out on these graphs.

5.2 Layouts

Apart from native layouts – `NATIVE` (§ 5.2.1) – we implemented two proper – `FMMM` (§ 5.2.2) and `STRESS` (§ 5.2.3) – as well as three garbage – `RANDOM_UNIFORM` (§ 5.2.4), `RANDOM_NORMAL` (§ 5.2.4) and `PHANTOM` (§ 5.2.5) – layouts (*cf.* § 3) that will be described in the following subsections. The choice of layout algorithms was driven by the desire to get a good mixture of different layouts with a great variety of aesthetic quality.

As with graph generators, layout tools are implemented as small stand-alone programs that read a graph description from a GraphML file, compute the layout and output it to a GraphML file, optionally alongside with some meta-information such as the layout's bounding box in JSON format. Our driver script reads a configuration file `layouts.cfg` that specifies what layouts are desired for graphs of different sizes. Since the time and memory requirements of the layout algorithms vary greatly, it makes sense to compute certain layouts only for smaller graphs. The driver will then loop over the graphs in the database, check whether the desired layouts already exist and if not generate them and update the database accordingly.

Upon insertion into the database, each layout is assigned a unique ID that is chosen randomly. We have abandoned the idea of using a fingerprinting technique as we do for graphs because dealing with the inevitable collisions was too much hassle compared to the little inefficiency of having some layouts in the database under two names.

5.2.1 Layouts from External Sources (`NATIVE`)

Some graphs come along with an existing layout that was either hand-crafted or fell out of the graph generation process. Many of our graph generators described in section 5.1 produce such *native* layouts. The quality of a native layout is expected to be high. There is no further commonality among native layouts. Please refer to figures 5.2, 5.7, 5.9, 5.11 and 5.13 for an inspiration. `NATIVE` is not a layout algorithm (*cf.* def. 1.8) in the strict sense because it does not derive a layout for a given graph from first principles. Actually, it does nothing but to create a symbolic link.

5.2.2 Fast Multipole Multi-Level Method (`FMMM`)

The *Fast Multipole Multi-Level Method (FMMM)* introduced by Hachul and Jünger [18] in 2005 and *Stress Minimization* (see the following section) suggested by Kamada and Kawai [25] in 1989 and thenceforth significantly optimized stand in as two examples of state-of-the-art general-purpose layout algorithms. We've used the implementation available in the OGDF. Both layout algorithms can be classified as *energy-based* (*a.d. force-directed*) algorithms. Please see Kobourov [29] for a good and relatively recent summary and literature survey on the topic.

The basic idea of an energy-based layout algorithm is to define a multivariate *energy function* for layouts Γ of a graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ for $n \in \mathbb{N}$ as

$$f_G : \begin{array}{lcl} \mathbb{R}^{2\times n} & \to & \mathbb{R} \\ (\Gamma(v_1), \dots, \Gamma(v_n)) & \mapsto & \textit{implementation-defined} \end{array} \tag{5.1}$$

that receives as inputs the (ordered) sequence of vertex coordinates $\Gamma(v_1), \dots, \Gamma(v_n)$ and outputs a single scalar quantity. f ought to be a pure function that produces identical output for identical inputs. We use the subscript "G" to indicate that it may also make use the information about the graph G.

An energy-based layout algorithm will try to find vertex coordinates $\Gamma(V)$ such that $f_G(\Gamma(V))$ will be minimized. The usual global and local numerical optimization procedures for highly-dimensional problems may be employed for this purpose.

The name of these algorithms stems from the physical interpretation of a graph as a system of spheres (vertices) that exercise a repulsive force upon each other and are connected by springs (edges) which have a preferred length and exercise a contracting or extending force if stretched or compressed respectively. The function f_G describes the potential energy in the system. The idea is that if such a system would be set up and then allowed to let go, it would rearrange itself into a position that is supposed to be a good layout. A straight-forward way to solve the problem numerically is therefore to form the gradient

$$\frac{\partial f_G}{\partial \Gamma}(\Gamma(V)) \tag{5.2}$$

of the energy function with respect to vertex coordinates and interpret it as a *force* upon vertices. Starting with an initial layout (which can be a random arrangement of vertices) this force introduces an *acceleration* acting upon the vertices which therefore start to move, transforming potential into kinetic energy. By continuously removing kinetic energy from the system via a non-conservative frictional force, the simulation of the system will eventually reach a steady state. The frictional force may be varied during the course of the simulation allowing more rapid movement at the beginning and only fine adjustments near the end. This has an analogy of reducing the temperature of a viscous medium embedding the system. The strategy is also known as *simulated annealing* for its similarity to the metallurgic process. The idea was brought to numeric optimization by Kirkpatrick, Gelatt, and Vecchi [27]. These strategies can be applied to arbitrary numeric optimization problems but in the context of graph drawing, the physical analogy is especially illustrative.

The `FMMM` layout uses the FMMM implementation found in the OGDF at its default settings. It produces good-quality layouts and runs very fast even for big graphs. FMMM layouts are very good at "discovering structure" of graphs even in higher dimensions than two such as for graphs from our `TORUS2`, `TORUS3` (§ 5.1.2) or `BOTTLE` (§ 5.1.6) generators. On the other hand, FMMM is not so good at ensuring precise angles and sometimes even tends to "tie a knot" into graphs that would actually be planar. The latter phenomenon is a case of the algorithm's failure to find a global optimum and getting stuck in a sub-optimal local optimum instead.[8] These characteristics are illustrated in figure 5.14.

The FMMM (*a.d.* FM[3]) was invented by Hachul and Jünger [18] in 2005. We will describe its key aspects in the remainder of this section in moderate detail.

The energy function used in the FMMM is ugly to write down and not actually needed so let us mention its derivative, the force function, instead. For a pair of vertices at distance d there exists a repulsive force

$$F_{\text{node}}(d) = -\frac{\alpha}{d} \tag{5.3}$$

and in addition for a pair of connected vertices an attractive force

$$F_{\text{edge}}(d) = \beta \log\left(\frac{d}{l}\right) d^2 \tag{5.4}$$

where l is the desired edge length (which might be different for different edges) and α and β are positive constants. Note that the force of an edge in equation 5.4 can actually become repulsive if the edge is contracted below its desired length l causing the result of the logarithm to become negative. Equations 5.3 and 5.4 are taken from the original publication [18, § 2.1][9] but the FMMM is actually flexible enough to work with other force models as well and the OGDF actually provides a plugin framework.

When solving the resulting differential equation naively, force-directed models quickly become slow for even moderately-sized graphs because the number of node-node interactions grows quadratic in the size of the graph. The major contribution of the FMMM is a way to keep the overall runtime of the algorithm sub-quadratic nevertheless. It achieves this by the combination of two strategies.

[8] Interestingly, we've found that feeding the algorithm the native, planar, layout as additional information (which we normally don't) does not reduce this tendency significantly and sometimes even seems to have an adverse effect.

[3] The superscript "3" is not a footnote mark but an exponent as in $\mathrm{M}^3 = \mathrm{MMM}$. Except on this page where it just so happens to be a footnote, too.

[9] The inverse formula $1/d$ for F_{node} rather than the inverse-square $1/d^2$ as it would be expected from Coulomb's law was taken directly from the original paper [18]. Given that they repeatedly refer to the nodes as "point charges" and also seem to apply reasoning taken directly from electrostatics, this might be a typo in the paper. However, then again, F_{edge} also doesn't follow Hooke's law and is still referred to as a "spring force". (Of course, there are also physical examples like gas springs that don't follow Hooke's law either.)

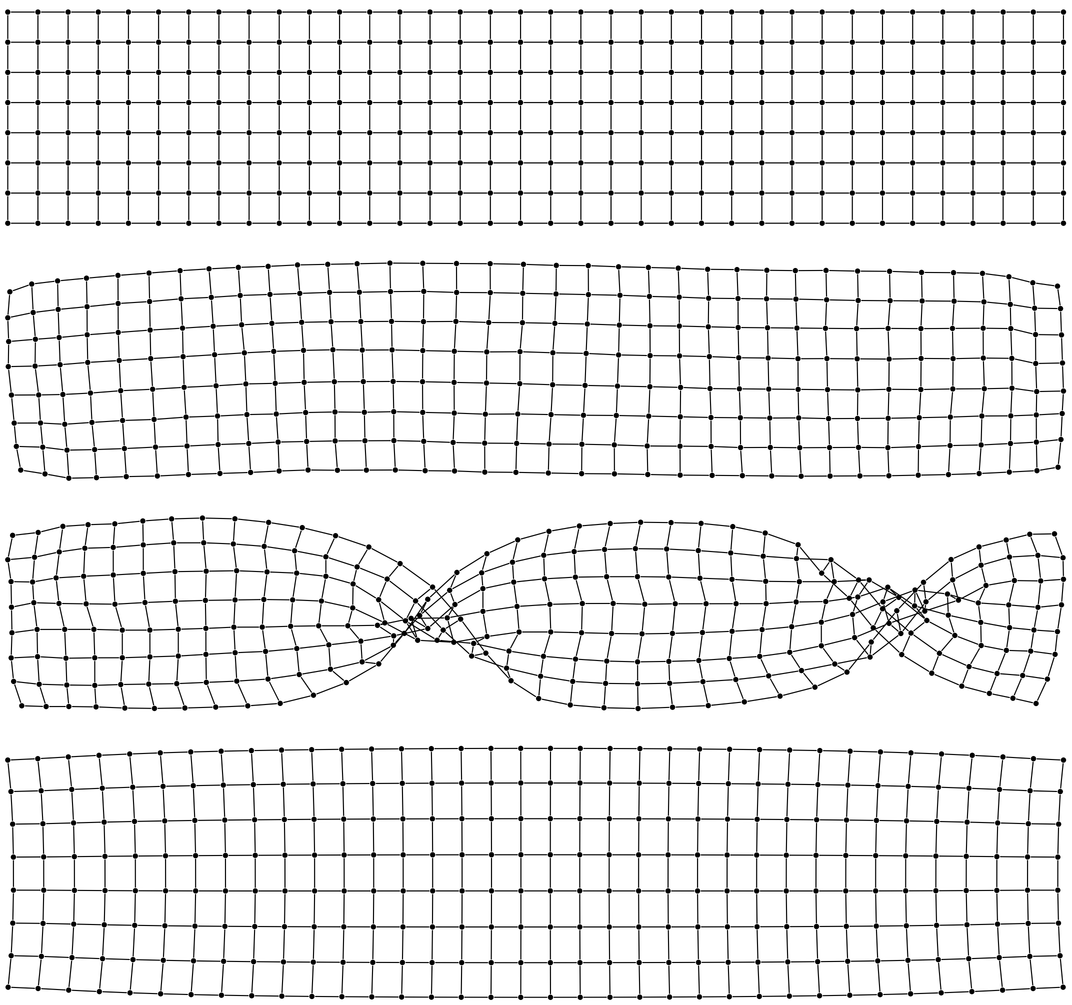


Figure 5.14: Characteristic features of FMMM and STRESS layouts illustrated. While FMMM is generally good at finding a faithful representation of the graph's structure, it doesn't produce very precise angles. Compare the NATIVE layout of a regular grid at the top with the force-directed layouts in the middle. The upper middle one is about as good as it gets with FMMM. Occasionally, the algorithm also "ties a knot" (or actually two, in this case) into the graph as can be seen in the layout at the lower middle. The two middle layouts were computed using exactly the same inputs except for a different random seed. Shown at the very bottom is a STRESS layout which is reproducible in this quality. The graph for this example was chosen with guile – FMMM struggles especially hard with narrow stripes as the graph here where it is actually more likely than not to mess up. STRESS layouts are generally not affected by such phenomena. The layouts shown in this figure were rotated afterwards in order to align the major principal axis horizontally; the algorithms don't do this automagically.

5.2.2.1 The Multi-Level Approach

Rather than laying out the whole graph G at once, a multi-level layout algorithm will use a series of graphs $G = G_0, G_1, \ldots, G_k$ with geometrically decreasing sizes chosen such that G_k is of constant size. It will then start by finding a layout for G_k (which is supposed to be quick) and use that coarse layout as an aid for finding the layout for G_{k-1} more quickly and so on until a layout for the original graph is obtained.

Ideally, the coarse graphs should still be a faithful approximation of the structure of the larger graph. A seemingly straight-forward way to achieve this is the repeated contraction of randomly chosen edges, which preserves the graph's clustering. Other coarsening strategies have been investigated as well [29]. Of course, if the layout for G_i should be any good for finding a layout for G_{i-1}, it has to take into account the additional space requirements for the omitted nodes. This is why it is important that the desired edge length in equation 5.4 is allowed to depend on the edge even if the edges in the original graph all have unit weight.

The FMMM does not contract random edges but uses a niftier approach of labeling each vertex in the graph G_i as either a "sun", "planet" or "moon" node where the set of sun nodes is built in an iterative process by choosing a random vertex as sun node and then marking (deleting) all vertices with distance 2 or less. This is repeated until all vertices are marked (deleted). The set of planet nodes is then defined as those vertices that are direct neighbors to a sun node and the remaining vertices are labeled as moon nodes. This labeling can be done in linear time. The next coarse graph G_{i+1} contains as vertices the "solar systems" of G_i where a solar system consists of a sun, its adjacent planets and those moons that are closest to it. When going into the other direction, the layout for G_{i+1} is used to determine the coordinates of the sun nodes in G_i. The remaining vertices are then initially laid out according to a set of rules that make use of the information collected during the construction of the solar systems and their positions refined via energy minimization.

5.2.2.2 The Multipole Approach

The second optimization employed by the FMMM is to avoid the computation of the forces between each individual pair of vertices and instead estimating the force acting upon a node by an approximation that is faster to compute. In the FMMM only F_{node} is approximated while F_{edge} is computed exactly. This seems to be a reasonable trade-off as the complexity of node-node interactions will always grow quadratic in the graph size while the number of edges had better grow only moderately or a drawing of a large graph that is not sparse will be an unpleasant mess anyway.

The speedup of the calculation of F_{node} builds upon the superposition principle. The combined potential of a number of nodes inside a sphere (which happens to be a circle in two dimensions) acting upon a node *outside* of this sphere can be described exactly by a function of the distance to the center of the sphere only. A straight-forward expansion of this potential function to an infinite power series was presented by Greengard [17] and is used in the FMMM. The derivative of this series (which is easily computed) gives the desired force. By evaluating the series only to a constant (the authors suggest to use four) number of terms, a constant-time approximation for the combined force exercised by all contained nodes can be computed.

At least the OGDF implementation of the FMMM also seems to use simulated annealing techniques but the original paper [18] does not mention how this is applied and we did not dwell deep enough into the actual implementation to find out ourselves.

5.2.3 Stress Minimization (`STRESS`)

Stress Minimization as suggested by Kamada and Kawai [25] is another form of force-directed layout with a special energy function. For a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ for $n \in \mathbb{N}$, the energy function for stress minimization is defined as

$$\operatorname{stress}_G(p_1, \ldots, p_n) = \frac{1}{2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{K}{\operatorname{dist}(v_i, v_j)^2} \Big(\|p_i - p_j\| - l \operatorname{dist}(v_i, v_j) \Big)^2 \tag{5.5}$$

where $K \in \mathbb{R}$ is a constant and $l \in \mathbb{R}$ is the desired edge length (which would be 100 in our case, *cf.* def. 1.6). The stress for a given layout Γ is therefore

$$\operatorname{stress}_G(\Gamma(V)) = \frac{1}{2} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{K}{\operatorname{dist}(v_i, v_j)^2} \Big(\operatorname{dist}_\Gamma(v_i, v_j) - l \operatorname{dist}(v_i, v_j) \Big)^2 . \tag{5.6}$$

Recall that dist is the graph-theoretical distance (length of shortest paths) between vertices (*cf.* def. 1.2) and $\operatorname{dist}_\Gamma$ is the node distance in layout Γ (*cf.* def. 1.3). Stress minimization therefore aims to achieve $\operatorname{dist}_\Gamma(v_i, v_j) \approx l \operatorname{dist}(v_i, v_j)$ which is a very interesting and distinctive approach. Kamada and Kawai [25] originally used a gradient-based (Newton-Raphson) approach to minimize the stress numerically. The gradient of the stress function was formed analytically rather than using numeric differentiation techniques.

Not only is this approach very intuitive and elegant, it also produces very high-quality layouts that are usually more regular than those found by the FMMM (*cf.* fig. 5.14 and especially fig. 5.15).

Unfortunately, the computation of the stress_G function requires an all-pairs shortest path matrix of G to be computed upfront. Usually [25, 29], the Floyd–Warshall algorithm [12] is suggested for this. However, since it has complexity $\mathcal{O}(n^3)$ regardless of the graph and there exist algorithms such as repeated execution of the Dijkstra algorithm that can run in $\mathcal{O}(nm + n^2 \log(n))$ where $m = |E|$ is the number of edges [37], it seems reasonable to expect better performance of the latter, given that large graphs are often very sparse.

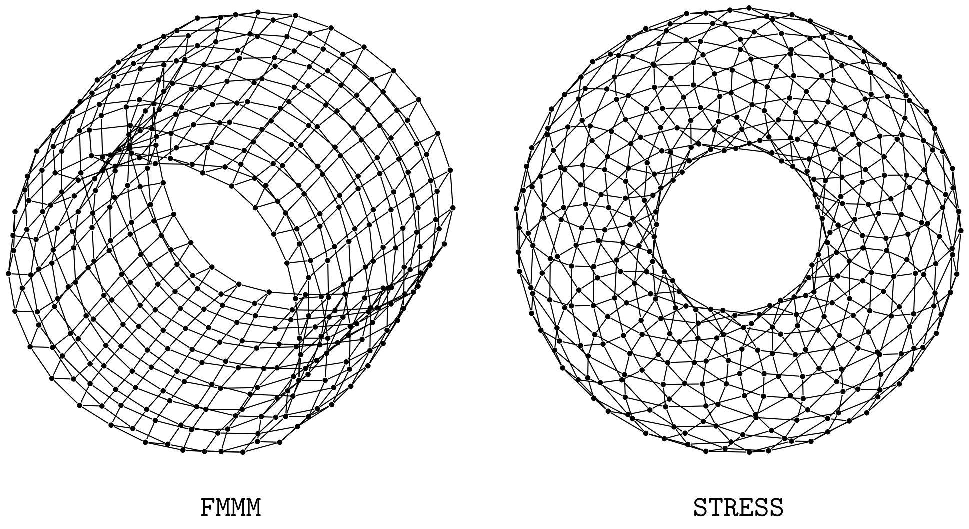


FIGURE 5.15: Comparison of the FMMM and STRESS layouts for a moderately sized ($n = 400$ and $m = 800$) graph generated by TORUS3D. The quality of the FMMM is more difficult to predict.

Gansner, Koren, and North [13] simplified equation 5.5 to obtain a form (*a.d.* “binary stress” [31]) that is suitable for efficient global optimization using a technique known as *majorization* which can be solved using numerical linear algebra. This has desirable properties such as guaranteed conversion and is also very fast given that the bulk of the workload is made up by matrix computations for which highly optimized software libraries are abundant.

Brandes and Pich [4] optimized the procedure even further by avoiding the computation of the full all-pairs shortest paths matrix and instead rely on probabilistic sampling.

Unfortunately, we don’t know what version of stress minimization is implemented in the OGDF (and therefore used in STRESS) as the documentation is unclear about this and we did not look at the details of their implementation.

