Drawing Area- and Order-Aware Packings of Rectangles

Daniel Lucas

Bachelor thesis
September 2018

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Christoph Daniel Schulze

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

KIELER is a research project about enhancing the graphical model-based design of complex systems. Amongst others, the project contains an editor for SCCharts, a visual language designed for modeling safety-critical reactive systems. The editor uses automatic layout algorithms to relieve developers of the tedious task of modeling SCCharts using drag and drop. The automatic layout algorithms are implemented in ELK, an official Eclipse project that implements an infrastructure to connect diagram editors or viewers to automatic layout algorithms.

The editor uses an algorithm called *Box Layout* to place box-like areas in SCCharts called *regions*. The algorithm is not optimal as it produces large amounts of unwanted whitespace. Due to the rectangular shape of regions, determining their layout is essentially a packing problem.

This thesis proposes five packing algorithms aiming to optimize the following goals: minimizing whitespace, approximating a given desired aspect ratio, and following a given order. Several definitions of what it means to follow an order are introduced and discussed. The presented algorithms are implemented in the context of ELK and evaluated with randomly generated test instances as well as real-world examples. The algorithm found to be superior to Box Layout uses an approximated bounding box and fills the bounding box row-wise. Inside the rows, rectangles are organized as stacks. After an initial packing, the algorithm compacts rows and fills them with elements from subsequent rows as long as improvements are made to the packing.

Contents

List of Figures

List of Tables

Acronyms

DAR Desired Aspect Ratio

ELK Eclipse Layout Kernel

EPL Eclipse Public License

GrAna Graph Analysis

IDE Integrated Development Environment

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

KLighD KIELER Lightweight Diagrams

OPL Optimization Programming Language

SCCharts Sequentially Constructive Statecharts

Chapter 1

Introduction

When developers use graphical modeling tools, their goal is to describe or plan a technical entity. Modeling can be very tiresome and time consuming when doing it manually. For example, when introducing a new element to the model in a *drag & drop* based editor, the alignment of elements may have to be reconsidered. A *drawing* or *layout* maps every element to a distinct and unique place on a given drawing area. Automatic graph layout is concerned with automatically computing a drawing for a given graph, which may represent a model. The elements of a model can be portrayed by nodes and edges. A drawing determines how and where the graph's elements are placed on the given drawing area. The goal is to free the developers of mechanical work such as manually placing elements. In order to achieve this, textual specifications of the models can be used, which are parsed, converted into an abstract data structure, and then used by an algorithm to compute a layout. The corresponding algorithms are called *graph drawing algorithms* [DET+94]. Apart from freeing the developer of work, graph drawing algorithms are designed to optimize for specific aesthetic criteria, but will usually not produce perfect results. Research has been done on graph drawing to find a common understanding of aesthetic criteria [TDB88; Pur02]. A tool that focuses on textual specification and automatic generation of drawings is the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER), an editor and research project about enhancing the graphical model-based design of complex systems (see Section 2.3.1 for a detailed description of KIELER). Underlying KIELER, the Eclipse Layout Kernel (ELK) provides an infrastructure to connect diagram editors such as KIELER to layout algorithms implemented in ELK (see Section 2.3.2 for a detailed description of ELK).

An example of a modeling language that can be described textually is given by Sequentially Constructive Statecharts (SCCharts), a visual language designed for modeling safety-critical reactive systems. SCCharts are based on a statechart notation. In these states, especially macro states, it is possible to define multiple concurrent regions [HDM+14]. An example of SCCharts is given in Figure 1.1 (see Section 2.2 for a detailed description of SCCharts).

Layout algorithms are designed to optimize quantifiable aesthetic criteria, but unanticipated input graphs may produce unsatisfactory results. Thus, layout problems may only be discovered when using a finished modeling tool such as KIELER with fully developed and implemented layout algorithms as those provided by ELK. This is the case in the example given in Figure 1.2. A waste of space can be observed in the regions *A* to *K*, caused by the region *L* in the same row. The example was created illustrating SCCharts with KIELER. The layout algorithm producing the results is implemented in ELK. In order to allow the user to understand given information quickly, SCCharts offer the option to hide the contents of one or

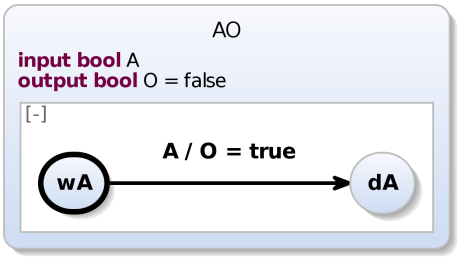


Figure 1.1. An example of SCCharts. Taken from the SCCharts Online Compiler, which is accessible via `http://www.sccharts.com/`.

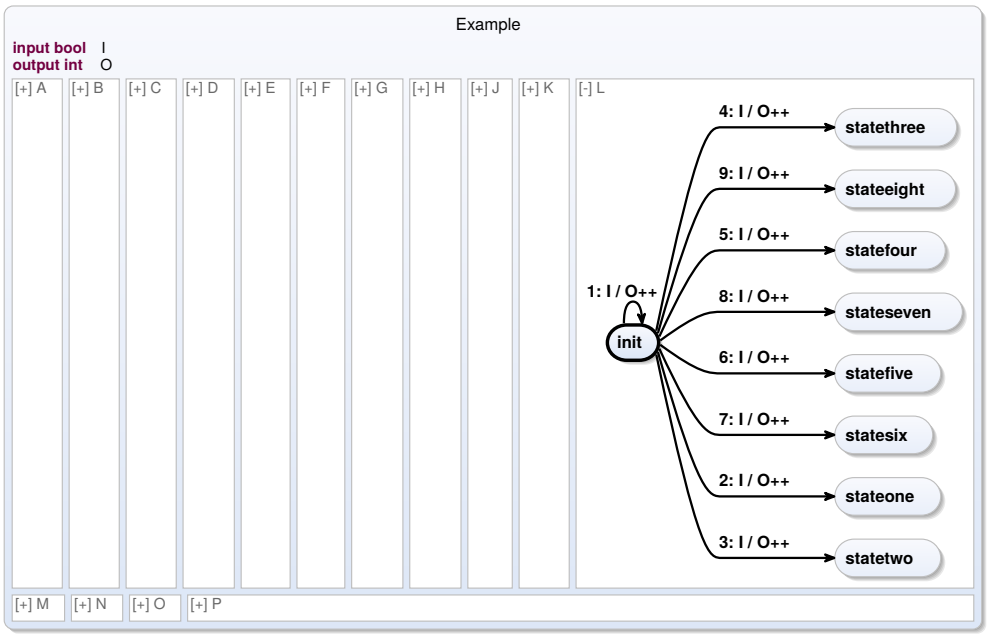


Figure 1.2. An example of the current layout. Regions A-K take up the majority of the state 'Example'.

multiple regions by collapsing them. Only their name and a button for expansion are visible in collapsed regions. The variable size of the regions produces a layout problem, because elements cannot be placed uniformly. The current algorithm places the regions successively in a row. To maintain a legible ratio between width and height, the row is separated into multiple rows. The region with the maximum height in a row determines the height of every region in that particular row in order to properly align the elements and eradicate empty spaces. Ultimately, this leads to a waste of space since the collapsed regions might take up more space than they need to. Consequentially, a lot of empty space is visible inside the regions.

As Figure 1.2 illustrates, the layout problem is solely concerned with the arrangement of the regions. Their internal layout is not considered by this thesis. Since there is only one type of element to be considered, a graph containing nodes and edges is not a viable mathematical

structure to define this problem. Rather, regions are abstracted as rectangles of variable size with minimal dimensions. Hence, the goal of this thesis is to pack a set of rectangles onto the available drawing area in a way that produces a layout that is superior to the layout illustrated in Figure 1.2.

A problem such as this is called a *packing problem*, which is a broad research field with a variety of subproblems [FDK02]. Packing problems are a class of optimization problems with the goal to pack objects into one or multiple containers, either with the lowest number of containers or as densely as possible. An arrangement of rectangles is called *packing*.

Purchase provides formal metrics for measuring the aesthetic quality of a drawing of a graph. She describes the seven most common aesthetic criteria in detail [Pur02]. Unfortunately, these criteria are constrained to drawings of graphs containing nodes and edges as most are concerned with the layout and alignment of edges. This thesis' problem does not contain any edges. Thus, these criteria cannot be utilized as a paradigm for the improved drawing. However, *maximizing node orthogonality* may be applied when interpreting the rectangles as nodes. Di Battista et al. further describe minimal used area and ratio between longest and shortest side of a drawing as criteria, both of which may be applied to this thesis' work [DET+99]. Since this thesis cannot follow a general guide for a proper legible drawing, further criteria will be described in Section 1.1.

Apart from finding a legible position for all regions, it might be beneficial to adopt the order in which the regions were specified by the user in the textual description of SCCharts. The order of the elements in the specification might be the same as the order in the developer's mind. The way a developer pictures a model in their head is called *mental map* [PHG06]. Arranging elements contrary to the developer's mental map and expectations may cause confusion and cost time to survey the unanticipated product.

1.1 Problem Statement

This section will formalize the problem to be solved and the objectives of this thesis. Definitions will be examined for the problem as well as metrics, which can be used to assess the quality and usability of a computed drawing.

Let $R = \{r_1, \ldots, r_n\}$ be a set of $n \in \mathbb{N}$ rectangles. Let A be a *drawing area* where the rectangles can be placed. The drawing area A has its origin at the top-left corner, spanning horizontally to the right and vertically to the bottom. A point on the drawing area can be distinctly described by a tuple $(x, y) \in \mathbb{R} \times \mathbb{R}$ where x describes a value on the horizontal x-axis and y describes a value on the vertical y-axis. A rectangle i is defined by the tuple (x_i, y_i, w_i, h_i) with x-coordinate $x_i \geqslant 0$, y-coordinate $y_i \geqslant 0$, current width $w_i > 0$, and height $h_i > 0$. Rotation of rectangles is not allowed. Regarding the use case in Chapter 1, minimum dimensions are required for each rectangle, as there is a given minimal amount of information that needs to be presented. For example, a collapsed region has to be at least big enough to display the name and an expand button, whereas expanded regions have to be able to show all of their internal contents. The minimal dimensions of the expanded regions are computed

before the arrangement of the rectangles, as the layout of their internal elements is computed beforehand. Hence, the denotation of a rectangle i is expanded in order to formalize the minimal dimensions. Let the variables $w_i^{min}, h_i^{min} \in \mathbb{R}_{>0}$ denote the minimal dimensions for a rectangle i, where w_i^{min} determines the minimum-width and h_i^{min} the minimum-height. Therefore, a rectangle is now denoted by $(x_i, y_i, w_i, h_i, w_i^{min}, h_i^{min})$. Initially, $w_i = w_i^{min}$ and $h_i = h_i^{min}$ hold. Later, a rectangle might need to be enlarged to make it fit into its surroundings. However, it can never be smaller than its minimal dimensions. Therefore,

$$w_i^{min} \leqslant w_i \wedge h_i^{min} \leqslant h_i$$

must always hold.

The coordinate system in this problem spans vertically down and horizontally to the right with $(0, 0)$ being in the top-left corner. No negative coordinates are allowed as the axes represent the drawing area's borders.

Moreover, the rectangles must be packed in a way that their borders are orthogonal to the drawing area's borders. A packing with this constraint is called orthogonally oriented [Ste97].

Under the assumption that there is an order to the rectangles, let $\Pi(i)$ denote the position in the order for rectangle i. A definition of an order implies a rule that a drawing should follow. Such definitions restrict the positioning of rectangles' coordinates relative to other rectangles. Next, a very simple depiction of what it means for an order to be respected is introduced. Different definitions of orders will be discussed in detail in Section 3.1. Order Π is followed if the following holds for any two rectangles $n, k \in R$ with $n \neq k$:

$$\Pi(n) < \Pi(k) \iff (x_n < x_k) \vee (y_n < y_k) \tag{1.1.1}$$

Equation 1.1.1 demands that the top-left corner of a rectangle k that follows a rectangle n according to order Π is located lower than or right of n's top-left corner. On the other hand, it allows k to be placed fully to the left of n as long as the top-left corner of k is below the top-left corner of n.

The *aspect ratio* of an object with width W and height H is calculated by W/H. A function that maps every rectangle to a unique point on the drawing area is called a *drawing*, in the following denoted by B. Let an aspect ratio indicating the drawing's favored aspect ratio also be given. This aspect ratio is called the Desired Aspect Ratio (DAR). The drawing's dimensions can be determined by identifying its *bounding rectangle*, which is the minimal surrounding rectangle of the drawing [FDK02]. Consequentially, the bounding rectangle has the dimensions width w_B and height h_B, making it possible to calculate the aspect ratio for the drawing. These values are easily computed by $w_B = \max_{i \in R}\{x_i + w_i\}$ and $h_B = \max_{i \in R}\{y_i + h_i\}$ under the assumption that every rectangle in R has already been placed on the drawing area and that the top-left corner of the bounding box is located at $(0, 0)$. These equations describe the biggest x- and y-coordinates that any rectangle adopts.

As illustrated in Figure 1.3, only considering the aspect ratio of a drawing as a target metric is not enough. The *max scale measure* assesses a drawing and determines whether it is compact and uses the given drawing area effectively [RH18]. Let the dimensions of drawing area A be given by w_A and h_A. These values may be implicitly given by the editor's window

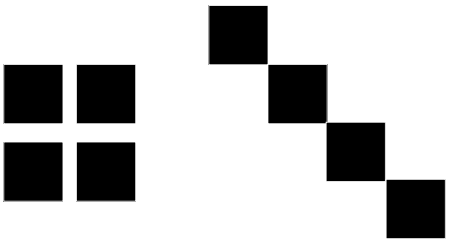


Figure 1.3. Two drawings of four equal sized rectangles. Both drawings have an aspect ratio of 1, however the left drawing is more compact.

or the user's monitor size. This thesis assumes that the dimensions of the drawing area is always big enough to fit all rectangles. The max scale measure states that the largest scale factor s computed by $\min\{w_A/w_B, h_A/h_B\}$ over different drawings B can be used to display a drawing within A as compact as possible.

Apart from aspect ratio and scale factor, the area taken up by the packed rectangles can be used as a metric. Minimizing the used area is a common target function for most packing problems. The used area is computed by $w_B \cdot h_B$. A similar metric is whitespace, which is the difference between the used area and the summed up area of the given rectangles.