5.2.4 Random Placement of Nodes (RANDOM_UNIFORM and RANDOM_NORMAL)

In order to generate *really bad* layouts, we wrote two layout algorithms that assign random coordinates to vertices. We have experimented with different probability distributions, namely uniform distributions (RANDOM_UNIFORM) and normal (Gaussian) distributions (RANDOM_NORMAL). Both generate independent x and y coordinates in the unit interval or according the standard normal distribution (with $\mu = 0$ and $\sigma = 1$) respectively and then normalize (*cf.* def. 1.6) the layout afterwards.

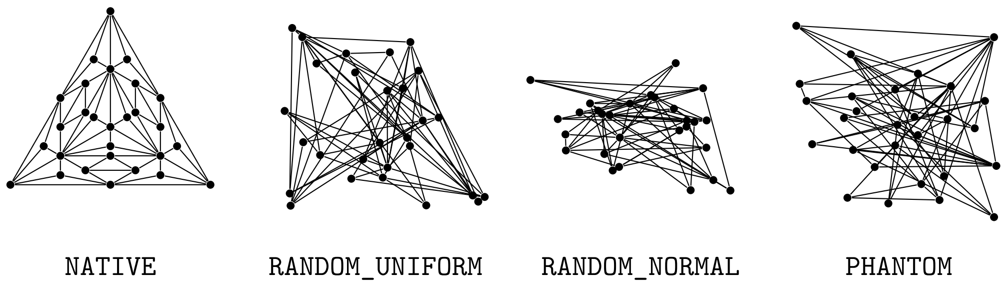


FIGURE 5.16: Comparison of the garbage layouts. The picture on the left shows the NATIVE layout of the graph (generated by MOSAIC2). Sown in the middle are a RANDOM_UNIFORM and RANDOM_NORMAL layout which both look "artificial" in some sense. The picture on the right shows a PHANTOM (§ 5.2.5) layout of the graph which looks more "reasonable".

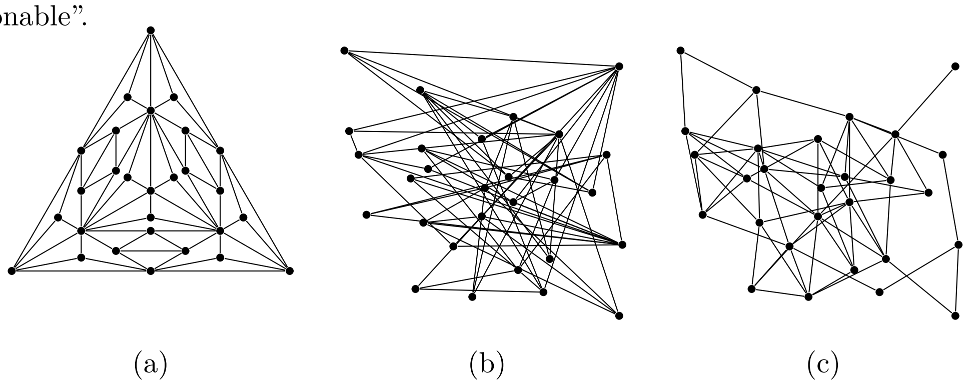


FIGURE 5.17: The picture in (a) shows the graph's (which happens to be the same as in figure 5.16) native layout. The picture in (b) shows the PHANTOM layout. The fore-directed layout of the "phantom" graph that was used to generate the layout is shown in picture (c). Note that the vertex coordinates in (b) and (c) are identical.

Examples of both layouts are given in figure 5.16 which also shows that both random layouts look somehow "artificial". Therefore, we did not use RANDOM_UNIFORM and RANDOM_NORMAL for our experiments and instead went ahead implementing a third garbage layout algorithm that is described in the following section.

5.2.5 Phantom Layouts (PHANTOM)

Given that the random placement of nodes did not seem to create realistic examples of poor layouts, we turned to another strategy that we also ended up using for the final experiment. Instead of assigning random coordinates to vertices of a given graph $G = (V, E)$, the PHANTOM layout algorithm first generates a "phantom" graph $G' = (V, E')$ with $|E'| = |E|$. The generation of this graph is shown in algorithm 5.18. The PHANTOM

```
INPUT: Graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $|E| = m$.
OUTPUT: Random simple graph $G' = (V, E')$ with $|E'| = m$.
ROUTINE
    $E' \leftarrow \emptyset$
    WHILE $|E'| < m$ DO
        Choose random $i, j \in \{1, \ldots, n\}$ according to a uniform distribution
        IF $i \neq j \ \wedge \ \{v_i, v_j\} \notin E'$ THEN
            Insert $\{v_i, v_j\}$ into $E'$
        END
    END
    RETURN $(V, E')$
END
```

ALGORITHM 5.18: Algorithm for the construction of the "phantom" graph for the `PHANTOM` layout (see text). In our implementation we actually use the function `ogdf::randomSimpleGraph` provided by the OGDF.

algorithm then proceeds to compute a force-directed (FMMM) layout Γ' of G' and finally outputs the layout Γ for the actual graph G with $\Gamma(v) = \Gamma'(v)$ for each $v \in V$.

We find that these `PHANTOM` layouts have the desired properties that they look both, terrible and still plausible. An example `PHANTOM` layout together with the "phantom" graph is shown in figure 5.17 while figure 5.16 shows the same layout in comparison with `RANDOM_UNIFORM` and `RANDOM_NORMAL` layouts.

5.2.6 Summary

We have presented and discussed various layout algorithms in order to obtain layouts in different qualities. The discussed algorithms were:

- `NATIVE` (*proper*) — This is not a layout algorithm (*cf.* def. 1.8) but merely a way to refer to layouts that were obtained from external sources such as byproducts from graph generation.
- `FMMM` (*proper*) — Uses the Fast Multipole Multi-Level Method algorithm from Hachul and Jünger [18] implemented in the OGDF [7]. The output is sensitive to the random seed and might vary in quality (*cf.* fig. 5.14). This algorithm has sub-quadratic complexity.

- `STRESS` (*proper*) — Uses the stress minimization [25] algorithm implemented in the OGDF [7]. Unlike `FMMM`, the output of `STRESS` is very predictable and generally of high quality but the complexity (which could be as bad a cubic although we don't know exactly what optimizations are implemented in the OGDF) can make its application impractical for larger graphs.
- `RANDOM_UNIFORM` (*garbage*) — Assigns random coordinates to each vertex according to a uniform distribution over the unit interval. The layouts look abysmal but don't seem to be realistic examples of bad layouts any real layout algorithm might actually produce.
- `RANDOM_NORMAL` (*garbage*) — Like the previous algorithm except that a normal (Gaussian) distribution is used instead. The same remarks apply though to a lesser extent.
- `PHANTOM` (*garbage*) — Generates a "phantom" graph that has the same number of vertices and edges and computes a force-directed layout for that graph; then assigns the same coordinates to the vertices in the original graph. We prefer these as the "best garbage layouts".

Easy ways to obtain even more layouts will be discussed in the following chapter.

6 Data Augmentation

We wish to have a means to derive layouts Γ' from existing layouts Γ in such a way that we know how the quality of Γ and Γ' relate to each other. To this end, we apply *layout worsening* (§ 6.1) and *layout interpolation* (§ 6.2). The theoretical considerations for this were explained in chapter 3. In the remainder of the present chapter, we will explain how we've achieved this by means of *layout transformations*.

We only apply layout transformations to primary layouts. A *primary* layout is one that was produced by one of the layout algorithms described in section 5.2. We apply layout transformations to primary layouts only. If we also were to apply transformations to transformed layouts again, the process would never stop.

6.1 Layout Worsening

Improving an existing layout is a difficult challenge. However, any fool ought to be able to ruin an existing good layout. Therefore, we start with a proper layout Γ that we assume is of reasonable quality and then apply some unary transformation to it in order to degrade its quality. There are several approaches that could be thought of and we implemented more than one. The individual "worseners" we ended up implementing are described in the remainder of this section. Recall from definition 3.1 that each of them is a probabilistic algorithm that takes two inputs: the layout Γ to degrade and a parameter $0 \leq r \leq 1$ that controls how much to degrade it with $r = 0$ implying $\Gamma' = \Gamma$ and $r = 1$ being a request to ruin the layout beyond any restriction.

Each worsener is again a small program that reads the graph and parent layout Γ from a GraphML file and receives any number of parameters $r_1, \ldots, r_n$ as command-line arguments. It will then generate and output layouts $\Gamma'_{r_1}, \ldots, \Gamma'_{r_n}$ in GraphML format again.

The driver script reads a configuration file `worsening.cfg` which lists the desired values of r for each worsening strategy. It will then loop over the primary layouts in the database and check whether the desired worsened layouts already exist. Those that don't, it will compute and update the database accordingly.

6.1.1 Adding Noise to Coordinates (PERTURB)

The most straight-forward way to degrade a given vertex layout is to add some (white) noise to it. This is exactly what the PERTURB worsener does. It displaces the coordinates of each node by independently adding some Gaussian noise. This can be expressed as a simple formula

$$\Gamma'_r(v) = \Gamma(v) + r\Delta^2(v) \tag{6.1}$$

where Δ^2 is a two-dimensional normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 100$ (*cf.* def. 1.6). Since this transformation will usually produce a layout that is not normalized, a normalization step has to be carried out afterwards.

Figure 6.1 shows an example of the application of the PERTURB operation for different rates of degradation. This worsening also has a physical interpretation: adding Gaussian noise to the coordinates can be thought of as increasing the temperature of the system, causing particles to oscillate around their zero position.

6.1.2 Flipping Nodes and Edges (FLIP_NODES and FLIP_EDGES)

Another strategy towards ruining a layout is pairwise exchange of the coordinates of vertices. This transformation for layout Γ of graph $G = (V, E)$ can also be expressed as a simple formula

$$\Gamma'_r(v) = \Gamma(\pi_r(v)) \tag{6.2}$$

where $(\pi_r)_{0 \leq r \leq 1}$ is a family of appropriate permutations such that $\pi_r(v) \neq v$ with probability r. This leads directly to the FLIP_NODES worsener. An example of its application is shown in figure 6.2.

As can be seen from the example, the effects of flipping arbitrary pairs of nodes are quite dramatic. The layout appears completely ruined unless $r \ll 1$. A smoother effect can be achieved by restricting the permutation π_r to τ_r only flipping the coordinates for pairs of adjacent vertices. In this case, however, it has to be taken into consideration that flipping *every* edge does not have the expected devastating effect as it merely "rolls over" the graph. Therefore, the probability for $\tau_r(v) \neq v$ should only be $r/2$. This leads to the FLIP_EDGES algorithm, an example for which is shown in figure 6.3.

6.1.3 Affine Deformations using Moving Least Squares (MOVLSQ)

The layout worsening algorithms described so far all operate locally in the sense that they alter the coordinates of each vertex or pair of vertices in isolation. Let us now introduce an algorithm that operates on the layout as a whole. The idea of this transformation is to deform the drawing area as if the graph were drawn on a sheet of rubber, then $k \in \mathbb{N}$ needles were poked into this sheet at random positions and moved to new (random) positions, "dragging" the drawing with them as the rubber is deformed. Note that

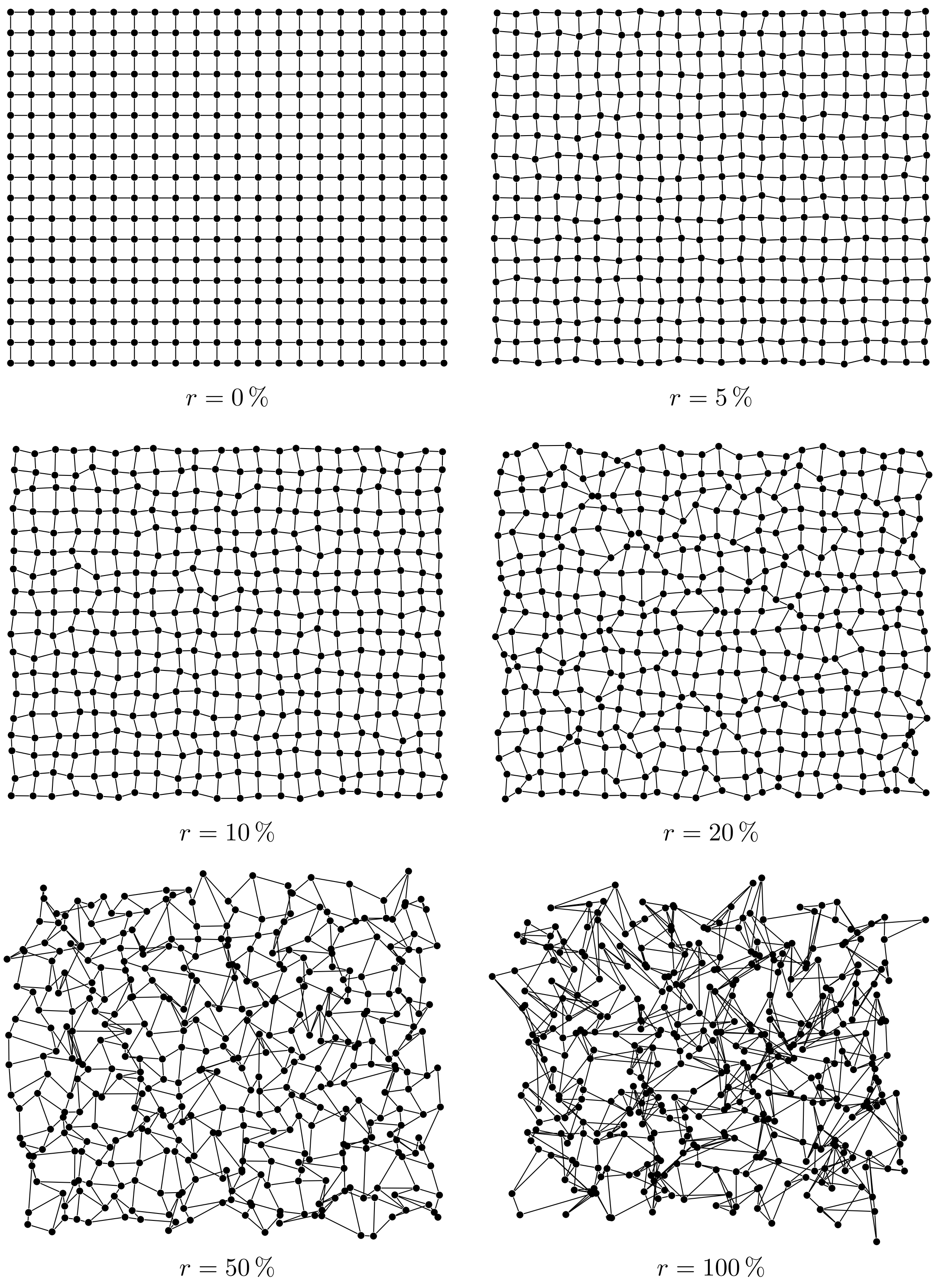


FIGURE 6.1: `PERTURB` application illustrated on a regular grid for increasing rates of degradation.

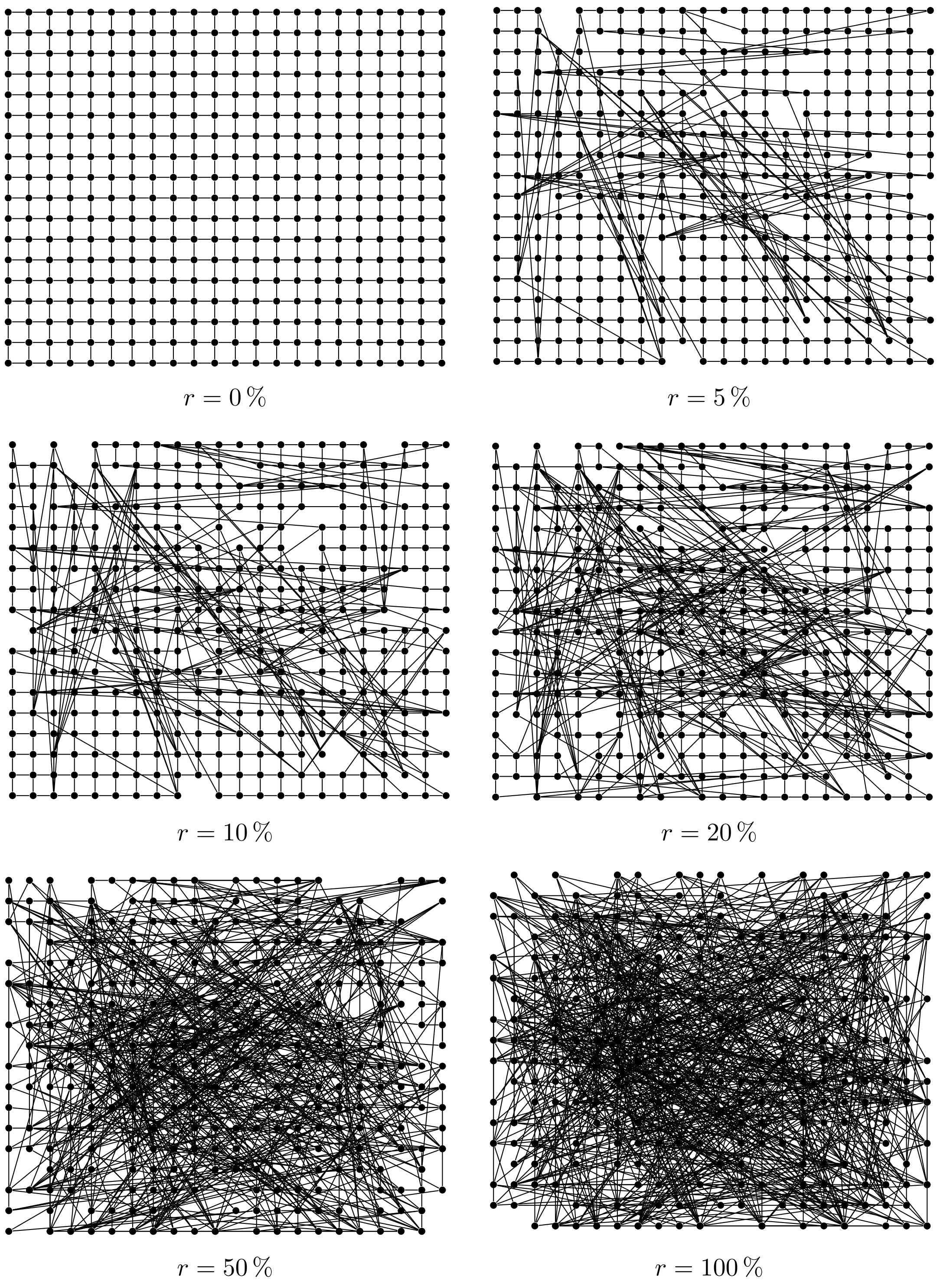


FIGURE 6.2: `FLIP_NODES` application illustrated on a regular grid for increasing rates of degradation.

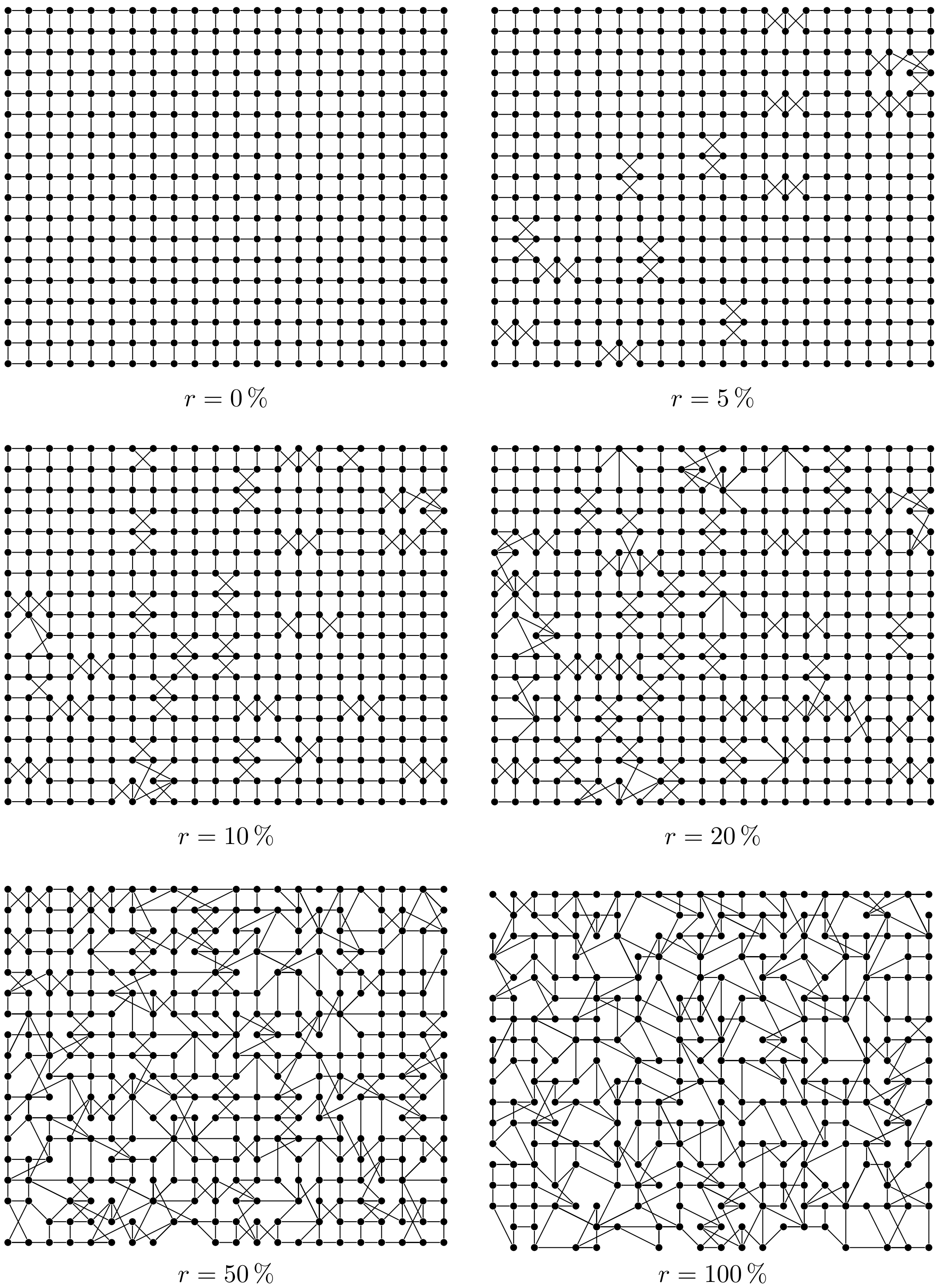


FIGURE 6.3: `FLIP_EDGES` application illustrated on a regular grid for increasing rates of degradation.

since our graph drawings – by definition – always draw edges as straight lines, the transformation only "drags" the coordinates of vertices. Once the new coordinates of the vertices are fixed, the edges are again drawn as straight lines between them. The "rubber sheet" analogy is not to be taken literally. We do not simulate an actual physical deformation process. Instead, the `MOVLSQ` algorithm proceeds as follows.

Let Γ be a layout for the graph $G = (V, E)$ with $|V| = n$ and rectangular bounding box[1] $B \subset \mathbb{R}^2$. The `MOVLSQ` worsener picks $k \in \mathbb{N}$ points $p_1, \ldots, p_k$ selected independently according to a uniform random distribution over B. These coordinates are referred to as the transformation's *source control points.* Another set of points $c_1, \ldots, c_k$ is chosen according to the same distribution and referred to as the *destination control points.* The parameter k is chosen randomly according to a geometric distribution[2] with parameter $n^{-1/2}$ (see footnote) but ensuring $k \geq 5$. These (ordered) sets of points are chosen upfront and can be reused if the transformation is to be performed for multiple values of r. In order to obtain the transformed (worsened) layout Γ'_r for given value of $0 \leq r \leq 1$, control points $q_1, \ldots, q_k$ are chosen as $q_i = (1 - r)p_i + rc_i$ for $i \in \{1, \ldots, k\}$. The transformed coordinates of the vertices are then computed according the affine deformation using moving least squares described by Schaefer, McPhail, and Warren [47, § 2.1].

The essential step of this transformation is reproduced in algorithm 6.4. For the mathematical background, please refer to the cited source. An example of the `MOVLSQ`'s application is given in figure 6.5.

6.1.4 Summary

We have presented four ways to ruin a good layout by a configurable rate. The discussed unary layout transformations were:

- `PERTURB` — Adds white (Gaussian) noise independently to the coordinates of each node. The result is easy to control and the complexity is linear.
- `FLIP_NODES` — Swaps coordinates of randomly selected pairs of nodes. The effect of this is devastating even if only a very small fraction of nodes is affected. The complexity is linear.
- `FLIP_EDGES` — Like `FLIP_NODES` but restricted to flip only connected pairs of vertices. The complexity is the same but the effect is more controlled.

[1]The rectangular bounding box $B \subset \mathbb{R}^2$ of a (finite, non-empty) point set $P \subset \mathbb{R}^2$ is the minimal axes-aligned convex hull. It contains all points $(x, y) \in \mathbb{R}^2$ such that $x_{\min} \leq x \leq x_{\max}$ and $y_{\min} \leq y \leq y_{\max}$ where $x_{\min} = \min\{x : \exists y \in \mathbb{R} : (x, y) \in P\}$ and likewise for the other limits.

[2]A geometric distribution is a discrete probability distribution parameterized by a single parameter $0 < p < 1$. Let X be a Boolean random variable that is true with probability p. Furthermore, let Y be a discrete random variable that counts the number successive zeros while observing a sequence of realizations of X. Then Y is distributed according to a geometric distribution with parameter p. The probability for $Y = n$ for given $n \in \mathbb{N}_0$ is $p(1 - p)^n$. Consult Weisstein [0] for more information.

```
INPUT: Graph $G = (V, E)$ with layout $\Gamma$ and set of parameters $R \subset \mathbb{R}$.
OUTPUT: Set of transformed layouts $\{\Gamma'_r : r \in R\}$.
CONSTANTS: Exponent $0 < \alpha \leq 1$.
ROUTINE
    Determine rectangular bounding box $B \subset \mathbb{R}^2$ of $\Gamma(V)$
    Choose $k \geq 5$ according to a geometric distribution with parameter $n^{-1/2}$
    Choose $p_1, \ldots, p_k$ and $c_1, \ldots, c_k$ according to a uniform distribution over $B$
    FOREACH $r \in R$ DO
        $q_i \leftarrow r c_i$ for $i \in \{1, \ldots, k\}$
        FOREACH $u \in V$ DO
            $v \leftarrow \Gamma(u)$
            $w_i \leftarrow \|p_i - v\|^{-2\alpha}$ for $i \in \{1, \ldots, k\}$
            $p^* \leftarrow \sum_{i=1}^{k} w_i\, p_i / \sum_{i=1}^{k} w_i$
            $q^* \leftarrow \sum_{i=1}^{k} w_i\, q_i / \sum_{i=1}^{k} w_i$
            $\hat{p}_i \leftarrow p_i - p^*$ for $i \in \{1, \ldots, k\}$
            $\hat{q}_i \leftarrow q_i - q^*$ for $i \in \{1, \ldots, k\}$
            $M \leftarrow \sum_{i=1}^{k} w_i \, |\hat{p}_i\rangle \, \langle\hat{p}_i|$
            $a_i \leftarrow M^{-1} \, \langle v - p^* | w_i \hat{p}_i \rangle$
            $\Gamma'_r(u) \leftarrow q^* + \sum_{i=1}^{k} a_i \hat{q}_i$
        END
    END
END
```

ALGORITHM 6.4: Essence of the affine deformation using moving least squares described by Schaefer, McPhail, and Warren [47] applied to unary layout transformations. We use the same symbols as in their publication for the sake of easier comparison. While the algorithm might seem laborious, note that all memory can be allocated before the outermost loop is entered. From that point on, the algorithm can operate using a constant amount of memory. Note further that M is a 2×2 matrix and therefore easily invertible using a closed formula. We fixed $\alpha = 1$ in our code.

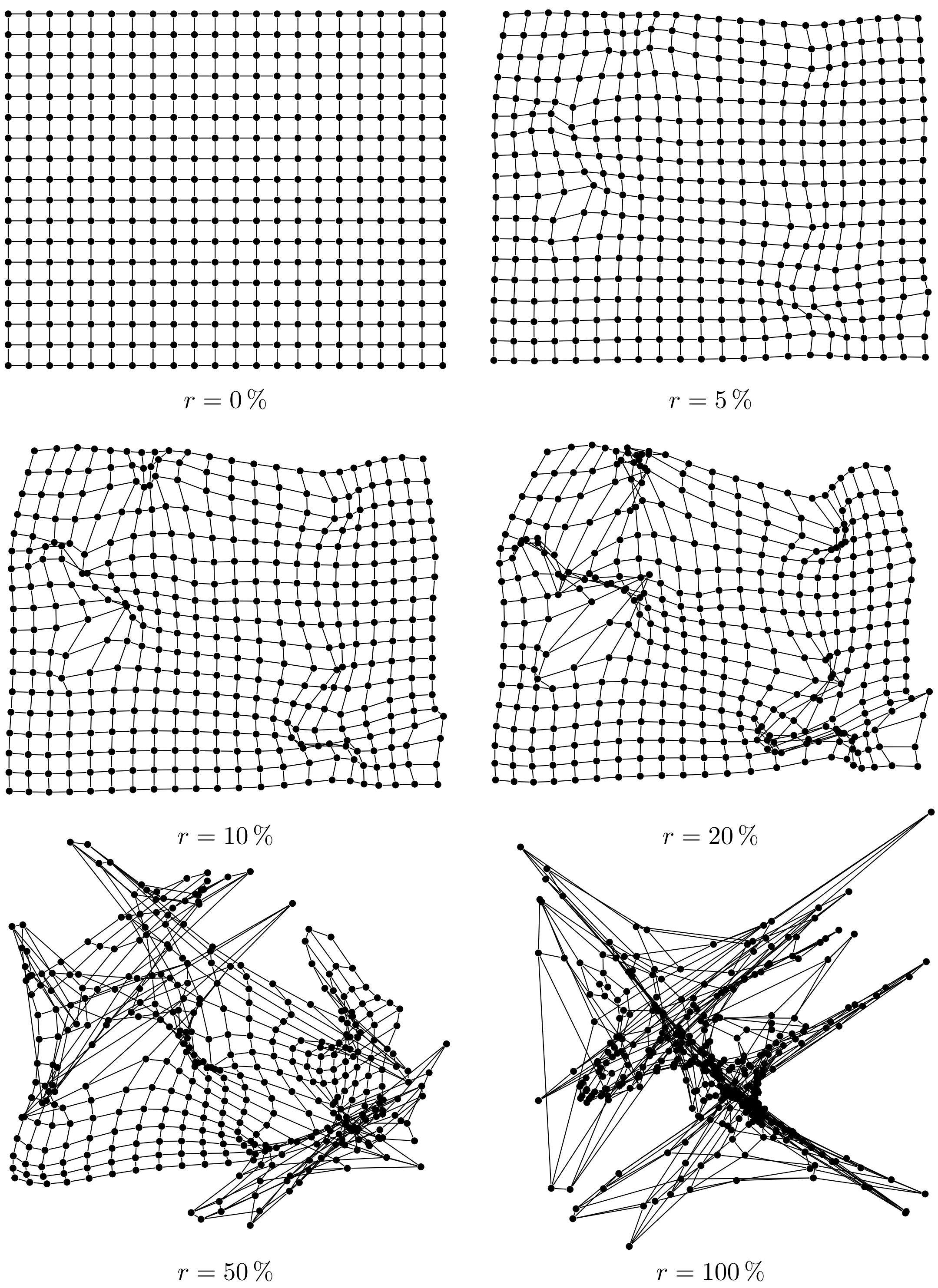


FIGURE 6.5: `MOVLSQ` application illustrated on a regular grid for increasing rates of degradation.

- `MOVLSQ` — Applies affine deformations using moving least squares suggested (although for a different purpose) by Schaefer, McPhail, and Warren [47] to the drawing area in order to move nodes in concert. Since we choose the number of control points proportional to the square root of the graph size and the algorithm has to[3] loop over the control points in order to find the new position of each node, the complexity is $\mathcal{O}(n^{3/2})$ with n being the number of vertices.

6.2 Layout Interpolation

Given two parent layouts Γ_{A} and Γ_{B} of a graph G we wish to obtain intermediate layouts Γ' for G that are "between" the parents Γ_{A} and Γ_{B} for some definition of "in between". We ended up implementing two "interpolators" that are described in the remainder of this section. Recall from definition 3.2 that these a probabilistic algorithms that take three inputs: the layouts Γ_{A} and Γ_{B} to interpolate between and a parameter $0 \leq r \leq 1$ that controls how much weight to give to each parent layout with $r = 0$ implying $\Gamma'_0 = \Gamma_{\mathrm{A}}$ and $r = 1$ implying $\Gamma'_1 = \Gamma_{\mathrm{B}}$. As we shall see, we had to make some trade-offs here and can only provide $\Gamma'_0 \cong \Gamma_{\mathrm{A}}$ and $\Gamma'_1 \cong \Gamma_{\mathrm{B}}$ for one of the interpolators (`XLINEAR`) while the other (`LINEAR`) has other problems of its own. It will be explained later in section 6.2.2 what "$\cong$" means here.

Interpolators are individual programs that read two existing layouts (of the same graph) from GraphML files and accept any number of parameters $r_1, \ldots, r_n$ as command-line arguments. They will then compute and output interpolated layouts $\Gamma'_{r_1}, \ldots, \Gamma'_{r_n}$ in GraphML format again.

The driver script reads a configuration file `interpolation.cfg` that lists the desired values of r for each interpolation strategy. It will then loop over the graphs and primary layouts (*cf.* § 6) in the database as detailed in algorithm 6.6 and compute missing interpolated layouts. Finally, it updates the database.

6.2.1 Linear Layout Interpolation (`LINEAR`)

Given two parent layouts Γ_{A} and Γ_{B} of a graph $G = (V, E)$ one could hope to quickly interpolate an intermediate layout Γ' via

$$\Gamma'_r(v) = (1 - r)\Gamma_{\mathrm{A}}(v) + r\Gamma_{\mathrm{B}}(v) \tag{6.3}$$

for each vertex $v \in V$ where $0 \leq r \leq 1$ is the interpolation parameter. This leads to the `LINEAR` interpolation algorithm.

[3]Schaefer, McPhail, and Warren suggests to evaluate the deformation on a regular grid and then interpolate intermediate points. This was in the context of the deformation of raster graphics. We don't use a regular gird but the coordinates of vertices directly.

```
CONSTANTS: Tolerance $0 < \delta \ll 1$
ROUTINE
    FOREACH interpolation algorithm $\mathcal{A}$ DO
        Let $R_{\text{want}}$ be the set of desired interpolation rates for algorithm $\mathcal{A}$.
        FOREACH graph $G$ in the database DO
            Let $\Gamma_1, \ldots, \Gamma_n$ be the primary layouts for $G$ found in the database.
            FOREACH $i \in \{1, \ldots, n\}$ DO
                FOREACH $j \in \{i + 1, \ldots, n\}$ DO
                    $R_{\text{have}} \leftarrow \{r : \text{layout } \mathcal{A}(\Gamma_i, \Gamma_j, r) \text{ exists in the database}\}$
                    $R_{\text{need}} \leftarrow \{r \in R_{\text{want}} : \forall\, r' \in R_{\text{have}} : |r' - r| > \delta\}$
                    IF $R_{\text{need}} \neq \emptyset$ THEN
                        Compute $\mathcal{A}(\Gamma_i, \Gamma_j, r)$ for all $r \in R_{\text{need}}$ in a single invocation.
                    END
                END
            END
        END
    END
END
```

ALGORITHM 6.6: Actions of the driver script for layout interpolation. Note that this procedure assumes that interpolation algorithms are symmetric in the sense that $\mathcal{A}(\Gamma_i, \Gamma_j, r) = \mathcal{A}(\Gamma_j, \Gamma_i, 1 - r)$ and that the desired interpolation rates are symmetric in the sense that $r \in R_{\text{want}} \Rightarrow 1 - r \in R_{\text{want}}$. We don't know of a good reason to break this assumption.

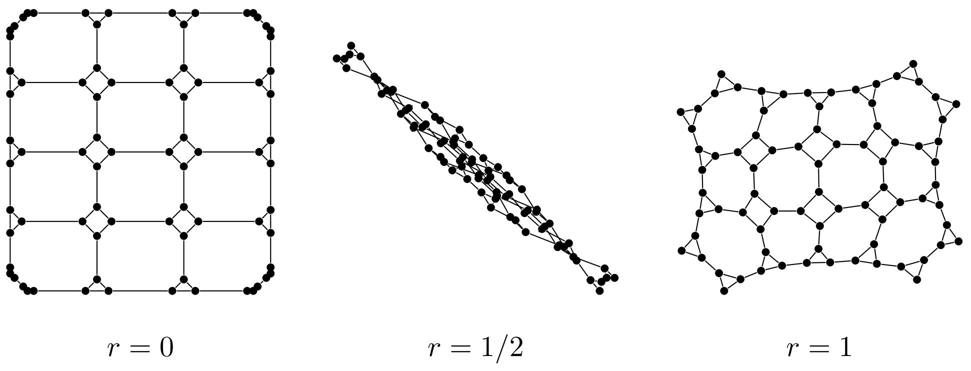


Figure 6.7: The problem with the naive `LINEAR` interpolation illustrated on a seemingly innocent but pathological set of inputs.

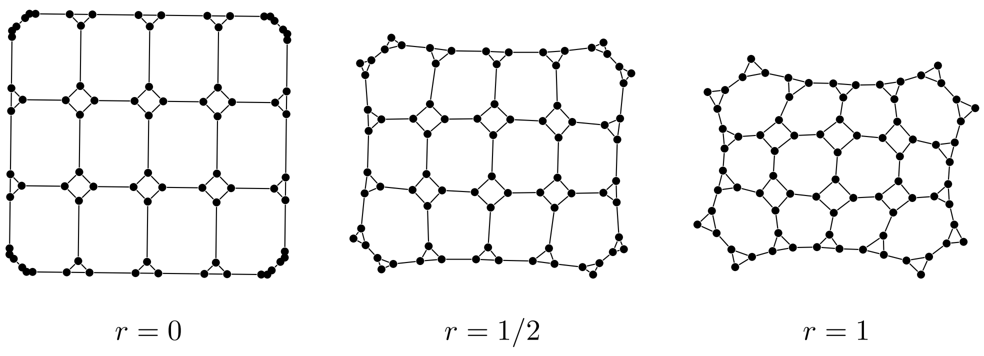


Figure 6.8: The `XLINEAR` interpolation avoids the pathological cases of the `LINEAR` interpolation by preprocessing the parent layouts. Note how it rotated the parent layouts to align the principal axes. (*cf.* fig. 6.7).