This thesis' goal is to develop and implement a packing algorithm that calculates w_i, h_i, x_i, and $y_i, \forall i \in R$, meets the following requirements, and ultimately presents a solution to the layout problem that was described in the introduction.

1. All rectangles should be packed compactly on the drawing area without any two rectangles overlapping each other.

2. The drawing must be orthogonally oriented.

3. The drawing must respect the given order Π.

4. The scale factor s should be maximized.

5. The area of the drawing's bounding rectangle should be minimized.

6. Let the aspect ratio of the drawing's bounding rectangle approach the desired aspect ratio.

7. $\forall i \in R$, let w_i^{min} and h_i^{min} be as close to h_i and w_i as possible.

8. To produce an aesthetically pleasant drawing, all whitespace openings inside the drawing and nonuniform borders should be eliminated by enlarging the rectangles.

Some of the goals described above are in conflict with one another. This will be considered in the evaluation. As the layout problem in Chapter 1 was observed in the modeling tool KIELER, the proposed algorithm should be implemented in ELK and called by KIELER to remove the issue from the project.

1.2 Related Work

Since this thesis proposes packing algorithms that have essential characteristics of layout algorithms, related work may cover either packing or layout algorithms. Both are broad research fields.

Introductory work regarding graph drawing and automatic layout is given by Battista et al. [DET+99; DET+94; TDB88].

Generally, this thesis covers a two-dimensional rectangular packing problem with the goal of packing multiple rectangular objects of variable size into a larger given rectangle [DD92]. The problem is NP-complete [Kor03; DD92; BCR80]. Schneider et al. [MS11] have proven that most restrictions of the two-dimensional rectangular packing problem remain NP-complete.

Most publications regarding two-dimensional rectangular packing problems optimize the packing of rectangles in terms of use of space. However, this thesis considers more restrictions. Besides a packing, this thesis considers the arrangement of rectangles to be a drawing that aesthetic criteria are applied to. The use case requires that a viewer of the packing can easily read and assess the contents represented by the rectangles. Furthermore, the viewer's mental map should be sustained. The order restriction given in this problem can help structuring the packing and make it easier for a viewer to assess the packing. This work also differs in terms of drawing area awareness. It is common to restrict the container the rectangles are to be placed in [DD92]. However, this thesis additionally considers the given desired aspect ratio for the packing inside the given drawing area. Another restriction is the execution time of a proposed algorithm as the use-case requires a layout in near real-time. Hence, optimal packing algorithms usually exceed realistic time for this requirement.

Orthogonal Packings in Two Dimensions

Baker, Coffman, and Rivest give an introduction to bin packing and illustrate its importance in real-life applications [BCR80]. Bin packing is concerned with packing a set of rectangles into a container of fixed width and unlimited height. They give a general explanation of the complexity of bin packing, which is NP-complete. Next to formal basics, they introduce approximation algorithms and give an assessment on their limits and worst-case bounds. They focus on *bottom-up left-justified packing*, which is simply called BL packing. In this approach, rectangles are placed at the lowest possible location. There, the rectangle is then left-justified in that vertical position. Their work shows that it is possible to find a packing that is at most two times as high as the optimal packing by using the BL packing approach.

Optimal Rectangle Packing

Korf covers a packing problem where a set of rectangles with fixed orientations is to be packed into another rectangle with minimal dimensions without overlapping any of the given rectangles. The goal is to pack the rectangles using the smallest possible area [Kor03]. They introduce a branch-and-bound algorithm to optimally solve the problem. They compare

their results to other algorithms and their techniques can be applied to further problems of packing non-rectangular shapes in non-rectangular enclosing containers. Furthermore, this work contains a proof showing the NP-completeness of the rectangular packing problem.

Disconnected Graph Layout

Freivalds, Dogrusoz, and Kikusts present a paper covering a new approach and reviewing existing algorithms for disconnected graph layout [FDK02]. Components of a graph are usually placed by considering the bounding rectangle of each component. Intuitively, this costs a lot of space in case a component itself does not have a compact layout. Viewing components as rectangles makes the usual approach a basic packing problem. Furthermore, they give an overview over existing methods for placing rectangles. However, their approach is based on polyomino representation of components. Their new approach produces more compact and uniform drawings than the methods they compared their algorithm to.

A Layout Adjustment Problem for Preserving Orthogonal Order

Hayashi, Inoue, Masuzawa, and Fujiwara present an article considering a layout adjustment problem for minimizing the area under the constraints that rectangles should not overlap and that the orthogonal order should be preserved [HIM+98]. Layout adjustment generally seeks to improve a given layout. They proposed a corresponding heuristic for the problem. Lastly, they compared their algorithm to a solution developed by Misue et al. on the same issue [MEL+95]. Their algorithm produces a smaller area, while having the same computation time.

1.3 Outline

First, preliminaries will be explained (Chapter 2). A detailed general description of packing problems (Section 2.1) and SCCharts (Section 2.2) as well as the technology used for implementing the algorithm (Section 2.3) are in focus. The context of the editor KIELER, in which the issue in Chapter 1 appeared, will be explained (Section 2.3.1). This is followed by some technical background and ELK (Section 2.3.2). Next, five algorithms are proposed that fulfill the needs as described in Section 1.1 (Chapter 3). Next follows the implementation of the proposed algorithms (Chapter 4), whose technologies are explained in Section 2.3. The implementation chapter is split in two: the actual implementation of the algorithm in ELK and the process of calling the algorithm to fix the layout problem. Next, after presenting results, a quantitative and performance evaluation is conducted (Chapter 5). Lastly, a conclusion is drawn and future work will be discussed (Chapter 6).

Chapter 2

Preliminaries

This chapter introduces the basics needed to understand this thesis. First, packing problems will be introduced. This is followed by the visual synchronous language SCCharts providing the layout problem motivating this thesis as described in Chapter 1. As mentioned before, the developed algorithm will be implemented in existing technology. Consequentially, the used technologies KIELER and ELK will be introduced as well. Lastly, this chapter introduces CPLEX, a tool for implementing linear programs and Graph Analysis (GrAna), a tool for evaluating graphs.

2.1 Packing Problems

Packing problems are a broad research field and extensive research has been done on different variations [FDK02]. Generally, packing problems are a class of optimization problems with the goal to pack objects into containers. There are different variations of the problem regarding quantity and sizes of objects or containers. Additionally, there are different versions of the packing problem regarding the dimensions of objects and containers such as two and three dimensional problems. Accordingly, the research field of packing problems contains subproblems with individual characteristics such as stock cutting, loading, and knapsack problems [DD92].

Packing problems in general are NP-complete [Kor03; DD92; BCR80] and algorithms that optimally solve given problems have been found [Kor03].

Dowsland and Dowsland [DD92] provide an overview of the variety of packing problems while focusing on presenting models and solutions. Different types of problems are introduced in the following:

Two-dimensional rectangular packing The main problem can be described as packing rectangular objects of variable size into a larger given rectangle. Generally, it is required to place rectangles parallel to the container's axes. Usually, the goal is to pack the objects as densely as possible in order to minimize wasted space. More subproblems are known for this problem, for example regarding the quantity, dimensions, and sizes of objects. Schneider et al. [MS11] have proven that most restrictions of this problem remain NP-complete.

Pallet loading Instead of packing objects of different sizes, the pallet loading problem considers packing identical rectangular objects onto a rectangular loading area. However,

the objects may be turned as long as its borders are parallel to the loading area's axes, ultimately causing this problem to not be trivial. The goal is to maximize the number of cases fitting onto the loading area. This problem is mainly given in logistics and transportation. Pallet loading is a reduced two-dimensional rectangular packing problem. A simple and effective approach to this problem is to find a pattern fitting the loading area's boundary and to extend this pattern towards the center. Pallet loading provides heuristic solutions that are often close to the optimal solution.

Bin packing Two main problems are distinguished in this class of problems. First, given are a set of (rectangular) pieces and a (rectangular) bin with fixed width and unlimited height. The goal is to fit all the pieces into the bin while minimizing the bin's height needed to do so. This variation is called *strip packing*. The other version of this problem, usually called *two-dimensional bin packing*, allows multiple bins with fixed width and height. Here, the goal is to minimize the number of bins used to fit all pieces. Placing the pieces parallel to the bin's axes is usually required. This is because some motivations for bin packing problems originate in scheduling where jobs are presented as rectangular objects that cannot be turned in any direction.

Three-dimensional packing Applications such as packing ship containers require extending the previous class of problems as the available space is three-dimensional. The available space, usually called container, is enclosed by 6 orthogonal walls, thus forming a cubic object. The objects to be packed are called cargo. Adding further dimensions increases the combinatorial complexity and lowers the effectiveness regarding execution time of exact solution procedures. Similar to the other classes of problems, there are different variations of three-dimensional packing regarding requirements and goals. An example for such requirement is load stability, which requires the cargo to be packed evenly in the horizontal dimension.

Non-rectangular packing When either the objects or containing space are not rectangular, the problem is an instance of non-rectangular packing. If the containing area is rectangular but parts of it are unusable, usually small modifications to existing algorithms are needed, because the container can now be modeled as a set of smaller rectangles. More fundamental modifications are required, if the container is of non-rectangular form. The number of different variations of this problem correlates with the variety of geometrical forms that a container or object can adopt. In research, packing of circles and spheres is most common, whereas more practical approaches consider non uniform shapes.

2.2 SCCharts

SCCharts are a visual language designed for modeling safety-critical reactive systems. They were introduced by von Hanxleden et al. in 2014 [HDM+14]. SCCharts use a statechart notation similar to statecharts as introduced by Harel [Har87]. Moreover, they can be viewed as

an extension to SyncCharts, which were proposed by André [And95]. Features such as hierarchy, concurrency, and modularity are found in SyncCharts and SCCharts. Furthermore, determinate concurrency based on a synchronous model of computation (MoC) is provided. The semantics of SCCharts are defined on a determinate basis, but the safety-critical focus is also reflected in the definition of the language. Several projects have been completed using SCCharts, such as modeling software for a model railway controller [The06]. Figure 1.1 illustrates a simple example of SCCharts, which illustrates a root state called *AO*, an interface declaration containing boolean input and output variables, and a region. The latter contains two states connected by a transition with a transition trigger and effect. Each region contains exactly one initial state, which is *wA* in this case.

This thesis will concentrate on macro states and parallel regions since they are the origins of the layout problem stated in Section 1.1. Macro states are extended simple states because they also have inner behavior, which is specified in one or multiple regions containing further elements such as states and transitions. When a macro state is entered, all of its inner regions are executed concurrently. When two regions are declared inside the same state, they share a *parent scope* and are considered to be parallel. During execution, every region sharing the same parent scope will be run inside its own thread.

Like any model, SCCharts can become complex. A user might not require all information in an instance of SCCharts in every moment. Hence, they allow to hide information by expanding or collapsing regions. This results in showing the regions' inner behavior or hiding it by only revealing the regions' name. Ultimately, this results in variable sizes of regions given by the size of their inner behavior and length of name. The variable size is the main reason for this thesis' layout problem.

2.3 Technologies Used

The goal of this thesis is to solve the layout problem introduced in Chapter 1 that occurred when drawing SCCharts with the tool KIELER. Layout algorithms used by KIELER are provided by ELK. Both technologies are portrayed in detail in this section. Additionally, CPLEX and GrAna are presented, which are necessary for the implementation and evaluation of the proposed algorithms.

2.3.1 KIELER

Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) is a research project about enhancing the graphical model-based design of complex systems [FH09]. A key element is the ability to automatically compute layouts of graphical components within the modeling environment. Enhancements to its predecessor are the integration of a variety of modeling languages and the integration of the project into the rich client platform Eclipse[1]. The project

[1] http://www.eclipse.org/

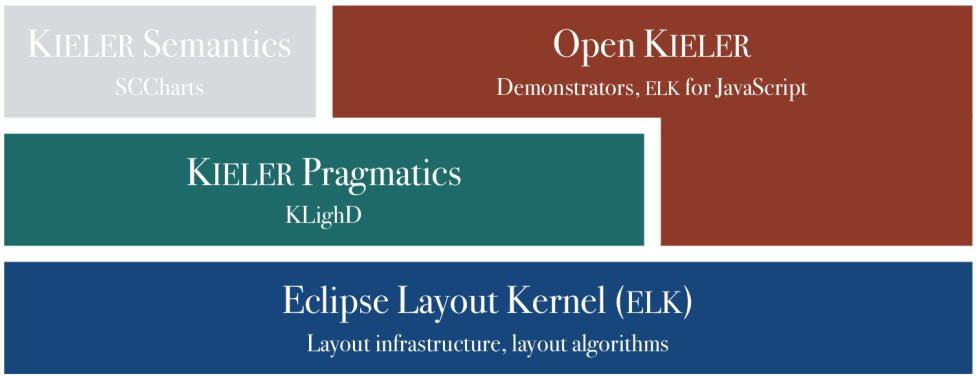


Figure 2.1. Division of the KIELER research project into layout(ELK), pragmatics, semantics, and open KIELER.

is developed as open source software, licensed under the Eclipse Public License (EPL)[2]. The project is divided into layout, pragmatics, semantics, and open KIELER as illustrated in Figure 2.1.

Layout

When a user edits a model with a drag- and drop based editor, inserting new elements can be a troublesome experience as existing elements may have to be moved out of the way. Much effort and time is needed to complete a task, and sometimes the information becomes second-tier due to struggles with graphical components. Layout algorithms can free the user of these tasks. The subproject layout provides automatic layout algorithms as the key element. Other features are enabled, such as *focus & context* [CMS99] and *dynamic view creation* [SSH13].

As mentioned before, the contributed automatic layout algorithms are implemented in ELK, which is presented in detail in Section 2.3.2.

Pragmatics

The term *pragmatics* in linguistic studies describes how context contributes to meaning [Cru11]. The following passage demonstrates how the meaning of this word is implemented and executed in KIELER.

A set of practical aspects that influences the daily work of modelers in the context of *model-driven engineering* [Sch06] is denoted with the term *pragmatics*. This includes the creation and modification of models and the synthesis of diverse views on those models depending on different stakeholders and purposes.