Alas, this naive approach doesn't work too well. To see this, consider a simple graph with two vertices v_1 and v_2 connected by an edge. Now assume that $\Gamma_{\mathrm{A}}(v_1) = \Gamma_{\mathrm{B}}(v_2) = (-50, +50)$ and $\Gamma_{\mathrm{A}}(v_2) = \Gamma_{\mathrm{B}}(v_1) = (+50, -50)$. The drawings for both layouts will be identical. However, interpolation according to equation 6.3 will yield a degenerated layout $\Gamma'_{1/2}$ with $\Gamma'_{1/2}(v_1) = \Gamma'_{1/2}(v_2) = 0$. This is very unfortunate as it doesn't meet our expectation at all that the quality of the intermediate layout should be intermediate between the quality of the parent layouts. Here, we had two parent layouts of identical good quality and produced an intermediate layout of the worst quality possible. Another example of this problem with a bigger graph is shown in figure 6.7.

6.2.2 Linear Layout Interpolation with Prearrangement (XLINEAR)

The root cause of the problem with the LINEAR interpolation described in the previous section is that it does not honor the fact that the parent layouts already have a very similar structure. The pathological cases can be mitigated by interpolating not between the original layouts Γ_A and Γ_B but instead preprocess them to obtain layouts $\hat{\Gamma}_\mathrm{A}$ and $\hat{\Gamma}_\mathrm{B}$. Then linear interpolation is performed between $\hat{\Gamma}_\mathrm{A}$ and $\hat{\Gamma}_\mathrm{B}$. Compare figures 6.7 and 6.8 to see how the problem exposed in the former is fixed in the latter.

Unfortunately, this has the undesirable consequence that $\Gamma'_0 = \hat{\Gamma}_\mathrm{A} \neq \Gamma_\mathrm{A}$ and $\Gamma'_1 = \hat{\Gamma}_\mathrm{B} \neq \Gamma_\mathrm{B}$ voiding a property of interpolated layouts that we went to require in the first place. However, $\hat{\Gamma}_\mathrm{A}$ is sufficiently similar to Γ_A and likewise $\hat{\Gamma}_\mathrm{B}$ to Γ_B that we may still safely assume that their aesthetic values are on par (we may write this as $\Gamma \cong \hat{\Gamma}$).

Figure 6.10 shows the XLINEAR interpolation between two proper layouts while figure 6.11 shows the XLINEAR interpolation between a proper and a garbage layout. The graph used in both examples is the same and was taken from the ROME collection.

The preprocessed layouts are determined as follows. Given a layout Γ of a graph $G = (V, E)$ we perform a principal component analysis to obtain the primary axes c_1 and c_2 for $\Gamma(V)$ and the standard deviations σ_1 and σ_2 along them. Writing c_1 and c_2 as the column vectors of a 2×2 matrix C allows us to define

$$\bar{\Gamma} = \Gamma C \tag{6.4}$$

and defining the 2×2 matrix

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \tag{6.5}$$

allows us to define

$$\tilde{\Gamma} = \bar{\Gamma}\Sigma^{-1} \ . \tag{6.6}$$

We also introduce the following four auxiliary 2×2 matrices.

$$\begin{aligned} I_{00} &= \begin{pmatrix} +1 & 0 \\ 0 & +1 \end{pmatrix} & I_{01} &= \begin{pmatrix} +1 & 0 \\ 0 & -1 \end{pmatrix} \\ I_{10} &= \begin{pmatrix} -1 & 0 \\ 0 & +1 \end{pmatrix} & I_{11} &= \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \end{aligned} \tag{6.7}$$

Turning back to our parent layouts Γ_A and Γ_B we set

$$\hat{\Gamma}_\mathrm{A} = \bar{\Gamma}_\mathrm{A} \qquad \text{and} \tag{6.8}$$

$$\hat{\Gamma}_\mathrm{B} = \bar{\Gamma}_\mathrm{B} I^* \tag{6.9}$$

where I^* is chosen among the I_{ij} such that the quantity

$$\sum_{v \in V} \left\| \tilde{\Gamma}_\mathrm{A}(v) - \tilde{\Gamma}_\mathrm{B}(v) I^* \right\|^2 \tag{6.10}$$

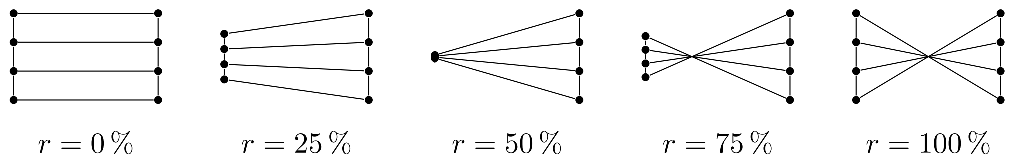


FIGURE 6.9: A remaining pathological case even for `XLINEAR` interpolation. While the interpolation does exactly what it is supposed to do, the quality of the intermediate layout is arguably much worse than that of either of its parents.

is minimized.

This is deemed sufficient in order to eliminate undesired effects introduced by rotations and inflections of the parent layouts. There is still a problem which we don't know how to solve, though. A pathological case is illustrated in figure 6.9.

6.2.3 Summary

We have presented one and a half ways to obtain layouts Γ'_r "in between" two other layouts Γ_A and Γ_B at a configurable rate $0 < r < 1$. The discussed binary layout transformations were:

- `LINEAR` — Assigns $\Gamma'(v) = (1-r)\Gamma_\mathrm{A}(v) + r\Gamma_\mathrm{B}(v)$ which is simple and efficient but can lead to quirky effects (*cf.* fig. 6.7). The computational complexity is linear.
- `XLINEAR` — Attempts to mitigate the problems encountered with `LINEAR` interpolation by trying to align the layouts in the most favorable way before applying the interpolation. This requires a PCA to be performed and voids the property that $\Gamma'_{0/1} = \Gamma_{\mathrm{A}/\mathrm{B}}$. Furthermore, there are still pathological cases where it produces undesirable intermediate layouts (*cf.* fig. 6.9). Its computational complexity is still mostly linear (the complexity of PCA was already discussed in § 4.1).

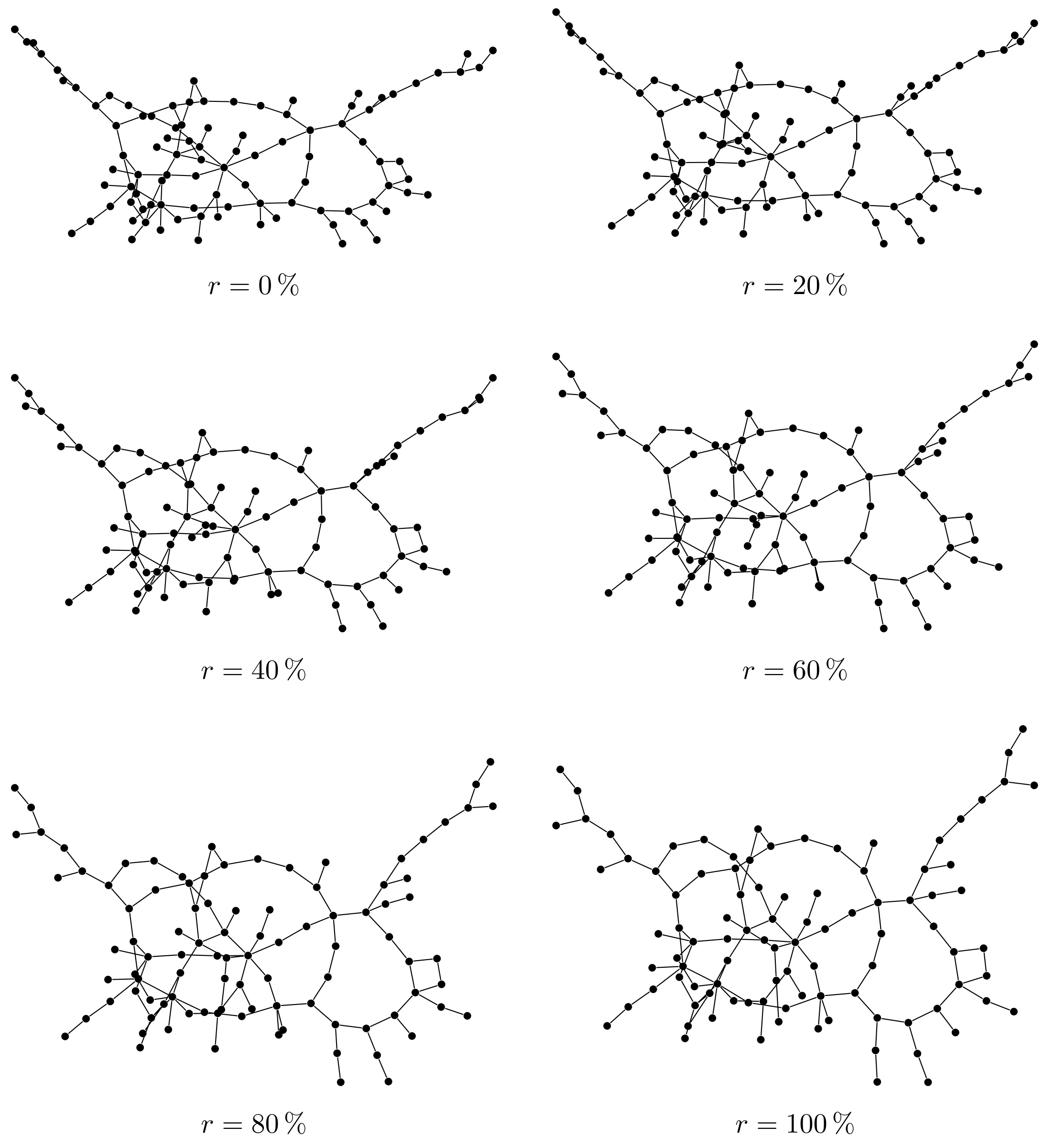


FIGURE 6.10: `XLINEAR` interpolation between two proper (`FMMM` and `STRESS`) layouts shown at differents steps.

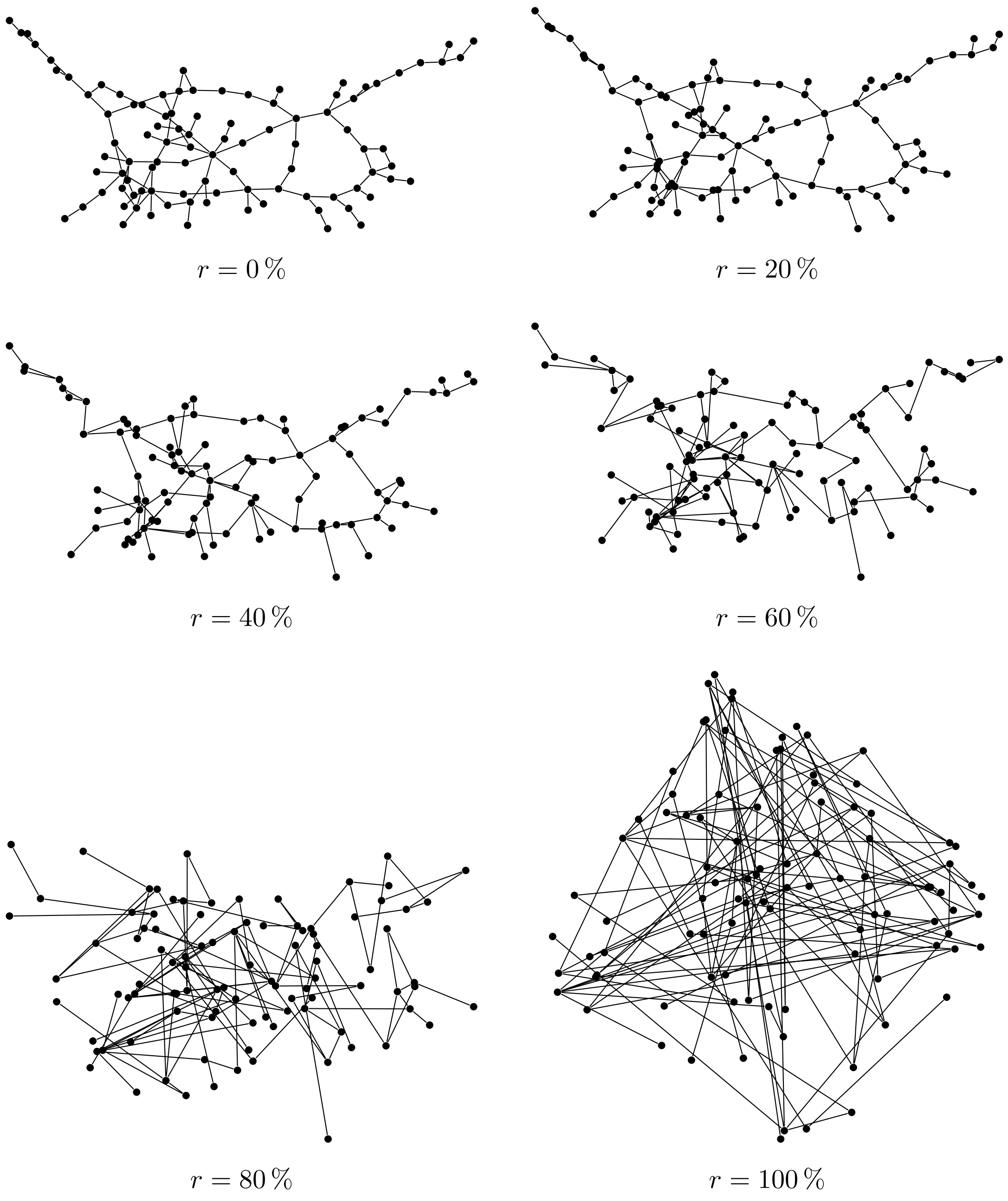


FIGURE 6.11: `XLINEAR` interpolation between a proper (`FMMM`) and a garbage (`RANDOM_UNIFORM`) layout shown at differents steps.

7 Feature Extraction

In chapter 4 we have introduced the notion of *properties*. Those properties are multisets of scalars that can be computed for a given graph and layout which we believe to be somehow valuable syndromes of the layout's aesthetic value. We then explained in chapters 5 and 6 how we proceeded in order to obtain a large corpus of sample data on which we want to try our methods. In the present chapter, we will explain our strategies for converting the properties (which are multisets of generally unbounded size) into fixed-size *feature vectors* that condense the information in a form that is approachable by a discriminator.

Each property introduced in chapter 4 is implemented in an individual program that reads a layout from a GraphML file and accepts various parameters that select its *modus operandi* as additional command-line arguments. It then computes the requested property and performs further data processing that will be described in the remainder of this chapter.

In principle, the collection of raw event data and its processing could be split in two phases. However, we've decided against this mostly for performance reasons. For some properties, the amount of data can become very large and, ideally, we would never want to keep its entirety in memory at any point in time. On a technical level, this is elegantly solved without additional overhead using C++ iterators. However, all our tools also accept a command-line option that will cause them to not perform any analysis and simply output the raw stream of event data.[1] Conversely, we have also written a tool that can read and analyze such a stream. Separating different stages of the work-flow into different programs might elegant from a software engineering point of view, however, formatting millions of floating-point numbers as ASCII text to send them through a POSIX pipe just to parse them back immediately afterwards seems to be too much overhead. Not to mention that some of the analyses we'll present will require multiple passes so the stream would have to be buffered in memory or written to a file on disk, thereby reducing time and space efficiency even further.

The driver script reads a configuration file `properties.cfg` that lists what property to compute for which size classes of graphs. Recall from chapter 4 that the complexity for computing the different properties ranges from $\mathcal{O}(n)$ to $\mathcal{O}(n^3)$ so it is not feasible to compute everything for everything unless one would be willing to consider only small graphs, which we are not. It then loops over all layouts in the database and checks which properties still need to be computed for it. It will then call the aforementioned programs

[1]The respective option is `--kernel=raw` (or its short form `-k raw`).

with the respective arguments in order to compute all missing properties and insert them into the database.

The programs output possibly several text files with the analyzed data (this could be a raw list of events, a histogram or a sliding average evaluated at a number of support points). We will explain shortly why multiple outputs are needed. The text file is in a format approachable by tools like `gnuplot` and actually not needed for the further analysis. However, it may be viewed in the web front-end as plotted chart in order to manually investigate the data set. We also use this output to produce the various plots in this document.

Furthermore, the programs output separate metadata in JSON format which is captured by the driver and stored in a relational database. This metadata contains a fixed number of scalars that will then be further used to train and test the discriminator model that we will introduce in chapter 8.

7.1 Basic Statistic Properties

Let us first introduce some very elementary statistic measures.

Definition 7.1 (Generalized Mean, Arithmetic Mean, Root Mean Squared): *For a non-empty finite multiset $X = [x_1, \dots, x_n] \subset \mathbb{R}$ with $n \in \mathbb{N}$ the generalized mean with exponent $p \in \mathbb{R}_{>0}$ of X is defined as*

$$\operatorname{mean}_p(X) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{1/p} . \tag{7.1}$$

The special case $p = 1$ is referred to as the arithmetic mean while the special case $p = 2$ is called root mean squared (RMS). Notation wise, $\operatorname{mean} = \operatorname{mean}_1$ *and* $\operatorname{rms} = \operatorname{mean}_2$ *will be used.*

Definition 7.2 (Population & Sample Standard Deviation): *For a non-empty finite multiset $X = [x_1, \dots, x_n] \subset \mathbb{R}$ with $n \in \mathbb{N}$ the population standard deviation is defined as*

$$\operatorname{stdevp}(X) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \big(x_i - \operatorname{mean}(X)\big)^2} \tag{7.2}$$

and the sample standard deviation is defined as

$$\operatorname{stdev}(X) = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} \big(x_i - \operatorname{mean}(X)\big)^2} \tag{7.3}$$

provided that $n \geq 3$.

The following relationship between mean and standard deviation of a multiset X exist.

$$\operatorname{stdevp}(X) = \sqrt{\operatorname{rms}(X)^2 - \operatorname{mean}(X)^2} \tag{7.4}$$

$$\operatorname{stdev}(X) = \sqrt{\frac{|X|}{|X| - 1}\Big(\operatorname{rms}(X)^2 - \operatorname{mean}(X)^2\Big)} \tag{7.5}$$

For our analysis, we compute the arithmetic mean and RMS for each property which therefore also implicitly captures the standard deviation without us having to compute it explicitly which is desirable in order to avoid division by zero for pathologically small data sets.

7.2 Histograms

Histograms are a common technique in order to aggregate large amounts of event data into a more approachable form that allows an estimation of the density distribution. An important choice to make when building a histogram is the number of bins. Figure 7.2 illustrates the effect of this choice. There is no shortage of recommendations how to choose the number of bins – or, equivalently, the bin width – for a histogram. Wikipedia [19] alone lists eight formulae to guide with this decision. We have tried several of them and were able to achieve the least disappointing results using *Scott's normal reference rule* [48] which minimizes the integrated mean squared error of a Gaussian distribution.

Definition 7.3 (Scott's Normal Reference Rule): *Let $X \in \mathbb{R}$ be a multiset with $|X| = n$ and $\operatorname{stdev}(X) = \sigma$. Then Scott's normal reference rule suggests to use*

$$h \approx 3.5\,\sigma\,n^{-1/3} \tag{7.6}$$

as the histogram bin width.

Unfortunately, the mathematical property that is optimized by this rule turned out to be of little value for our intents and purposes. As the example in figure 7.2 shows, the bin width chosen according to this rule is not ideal at all to visualize the structure of our data. Even though Scott already acknowledged in his original paper that "the data-based algorithm leads to [bin widths] that are generally too big for all our models of non-Gaussian data" and suggested that "a correction factor may be applied" to equation 7.6, we were unable to find a factor that would be satisfactory at least for the large majority of our data. Eventually, we came to the conclusion that the data we're looking at has very sharp features that are much less common is other statistics domains so the recommendations commonly found in the literature might not be fully applicable to our problem. Given an appropriate user interface and some experience, humans can quickly figure out a good choice of bin width interactively. However, this is of no help to us either because a requirement for human intervention would reduce the purpose of our effort *ad absurdum*. In the end, we decided to defer the problem and not settle for any specific histogram width at all.

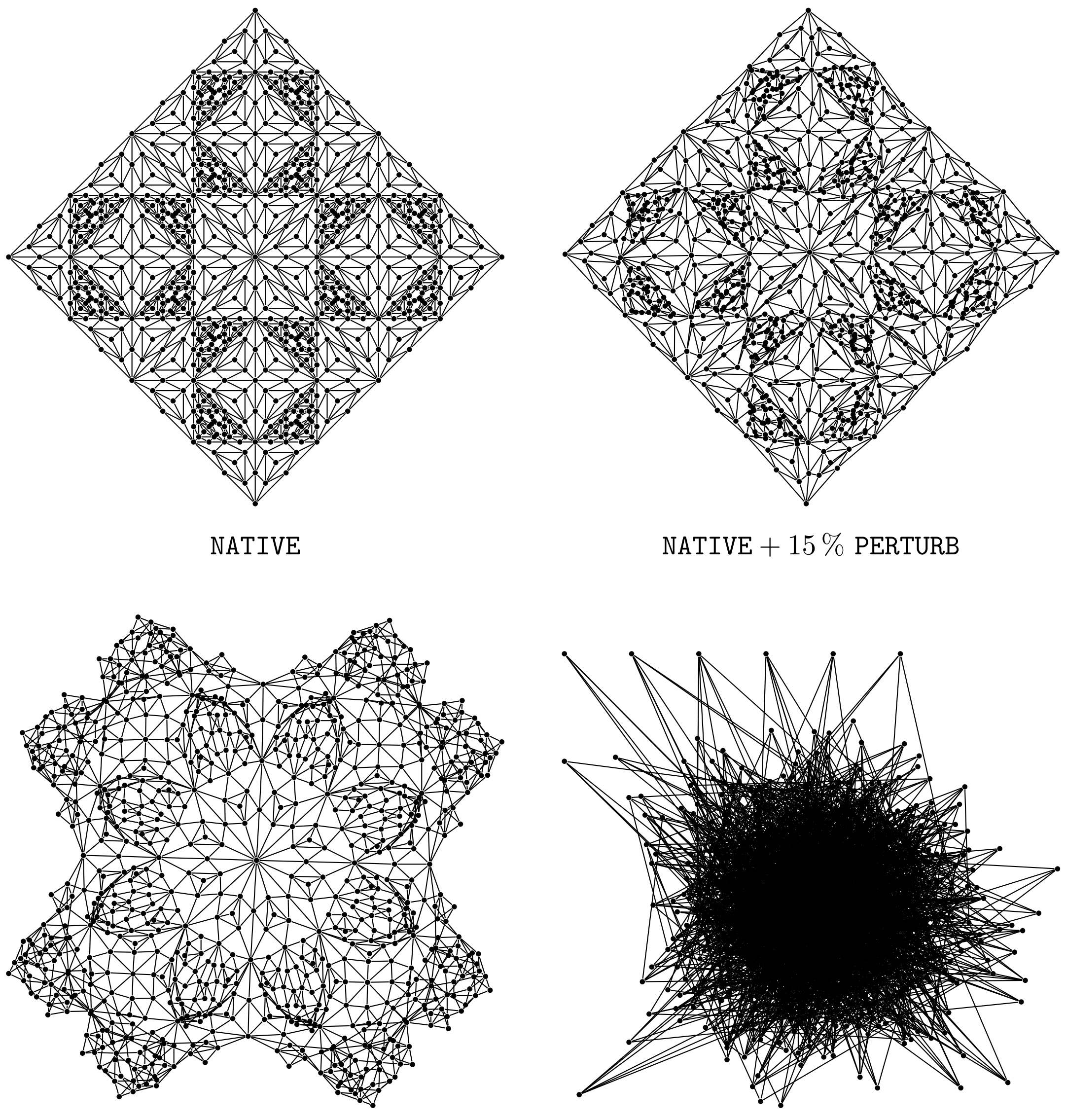


Figure 7.1: Example layouts (all of the same graph) that will be used thoughout this chapter. The graph was generated using the `MOSAIC2` generator. It has 833 nodes and 2 224 edges.

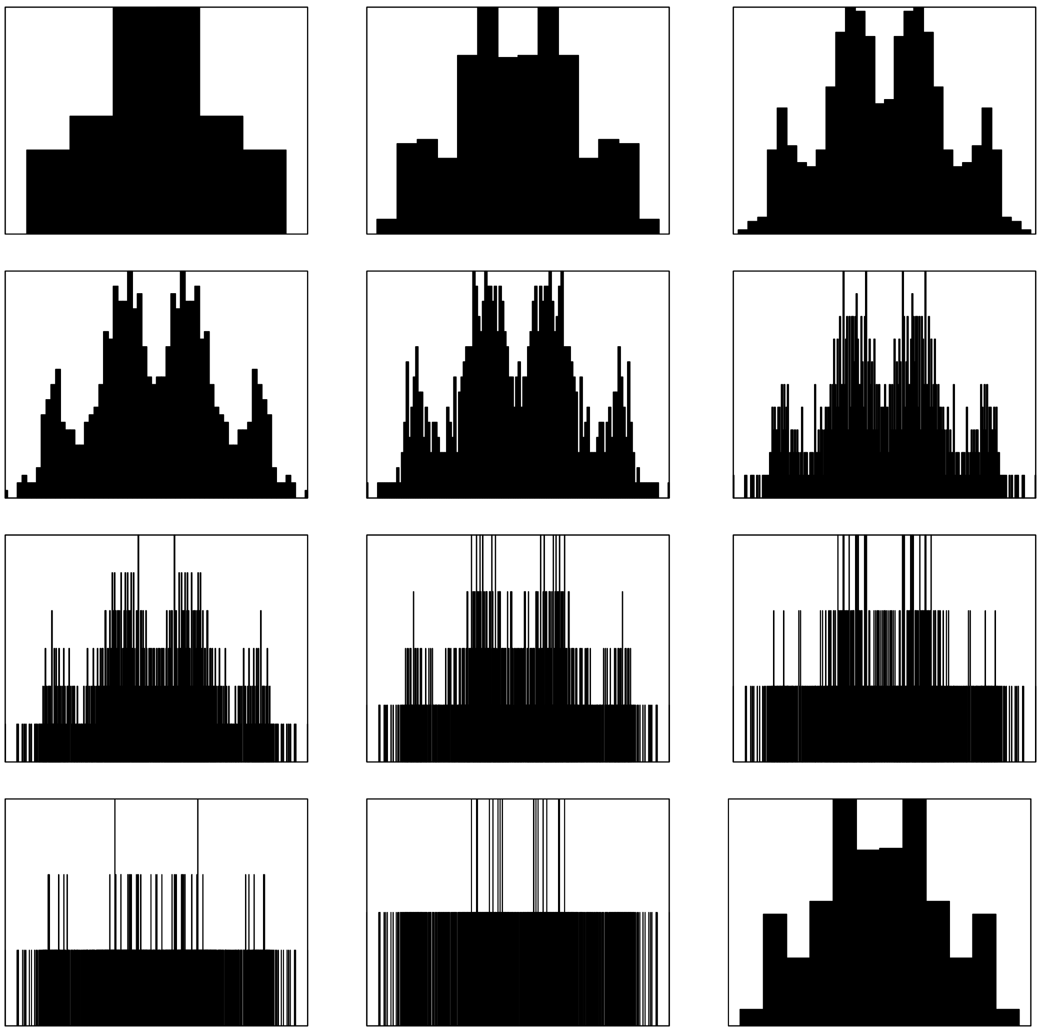


Figure 7.2: Each histogram shows the same data (which is the `PRINCOMP1ST` property of the layout shown in fig. 7.1) but using different bin widths / counts. The bin count doubles for each histogram starting from the top left except for the histogram in the bottom right corner for which Scott's normal reference rule was used to determine the bin width. It can be seen that the histograms in the top row fail to present relevant information by using bins that are too wide to show any interesting details. The histograms in the second row give a fairly good impression of the distribution and let the peaks corresponding to the denser square tiles of the layout become clearly visible. However, starting with the third row, the insight gained from the histograms diminishes again as there is not enough data for a resolution that fine. The two histograms in the last row (except the one on the right) basically allocate each event into its own bin thereby reducing the value of the histogram close to zero. The number of bins chosen according to Scott's normal reference rule is also not ideal because it is too small.

7.3 Sliding Averages

A histogram with constant bin width $h \in \mathbb{R}_{>0}$ for a multiset of events $X = [x_1, \dots, x_n] \subset \mathbb{R}$ with $n \in \mathbb{N}$ may be formalized as a function

$$\begin{aligned} H_h : \quad \mathbb{R} &\to \mathbb{R}_{\geq 0} \\ x &\mapsto \frac{1}{n} \sum_{i=1}^{n} \delta_{\lfloor x/h \rceil, \lfloor x_i/h \rceil} \end{aligned} \tag{7.7}$$

where δ is the *Kronecker delta*[2] which is defined as

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{7.8}$$

for $i, j \in \mathbb{Z}$. The summation function H_h may be generalized to use an arbitrary *kernel* (or *filter*) function $f : \mathbb{R}^2 \to \mathbb{R}_{\geq 0}$ instead of the Kronecker delta. This gives the normalized sliding average

$$\begin{aligned} F_f : \quad \mathbb{R} &\to \mathbb{R}_{\geq 0} \\ x &\mapsto \frac{\sum_{i=1}^{n} f(x, x_i)}{\int_{-\infty}^{+\infty} \mathrm{d}y \ \sum_{i=1}^{n} f(y, x_i)} \end{aligned} \tag{7.9}$$

Which provides a continuous density distribution. A natural choice for the kernel is the *Gaussian function*[3] g_σ with $\sigma \in \mathbb{R}_{>0}$ defined as

$$g_\sigma(\mu, x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{7.10}$$

where the normalization factor is expendable here because it will be canceled in F anyway.

Like histograms, sliding averages using Gaussian kernels can be parameterized on the filter width but unlike the Kronecker delta, the Gaussian kernel will be smooth and not damp local features as aggressively. Of course, the question remains how to choose the filter width σ. Figure 7.3 shows the same data as in Figure 7.2 but this time analyzed with a Gaussian filter using as filter width $\sigma = h/2$ where h is the bin width that would otherwise have been used for the histogram. Interestingly, Scott's normal reference rule seems to be a reasonable heuristic for choosing a Gaussian filter width for our application.

Unfortunately, however, evaluating sliding averages is much more expensive than creating histograms. A histogram for $n \in \mathbb{N}$ events with $m \in \mathbb{N}$ bins can be constructed with $\mathcal{O}(n+m)$ effort and after that, querying for the frequency at any point along the abscissa

[2] E. W. Weisstein. "Kronecker Delta". In: *MathWorld – A Wolfram Web Resource*. URL: `http://mathworld.wolfram.com/KroneckerDelta.html` (visited on 2018-03-12).

[3] E. W. Weisstein. "Gaussian Function". In: *MathWorld – A Wolfram Web Resource*. URL: `http://mathworld.wolfram.com/GaussianFunction.html` (visited on 2018-03-12).

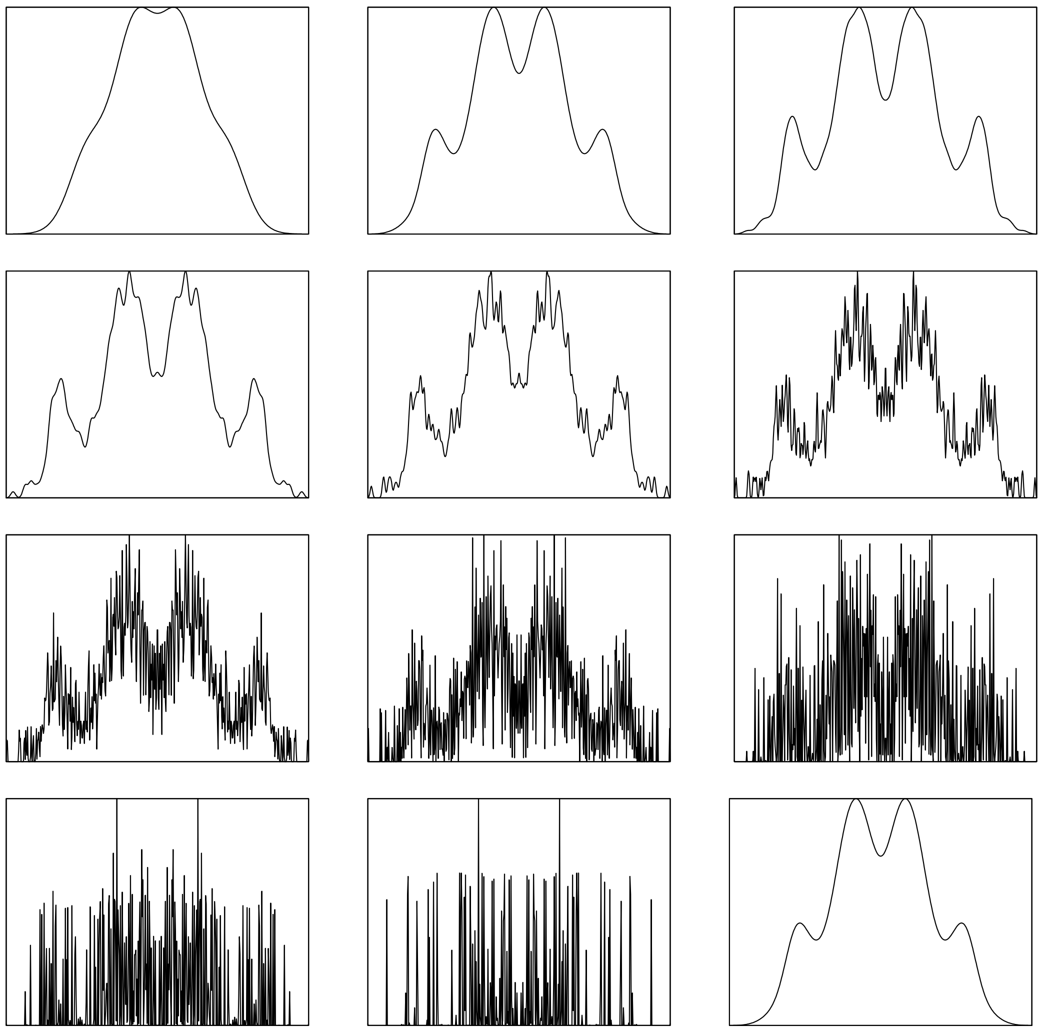


Figure 7.3: The same data as in figure 7.2 (which is the `PRINCOMP1ST` property of the layout shown in fig. 7.1) but this time analyzed using a Gaussian filter with $\sigma = h/2$ where h is the bin width that would otherwise have been used for the histogram. The filter width halves for each plot starting from the top left except for the plot in the bottom right corner for which Scott's normal reference rule was used to determine the filter width. The sliding averages in the top row are still very coarse but, with the exception of the very first image, actually capture useful information. Importantly, so does the image in the bottom right corner that had its filter width chosen according to Scott's normal reference rule. In the second row, the two plots to the left and in the middle look very reasonable while the one right already seems too noisy. The plots in the third row barely capture any information any more while the two pictures on the left and in the middle of the bottom row are completely worthless.

is an $\mathcal{O}(1)$ operation. For sliding averages of the same data using m support points, on the other hand, the setup is a $\mathcal{O}(nm)$ operation as the weighted sum has to be computed for each support while every subsequent query of a value can be done in $\mathcal{O}(\log(m))$ via interpolation between supports (which have to be found via bisection first unless they are distributed equidistantly). Alas, to make matters worse, the number of a sliding average's support points for a satisfactory result usually has to be much larger than the number of bins for an equivalent histogram would be. The problem could be mitigated if the event data were *sorted* in which case one could stop the summation in equation 7.9 once the filter drops below a certain threshold. This, however, we cannot do either because sorting would require us to compute the entire data set upfront which we want to avoid for reasons discussed in section 4.4.

One optimization we were able to implement was to use an adaptive strategy for the selection of support points in order to be able to reduce their number. The idea is that we first distribute a small fixed number of support points equidistantly over the data range. Then we intensify the support points recursively by probabilistic sampling of additional points in the intervals between existing supports. Once the newly added points are found to be approximated to satisfactory accuracy via linear interpolation between the existing enclosing supports, the recursion is stopped. This procedure causes us to use densely spaced support points in regions where the distribution function has a high curvature to achieve reasonable approximation while using only loosely spaced support points in regions of low curvature to be more economic. An example of this evaluation strategy is shown in figure 7.4 and the algorithm is detailed in 7.5. Unfortunately, there is a non-zero risk that the initial sampling process will completely miss a very sharp peak in the function. Therefore, if a number of support points is specified on the command-line, our programs will honor it and not use the adaptive strategy. We have used a fixed generous hand-tuned number of support points for most plots in this printing to ensure high quality but use the automatic adaptive strategy in our unattended experiments.

7.4 Entropy

In many of the examples we introduced in chapter 4, we mentioned that a distinguishing property of regular versus not-so-regular layouts is that the former have a less uniform distribution of the various properties. We wish to capture this information. *Information Entropy*, pioneered by the work of Shannon [49], provides a measure for this that is conveniently expressed as a single scalar value.