The main subproject of pragmatics is KIELER Lightweight Diagrams (KLighD), which offers a model browser that can be adapted to any kind of graph-based representation. Graphical or textual representations are synthesized from a domain model and dismissed if they are

[2] http://rtsys.informatik.uni-kiel.de/~kieler/epl-v10.html

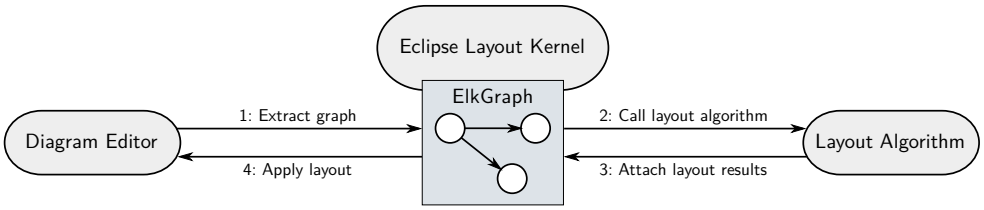


Figure 2.2. How ElkGraph is used as a key element between editing tools and layout algorithms. Source: [SSR16].

not needed anymore. Basically, KLighD obtains nodes, edges, and hierarchy levels from the domain model using a *diagram sysnthesis*.

Semantics

In linguistic studies, the word *semantics* describes the meaning of language. It handles the correlation between signifiers, such as words, phrases, or symbols, and their denotation. This also applies to KIELER.

Semantics in KIELER context essentially focus on SCCharts as described in detail in Section 2.2.

2.3.2 ELK

The Eclipse Layout Kernel (ELK) is an official Eclipse project[3] that implements an infrastructure to connect diagram editors or viewers to automatic layout algorithms. Some of these are already implemented and ready to use.

ELK can be separated into two parts: *kernel* and *algorithms*. The kernel provides an interface connecting editors with the algorithms. A key element here is the *ElkGraph*, passing diagrams to the layout algorithms and computed layout information to the editor as a data structure as illustrated in Figure 2.2. In order to use ELK, a tool developer usually only has to provide a transformation producing an ElkGraph from their editor's domain model. Since this thesis implements an algorithm in ELK to be called in the KIELER pipeline, this section will focus on information relevant for implementing an algorithm, as opposed to using ELK as a library behind an editor.

A key for algorithm developers is to understand how informations are handled, stored, and accessed. Furthermore, it is paramount to know how to place elements on the drawing area and how the drawing area is organized.

The term *graph* denotes a set of nodes and edges and other elements belonging to it such as given ports and labels. Figure 2.3 illustrates simple graph terminology. We distinguish between *simple nodes* and *hierarchical nodes* whereas the former does not contain any child nodes and the second one does. *Simple edges* connect nodes sharing the same parent node,

[3] //www.eclipse.org/elk/

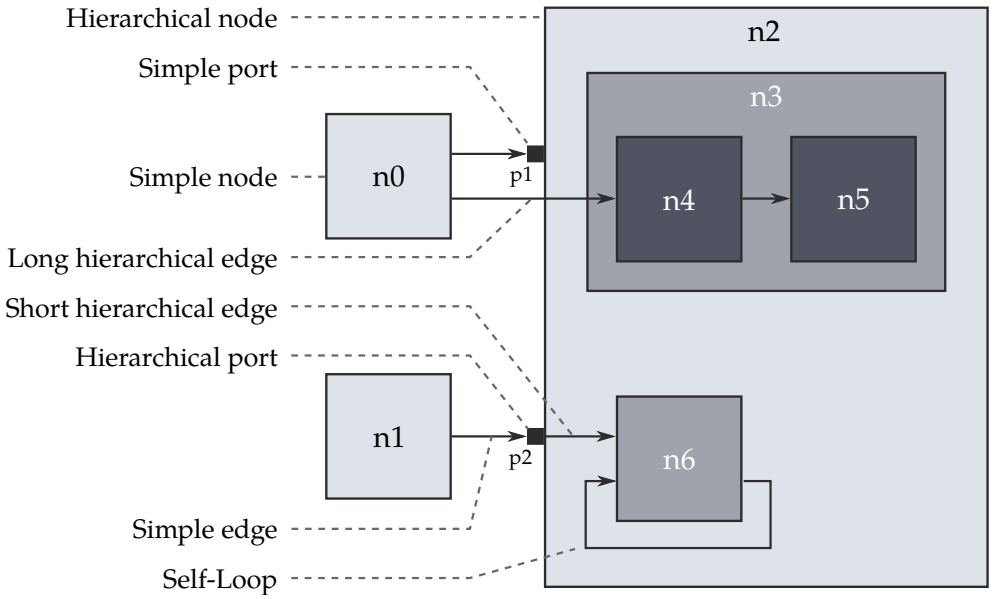


Figure 2.3. Illustration of simple graph terminology. Source: [SSR16].

where *hierarchical edges* are edges that are not simple. *Ports* are connection points on nodes for edges. How these terms are implemented in the ElkGraph meta model is illustrated in a class diagram in Figure 2.4. The common graph elements are represented by *ElkNode*, *ElkEdge*, and *ElkPort*. Additionally, the ElkGraph meta model consists of abstract elements providing abstraction and modularity, both of which are beneficial for extensibility and maintainability. The elements of the ElKGraph meta model are described in detail in Table 2.1.

The coordinate system used in ELK is a two dimensional coordinate system with the origin in top-left corner spanning horizontally to the right and vertically to the bottom. The coordinates of a single element point to the element's top-left corner. For the algorithm implementation following in Chapter 3, a coordinate system equivalent the one presented here is used.

Finally, ELK provides a series of algorithms, two of which are important for this thesis: *Box Layout*[4] and *Elk Layered*[5]. The former is an algorithm for packing unconnected boxes, usually given by graphs without edges. The latter is a layer-based algorithm as introduced by Sugiyama et al. [STT81]. It arranges as many edges as possible into one direction by placing nodes into subsequent layers. The implementation offers different routing styles and supports hierarchical layout.

Regarding the layout problem described in Chapter 1, Elk Layered is used when placing the contained elements inside of a region. In order for a region to be big enough to display all elements, the computed inner layout provides its dimensions. The Box Layout algorithm then places the regions, which ultimately produces the layout problem.

[4] https://www.eclipse.org/elk/reference/algorithms/org-eclipse-elk-box.html

[5] https://www.eclipse.org/elk/reference/algorithms/org-eclipse-elk-layered.html

Table 2.1. Description of the elements in the ELK meta model (see Figure 2.4).

Meta Element	Description
IPropertyHolder	Instances of this type hold a map of values to properties. Developers can define properties for internal and external use, where the latter may be set by clients to configure an algorithm.
ElkGraphElement	Abstract super class. May hold one or more labels.
ElkShape	Superclass of rectangular graph elements: nodes, ports, and labels. Contains properties: width, height, x-, and y-coordinates.
ElkConnectableShape	Edges can be connected to ports or nodes, both of which are connectable shapes. They serve as sources and targets for edges.
ElkNode	Primary ingredient of ElkGraphs. As mentioned before, ElkNodes can contain children (hierarchical nodes). The relation *contained edges* describes edges that are contained inside the node as opposed to incoming or outgoing edges.
ElkLabel	Elements of a graph may have labels that further describe them. These labels are rectangular ElkShapes that can be placed on the drawing area. A property of each ElkLabel is the text it should display.
ElkPort	Ports are connection points for edges that are connected to exactly one node. They can be placed along all sides of a node unless restricted otherwise.
ElkEdge	An ElkEdge has at least one source and one target, allowing it to be a hyperedge (an edge with more than one source or target).
ElkEdgeSection	Describes the path an edge takes through a drawing.

2.3.3 CPLEX

CPLEX is a commercial product developed by IBM for developing and solving complex minimization problems. The product comes with its own Integrated Development Environment (IDE), where minimization problems can be expressed with the Optimization Programming Language (OPL). Moreover, the product offers a Java library to implement linear programs directly in the developers project.

2.3.4 GrAna

Graph Analysis (GrAna)[6] is a test framework to examine graphs on their structural properties such as node or edge count and properties of a computed drawing such as area and whites-

[6] https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=12288271

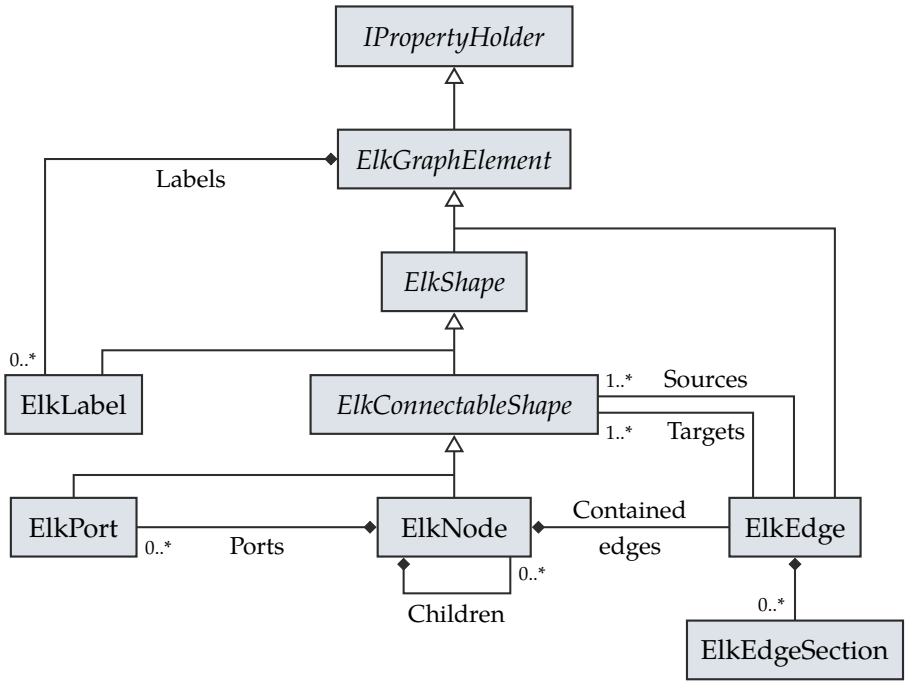


Figure 2.4. ElkGraph meta model. Source: [SSR16].

pace [Rie10]. An analysis can be either performed on a single graph or batch-like on a set of graphs. In the latter case, the results are written to a specified comma-separated value file. All specifications including input- and output files, layout options, and analyses are defined in a *.grana* file. The required plug-ins are part of the KIELER pragmatics project.[7]

[7] https://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/pragmatics/browse

Chapter 3

Theoretical Analysis

First, this chapters discusses and proposes different ordering constraints. Next, this chapter introduces several algorithms aiming to solve the problem as formulated in Section 1.1. This includes two heuristics, a linear program and a binary search algorithm. Additionally, this chapter introduces a simple placement algorithm for a special case, with one big rectangle and multiple same-sized smaller ones. Said special case is altered from the real world motivation for this thesis, where the smaller rectangles may vary in width since they display the corresponding name label in full size.

3.1 Ordering Constraint

This section discusses three possible definitions of following an order. As we recall from Chapter 1, the order of the rectangles is given by the order the elements were specified textually by the developer in the context of SCCharts. Hence, the basic goal of the ordering constraint is to order the packing of rectangles in a way that prevents cases such as illustrated in Figure 3.1, where the rectangles are packed reversely to the given order.

Next, three different definitions of following an order will be introduced. As we recall, the coordinate system spans vertically down and horizontally to the right. The variables w_i, h_i, x_i, and y_i denote the width, height, x-coordinate, and y-coordinate of a rectangle $i \in R$. The order is denoted by Π. Let two arbitrary rectangles be denoted by $n, k \in R$ with $n \neq k$. Three possible definitions of what it means to follow an order are given Equations 3.1.1, 3.1.2, and 3.1.3. The equations are given in ascending order regarding their strictness.

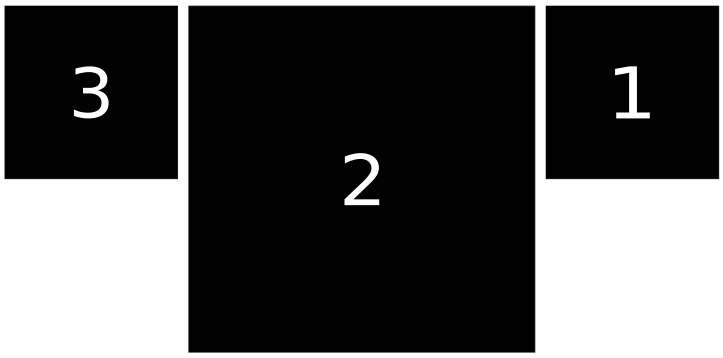


Figure 3.1. An example of a poor packing regarding the ordering constraint.

$$\Pi(n) < \Pi(k) \iff (x_n < x_k) \vee (y_n < y_k). \tag{3.1.1}$$

$$\Pi(n) < \Pi(k) \iff (x_n + w_n \leqslant x_k) \vee (y_n + h_n \leqslant y_k). \tag{3.1.2}$$

$$\Pi(n) < \Pi(k) \iff (x_n + w_n \leqslant x_k \wedge y_n \leqslant y_k) \vee (y_n + h_n \leqslant y_k). \tag{3.1.3}$$

Equation 3.1.1 states that the top-left corner of a subsequent rectangle must be placed further below or right of a preceding rectangle's top-left corner. Equation 3.1.2 adds a further restriction by also adding the preceding rectangle's width and height. Hence, a rectangle can only be placed below the bottom border or right of the right border of a preceding rectangle. Lastly, Equation 3.1.3 adds a further restriction, which forbids a rectangle's top-left corner to be placed at smaller y-coordinates as a preceding rectangle's top-left corner.

Figure 3.2 illustrates two simple drawings where different ordering constraints hold. There are several conclusions that can be drawn from this example. One is that Equation 3.1.1 prevents situations such as illustrated in Figure 3.1, but it may break the western horizontal reading order of left to right. Another is that in order to maintain said reading order, further unused whitespace is produced. Thus, there is a trade-off between reading order and whitespace.

Further, stricter definitions help to sustain the developer's mental map in case the size of one or multiple rectangles changes after an initial packing has been found. Sustaining the user's mental map is stated as a requirement in Section 1.1. Chapter 1 also stated that the size of rectangles is variable. An example of a rectangle changing its size is given by the expansion or collapse of regions in SCCharts. After the expansion or collapse, a new packing for the altered rectangles is calculated. Naturally, stricter definitions then dictate more precisely were the rectangles are to be placed. Thus, stricter definitions are more likely to sustain the developer's mental map. For small instances such as given by Figure 3.2, it is not a big issue for a viewer to trace the changes with a tolerant order definition such as Equation 3.1.1. However, considering instances that contain more rectangles, the change of size of a single rectangle has further consequences for subsequent rectangles, which makes it harder to trace the changes of a packing. Again, there is a trade-off between sustaining the developer's mental map and whitespace.