Definition 7.4 (Entropy of Histogram): *Let H be a histogram with $n \in \mathbb{N}$ bins that have the values (relative frequency counts) $H_1, \ldots, H_2 \in \mathbb{R}_{\geq 0}$ such that $\sum_{i=1}^{n} H_i = 1$. Then the entropy of H is*

$$S(H) = -\sum_{i=1}^{n} H_i \log_2(H_i) \tag{7.11}$$

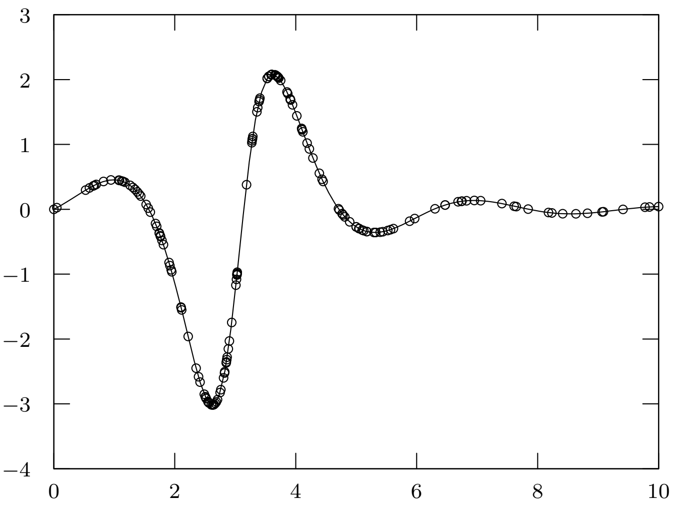


FIGURE 7.4: Example of a function sampled according to our probabilistic adaptive strategy. The support points are drawn explicitly. It can be seen that their placement is random but nevertheless their density is higher in those regions where the function has a high curvature.

where we use the convention that bins with $H_i = 0$ *shall contribute a zero term to the sum.*[4]

Unfortunately, the entropy according to definition 7.4 is highly dependent on the choice of bin width which we have just given up to determine reliably. In order to still get a measure of predictability of the distribution, we therefore don't consider the entropy of any single histogram but of a *series* of histograms with changing bin widths. We have found that the entropy as a function of the logarithm of the bin count can usually be approximated very closely by a linear regression. This allows us to use the parameters (intercept and slope) of the regression function instead of the entropy of any particular histogram which alleviates us from the problem that we don't know how to choose a good bin width automatically. Yet, the regression curves for different layouts show a good amount of variability which looks promising. Figure 7.6 shows a few examples of such regressions.

It might be tempting to generalize the entropy definition from definition 7.4 to continuous data in order to apply it to sliding averages as well. This leads to the following definition.

DEFINITION 7.5 (DIFFERENTIAL ENTROPY): *Let* $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ *be a non-negative steady function normalized such that* $\int_{-\infty}^{+\infty} \mathrm{d}x\ f(x) = 1$. *The differential entropy of* f *is defined*

[4]Because the value of "$0 \cdot \log_2(0)$" is undefined.

```
INPUT: Reasonably smooth function $f : \mathbb{R} \to \mathbb{R}$ defined on interval $x_\mathrm{A} < x_\mathrm{B}$.
OUTPUT: List $L$ of support points $(x, f(x)) \in \mathbb{R}^2$.
CONSTANTS: Initial number of supports $n \in \mathbb{N}$, splitting factor $k \in \mathbb{N}$, relative
           tolerance $0 < \epsilon \ll 1$ and recursion limit $r_\mathrm{max} \in \mathbb{N}$.
ROUTINE
    Set $x_i \leftarrow x_\mathrm{A} + i(x_\mathrm{B} - x_\mathrm{A})/(n+1)$ for $i \in \{0, \ldots, n+1\}$
    Set $y_i \leftarrow f(x_i)$ for $i \in \{0, \ldots, n+1\}$
    $L \leftarrow \{(x_0, y_0), \ldots, (x_{n+1}, y_{n+1})\}$
    $\delta \leftarrow \epsilon \sum_{i=0}^{n+1} |y_i| / (n+2)$
    FOR $i \leftarrow 0$ TO $n$ DO
        RECURSE$(L, x_i, y_i, x_{i+1}, y_{i+1}, \delta, 1)$
    END
    Sort $L$ by the value of the first tuple element
END
SUBROUTINE RECURSE$(L, x_\mathrm{a}, y_\mathrm{a}, x_\mathrm{b}, y_\mathrm{b}, \delta, r)$
    Let $(x_0, y_0) \leftarrow (x_\mathrm{a}, y_\mathrm{a})$ and $(x_{k+1}, y_{k+1}) \leftarrow (x_\mathrm{b}, y_\mathrm{b})$ for convenience
    Choose random values $0 < t_1 < \cdots < t_k < 1$ independently from the unit interval
    Set $x_i \leftarrow (1 - t_i)x_\mathrm{a} + t_i x_\mathrm{b}$ for $i \in \{1, \ldots, k\}$
    Set $y_i \leftarrow f(x_i)$ for $i \in \{1, \ldots, k\}$
    Append $(x_i, y_i)$ to $L$ for $i \in \{1, \ldots, k\}$
    Set $z_i \leftarrow (1 - t_i)y_\mathrm{a} + t_i y_\mathrm{b}$ for $i \in \{1, \ldots, k\}$
    IF $r < r_\mathrm{max}$ AND $\max_{i=1}^{k}\{|y_i - z_i|\} > \delta$ THEN
        FOR $i \leftarrow 0$ TO $k$ DO
            RECURSE$(x_i, y_i, x_{i+1}, y_{i+1}, \delta, r+1)$
        END
    END
END
```

ALGORITHM 7.5: The adaptive sampling algorithm that we used for evaluating sliding averages more quickly (*i.e.* less slowly). We have chose $n = 17$, $k = 2$, $r_\mathrm{max} = 10$ and $\epsilon = 1/20$ for our code.

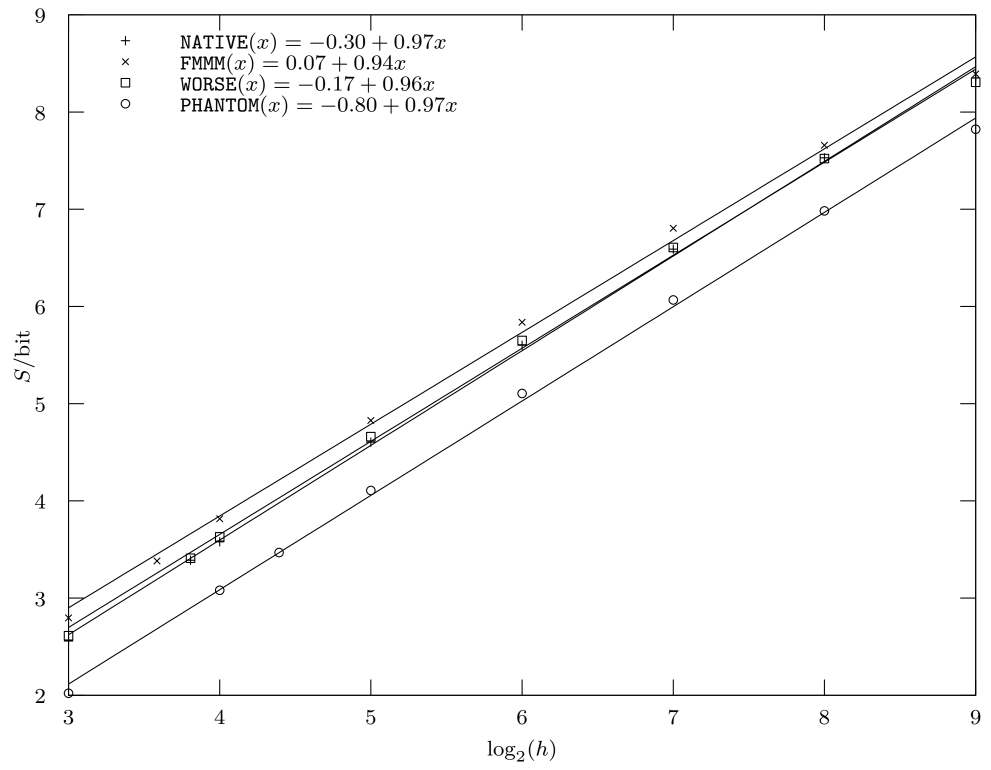


FIGURE 7.6: Histogram entropy for the `PRINCOMP1ST` property computed for the four layouts shown in figure 7.1 plotted as a function of the logarithm of the histogram bin count. It can be clearly seen from the plot that the linear regression fits the data very well. Another interesting observation to make is that the bin widths chosen according to Scott's normal reference rule (which are easily identifiable in the lower left region as they are the only data points with a bin count that is not a power of two) all yield approximately the same entropy for the histogram. While this is a remarkable property *per se*, it is unfortunately not helpful for us and lead us to abandon this heuristic altogether. Speaking of the data itself, it is apparent that the `PHANTOM` layout stands out. The `NATIVE` layout and its worsened companion are almost indistinguishable while the `FMMM` layout is closer to them than to the `PHANTOM` layout. The most surprising observation, however, is the fact that the entropy for the `PHANOM` layout is actually the *lowest* which is unexpected even if taking into consideration that the construction of this layout does not assign random coordinates to vertices (*cf.* § 5.2.5).

as

$$\bar{S}(f) = -\int_{-\infty}^{+\infty} \mathrm{d}x \; x \log_2(x) \tag{7.12}$$

where we use the convention that the integrand shall be zero for those $x \in \mathbb{R}$ where $f(x) = 0$.

While this definition of differential entropy was actually proposed by Shannon himself in his original paper [49], it has been argued [23] since that equation 7.12 is *not* a correct information measure and does not provide the continuous analog of equation 7.11 for the limit $n \to \infty$. An obvious observation to make is, for example, that $\bar{S}(f)$ according to equation 7.12 may even become negative, such as for the uniform distribution over an interval narrower than 1. On the other hand, differential entropy has been studied for various distribution functions [33] even if, unlike discrete entropy, it might not have a straight-forward interpretation. Furthermore, it can be shown [9, thm. 9.3.1, eq. 9.30] that if a Riemann integrable density function f is sampled into a histogram H_Δ with bin width $\Delta \in \mathbb{R}_{>0}$ then

$$\lim_{\Delta \to 0} S(H_\Delta) + \log_2(\Delta) = \bar{S}(f) \ . \tag{7.13}$$

Given that no better alternative was available, we decided to simply use differential entropy of sliding averages without further ado.

7.5 Special Considerations for Local RDF

The `RDF_LOCAL` property (§ 4.4) is the only property that is already parameterized in its own right. We might use this in order to perform some analysis. Figure 7.7 shows the differential entropy of the sliding average of `RDF_LOCAL`(d) as a function of $\log_2(d)$. We decided to not use this for a regression, however, because there is no theoretical consideration that would justify this. Instead, we put the entropy values for each value of d directly into the feature vector (which therefore becomes considerably larger). We use the differential entropy of the sliding average rather than using the regression technique to get a linear entropy function in order to avoid having two variables that need to be varied (and therefore make the computation even costlier).

7.6 Other Data

There are a few more bits of information that didn't fit in any of the previous discussion. If we have a PCA available, we also want to capture the orientation of the layout, which might be important (see fig. 1.2 and Giannouli [14]). For this, we also include the coordinates of the principal axes into the feature vector (which are two values per component).

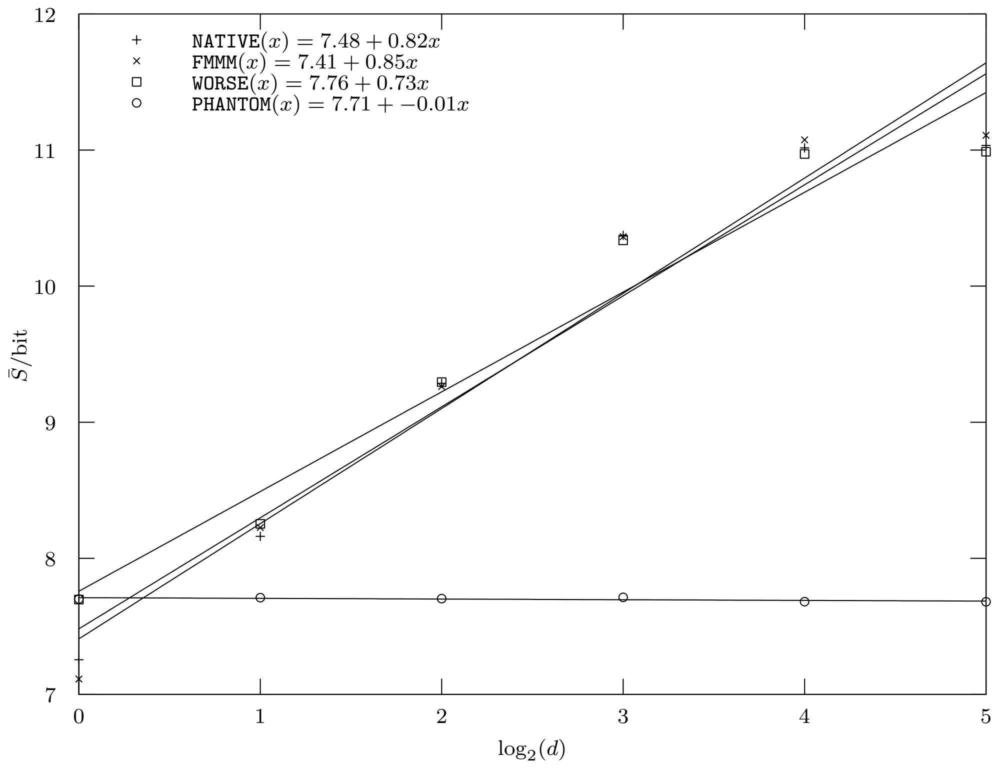


FIGURE 7.7: Differential entropies of the density functions obtained via computing sliding averages (with a Gaussian filter width chosen according to Scott's normal reference rule) for the `RDF_LOCAL`(d) properties for for the four layouts shown in figure 7.1 plotted as a function of $\log_2(d)$. The first thing to notice is that – unlike pointed out for the entropy of histograms – using Scott's normal reference rule for the selection of the filter width does not seem to predetermine the entropy value. (Note that the ordinate does not start at zero, though.) The distribution for the `PHANTOM` layout clearly stands out. It meets the expectation that its value is mostly independent of the parameter d as the layout does not make use of the graph's structure at all. The data points for the `NATIVE` and `FMMM` layout are almost identical with a smaller difference to the worsened layout, which is all as expected. The linear regression curves for all layouts but `PHANTOM` do not approximate the data very well.

Finally, we also want to capture the size of the graph itself, which might have an influence on the importance and reliability of some properties. For this purpose, we also put the *logarithm* of the number of vertices and edges into the feature vector. This also captures the sparsity implicitly. It might be worthwhile to include other information (such as the diameter) as well but we did not look into this.

7.7 Summary

We have introduced mean and RMS and discussed the difficulties of finding good bin widths for histograms and presented our solution of using a linear regression of the entropy as a function of the logarithm of the bin count. We also introduced sliding averages and differential entropy. `RDF_LOCAL`(d) is a special case as it is already parameterized. All in all, our feature vector for a layout has the following entries, which happen to sum up to 58 given that we compute `RDF_LOCAL`(2^i) for $i \in \{0, \dots, 9\}$ which was chosen like this because none of our graphs had a diameter in excess of $2^9 = 512$.

- Mean and RMS for each property.
- Slope and intercept of the linear regression function found for histogram entropy as a function of the bin count for all properties except `RDF_LOCAL`.
- Differential entropy for `RDF_LOCAL`.
- Principal components (four scalar values).

For the corresponding graph, the feature vector only contains two entries which are

- the logarithm of the number of vertices and
- the logarithm of the number of edges.

If any entry in the feature vector is not available – maybe because the computation timed out or because something is mathematically undefined for a pathological instance, such as a graph with no edges – we record a special “null” value that will later be dealt with.

On a technical level, the feature vector is implemented by dynamically creating a *view* in the SQL database which pulls in all needed data and is keyed by the layout ID.

In the next chapter, we will discuss how the extracted features are used to train and test an automatic discriminator model.

8 Discriminator Model

Using the condensed information collected in the feature vector we would like to train a neural network that predicts an ordering relation between the aesthetic value of two layouts. In this chapter we will explain the structure of the neural network that we have chosen and explain the training and testing of it. Since it discriminates between two layouts, we refer to it as a *discriminator* (in order to distinguish it from, say, a classifier).

At the user level, our system allows to specify two layouts by their IDs i and j and outputs a prediction $-1 \leq p \leq +1$ for the aesthetic preference between the layouts Γ_i and Γ_j where a value of $p < 0$ or $p > 0$ is a result in favor of Γ_i or Γ_j respectively and $|p|$ is a measure of the discriminator's certainty about its prediction. It is currently not possible to ask the discriminator about layouts that are not already saved in the internal database. This is an obvious annoyance and we are thinking about providing a more convenient interface in the future. Specifying layout IDs that belong to different graphs is an error.

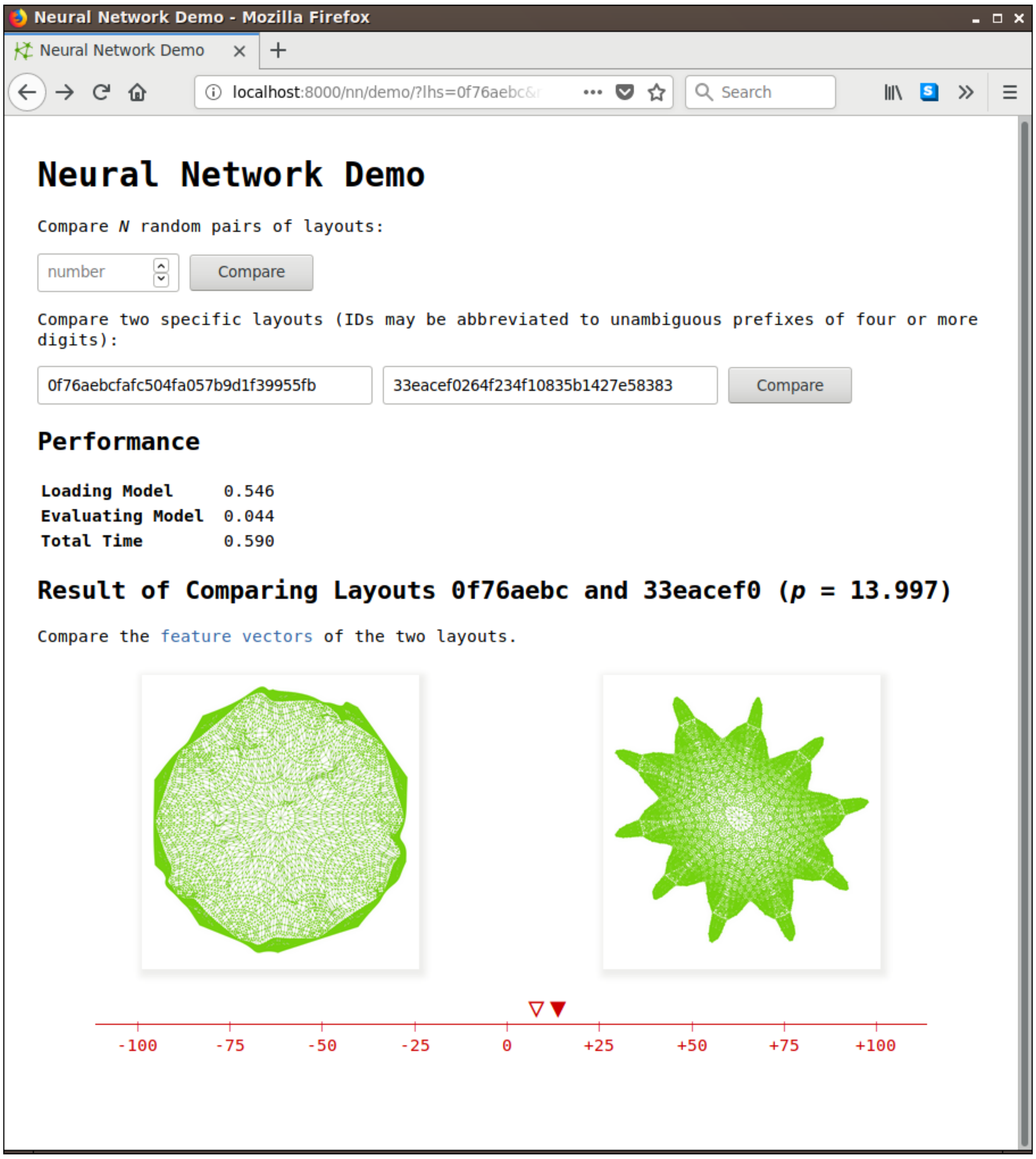
We provide a command-line tool and a web front-end to interact with the discriminator. The command-line tool would be used like this

```
$ compare 0f76aebc 33eacef0
0f76aebcfafc504fa057b9d1f39955fb 33eacef0264f234f10835b1427e58383   +0.13997
33eacef0264f234f10835b1427e58383 0f76aebcfafc504fa057b9d1f39955fb   -0.07941
```

where the user asked for a prediction concerning the layouts with the unambiguous ID prefixes `0f76aebc` and `33eacef0` and the tool outputted the full IDs next to the numbers $+0.13997$ and -0.07941 indicating a preference of $+14\,\%$ in favor of the second layout and $-8\,\%$ if the layouts are compared in reversed order. We will discuss shortly why the numbers are not exactly symmetric. An impression of the web UI is shown in figure 8.1.