3.2 Heuristics

As we recall from Section 1.1, the values of w_A, h_A, w_r^{min}, and h_r^{min} for the drawing area A and every rectangle $r \in R$ as well as the order function Π are given. The values of w_B and h_B of a drawing B are given by the rectangles' calculated coordinates and dimensions. As we also recall, metrics for a drawing are aspect ratio, area, whitespace, and scale measure.

Next, this chapter introduces two heuristics for packing the rectangles, as well as a simple heuristic for a special case with one big rectangle and multiple same-sized smaller ones. The first heuristic focuses on simply following the order as defined in Equation 3.1.2 while

(a) Equation 3.1.1 holds.

(b) Equations 3.1.2 and 3.1.3 hold.

Figure 3.2. Two simple drawings that fulfill different ordering constraints.

sequentially placing rectangles and approximating the desired target metric (e.g. aspect ratio or area) similar to a greedy algorithm. Since all following rectangles must be placed to the right or below a given rectangle according to Equation 3.1.2, the first heuristic is called *Down-Right Heuristic*. The second heuristic focuses on aligning rectangles to achieve a more ordered look by also following Equation 3.1.2. It is called the *Row-filling and Compaction Heuristic* as it repeats two phases, *row-filling* and *compaction*, as long as improvements are made. An algorithm considering the special case of having one big rectangle and multiple same-sized smaller ones is called *Special Case Heuristic*.

3.2.1 Down-Right Heuristic

In this approach, the main idea is to strictly fulfill the order as defined by Equation 3.1.2 while prioritizing either a small packing area, achieving the desired aspect ratio, or maximizing the scale measure when comparing different candidate positions. Thus, rectangles are placed sequentially according to the target metric.

The first step is to place the rectangle i with $\Pi(i) < \Pi(k), \forall k \in R$ at the coordinates $(0, 0)$ since it is the first rectangle according to the order and must be placed either to the left of or above all other rectangles. Then, at least two candidate positions are present when placing the second rectangle according to Π: to the right of or below i. The algorithm assesses the positions by calculating aspect ratio, drawing area, and scale measure for each possible drawing. Thus, the algorithm places the rectangle at the candidate position whose drawing has the best value for the target metric. This assessment is done for each subsequent rectangle and their respective candidate positions. Naturally, the candidate positions become more complex and numerous with every added rectangle. Hence, the algorithm defines multiple distinct candidate positions that will be introduced next.

Let $R_{lastPlaced}$ always hold the rectangle that was most recently placed onto the drawing area. The four candidate positions the algorithm assesses are as follows:

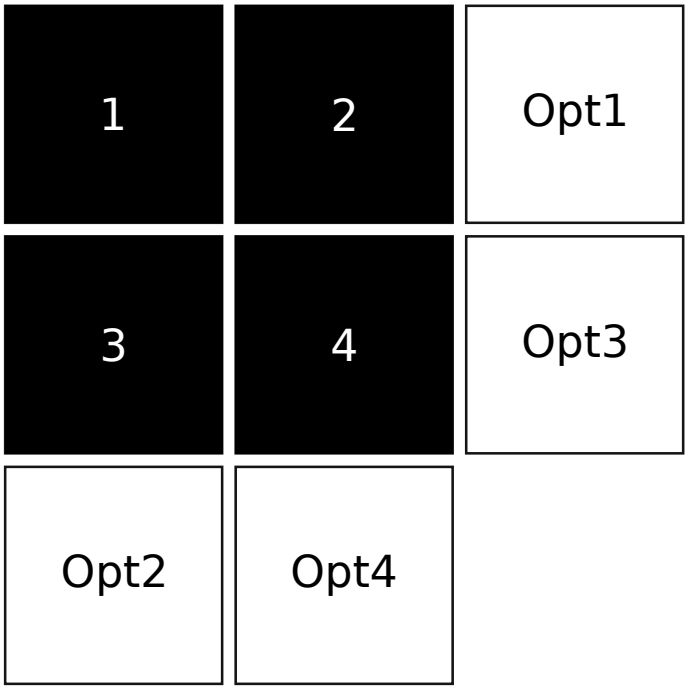


Figure 3.3. Candidate positions of the Down-Right Heuristic. In this example, the variable $R_{lastPlaced}$ holds rectangle 4.

Opt1	Top-right of whole current drawing
Opt2	Left-below whole current drawing
Opt3	Right of $R_{lastPlaced}$
Opt4	Below $R_{lastPlaced}$

The candidate positions are illustrated with a trivial example in Figure 3.3. With the order restriction in mind, it is clear that the candidate positions 3 and 4 are not preferable in this example. Thus, consider Figure 3.4 where these positions are viable candidate positions.

A *packing strategy* determines what metric the algorithm prioritizes. The strategies are as follows: *Max-Scale Driven*, *Area Driven*, and *Aspect Ratio Driven*. Each strategy causes the algorithm to choose the candidate position with the best value for the corresponding target metric. Considering Figure 3.3, it is possible for two positions to produce the same value for a specific metric. Hence, tie-breakers are needed to decide what position to choose. For these strategies, tie-breakers are primarily the other metrics. The decision order for each strategy is listed in Table 3.1.

The scale measure already considers the aspect ratio in its calculation by using the drawing's dimensions. That is why the area is considered before the aspect ratio in the max-scale driven strategy. It is the same for the area driven strategy that considers the scale measure before the aspect ratio. Lastly, the aspect ratio driven strategy considers the area before the scale measure, since the scale measure also favors smaller drawings as described in

Table 3.1. The packing strategies of the Down-Right Heuristic and their corresponding priorities regarding the metrics.

Strategy	Priority 1	2	3
Max-scale Driven	Scale Measure	Area	Aspect Ratio
Area Driven	Area	Scale Measure	Aspect Ratio
Aspect Ratio Driven	Aspect Ratio	Area	Scale Measure

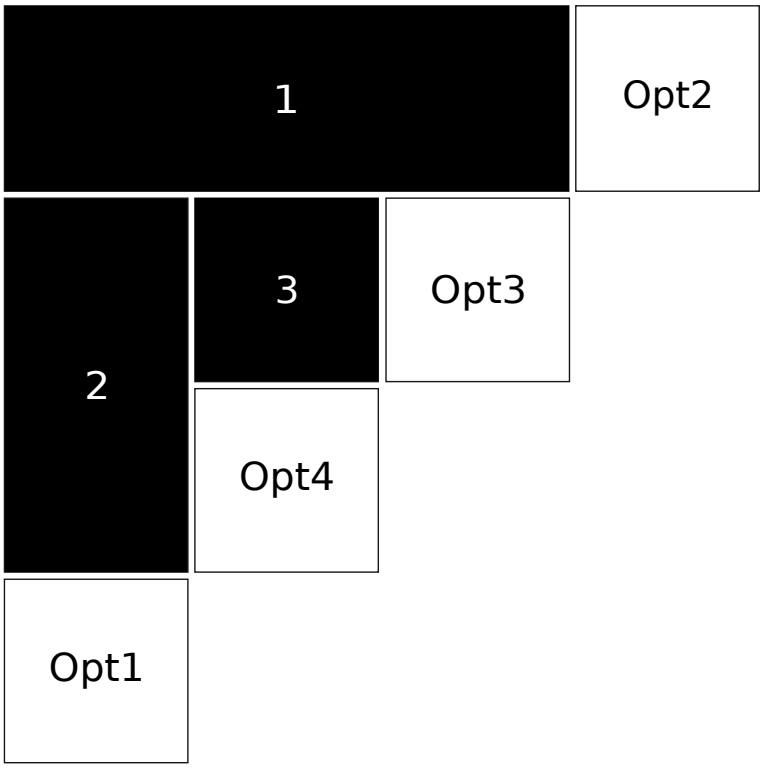


Figure 3.4. A case where candidate positions 3 and 4 cannot be assessed numerically. In this example, the variable $R_{lastPlaced}$ holds rectangle 3.

Chapter 1.

However, some special cases remain after assessing the candidate positions with these strategies. There are cases where the multiple candidate positions have the same value for all three metrics. These special cases are introduced in the following.

Consider Figure 3.3. It is clear that candidate positions 1 and 3 will produce the same metric values, as will candidate positions 2 and 4. This is because they will equally enlarge the dimensions of the drawing. When deciding between two of these positions producing the same values, the algorithm will prefer the positions 1 and 2. Placing the rectangles at position 3 or 4 would create whitespace at the positions 1 or 2 that cannot be eliminated because the

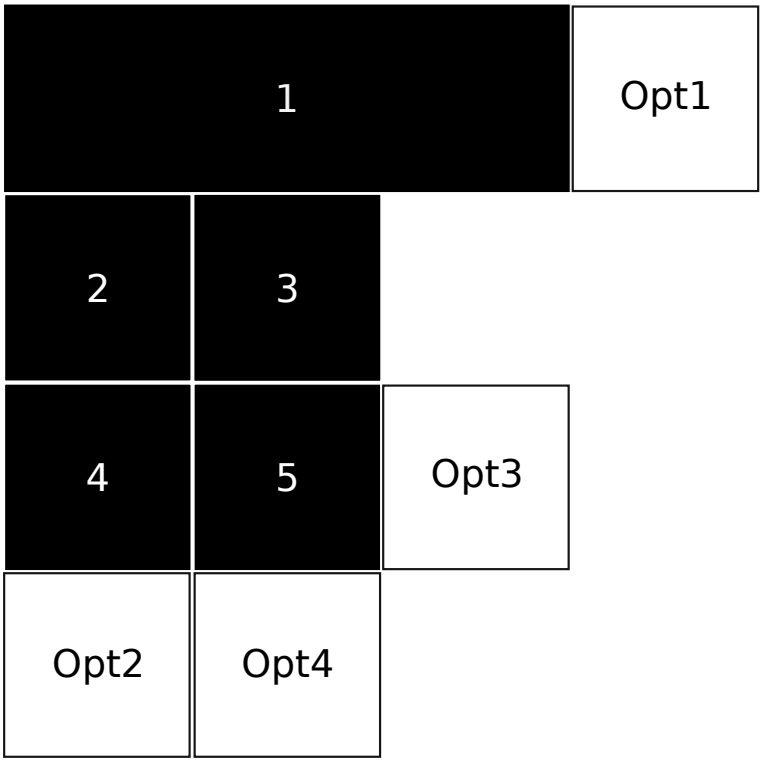
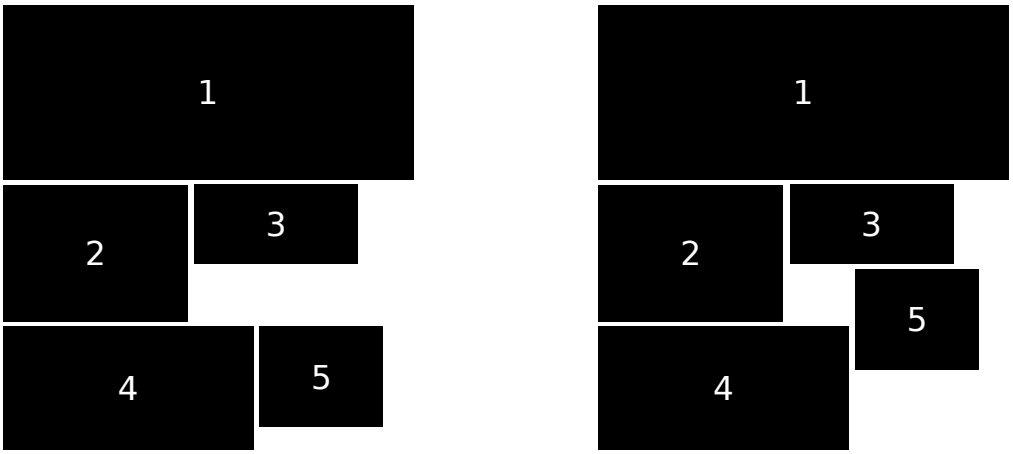


Figure 3.5. A packing where shifting the rectangle placed at candidate position 3 or 4 is important to avoid whitespace. In this example, the variable $R_{lastPlaced}$ holds rectangle 5.

order definition forbids placing other subsequent rectangles there.

For the last case, consider Figure 3.4. As opposed to the other candidate positions, the positions 3 and 4 do not increase the dimensions of the drawing and thus, depending on the target metric, are the two best positions. Neither can be be considered to be the better candidate position as both produce equal values for the metrics. As a solution to this problem, the area composed by the most recently placed rectangle and the rectangle to be placed in either position is considered. The area is calculated by $(w_i + w_j) * \max\{h_i, h_j\}$ for position 3 and $\max\{w_i, w_j\} * (h_i + h_j)$ for position 4 with i being the most recently placed rectangle and j being the rectangle to be placed next. The algorithm will choose the candidate position with the smaller calculated area. However, it is clear that the simple example in Figure 3.4 cannot be solved with this approach as both positions in combination with $R_{lasstPlaced}$ produce the same calculated area. Thus, if the area calculation still does not provide a distinct solution, position 3 is taken since it rather preserves the natural reading order from left to right.

Lastly, the algorithm is extended by shifting rectangles placed at candidate positions 3 and 4. Consider Figure 3.5 where rectangle 5 is the most recently placed rectangle. When a rectangle is placed at the candidate position 3, unused whitespace is present above the rectangle. It is the same for candidate position 4, where unused whitespace is present to its left. This issue can be solved by shifting the rectangle upwards when considering candidate



(a) Shifting rectangles disallowed. **(b)** Shifting rectangles allowed.

Figure 3.6. Two differrent packings produced by the Down-Right Heuristic. (b) has a whitespace opening that cannot be closed by resizing the rectangles, whereas the opening in (a) can be closed by enlarging rectangle 3.

position 3 and left for candidate position 4. The rectangle is shifted as far as the already placed rectangles, order, and drawing area allow. Therefore, the algorithm finds the closest neighbor that restricts the positioning of the rectangle and places it at the closest neighbor's bottom or right border. First, consider candidate position 3, where the x-value of the rectangle to be placed i is given by the right border of $R_{lastPlaced}$. A vertical order constraint between i and an already placed rectangle j exists, if $x_i < x_j + w_j$. The algorithm checks every placed rectangle and finds the closest upper neighbor with a vertical order constraint. In Figure 3.5, the closest upper neighbor is rectangle 1. This neighbor's bottom border provides the y-value for the rectangle to be placed. Similarly, candidate position 4 searches for the closest horizontal neighbor with the restriction $y_i < y_j + h_j$, whose right border provides the x-coordinate for the rectangle to be placed. In Figure 3.5, there is no closest left neighbor and the rectangle is shifted to the border of the drawing area. It is possible for the shift to affect the drawing's dimensions. Thus, the shift has to be calculated and considered for the metrics before comparing the candidate position with others. The shift will also influence further placements since the candidate positions 3 and 4 are in different locations on the drawing area after the shift.

However, the shift might cause whitespace openings that cannot be closed by expanding rectangles. An illustration of this case is given in Figure 3.6. Hence, including the shift causes a tradeoff between the size of the packing and meeting the requirement of eliminating all whitespace openings as described in Section 1.1. Therefore, a solution for the main problem of this thesis is only given by this algorithm, if the shift is disallowed.

Regarding time complexity, the simple implementation of the Down-Right Heuristic is easily in $\mathcal{O}(n^2)$, although more advanced implementations might improve on that bound. The rectangles are placed sequentially. For each rectangle, four candidate positions are assessed

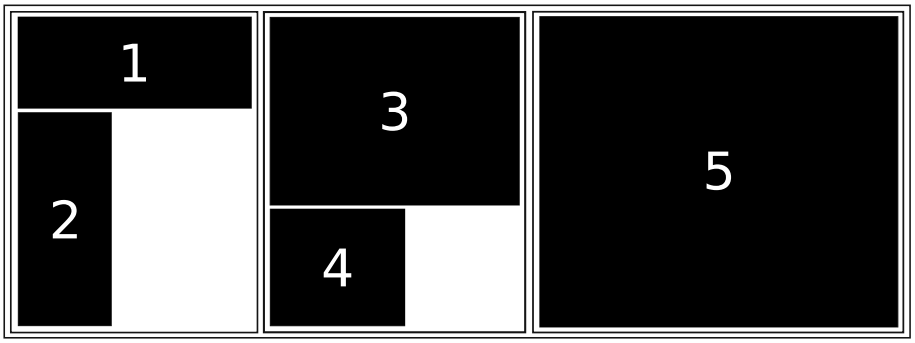


Figure 3.7. An illustration of rows and stacks. One or multiple rectangles form a stack and multiple stacks form a row.

and calculated, two of which are in constant time. The other two, however, search through all previously placed rectangles for vertical and horizontal constraints. For example, placing a third rectangle will cause the algorithm to search two other rectangles in the drawing area to assess their position and determine the third rectangle's possible position through the order restriction. Hence, the algorithm has to do $1 + 2 + \cdots + (n-1) + n = \sum_k^n k$ operations. The latter equals $\frac{n(n-1)}{2}$ according to the triangle number, thus yielding a time complexity of $\mathcal{O}(n^2)$.

3.2.2 Row-Filling and Compaction Heuristic

The Row-Filling and Compaction Heuristic focuses on aligning the rectangles in order to achieve an ordered outcome. This heuristic depends on approximated drawing dimensions used as bounds. Therefore, the Down-Right Heuristic provides drawing dimensions that can be achieved for the given rectangles and already approximate the desired outcome depending on the chosen packing strategy. The calculated drawing dimensions can also be referred to as *approximated bounding box*.

The basic design of this algorithm is to repeatedly execute its two phases as long as improvements are made. However, a basic initialization phase is preceding. The two phases also provide the name of the algorithm, as the phases are called *compaction* and *row-filling*.

First of all, two data types are introduced that are used to order and access rectangles on the drawing area: *stacks* and *rows*. They are essential for understanding the phases of this algorithm. A simple illustration of rows and stacks is given in Figure 3.7. Multiple rectangles form a stack, whereas rows are made up of stacks. In this example, there are three stacks. One consists of the rectangles 1 and 2, another consists of the rectangles 3 and 4, and the last one consists of the single rectangle 5. These three stacks are assigned to the same row. Let the elements both types are made of be called *children*. Both types have at least one child. Therefore, commonalities between stacks and rows are width and height given by their children, as well as coordinates indicating their top-left corner. However, a row's x-coordinate is always equal to zero, as anything else would lead to empty unused space. Consequentially,

the first stack of a row shares the x-coordinate of zero. Moreover, the children of a row all share the same y-value as the stacks are aligned horizontally, whereas all rectangles in a stack share the same x-value as they are aligned vertically. The dimensions of a row are given by its children's maximum height and sum of widths. In contrast, a stack's dimensions are given by its children's maximum width and sum of heights. Regarding their children, rows are ordered from left to right, whereas stacks are ordered downwards.

Algorithm 1: Basic structure of the Row-Filling and Compaction Heuristic.

Data: Rectangles, desired aspect ratio, approximated bounding box.
Result: Coordinates for every rectangle in a tight packing.
1 Initial packing
2 **while** *packing was improved* **do**
3 Compaction by filling stacks with subsequent rectangles
4 Row-filling with stacks
5 Row-Filling with single rectangles

Algorithm 1 illustrates the basic structure of the Row-Filling and Compaction Heuristic. First, an initial drawing is computed, which is then improved in the compaction and row-filling phase. The former compacts each row by filling each stack with rectangles from the subsequent stack as long as the row's height permits. The row-filling phase then fills each row with stacks and rectangles from the subsequent row as long as the approximated bounding width is not exceeded.

Before sequentially improving the packing, the algorithm produces a basic drawing. This is done in an initialization phase. This phase can be found in line 1 of Algorithm 1. In this phase, the first rectangle is placed in a stack in the top-left corner of the drawing area. Said stack is assigned to the first row with the y-coordinate of 0. Next, another rectangle is placed in a new stack and assigned to the row. Afterwards, when placing a rectangle in this row, the algorithm tests whether the rectangle can be placed beneath the most recently placed stack without increasing the row's height. If so, the rectangle is placed in said stack. Otherwise, the rectangle is placed in a new stack if the bounding width is not exceeded. If it is, the rectangle is placed in a new stack in a newly created row. Said row's y-coordinate is equal the previous row's bottom border. This way, all rectangles are placed on the drawing area and assigned to stacks and rows. Since a high rectangle might have been placed later in a row after multiple rectangles have already been placed there, the initial drawing can be improved by the following two phases.

Next, the compaction phase starts examining each row separately. In Algorithm 1, this phase can be found in line 3. Since the height of a row is not necessarily equal to the height of each stack, it is possible for rectangles to fit underneath a previous stack. The stack that yields a rectangle is called a *yielding stack*, whereas the stack that takes a rectangle is called the *previous stack*. Starting at the first stack of a row, this phase tests if the first rectangle of the yielding stack can be fit underneath said previous stack bounded by the row's height. If so,

(a) Before. **(b)** After.

Figure 3.8. Drawings before and after a simple compaction. Since the sum of the heights of stack 1 and 2 is less or equal to the maximum height, rectangle 2 switches from stack 2 to stack 1. Stack 3 is shifted accordingly.

the respective rectangle will be assigned to the previous stack. This test of fitting a rectangle underneath the previous stack is done for each pair of stacks located side by side. In case a rectangle is assigned to a new stack, the yielding stack and every following stack might have to be shifted in their x-coordinate. Moreover, all rectangles in the yielding stack have to be shifted upwards in their y-coordinate. Other rows are not affected by a compaction since the height of a row is initially fixed by the biggest rectangle in that row and cannot be changed by compactions. Since the highest rectangle's height is equal to the row's height, said highest rectangle cannot be assigned to any other previous stack. A simple compaction is illustrated in Figure 3.8. That example also illustrates a shift of a following stack.

When a yielding stack contains more than one rectangle, the shift of the yielding stack's remaining rectangles is trivial: their y-coordinate has to be decreased by the moving rectangle's height. Besides vertical shifts, there are horizontal shifts to distinguish depending on the composition of the previous and yielding stack. The first case is illustrated in Figure 3.8 where the yielding stack only contains the moving rectangle. In this case, there are two more cases to distinguish. On the one hand, if the moving rectangle's width is smaller than or equal to the previous stack's width, the following stacks have to be shifted by decreasing their x-coordinate by the moving rectangle's width. On the other hand, if the moving rectangle's width is greater than the previous stack's width, the x-coordinates of the stacks following after the yielding stack have to be shifted to the right by $w_{movingRect} - w_{previousStack}$, which describes the increase of width of the previous stack, and to the left by $w_{movingRect}$, which describes the decrease of width of the yielding stack. Hence, the shift can be calculated by $(w_{movingRect} - w_{previousStack}) - w_{movingRect}$, which can be simplified to $-w_{previousStack}$. In the second case, the yielding stack contains more rectangles beside the moving rectangle. Hence, two types of stacks are affected: the yielding stack and all its following stacks. The yielding stack is affected by the change of width of the previous stack, whereas the following stacks are affected by the previous and yielding stack. There are three relevant cases to distinguish that are listed in Table 3.2. The table also demonstrates by what values the yielding and the following stacks have to be shifted by.

Table 3.2. This table demonstrates the shifts that have to be done to the yielding and following stacks, if the yielding stack contains more rectangles besides the moving rectangle r. There are three relevant cases. The width of the previous stack p and yielding stack y are given without the moving rectangle.

w_r compares to		Distance to shift	
w_y	w_p	yielding stack	following stacks
greater than	greater than	$w_r - w_p$	$w_y - w_r + w_r - w_p$
less equal	greater than	$w_r - w_p$	$w_r - w_p$
greater than	less equal	0	$w_y - w_r$

Lastly, the row-filling phase will be introduced. In Algorithm 1, this phase can be found in the lines 4 and 5. Its goal is to fill the rows to minimize the difference between each row's width and the approximated bounding width by taking rectangles or stacks from a subsequent row. This phase can be split into two tasks. The first is to check whether the first stack of the following row fits onto the end of the row and the second is to check whether the top rectangle of the first stack of the following row can be fitted underneath the last stack of the current row. Naturally, both tasks are linked to a number of shifts.

The task of moving a whole stack to a previous row is the only time this algorithm allows the row's height to be increased. This has the simple reason that increasing the row's width to minimize the distance to the approximated bounding width is paramount to meet the approximated dimensions and eliminate unused empty space since other rows' widths might be very close to the approximated bound. A simple illustration of filling a row with stacks from a following row is given in Figure 3.9. The increase of height does not bring negative effects either since another execution of the compaction phase is guaranteed and it fills the stacks as much as the updated height of the row allows. When a moving stack is assigned to a new row, multiple shifts are needed. All stacks and their respective children in the row that yields the moving stack have to be shifted to the left according to the width of the moving stack. The yielding row and its following rows are affected by the reallocation of the moving stack. The yielding row is only affected if the moving stack's height is greater than the taking row's previous height. In this case, the yielding row and every following row have to be shifted downwards by the difference between the moving stack's height and the height of the previous row excluding the moving stack. Every following row after the yielding row is also affected by the change of height of the yielding row. If the moving stack was the highest stack in the yielding row, the following rows are shifted upwards by the difference between the moving stack's height and the yielding row's new height.

The second and last task of the row-filling phase considers the first rectangle of the first stack of the yielding row. Said first rectangle is examined to fit underneath the last stack of the row that is to be filled. The rectangle changes stacks, if its dimensions added to the designated stack do not exceed the row's height and approximated width bound. Since the previous row's height does not increase, the yielding row does not have to be shifted vertically. If the

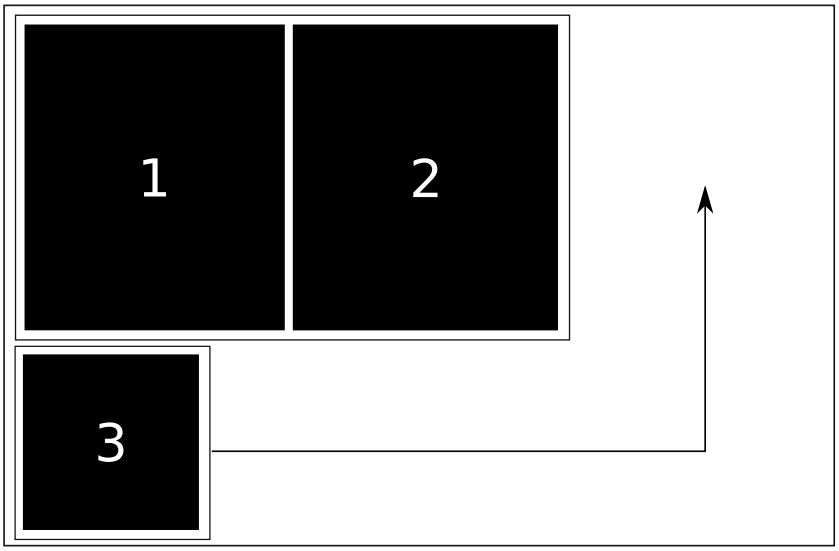


Figure 3.9. A simple illustration of the row-filling phase. The approximated bound given by the Down-Right Heuristic has a greater width than the first row, which is made up of the stacks 1 and 2. Taking stack 3 from the second row enlarges the first row in a way that does not break the approximated width.

yielding stack only contains the moving rectangle, the yielding row's elements have to be shifted left by the width of the moving rectangle. If the yielding stack contains more than the moving rectangle and the moving rectangle was the widest rectangle in its respective stack, the other stacks in the yielding row have to be shifted. Again, the elements are shifted by the difference between the moving rectangle's width and the yielding stack's width. Shifting all following rows is similar to the previously introduced shifts too. If the yielding stack was the biggest stack in the row and loses this status after the reallocation of the moving rectangle, the following rows have to be shifted upwards by the difference between the yielding stack's height including the moving rectangle minus the new height of the yielding row.

If both phases did not make any improvements to the drawing, the algorithm terminates. If either phase made an improvement, another iteration of the phases is executed as there could be other improvements to be made.