8.1 Siamese Neural Network Structure

Because the discrimination is an inherently binary task, we decided to use a *Siamese neural network* which is a structure originally proposed by Bromley *et al.* [6]. A network using this structure consists of two identical sub-networks (referred to as *shared model* from now on) that process the feature vector of the left-hand and right-hand input respectively. The output vectors of these sub-networks are then subtracted from each other and the difference is passed to a third, independent, sub-network that reduces this information into a single scalar quantity, which gives the predicted preference.


Neural Network Demo - Mozilla Firefox
Neural Network Demo
localhost:8000/nn/demo/?lhs=0f76aebc&
Search

Neural Network Demo

Compare *N* random pairs of layouts:

Compare two specific layouts (IDs may be abbreviated to unambiguous prefixes of four or more digits):

0f76aebcfafc504fa057b9d1f39955fb

33eacef0264f234f10835b1427e58383

Performance

```
Loading Model     0.546
Evaluating Model  0.044
Total Time        0.590
```

Result of Comparing Layouts 0f76aebc and 33eacef0 (p = 13.997)

Compare the feature vectors of the two layouts.

FIGURE 8.1: Screenshots of the web UI for querying the discriminator about its aesthetic judgment. The filled and outlined triangle indicates the result in forward and reverse order respectively. The user can also click on either layout in order to view detailed information about it. Clicking on the "feature vectors" link will show a table which details the deviation of each feature of the two layouts from the average over all layouts in the database as well as the difference between the two feature vectors.

The last sub-network also receives the feature vector of the graph. The structure of the network is shown in figure 8.2. If the shared model had an output layer with only a single value, this architecture would correspond to the computation of a single quantity that describes aesthetic value in an absolute sense. However, as we have already discussed, we don't believe that this is a viable approach. Therefore, we keep several dimensions in the output of the shared model which can reflect different aspects of each of the layouts and allow the third sub-network to make a final decision. The turn side of this approach is that we cannot guarantee through the network's structure that the discriminator will be symmetric in the sense that swapping the left and right hand input will invert the sign of the output only. As we have seen in the example at the beginning of this chapter, the output differs by a few percent if the inputs are swapped. We recommend that if the model were to be used in production, the discriminator should be run for both combinations of the inputs and the average of both outputs should be returned in order to enforce symmetry again. However, since we are more interested in studying the discriminator's behavior rather than actually using it, we omitted this trivial step and instead show the outputs for both combinations individually.

The shared model consists of two dense layers. The first layer (obviously) has as many inputs as the size of our feature vector for layouts (which is 58). Its output dimension, and therefore the input dimension for the second layer, was set to 10 mostly out of a gut feeling and after a little bit of experimentation to verify that increasing the number of dimensions does not improve the success rate. The output of the hidden layer also has 10 dimensions. This leads to $590 + 110 = 700$ trainable parameters for the shared model.

The additional feature vector with information specific to the *graph* rather than either layout is first passed through an auxiliary dense layer. This is probably not very useful (although it cannot do much harm either) at the moment because the feature vector only has size 2 and the auxiliary layer also has output dimension 2 as it would make no sense to use even less. However, if more entries shall be added to the graph's feature vector in the future, this auxiliary layer may help reduce the dimensionality of this additional information before it is given to the final layer.

Said final layer is also a dense layer and receives the difference of the outputs of the shared model concatenated with the output of the auxiliary layer and therefore has 12 inputs. Its output dimension obviously had better be 1.

The auxiliary layer adds 6 and the final layer 13 trainable parameters which gives a total of $700 + 13 + 6 = 719$ trainable parameters for the entire network.

The network was built using the Keras [26] framework with the TensorFlow [53] library as back-end. Unless noted otherwise in this writing, the default values provided by the library were used. Not having designed a neural network before, we found the recommendations in LeCun *et al.* [34] invaluable.

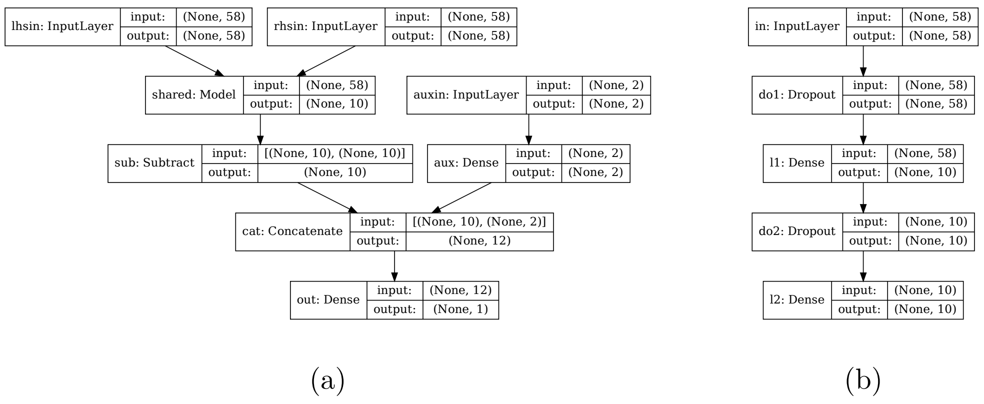

lhsin: InputLayer | input: (None, 58) | output: (None, 58)
rhsin: InputLayer | input: (None, 58) | output: (None, 58)
shared: Model | input: (None, 58) | output: (None, 10)
auxin: InputLayer | input: (None, 2) | output: (None, 2)
sub: Subtract | input: [(None, 10), (None, 10)] | output: (None, 10)
aux: Dense | input: (None, 2) | output: (None, 2)
cat: Concatenate | input: [(None, 10), (None, 2)] | output: (None, 12)
out: Dense | input: (None, 12) | output: (None, 1)
(a)
in: InputLayer | input: (None, 58) | output: (None, 58)
do1: Dropout | input: (None, 58) | output: (None, 58)
l1: Dense | input: (None, 58) | output: (None, 10)
do2: Dropout | input: (None, 10) | output: (None, 10)
l2: Dense | input: (None, 10) | output: (None, 10)
(b)

FIGURE 8.2: Structure diagram of the discriminator's neural network as output by the debugging feature of the Keras library. The diagram in (a) shows the overview of the entire model while the picture in (b) shows the internals of the shared sub-model – shown as the `Model` layer in (a). Only `Dense` layers can be trained. The `Subtract` and `Concatenate` layers do what you think they do; they have no parameters. An `InputLayer` in Keras parlance is merely a way to refer to parameters; it does not compute anything and has no parameters. The significance of the `Dropout` layers is discussed in section 8.2.

8.2 Regulation

In order to prevent overfitting of the model and harden it against missing input values, *dropout* is used. Dropout as a regularization technique was proposed by Srivastava *et al.* [50]. Please refer to their text for motivation and details.

For this purpose, a dropout layer is added before each of the dense layers in the shared model that discards 50 % and 25 % of the signals respectively during the training of the network.

For the dropout rate before the first layer, a large value of 50 % was chosen in order to make the network more robust against missing data which occurs frequently in our inputs. A lower dropout rate of 25 % before the second layer was chosen because the effect of missing inputs will already have been distributed by the first layer.

8.3 Metaparameters

8.3.1 Activation

A linear activation function for the first and auxiliary layer was chosen because we found this to be most robust with respect to inputs that are out of range after experimenting a bit with other functions and repetitively facing numeric overflow issues. The hidden layer uses the common and efficient rectified linear unit (ReLU) activation while a hyperbolic tangent was chosen as the activation function for the final layer because its output range and characteristics match exactly our requirements.

8.3.2 Initialization, Loss and Optimization

Neuron weights are initialized by setting the bias to zero and kernel to a random value drawn according to a truncated normal distribution with $\mu = 0$, $\sigma = 1/20$ and truncation at $\pm 2\sigma$.

A *mean squared error (MSE)* loss function and stochastic gradient descent (SGD) optimizer are used.

8.4 Normalization

Once the model is set up, the database is queried for all available data. Please refer to chapter 3 for the reasoning behind and explanation of the data available for training and testing. In order to collect it into a single huge vector, the driver script finds all of the following triples for each graph in the database.

- $\bigl(\Gamma, \Gamma', -1\bigr)$ where Γ is a proper and Γ' is a garbage layout.
- $\bigl(\mathcal{W}(\Gamma, r_i), \mathcal{W}(\Gamma, r_j), t_{ij}\bigr)$ for $r_i \neq r_j$ where Γ is a proper layout and $\mathcal{W}$ is a layout worsening algorithm. The expectation is set to $t_{ij} = (r_i - r_j)/r_{\max}$ where $0 < r_{\max} \leq 1$ is the maximum worsening rate ever used with algorithm $\mathcal{W}$ for any layout. Note that $\mathcal{W}(\Gamma, 0) = \Gamma$ is always available.
- $\bigl(\mathcal{I}(\Gamma, \Gamma', r_i), \mathcal{I}(\Gamma, \Gamma', r_j), t_{ij}\bigr)$ for $r_i \neq r_j$ where Γ is a proper layout, Γ' is a garbage layout and $\mathcal{I}$ is a layout interpolation algorithm. The expectation is set to $t_{ij} = r_i - r_j$.

The same pair of layouts is only used once. That is, if $(\Gamma_{\mathrm{A}}, \Gamma_{\mathrm{B}}, t)$ is already used, we don't use $(\Gamma_{\mathrm{B}}, \Gamma_{\mathrm{A}}, -t)$ too. What becomes the “left” and “right” layout is chosen randomly in order to ensure that t is unbiased. The normalization factor $r_{\max}$ for worsened layouts was introduced because not all worsenings are equally dramatic so this leads to more evenly distributed data.

After the list of layout combinations has been determined, the feature vectors for each of them has to be found. The values can be looked up directly in the database but before they are given to the network, they get *normalized* (*cf.* LeCun *et al.* [34]).

Suppose that a feature vector has $n \in \mathbb{N}$ entries and that there are $m \in \mathbb{N}$ layouts in the corpus of training and testing data. Let v_{ij} be the value of the j-th element in the feature vector of the i-th layout. The normalized feature vector for this layout will contains the value

$$\hat{v}_{ij} = \begin{cases} (v_{ij} - \operatorname{mean}(V_j)) / \operatorname{stdev}(V_j) & v_{ij} \neq \bot \\ \operatorname{mean}(V_j) & v_{ij} = \bot \end{cases} \tag{8.1}$$

where $\bot$ denotes a missing value and $V_j = [v_{kj} : 1 \leq k \leq n : v_{kj} \neq \bot]$ is the multiset containing all well-defined j-th elements of all feature vectors in the corpus.

This normalization step ensures that the inputs to the neural network have zero mean and unit standard deviation and that missing values represent no bias in either direction.

8.5 Training and Testing

We used test corpora with more than 10 k and less than 100 k labeled layout pairs, limited primarily by our willingness and ability to spend more computational resources. There are no arbitrary restrictions or assumptions in our setup that would limit the amount of data. No thorough investigation was performed, though. Repetition of our experiments is highly encouraged. The fellow researcher is invited to download and run our setup. Instructions how to do this are given in the appendix of this work.

Given the list of inputs and expected outputs prepared as described in the previous section, this list is shuffled randomly such that the model won't see the data in any particular order. 20 % of this data is kept aside for the purpose of testing the model once training is complete. The remaining data is used for training the model over 100 epochs using a validation split of 25 %.

Once training has completed, the 20 % of the original data previously set aside – which the network has never seen up to this point – are used for testing. That is, the model is queried for a prediction p for each pair $(\Gamma_\mathrm{A}, \Gamma_\mathrm{B})$ of the test triples $(\Gamma_\mathrm{A}, \Gamma_\mathrm{B}, t)$ and then p is compared to t. A test is considered a success if and only if $\operatorname{sign}(p) = \operatorname{sign}(t)$. A reproducible success rate above 95 % could be achieved. We will have a more detailed look at the actual test results in section 9.1. A similar interface to that shown in figure 8.1 is provided for manual inspection of the test cases.

9 Evaluation

Due to time constraints, we were only able to perform a very limited amount of evaluation of our discriminator model. The results we did obtain are presented in the current chapter. More work is clearly needed and we hope that the setup we've provided is a good starting point for this.

9.1 Accuracy

We used *cross validation* via *random subsampling* [30] in order to verify the reliability of our model. As already mentioned in section 8.5 we set a randomly chosen partition of 20 % of the data corpus is aside for testing and not use it for training. For the purpose of cross validation via random subsampling, a constant $k \in \mathbb{N}$ is chosen and the training and testing sequence is repeated k times – each time with a newly partitioned data corpus. By collecting the results of each run, not only can we estimate the reliability of our discriminator but also the reliability of this estimation. That is, we can provide a confusion matrix with errors. This is shown in table 9.1.

9.2 Contribution of Individual Properties

In order to assess the contribution that each property brings information wise, we've set up an experiment where we re-run the cross validation but either exclude a specific property from the feature vector or exclude all other properties and use only this single property. The results are shown in table 9.2.

`RDF_LOCAL` stands out as the clear winner of this comparison followed by `RDF_GLOBAL`. It is a little unfair, though, as `RDF_LOCAL` is not a single property but a group of properties so it is expected to contribute more. The difference is still pretty dramatic, though. Using `RDF_LOCAL` alone still yields a good result but significantly less than if all properties are used together. It should be noted, however, that this view is misleading as there are graphs for which we cannot compute `RDF_LOCAL` as they are too big. For those, the discriminator can only guess. In a more carefully set up experiment, such graphs would have to be excluded. It is also not clear from this experiment alone whether some properties are just plain useless or only have too much overlap with each other to make the exclusion of either of them insignificant. A more rigorous statistical analysis would be required to answer this question reliably.

	Cond. Neg.		*Cond. Pos.*		Σ	
Pred. Neg.	48.65 %	± 0.62 %	1.68 %	± 0.58 %	50.33 %	± 1.01 %
Pred. Pos.	1.24 %	± 0.41 %	48.43 %	± 0.67 %	49.67 %	± 1.01 %
Σ	49.89 %	± 0.49 %	50.11 %	± 0.49 %	100.00 %	± 0.00 %
Success Rate:					97.08 %	± 0.26 %
Failure Rate:					2.92 %	± 0.26 %
Average Number of Tests:						≈ 6 356
Number of Repetitions:						10

Table 9.1: Confusion matrix with errors obtained through cross validation via random subsampling and some additional information.

Property	*Sole Exclusion*		*Sole Inclusion*	
`RDF_LOCAL`	64.74 %	±6.13 %	86.32 %	± 0.82 %
`ANGULAR`	96.46 %	±0.61 %	76.98 %	± 6.23 %
`PRINCOMP2ND`	96.53 %	±0.48 %	51.45 %	± 6.90 %
`PRINCOMP1ST`	96.57 %	±0.80 %	55.89 %	± 7.74 %
`EDGE_LENGTH`	96.80 %	±0.46 %	43.90 %	±14.98 %
`RDF_GLOBAL`	97.11 %	±0.17 %	81.47 %	± 0.49 %
`TENSION`	97.26 %	±0.13 %	74.04 %	± 1.01 %
Baseline Using All Properties	97.08 %	±0.26 %		

Table 9.2: Cross validation results (success rates) with one property deliberately excluded (middle column) and only a single property included (right column) respectively. Rows are sorted by the middle column. It is immediately visible that `RDF_LOCAL` has by far the biggest contribution followed by `RDF_GLOBAL` (which doesn't add much when used *together* with `RDF_LOCAL` for apparent reasons but is almost as effective when used alone). `TENSION` and `ANGULAR` seem to contribute something while the `PRINCOMP1ST`, `PRINCOMP2ND` and `EDGE_LENGTH` appear close to useless. It could be that there is much overlap between the properties except RDF such that omitting any of them alone has little effect. The low contribution of `PRINCOMP2ND` is not very surprising. Please see the text for additional remarks. It should be noted that we used only 5 cross validation runs to obtain the entries in this table.

9.3 Performance

Our implementation was not designed with maximum performance in mind but rather guided by the desire to provide a flexible framework to which new tools can be added easily. Therefore, we do not present an analysis of the system's performance at this point. Practically speaking, most of the time is lost in our implementation by reading GraphML files from disk. There are, however, hard limits which we actually ran into that are not caused by our design decisions. For example, the OGDF routines sometimes simply crash with an out-of-memory error and other can take very long to run. Training the neural network only takes a few minutes on a normal PC and evaluating the model once it is trained and all inputs are available is so quick that it is hard to notice.

9.4 Summary

We were only able to do very limited evaluation of our discriminator model so the findings in this chapter have to be taken with caution. Nevertheless, we could show that our model reaches a reproducible success rate above 95 % for our data corpus.

The biggest contribution comes from the `RDF_LOCAL`(d) family of properties, followed by `RDF_GLOBAL`. The contribution of and overlap between the other properties remains to be studied in a more elaborate investigation.

Speaking of performance, most time is spent in our current system while waiting for slow I/O operations using the file system and parsing XML documents. Nevertheless, resource limits for the algorithmically more expensive operations (especially the computation of all-pair shortest path matrices) might be a concern, too.

10 Conclusions and Future Work

Unattended graph drawing can be a valuable part in the software engineering toolbox for a variety of business needs. Given the great variability of graphs people are dealing with and the manifold of layout algorithms available, we believe that a reliable method for automatic quantification that is not based on specific assumptions would be an important tool. Among other use cases, it could enable practitioners to quickly identify the "best" layouts in a large collection, guide them through the choice of a suitable layout algorithm for their domain or even promote the development of new algorithms.

We have defined the scope for automatic quantification of the aesthetic value of graph layouts and presented an approach towards automatic quantification that is based on the computation and statistical analysis of various elementary properties of a graph layout. These should, ideally, be searched from first principles and be influenced as little as possible by existing assumptions. We believe and – to some degree – have provided evidence that these features can be representative syndromes of aesthetic value. We think that inspiration from other disciplines and fields of science, especially astronomy, crystallography and thermodynamics can provide valuable directions.