Regarding time complexity, the Row-Filling and Compaction Heuristic is in $\mathcal{O}(x \cdot (n^2))$, where n is the number of rectangles and x is the number of times the two main phases are executed with $x < n$. In order to determine the complexity, the two phases are viewed separately. The compaction phase has a worst-case complexity of $\mathcal{O}(n^2)$. To illustrate that this is a tight bound, let no stack have more than one rectangle in it and let the height of the last rectangle be bigger than the sum of heights of all previous rectangles. In the worst case, all rectangles are in the same row. This case is illustrated with a simple example in Figure 3.10. Now, the number of stacks decreases with every successful compaction. Overall, $n - 2$ compactions are made, because the very first and very last stack are never taken by any other stack. With every compaction, all following stacks have to be shifted. In the first

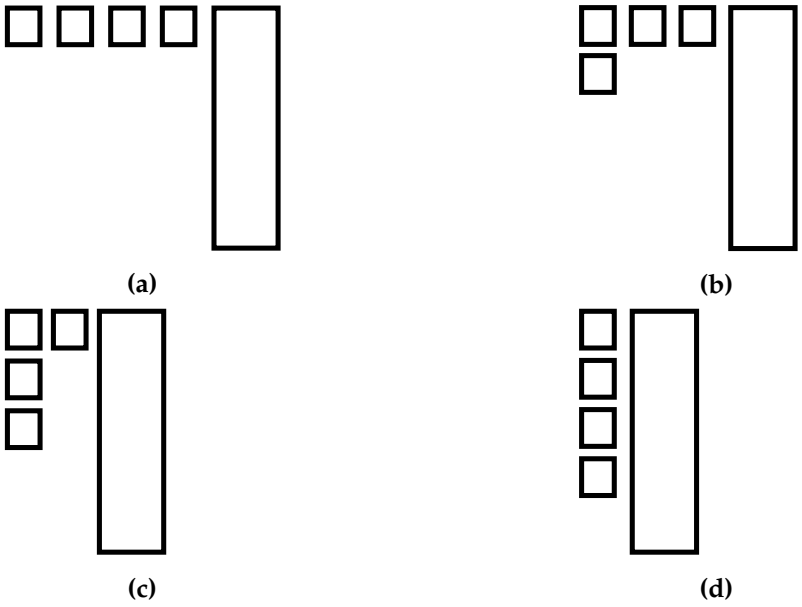


Figure 3.10. Illustration of the complexity of the compaction phase with a simple example of five rectangles assigned to unique stacks and a single row. Overall, three compactions are made. Three stacks are shifted in (b), two are shifted in (c), and one is shifted in (d).

compaction, $n-2$ stacks have to be shifted excluding the very first and second stack, then $n-3$ stacks and so on. This results in $\sum_{k}^{n-2} k \approx \sum_{k}^{n} k = \frac{n(n-1)}{2}$ operations, thus yielding a time complexity of $\mathcal{O}(n^2)$. The complexity of the row-filling phase is also $\mathcal{O}(n^2)$. For every row, rectangles are reassigned and according shifts are made, which can affect up to all rectangles in case a shift is caused by the first row. The number of rows can be the equal to the number of rectangles. This results in $\mathcal{O}(n^2)$. Consequentially, the complexity of the two phases are $\mathcal{O}(n^2 + n^2) = \mathcal{O}(n^2)$. The two phases are executed x times until no more improvements are made, which results in $\mathcal{O}(x \cdot (n^2))$. Presumably, x is smaller than n since the first compaction and row-filling phases will set the final position for rectangles in the first row. Consequentially, every following iteration will produce more final positions. Hence, x is smaller than n.

3.2.3 Heuristic for a Special Case

A case that led to the motivation for this thesis in Chapter 1 is the presence of one big rectangle that is surrounded by multiple smaller sized rectangles. Under the assumption that the smaller sized rectangles are all same-sized, a heuristic for this special case is introduced.

To find the biggest rectangle, one just needs to iterate through the ordered set of rectangles. Once the big rectangle is found, the remaining rectangles can be sorted into two lists. One list is for the rectangles that come before and the other is for the rectangles that come after the biggest rectangle in the given order. Next, it can be determined how many rows are needed to place the rectangles beside the big rectangle by dividing the big rectangle's height by the

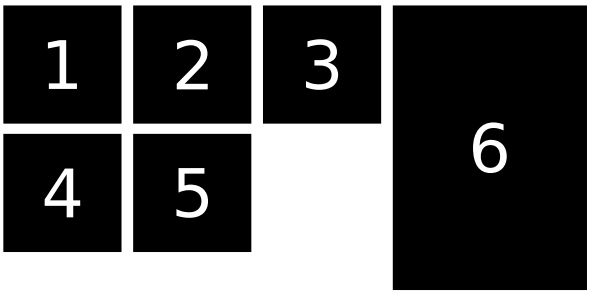


Figure 3.11. A simple illustration of the Special Case Heuristic. The height of rectangle 6 is greater than the height of a smaller rectangle multiplied by 2. Hence, the smaller rectangles can be placed in two rows beside the bigger rectangle.

height of the smaller rectangles. With the number of rows and the sizes of the two lists, it can be determined how many columns are needed for each side by dividing the size of a list by the number of needed rows. The result has to be rounded up. An example is given in Figure 3.11. To place 5 rectangles in two rows, $5/2 = 2.5$ columns are needed according to a simple division. However, since rectangles cannot be split, an integer amount of columns is needed. Since 2 rows and 2 columns won't fit 5 rectangles, the results is rounded up to 3 columns.

The algorithm starts placing the rectangles of the first list at $x = 0$ and $y = 0$ and sequentially fills the calculated number of columns and rows with the given rectangles. Next, the big rectangle is placed, which is followed by the second list which starts placing at $y = 0$ and $x = w_{smallRect} * columns + w_{bigRect}$.

3.3 Minimization Problem

As an alternative to approximate a solution heuristically, this section introduces an approach formulated as a linear program. As illustrated in Figure A.5 in the appendix, the linear program might produce results with whitespace openings that cannot be eradicated by enlarging rectangles. Hence, the implementation of this minimization problem only provides almost optimal values for the target metrics to compare the heuristics presented in Section 3.2 to.

As we recall from Section 1.1, the values of w_A, h_A, w_i^{min}, and h_i^{min} for the drawing area A and every rectangle $i \in R$ as well as the order function Π are given. The values w_B and h_B for a drawing B are given by the rectangles' coordinates and dimensions. The max scale measure provides the objective function. Alternatively, a function minimizing the drawing area may be used. Next, variables are introduced that allow us to shorten the equations needed. Let $y_i^{bo} = y_i + h_i^{min}$ be the y-value of the bottom border of a rectangle i. Likewise, let $x_i^{ri} = y_i + h_i^{min}$ denote the x-value of the right border of i. The first equation that the objective function is subject to avoids overlaps and is taken from the work of Wu et al. [WTZ+17]. Their inequation needed to be reworked since they use a coordinate system that spans vertically

up, whereas this thesis' problem definition specifies a coordinate system that spans vertically down. The second equation implies the following of the order given by Equation 3.1.2. Lastly follow the non-negativity constraints for the decision variables, which are the coordinates of the rectangles. The problem formulated as a linear program is as follows:

$$\begin{array}{ll} \text{maximize} & \min\{w_A/w_B, h_A/h_B\} \\ \text{subject to} & \max\{x_i - x_j^{ri}, x_j - xi^{ri}, y_j - y_i^{bo}, y_i - y_j^{bo}\} \geqslant 0, \forall i,j \in R \\ & \max\{x_j - (x_i + w_i), y_j - (y_i + h_i)\} \geqslant 0 \qquad \forall i,j \in R : \Pi(i) < \Pi(j) \\ & x_i, y_i \geqslant 0 \qquad \forall i \in R \end{array} \tag{3.3.1}$$

3.4 Binary Search

Similar to the minimization problem in Section 3.3, binary search produces results with whitespace openings that cannot be eradicated by enlarging rectangles. This is illustrated in Figure A.6 in the appendix. Hence, it also provides almost optimal values for the target metrics to compare the heuristics presented in Section 3.2 to.

Binary search examines an ordered solution space by dividing it in half and testing whether a solution lies in the lower half of the solution space. Hence, the drawing dimensions are used as the solution space. Newly calculated dimensions are used as bounds in a reformulated linear program, which determines whether a drawing for the given dimensions can be found. The reformulated linear program lacks a target function as its purpose it to determine whether a drawing with the given dimensions can be found at all.

The formulation for the linear program is similar to the one introduced in Section 3.3. Changes affect the target function and two new restrictions regarding the width and height of the drawing. Let the goal width and height be given by w_g and h_g. The new formulation of the linear program is then given as follows:

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & \max\{x_i - x_j^{ri}, x_j - xi^{ri}, y_j - y_i^{bo}, y_i - y_j^{bo}\} \geqslant 0, \forall i,j \in R \\ & \max\{x_j - (x_i + w_i), y_j - (y_i + h_i)\} \geqslant 0 \qquad \forall i,j \in R : \Pi(i) < \Pi(j) \\ & w_B \leqslant w_g \\ & h_B \leqslant h_g \\ & x_i, y_i \geqslant 0 \qquad \forall i \in R \end{array} \tag{3.4.1}$$

First, the starting dimensions are calculated by calculating the total width and height of all rectangles. Next, as illustrated in Figure 3.12, a rectangle meeting the desired aspect ratio with both dimensions greater than the total width and height is created. The big rectangle is downsized while maintaining the desired aspect ratio until a dimension bound given by the total width or height is met. The dimensions of the downsized rectangle then provide the initial upper bounds. Hence, the initial bounds meet the desired aspect ratio and provide a solution in any case, for example by placing all rectangles beside or below each other. Next,

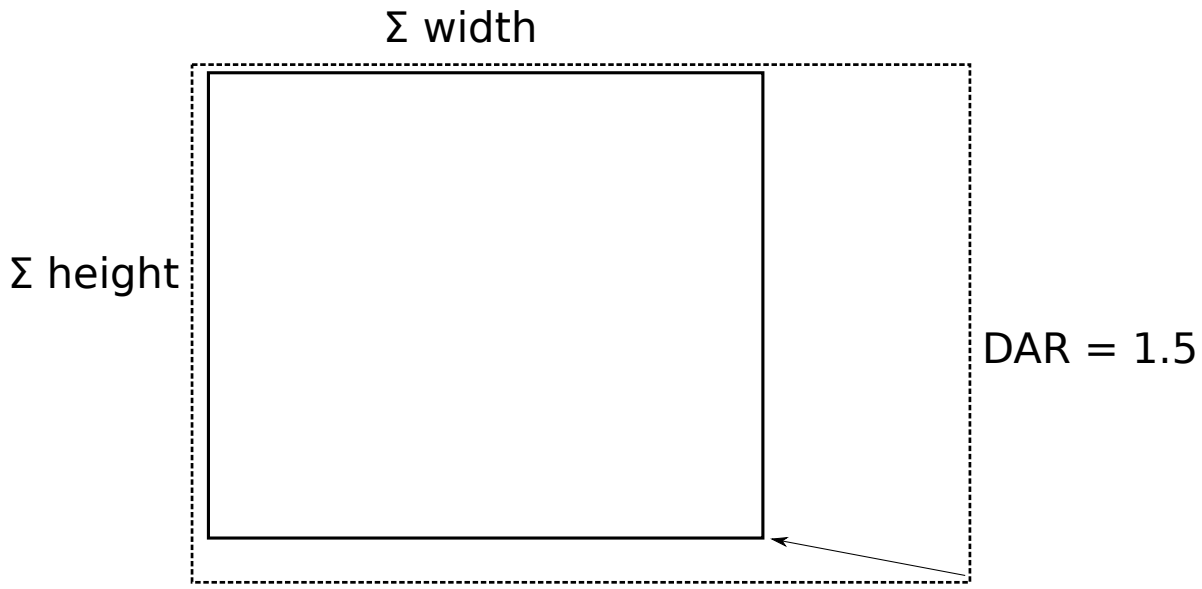


Figure 3.12. An illustration of how initial bounds for binary search are found.

the upper bounds are divided in half. These values are now the goal values w_g and h_g to be undercut. The lower bounds are initially zero. A linear program searches for a packing with dimensions smaller or equal to the goal bounds. If a solution is found, the goal values become the upper bounds. Else, the goal values become the lower bounds, as it is clear that no solution with smaller dimensions can be found. In either case, the middles between the upper and lower bounds then become the new goal values. This way, the goal values to be undercut will always preserve the desired aspect ratio.

These steps are repeated until upper and lower bounds are close enough together to not produce any other significant solution. The most recently found packing is then the best solution for the respective desired aspect ratio.

Chapter 4

Implementation

This chapter demonstrates how the presented algorithms were implemented in the context of ELK and then used by KIELER to solve the layout problem described in Chapter 1.

4.1 ELK Integration

This section describes the implementation and integration of the algorithms into ELK and making them viable layout algorithms that can be used and configured. First, the implementation of the calculation of the scale measure will be introduced. Next, the basic infrastructure of the implementation will be discussed. Finally, this section introduces the configurable options the implementation provides.

The scale measure, which was introduced in Chapter 1, is defined as $\min\{w_A/w_B, h_A/h_B\}$ with w_A, h_A, w_B, and h_B being the dimensions of a drawing area A and a drawing B. The KLighD synthesis does not provide actual dimensions of the available drawing area, but it provides the Desired Aspect Ratio (DAR). Since the scale measure calculates and compares the ratios of available dimensions to the drawing's dimensions, the actual dimensions of the drawing area are not needed as long as all drawings are assessed with the same values for the drawing area's dimensions. Hence, the calculation of the scale measure can be altered to $\min\{\text{DAR}/w_B, 1/h_B\}$. This altered calculation has been implemented.

4.1.1 Structure

Figure 4.1 illustrates the package structure, which is trivial as each algorithm is implemented in its own package. The package `firstiteration` implements the Down-Right Heuristic, whereas `seconditeration` implements the Row-Filling and Compaction Heuristic. An additional utility package containing object creation classes and other calculation classes is present. The algorithm packages often contain a class that offers a static method for executing the corresponding algorithm and, if possible, a class to expand the nodes if the corresponding configuration is set. Above all, the `LayoutProvider` class manages the configurations and calls methods inside the packages accordingly.

The two algorithms that are based on a linear program (see Section 3.3 and Section 3.4) are implemented with CPLEX. Since both models have the same restrictions regarding overlaps and ordering constraint, both algorithms call the same methods to include these restrictions in their CPLEX model. These restrictions are implemented in the `lp` package.

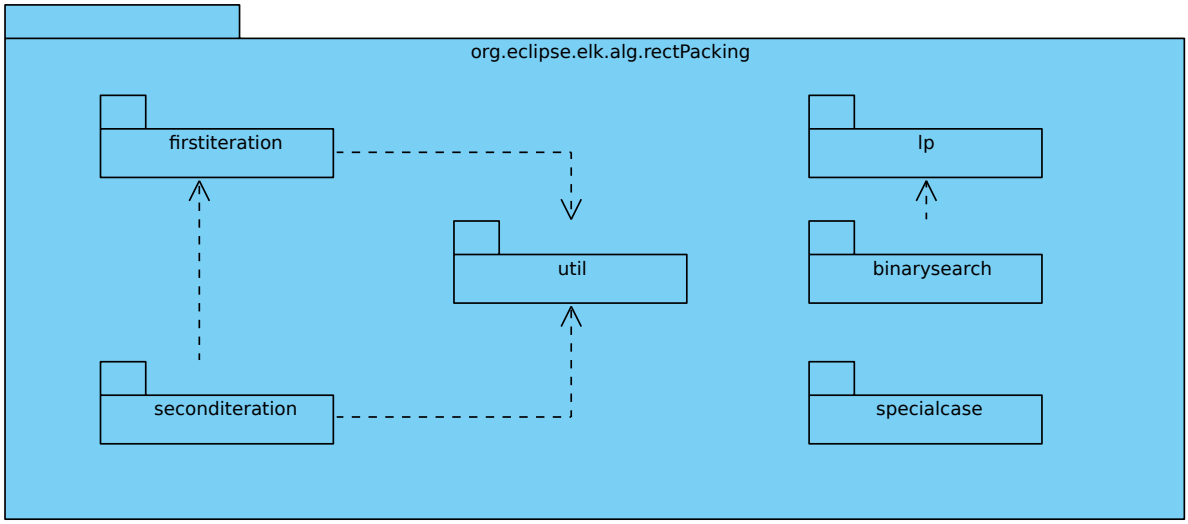


Figure 4.1. Package diagram of the implementation.

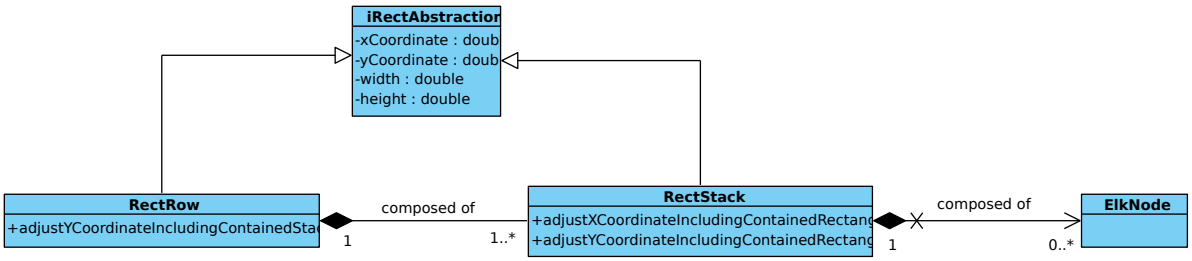


Figure 4.2. Class hierarchy of stacks and rows. The class *ElkNode* is provided by ELK.

Three important abstract classes are contained in the utility package. First, the class `DrawingData` is used to assess drawings with different placement options and save their corresponding values of the different metrics and potential coordinates for the newly placed rectangle. After finishing a single or all placements of rectangles, this class is also used to handle the metrics and return them between methods. It allows to easily store and access variables regarding a final or intermediate drawing.

The other two classes are called `RectRow` and `RectStack`. They represent the corresponding data structures rows and stacks as described in Section 3.2.2. Their hierarchy is illustrated in Figure 4.2. They contain the necessary fields such as x- and y-coordinates, width, height, and a list of their assigned children. `RectRow`'s children are `RectStacks`, whereas `RectStack`'s children are `ElkNodes`, a class provided by ELK representing nodes in a graph. Both classes also offer methods to shift their coordinates recursively. This means that calling these methods will not only adapt their own coordinates, but passes the changes to their children and children's children. Both classes also provide methods to add and remove a certain child while automatically adapting their own width and height accordingly. For example, a `RectStack` is

added to a RectRow, which adds the RectStack's width to its own width and checks whether the RectStack's height is greater than its current height. If it is, the RectRow adjusts its height accordingly. RectStack has a field that refers to its respective RectRow. This field is used to inform its parent about changes in the RectStack itself. For example, a RectStack's height increases due to a rectangle being added. Afterwards, the RectStack calls an update method on its respective RectRow. Consequently, the RectRow adjusts its dimensions accordingly.

4.1.2 Options

The implementation offers several options to influence the layout. Table 4.1 lists and describes them. The options enabling the algorithms based on linear programs lp and binarySearchLP will not end up in ELK due to their dependencies to CPLEX.

4.2 Adaption in KIELER

In the current version of KIELER, *ELK Layered* is used to layout each region's child graph and the regions in a macro state are arranged with *Box Layout*. For a detailed description of these algorithms, see Section 2.3.2.

To apply the developed algorithm, the use of Box Layout needs to be replaced with the use of the implemented algorithm as described in Section 4.1 instead. The KIELER semantics project[1] implements SCCharts. The package de.cau.cs.kieler.sscharts.ui.synthesis contains a class called StateSynthesis.xtend, which configures the default layout for regions in a state in a method called configureLayout. This method sets the layout options for the node containing the regions. Editing the following options will enable the use of the introduced layout algorithm when it is released in ELK. org.eclipse.elk.box has to be replaced with org.eclipse.elk.alg.rectPacking in the ALGORITHM option. Lastly, the SPACING_NODE_NODE option has to be removed, as it is not supported by this implementation as seen in Table 4.1.

[1] https://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/semantics/browse

Table 4.1. Description of implemented options in the ELK plug-in.

Option	Description
aspectRatio	Sets the desired aspect ratio.
padding	Sets the padding between drawing and border of the drawing area.
expandNodes	Boolean that indicates whether the nodes should be expanded after computing a layout to fill empty spaces and form a uniform border.
onlyFirstIteration	Boolean that indicates whether only the first iteration should be executed. The first iteration is the Down-Right Heuristic introduced in Section 3.2.1. Defaults to `false`. Node expansion is not possible for this algorithm.
strategy	Sets the packing strategy used by the Down-Right Heuristic in the first iteration. Defaults to the max-scale driven strategy.
lpShift	Boolean that indicates whether rectangles should be shifted as much as possible when being placed to the right or below the last placed rectangle as described in Section 3.2.1.
arCalcAbs	Boolean that indicates whether an absolute or relative calculation should be used to rank the aspect ratios of different drawings.
checkForSpecialCase	Boolean that indicates whether the algorithm should check for a special case with one big rectangle and multiple smaller same-sized ones. Defaults to false. If the special case is detected, the algorithm described in Section 3.2.3 is applied.
lp	Boolean that indicates whether a layout should be calculated by executing a linear program as described in Section 3.3. If enabled, this option excludes the execution of all other algorithms. Node expansion is not possible for this algorithm.
binarySearchLP	Boolean that indicates whether a layout should be calculated by executing a binary search algorithm as described in Section 3.4. If enabled, this option excludes the execution of all other algorithms. Node expansion is not possible for this algorithm.

Chapter 5

Evaluation

This chapter compares the proposed algorithms and the preexisting Box Layout with each other. The first section covers the quantitative evaluation of the presented algorithms. The algorithms are executed on randomly generated test instances. The resulting drawings are examined for the target metrics that were introduced in Chapter 1. This produces quantitative data that helps evaluating the algorithms. Next, the performance of the algorithms is examined, which covers their produced layouts and examines whether they meet the requirements stated in Chapter 1. This chapter then examines the execution time since fast algorithms are crucial for the intended use case. The last section evaluates the algorithms executed on a use case example taken from SCCharts from a student project at the University of Kiel.

All referenced test instances, R scripts, GrAna files, and comma seperated value files containing data are provided in the online repository of the Real-Time and Embedded Systems group at the University of Kiel[1].

5.1 Quantitative Evaluation

For this evaluation, I used GrAna to automatically analyze randomly generated test instances for the different algorithms. Every algorithm was executed with the same desired aspect ratio (1.3) and padding (none) to make the results comparable. The Down-Right Heuristic was tested with the shift of rectangles enabled and using the max-scale driven strategy. Every test instance was processed by each algorithm. An exception is the algorithm for the special case that was introduced in Section 3.2.3, which was only executed on test instances created for the specific situation. Chapter 1 introduced target metrics that can be used to assess different drawings. GrAna analyzes the test instances in terms of area, width, height, aspect ratio, and whitespace. Besides, the scale measure can be derived from the aspect ratio, drawing width, and drawing height as described in Section 4.1.

5.1.1 Experimental Objects

The test instaces were randomly generated.[2] Overall, 350 instances were created and tested. 50 out of the 350 instances were solely generated to test the special case. Said special case instances contain between 10 and 15 rectangles. The other 300 instances are partitioned into

[1] https://git.rtsys.informatik.uni-kiel.de/projects

[2] https://github.com/dalu2104/ElkT-RanGen

Table 5.1. Means and standard deviations of the aspect ratio of the introduced algorithms executed on their respective test instances with a desired aspect ratio of 1.3.

Algorithm	Mean	Standard Deviation
Box Layout	1.256	0.374
Down-Right Heuristic	1.312	0.177
Row-Filling and Compaction Heuristic	1.359	0.312
Linear program	1.293	0.029
Binary search	1.295	0.051

sets of 50, each containing instances with the same number of rectangles. The first set starts at 10 contained elements and the following sets each increase that number by 10 until 50 elements per instance are reached. In the last set, each instance contains between 50 and 100 elements. Regarding the special case, the big rectangle's dimensions are limited between 150 and 500, whereas the small rectangles' dimensions are bound by 1 and 50. The regular test instances were generated with dimension bounds of 1 and 200.

5.1.2 Experimental Results

Box Layout, Down-Right Heuristic, and Row-Filling and Compaction Heuristic are the only algorithms that were tested on all test instances. The Special Case Heuristic was only tested on the corresponding test instances.

Section 3.3 introduced a formulation of a linear program and Section 3.4 introduced a binary search that finds drawings with the help of another formulation of a linear program. However, it was not possible for these algorithms to solve large test instances in realistic time. I set the upper bound regarding execution time to about 30 minutes per test instance. Hence, the linear program only produced results for the test instances containing 10 elements, whereas the binary search additionally delivered results for the special case test instances containing between 10 and 15 elements.

Aspect Ratio

As we recall from Chapter 1, the aspect ratio is the ratio between the width and height of a drawing. Since the goal is to display a produced layout or packing in a given drawing area, it is important to produce drawings equal or close to the given desired aspect ratio.

Table 5.1 demonstrates the means and standard deviations of the aspect ratio for the different algorithms. Both, mean and standard deviation lead to the same observations. Linear program and binary search produce the best results with small deviations. Their results are also very similar. After a big gap regarding the standard deviation follows the Down-Right Heuristic, which tends to produce drawings that have a greater aspect ratio than the desired aspect ratio. Another big gap in standard deviation is found before the Row-Filling and Compaction Heuristic and the Box Layout. When comparing these two, the Row-Filling and

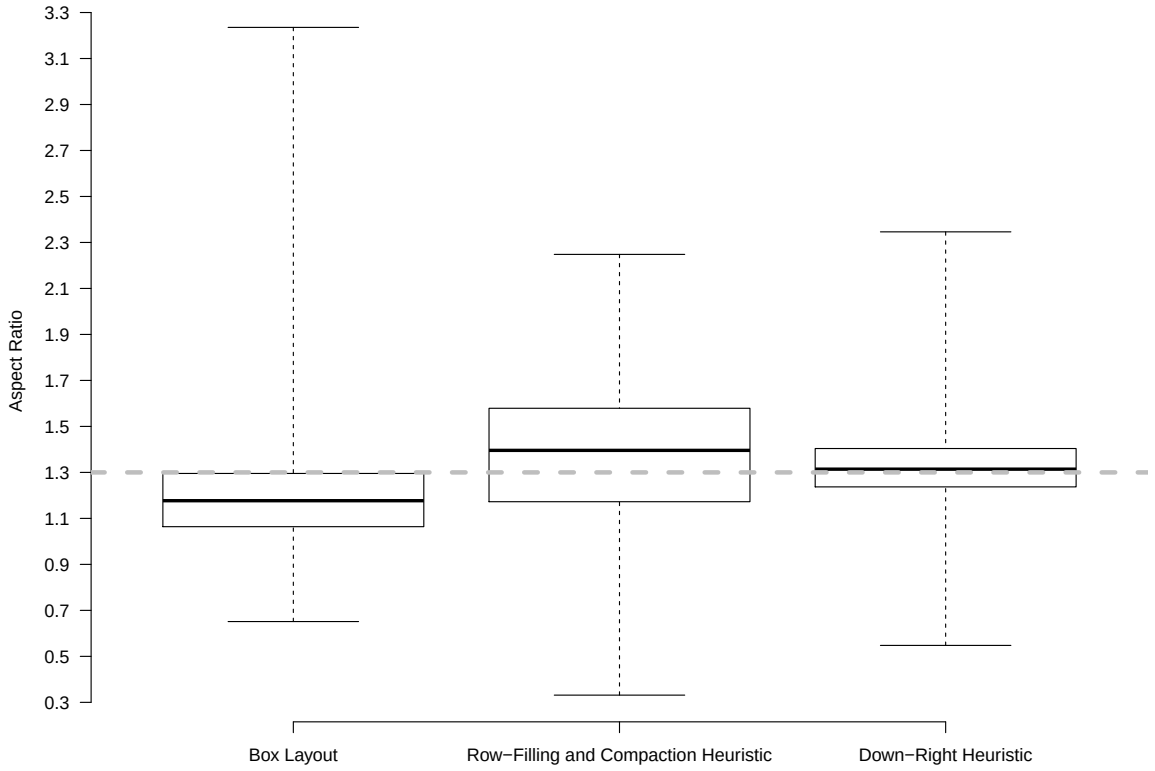


Figure 5.1. Box plot of the aspect ratio of the algorithms Box Layout, Row-Filling and Compaction Heuristic, and Down-Right Heuristic executed on all test instances. The dotted horizontal line indicates the desired aspect ratio of 1.3.

Compaction Heuristic produces results with less deviation. Moreover, Box Layout tends to undercut the desired aspect ratio, whereas the Row-Filling and Compaction Heuristic tends to surpass it.

Next, a box plot is illustrated. In general, box plots illustrate the distribution of data through their quartiles. The box in the center of a box plot is the core of the distribution, the outer bounds are the upper and lower quartile. That means that 50 percent of the values can be found inside the box. 25 percent of the values are smaller than the lower quartile, whereas 25 percent are bigger than the upper quartile. There is another line in the middle of the box representing the median. Box plots also include so called *whiskers* that are represented by lines extending orthogonally from the box. These whiskers illustrate variability outside the core distribution. The lower whisker is the lowest value of the given data set whereas the upper whisker is the highest value.