As we begun this work with an open-ended mindset, we started by developing a flexible framework to help us conduct our studies. We believe that the correct approach to investigate this topic further is not to restrict oneself to any one specific type of graph, property or data analysis but rather ensure that the system allows flexible addition of features and analyses.

Our main contribution is therefore a toolbox that might see continued use in the future in order to delve deeper into the subject. Our setup consists of many individual command-line tools that can be freely combined to build powerful applications. They are accompanied by a, by now, fairly elaborate driver "script" which acts as a coordinator and can bring data and analyses from various sources together and can be configured and adapted in various ways. The driver also provides a feature-rich web font-end for a convenient inspection and presentation of the data. We reckon that we have provided not so much of an experiment, let alone a theory, but rather a tool for experimentation.

The feasibility of our approach towards a quantification of aesthetic value of graph layouts was tested and demonstrated by training a neural network to become a fairly reliable ($> 95\,\%$) discriminator for pairs of layouts regarding their aesthetic value. Within the realm of the limited evaluation we could do, it became apparent that a localized definition of the radial distribution function which is well-known to physicists was the most reliable syndrome. Unfortunately, it is – among those we've given consideration –

the most expensive property to compute as it requires knowledge of an all-pairs shortest path matrix which can be a cubic operation. Evaluation of the sliding averages needed

We do not suggest that using a neural network as we did is the most effective way to approach the problem of automatic quantification of aesthetic value in the context of graph drawing. We do believe, however, that the fact that a network was apparently able to learn a great deal of knowledge from the data we provided it, is indicative of the fact that the features we're looking at are indeed syndromes of aesthetic value to a certain degree. Maybe future research can find a more white-box answer to the question what exactly those features are and how they correlate.

We have discussed the data generation and augmentation process in great length. The whole process is fully automated and does not require any human intervention or labeling of data. The setup is also self-contained and reproducible. We encourage others to re-run our experiments – ideally on larger machines than those that were available to us during the preparation of this work. Please refer to the appendix of this document for pointers on how to get started.

There are also many aspects of the topic that we were not able to address within the time and resource limits imposed on the preparation of this work. In the final section, we shall outline some of our ideas that we were not able to investigate further, yet.

10.1 Additional Properties

An obvious deficit of our current set of properties is that they focus a lot on the drawing of vertices and not so much on edges. Edge crossings and crossing angles would most certainly be a valuable addition.

Another idea that we find very interesting is to apply the technique of *shapelet analysis* [45] to the field of graph drawing. Shapelets are used in astronomy to study the structure of galaxies. They basically present a way to compute a series of weighted sums that preserve some spatial information by using a sequence of n-dimensional filter functions, such as provided by one of the popular polynomial bases. This filter is then centered at each object's location and the function value at the position of the other objects are summed up. The technique has already enjoyed successful application outside the field of astronomy, for example in the processing of microscopic imagery [51]. It seems promising to us that it might also be valuable in the field of graph drawing.

A fairly general concept that we find very exciting is the consideration of features of a layout with regard to some graph-theoretical property of the drawn graph. The example of local RDF that was presented and shown to be of value in this work has confirmed our intuition that this could be a powerful concept. We would like to investigate further examples – ideally ones that require less than cubic computational effort.

10.2 More Elaborate Data Analysis

Ignoring for a while the fact that we also input the mean and RMS into our model, the only really exciting measure we currently use is entropy. While it was challenging enough to get a reasonably reliable approximation of the entropy, there are myriads of other data analysis techniques that could also be applied to the properties we computed and might – especially in combination – provide deeper insights into the structure of the data.

A very important piece of work would also be to find and peruse ways to assess the overlap between the information provided by a collection of symptoms. We have performed some superficial analysis that lead us to the conclusion that local and – to a lesser extent – global RDF provide a distinguishing view of the layout's quality but all other properties remain yet to be picked apart.

10.3 Thorough Comparison With Existing Measures

The most painful omission from the present work is the lack of a comparison with existing quality measures. Such analysis should be performed with respect to accuracy as well as generality and efficiency. In particular, it would be interesting to see whether there is a correlation between, say, our discriminator failing to predict a layout pair correctly and the value of the stress function for both layouts. Such analysis would be relatively simple to set up even if it might take a lot of computational resources to be carried out over a large data set containing large graphs.

10.4 Conduction of a User Study

At the end of the day, any work on aesthetics in human perception has to be validated against the judgment of actual humans. As a very helpful side-effect, such an empirical study can produce more labeled data that may in turn be used to further reify the data acquisition and processing strategy. For example, we simply *assume* that the native layouts produced by our generators are "good" while there is no empirical evidence for this.

10.5 Generalization to More Complex Graph Drawings

Until now, we have considered straight-line drawings exclusively. There are, however, many other interesting ways to draw a graph and our current analysis will not be able to capture these specific features. For example, a particularly interesting kind of drawings are so-called *Lombardi* graph drawings [43] (named so after the graphical artist Mark Lombardi who created drawing that inspired the layouts. A Lombardi graph drawing hat the distinguishing property that the angle between the incident edges of a vertex with degree $n \in \mathbb{N}$ is always $2\pi/n$ which is obviously not always possible when using only straight lines so edges have to be drawn as bent curves. It would be very interesting to see how the `ANGULAR` property we've used would react to such a drawing, if only it could be approached by it.

10.6 Application as a Meta-Heuristic for a Genetic Layout Algorithm

The last thought we'd like to share is that our work might have an unanticipated application as a layout algorithm. Given that we already have

- various existing layout algorithms that produce layouts that are often good and seldom ideal and provide a good *initial population*,
- a set of unary layout transformations which can be parameterized in their intensity and act as a configurable *mutation function*,
- a set of binary layout transformations which can be parameterized (probabilistically, if one so wishes) in favor of wither parent and act as a *crossover function* and finally
- our discriminator as a means to predict which of two layouts has a higher aesthetic value and can therefore be used to construct a *fitness function*,

we have, in principle, every tool in the box that is needed in order to build a so-called *genetic algorithm* [39] which is a popular meta heuristic for hard optimization problems closely inspired by biological evolution. We would be delighted to see how it works out.

Bibliography

[1] H. Abdi and L. J. Williams. “Principal component analysis”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.4 (2010), pp. 433–459. ISSN: 1939-0068. DOI: 10.1002/wics.101.

[2] B. Bach *et al.* “Interactive Random Graph Generation with Evolutionary Algorithms”. In: *Graph Drawing*. Ed. by W. Didimo and M. Patrignani. Vol. 7704. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2012-09, pp. 541–552. DOI: 10.1007/978-3-642-36763-2_48.

[3] R. F. Boisvert *et al.* “Matrix Market: a web resource for test matrix collections”. In: *Quality of Numerical Software: Assessment and enhancement*. Ed. by R. F. Boisvert. Springer, 1997, pp. 125–137. ISBN: 978-1-5041-2940-4. DOI: 10.1007/978-1-5041-2940-4_9.

[4] U. Brandes and C. Pich. “Eigensolver Methods for Progressive Multidimensional Scaling of Large Data”. In: *Graph Drawing*. Ed. by M. Kaufmann and D. Wagner. Springer Berlin Heidelberg, 2007, pp. 42–53. ISBN: 978-3-540-70904-6. DOI: 10.1007/978-3-540-70904-6_6.

[5] U. Brandes *et al.* “GraphML Progress Report Structural Layer Proposal”. In: *Graph Drawing*. Ed. by P. Mutzel, M. Jünger, and S. Leipert. Springer, 2002, pp. 501–512. ISBN: 978-3-540-45848-7. DOI: 10.1007/3-540-45848-4_59.

[6] J. Bromley *et al.* “Signature verification using a “siamese” time delay neural network”. In: *Advances in Neural Information Processing Systems*. 1994, pp. 737–744.

[7] M. Chimani *et al.* “The Open Graph Drawing Framework (OGDF)”. In: *Handbook of Graph Drawing and Visualization*. Ed. by R. Tamassia. CRC Press, 2013. Chap. 17, pp. 543–570. ISBN: 9781420010268.

[8] A. Cimikowski and P. Shope. “A neural-network algorithm for a graph layout problem”. In: *IEEE Transactions on Neural Networks* 7.2 (1996-03), pp. 341–345. ISSN: 1045-9227. DOI: 10.1109/72.485670.

[9] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley series in telecommunications. Wiley, 1991. ISBN: 0-471-06259-6. URL: https://pdfs.semanticscholar.org/881c/f0ccc5a9dbb772d5a07671773f3c14b551c2.pdf (visited on 2018-03-13).

[10] P. A. M. Dirac. “A new notation for quantum mechanics”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (1939), pp. 416–418. DOI: 10.1017/S0305004100021162.

[11] G. H. Findenegg and T. Hellweg. *Statistische Thermodynamik.* 2nd ed. Springer Spektrum, 2015. ISBN: 978-3-642-37871-3. DOI: `10.1007/978-3-642-37872-0`.

[12] R. W. Floyd. “Algorithm 97: Shortest Path”. In: *Communications of the ACM* 5.6 (1962-06). ISSN: 0001-0782. DOI: `10.1145/367766.368168`.

[13] E. R. Gansner, Y. Koren, and S. North. “Graph Drawing by Stress Majorization”. In: *Graph Drawing.* Ed. by J. Pach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 239–250. ISBN: 978-3-540-31843-9. DOI: `10.1007/978-3-540-31843-9_25`.

[14] V. Giannouli. “Visual symmetry perception”. In: *Encephalos* 50 (2013), pp. 31–42.

[15] G. H. Golub and C. F. van Loan. *Matrix Computations.* 3rd ed. Johns Hopkins University Press, 1996. ISBN: 978-0801854149.

[16] *Graph Drawing.* URL: `http://www.graphdrawing.org/` (visited on 2018-02-15).

[17] L. F. Greengard. “The Rapid Evaluation of Potential Fields in Particle Systems”. AAI8727216. PhD thesis. New Haven, CT, USA, 1987.

[18] S. Hachul and M. Jünger. “Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm”. In: *Graph Drawing.* Ed. by J. Pach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 285–295. ISBN: 978-3-540-31843-9. DOI: `10.1007/978-3-540-31843-9_29`.

[19] “Histogram”. In: *Wikipedia – The Free Encyclopedia.* 2018-02-18. URL: `https://en.wikipedia.org/wiki/Histogram` (visited on 2018-03-12).

[20] J. J. Hopfield and D. W. Tank. ““Neural” computation of decisions in optimization problems”. In: *Biological Cybernetics* 52.3 (1985-07), pp. 141–152. ISSN: 1432-0770. DOI: `10.1007/BF00339943`.

[21] W. Huang, M. L. Huang, and C.-C. Lin. “Evaluating overall quality of graph visualizations based on aesthetics aggregation”. In: *Information Sciences* 330 (2016), pp. 444–454. ISSN: 0020-0255. DOI: `https://doi.org/10.1016/j.ins.2015.05.028`.

[22] W. Huang *et al.* “Improving multiple aesthetics produces better graph drawings”. In: *Journal of Visual Languages & Computing* 24.4 (2013), pp. 262–272. ISSN: 1045-926X. DOI: `https://doi.org/10.1016/j.jvlc.2011.12.002`.

[23] E. T. Jaynes. “Information Theory and Statistical Mechanics”. In: *Statistical Physics.* Ed. by K. Ford. Brandeis University Summer Institute Lectures in Theoretical Physics 3. New York: Benjamin, 1963, pp. 181–218. URL: `http://bayes.wustl.edu/etj/articles/brandeis.pdf` (visited on 2018-03-13).

[24] I. T. Jolliffe. *Principal Component Analysis.* 2nd ed. Springer, 2002. ISBN: 978-0-387-95442-4. DOI: `10.1007/b98835`.

[25] T. Kamada and S. Kawai. “An algorithm for drawing general undirected graphs”. In: *Information Processing Letters* 31.1 (1989), pp. 7–15. ISSN: 0020-0190. DOI: `10.1016/0020-0190(89)90102-6`.

[26] *Keras.* URL: `https://keras.io/` (visited on 2018-02-22).

[27] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. ISSN: 0036-8075. DOI: `10.1126/science.220.4598.671`. eprint: `http://science.sciencemag.org/content/220/4598/671.full.pdf`.

[28] R. Klapaukh. “An Empirical Evaluation of Force-Directed Graph Layout”. PhD thesis. Victoria University of Wellington, 2014.

[29] S. G. Kobourov. “Force-Directed Drawing Algorithms”. In: *Handbook of Graph Drawing and Visualization*. Ed. by R. Tamassia. CRC Press, 2013. Chap. 12, pp. 383–408. ISBN: 9781420010268.

[30] R. Kohavi *et al.* “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In: *Ijcai*. Vol. 14. 2. 1995, pp. 1137–1145.

[31] Y. Koren and A. Çivril. “The binary stress model for graph drawing”. In: *International Symposium on Graph Drawing*. Springer, 2008, pp. 193–205.

[32] O. H. Kwon, T. Crnovrsanin, and K. L. Ma. “What Would a Graph Look Like in this Layout? A Machine Learning Approach to Large Graph Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018-01), pp. 478–488. ISSN: 1077-2626. DOI: `10.1109/TVCG.2017.2743858`.

[33] A. C. G. V. Lazo and P. N. Rathie. “On the entropy of continuous probability distributions”. In: *IEEE Transactions on Information Theory* 24.1 (1978), pp. 120–122. DOI: `10.1109/TIT.1978.1055832`.

[34] Y. LeCun *et al.* “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade*. Ed. by G. B. Orr and K.-R. Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-49430-0. DOI: `10.1007/3-540-49430-8_2`.

[35] M. Lombardi. *Global Networks*. Ed. by R. C. Hobbs and J. Richards. Independent Curators International, 2003. ISBN: 0-916365-67-0.

[36] T. Masui. “Evolutionary Learning of Graph Layout Constraints from Examples”. In: *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*. UIST ’94. Marina del Rey, California, USA: ACM, 1994, pp. 103–108. ISBN: 0-89791-657-3. DOI: `10.1145/192426.192468`.

[37] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. ISBN: 978-3-540-77977-3. DOI: `10.1007/978-3-540-77978-0`.

[38] B. Meyer. “Self-Organizing Graphs – A Neural Network Perspective of Graph Layout”. In: *Graph Drawing*. Ed. by S. H. Whitesides. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 246–262. ISBN: 978-3-540-37623-1.

[39] M. Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press, 1996. ISBN: 9780585030944.

[40] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990. ISBN: 978-0-387-97297-8. URL: `http://algorithmicbotany.org/papers/#abop` (visited on 2018-02-28).

[41] H. Purchase. “Metrics for graph drawing aesthetics”. In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516. DOI: 10.1016/S1045-926X(02)90232-6.

[42] H. Purchase. “Which aesthetic has the greatest effect on human understanding?” In: *Graph Drawing.* Ed. by G. DiBattista. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 248–261. ISBN: 978-3-540-69674-2.

[43] H. C. Purchase *et al.* “On the Usability of Lombardi Graph Drawings”. In: *Graph Drawing.* Ed. by W. Didimo and M. Patrignani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 451–462. ISBN: 978-3-642-36763-2.

[44] “Quasicrystal”. In: *Wikipedia – The Free Encyclopedia.* 2018-02-18. URL: https://en.wikipedia.org/wiki/Quasicrystal (visited on 2018-03-12).

[45] A. Refregier. “Shapelets – I. A method for image analysis”. In: *Monthly Notices of the Royal Astronomical Society* 338.1 (2003), pp. 35–47. DOI: 10.1046/j.1365-8711.2003.05901.x.

[46] R. dos Santos Vieira, H. A. D. do Nascimento, and W. B. da Silva. “The Application of Machine Learning to Problems in Graph Drawing – A Literature Review”. In: *Proc. International Conference on Information, Process, and Knowledge Management.* 2015, pp. 112–118.

[47] S. Schaefer, T. McPhail, and J. Warren. “Image Deformation Using Moving Least Squares”. In: *ACM transactions on graphics (TOG).* Vol. 25. 3. ACM. 2006, pp. 533–540. DOI: 10.1145/1141911.1141920.

[48] D. W. Scott. “On optimal and data-based histograms”. In: *Biometrika* 66.3 (1979), pp. 605–610. DOI: 10.1093/biomet/66.3.605.

[49] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.4 (1948-10), pp. 623–656. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb00917.x.

[50] N. Srivastava *et al.* “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[51] R. Suderman, D. Lizotte, and N. Mohieddin Abukhdeir. “Theory and Application of Shapelets to the Analysis of Surface Self-assembly Imaging”. In: 91 (2014-04).

[52] R. Tamassia. *Handbook of Graph Drawing and Visualization.* Discrete Mathematics and Its Applications. CRC Press, 2013. ISBN: 9781420010268.

[53] *Tensor Flow.* URL: https://tensorflow.org/ (visited on 2018-02-22).

[54] *The GraphML File Format.* URL: http://graphml.graphdrawing.org/ (visited on 2018-01-30).

[55] *The NIST Reference on Constants, Units, and Uncertainty. Avogadro Constant.* 2014. URL: https://physics.nist.gov/cgi-bin/cuu/Value?na (visited on 2018-03-11).

[56] I. G. Tollis *et al. Graph Drawing: Algorithms for the Visualization of Graphs.* Pearson, 1999. ISBN: 9780133016154.

[57] C. Ware *et al.* “Cognitive Measurements of Graph Aesthetics”. In: *Information Visualization* 1.2 (2002), pp. 103–110. DOI: `10.1057/palgrave.ivs.9500013`.

[58] E. Welch and S. Kobourov. “Measuring Symmetry in Drawings of Graphs”. In: *Computer Graphics Forum* 36.3 (2017), pp. 341–351. ISSN: 1467-8659. DOI: `10.1111/cgf.13192`.

Index

Appendix

Using Our Software

Shortly after the official submission of this thesis, all source code will be made available at `http://klammler.eu/msc/` for anyone to experiment with it. Since time constraints hindered us from including a more comprehensive reference documentation in this appendix, this information will be made available on said web site and in the `README` file.

We are also thinking about a way to make the web UI publicly available but have yet to figure out a good way of doing so. If it should happen, the link provided above will give a pointer to the place where the interactive web page can be found. If it doesn't happen, it will always be possible to download the source code and run the web server (which is implemented using Python's `http.server` module) locally as we did it for a long time now. The problem with making the web UI publicly available on the internet is that the current implementation cannot handle concurrent requests and might also be vulnerable to malicious inputs.

The source archive contains a CMake project which can be configured and built using the normal procedure. System requirements and software dependencies will be specified in the `README` file. Several configuration options are available at the CMake level or via environment variables and many more via configuration files that will be explained in the `README` file, too. Running our experiment is just a matter of building the target `deploy` while the evaluation shown in chapter 9 can be repeated by building the `eval` target. In order to launch the web server locally on port 8000, build the `httpd` target. All these tasks can also be achieved by invoking the respective scripts directly which offers a greater level of control. Finally, building the `report` target will go and rebuild all imagery from scratch using our toolbox and then finally typeset this document with the experimental results available on the current machine.

We hope that this piece of software will be found useful and would be happy to assist with any difficulties that might arise. Please feel free to contact us at the e-mail address `moritz.klammler@student.kit.edu` or follow the link above to find an up-to-date contact address.