Figure 5.1 illustrates the distribution of produced aspect ratio values over the 350 test instances for the algorithms Box Layout, Row-Filling and Compaction Heuristic, and Down-Right Heuristic. The illustration provides new insights about how the means and standard

deviations presented in Table 5.1 came to be. The Box Layout produces drawings with an aspect ratio slightly below the desired aspect ratio, but it has greater deviations as the upper and lower whiskers are spread widely. The aspect ratio of the Row-Filling and Compaction Heuristic tends to surpass the desired aspect ratio and it has a greater distribution in its core as its upper and lower quartile are spread more widely. The algorithm seems to produce wider and lower drawings than desired as its median is close to the aspect ratio of 1.5. The distribution of the Down-Right Heuristic shows minimal deviations close to the desired aspect ratio. Its median is almost equal to the desired aspect ratio. Also, its upper and lower whiskers shows the smallest deviation out of the three algorithms.

A box plot was presented here as it makes the results easy to compare. Alternatively, Figures A.7, A.8, and A.9 in the appendix provide histograms of the distribution of the values. These illustrations suggest the same observations as made about Figure 5.1.

Overall, the results suggest that the Down-Right Heuristic is the best performing heuristic in terms of aspect ratio. They also suggest than Box Layout performs worse than the Row-Filling and Compaction Heuristic.

Scale measure

As we recall from Section 4.1, the scale measure is calculated by $\min\{\mathrm{DAR}/w_B, 1/h_B\}$ with w_B and h_B being the dimensions of a drawing B. The scale measure indicates by what factor a drawing can be scaled in the given drawing area. Thus, maximizing the scale measure produced on a set of rectangles is beneficial for a compact drawing.

The results for the scale measure were calculated with a fixed desired aspect ratio of 1.3 and multiplied with the factor of a hundred to make the values easier to observe and process in the presentation of the data. For example, let drawing B have dimensions of $w_B = 300$ and $h_B = 200$. The corresponding scale measure is $s = \min\{1.3/300, 1/200\} = \min\{0.00433, 0.005\} = 0.00433$. If this result is multiplied by a hundred, the result is 0.433, which is easier to read and compare as there are fewer unnecessary zeros.

The scale measures of two drawings of different sets of rectangles cannot be compared – an observation paramount for the illustration of the scale measure data. For example, let the drawings D and E illustrate two different sets of rectangles and let them have dimensions of $w_D = 1000, h_D = 1000, w_E = 10$, and $h_E = 10$. The scale measures multiplied by the factor 100 for D and E are $s_D = \min\{0.1, 0.0769\} = 0.0769$ and $s_E = \min\{10, 7.692\} = 7.692$. The results are similar due to the dimensions being multiples of one another. When comparing these two scale measures, drawing E seems to be superior. However, drawing E illustrates a different set of rectangles. This example shows that two scale measures of two drawings of different set of rectangles cannot be compared. Let drawing F handle the same set of rectangles as drawing E and let F have dimensions of $w_F = 8$ and $h_F = 12$, which leads to a scale measure of $s_F = \min\{12.5, 6.41\} = 6.41$. The scale measures s_E and s_F are comparable, because they assess drawings for the same set of rectangles. Said comparison yields that drawing E is superior since its scale measure is greater.

Figures 5.2 and 5.3 illustrate the scale measure data. The test instances are sorted in an

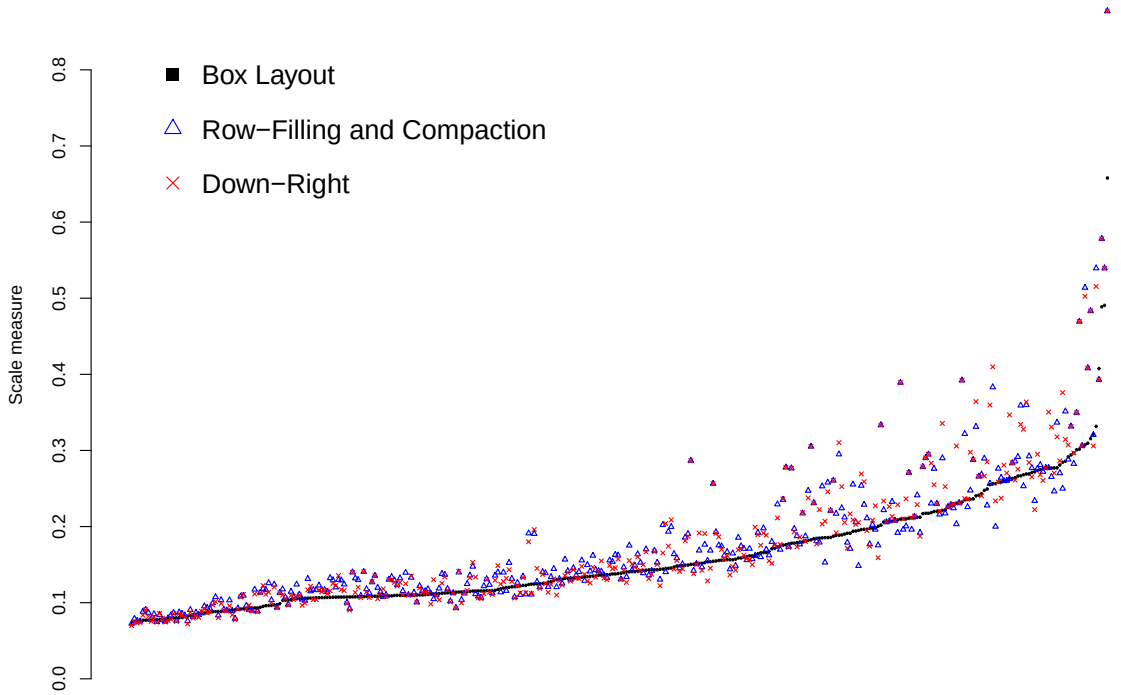


Figure 5.2. Scale measure results for algorithms that were run on all test instances. The test instances on the x-axis are sorted in ascending order by the scale measure results of Box Layout.

ascending order by the values of the scale measure of the Box Layout to make it easier to estimate the differences between the algorithms in comparison with each other.

Figure 5.2 illustrates the data for the three algorithms that were tested on all 350 test instances. The results demonstrate that the Row-Filling and Compaction Heuristic and Down-Right Heuristic produce better results than the Box Layout, as most of their scale measures are greater than the values of the Box Layout. Examining the data points, is it not clear whether the Down-Right Heuristic or the Row-Filling and Compaction Heuristic is better. The previous observations are supported by the means of the scale measure data: Box Layout yields 0.1593, Down-Right yields 0.1774, and Row-Filling and Compaction yields 0.1769. Box Layout performs the worst. The other two heuristics are close to one another with the Down-Right Heuristic being slightly better, similar to the observations made on Figure 5.2.

Finally, Figure 5.3 illustrates the scale measure data of all algorithms that were applicable to the test instances containing ten rectangles. These are the only test instances the linear program was able to produce drawings in realistic time for. The chart demonstrates that the Box Layout is the worst algorithm on these test instances as most of its values are worse than the other algorithms. Next follows the Row-Filling and Compaction Heuristic, which provides some scale measure values that are worse im comparison to Box Layout. The scale measures of the Down-Right Heuristic are mostly better than Box Layout, only a few values are worse.

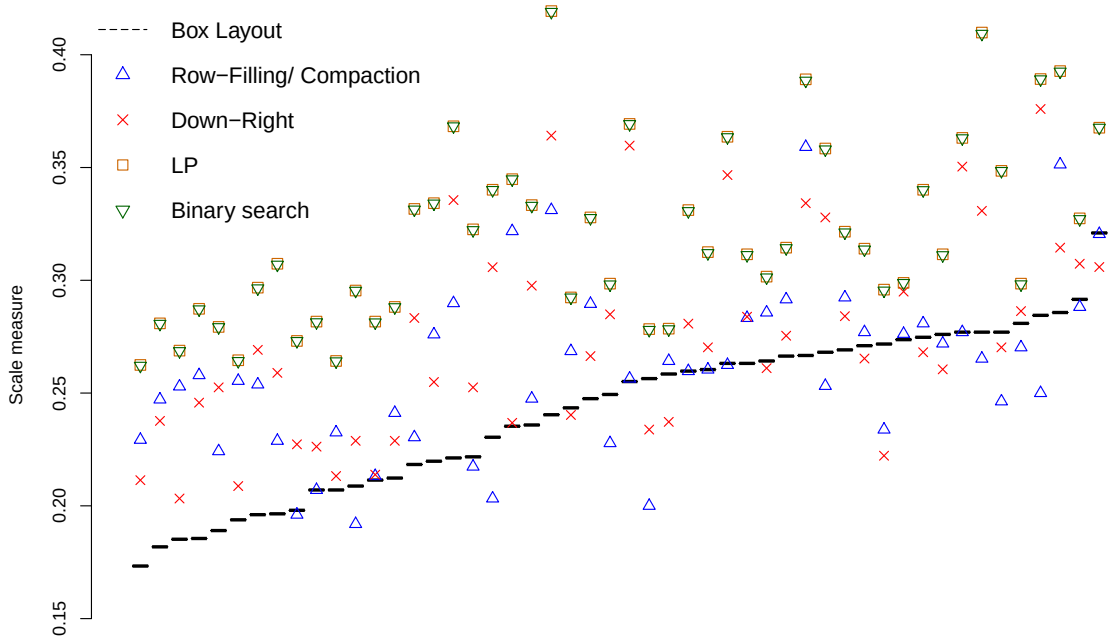


Figure 5.3. Scale measure results for algorithms that were run on the test instances containing 10 rectangles. The test instances on the x-axis are sorted in ascending order by the scale measure results of Box Layout.

Finally, binary search and linear program are almost entirely better than the Row-Filling and Compaction Heuristic and Down-Right Heuristic. Neither binary search nor linear program appears to be better as both produce equal results.

Overall, the results suggest that the Row-Filling and Compaction Heuristic and Down-Right Heuristic are equally useful heuristics in terms of scale measure.

Whitespace

The difference between the area of the drawing and the total area of all rectangles is called whitespace. It is used to assess the efficiency of a packing. If a drawing has a lot of whitespace, a lot of space in a drawing is unused. Hence, the drawing is deemed inefficient.

Figures 5.4 and 5.5 illustrate box plots of the produced whitespace data. The y-axes of both charts reveals the whitespace in percent of the total drawing area.

Figure 5.4 illustrates box plots of the data produced by the algorithms that were used on all test instances. The Box Layout's upper and lower quartiles lay far above the other two algorithms with its upper whisker providing the highest values of about 70% whitespace. Its lower whisker reaches out to 0% whitespace, which the other algorithms also achieve.

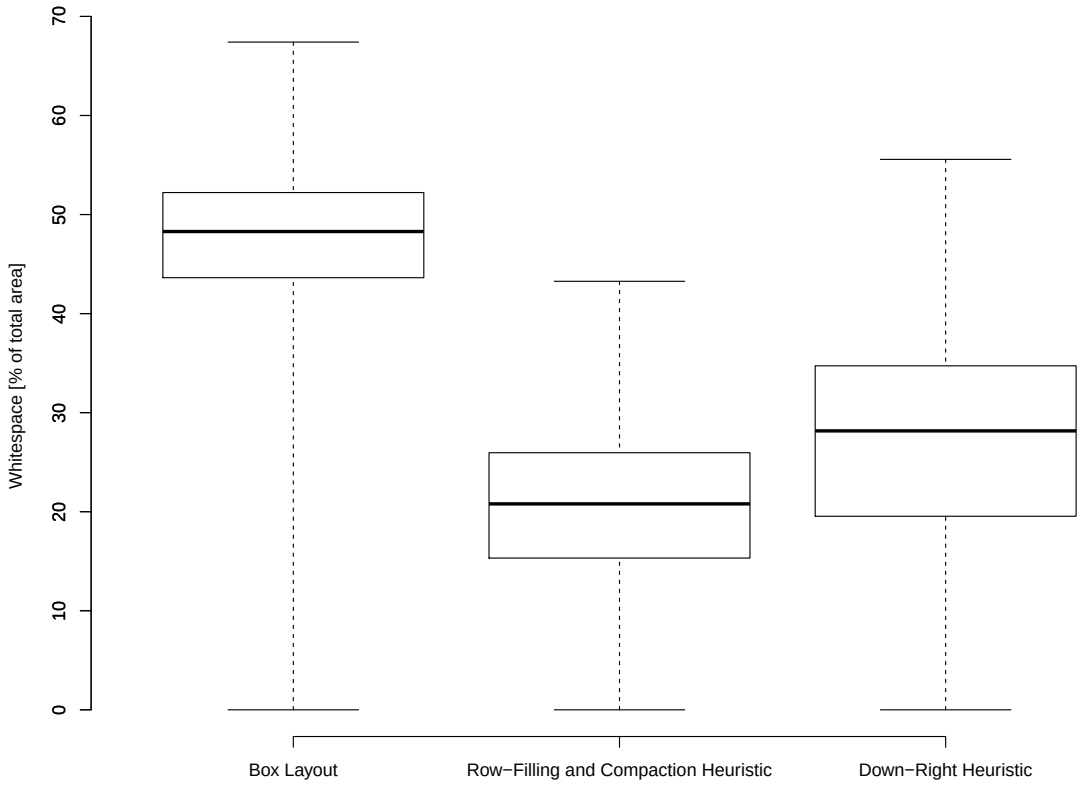


Figure 5.4. Whitespace distribution of algorithms executed on all test instances.

Its upper and lower quartiles have much bigger values than the other two algorithms. This makes the Box Layout by far the worst algorithm regarding this metric. Out of the remaining two algorithms, the Row-Filling and Compaction Heuristic proves to be the better one as all of its quartiles provide lower whitespace values than the Down-Right Heuristic's median. The core distribution given by the distance between the upper and lower whisker is greater for the Down-Right Heuristic. The means of this metric also support the previous observations as the whitespace mean of the Row-Filling and Compaction Heuristic is 19.968% of the total drawing area, whereas the Down-Right Heuristic's and Box Layout's whitespace means are 26.167% and 46.443% respectively.

Figure 5.5 illustrates the box plots of the data gained by the algorithms that were used on the test instances containing 10 rectangles. Similar to Figure 5.4, the image illustrates that Box Layout is the worst performing algorithm. The Down-Right Heuristic seems to be slightly better than the Row-Filling and Compaction Heuristic as every quartile and whisker is at lower values, excluding the median. The differences between the two algorithms are minimal, though, which is also demonstrated by their respective means of 20.332% and 21.655%. The best results were produced by the linear program and binary search, whose upper whiskers have about the same values as the medians of the better heuristics. Their medians are close to

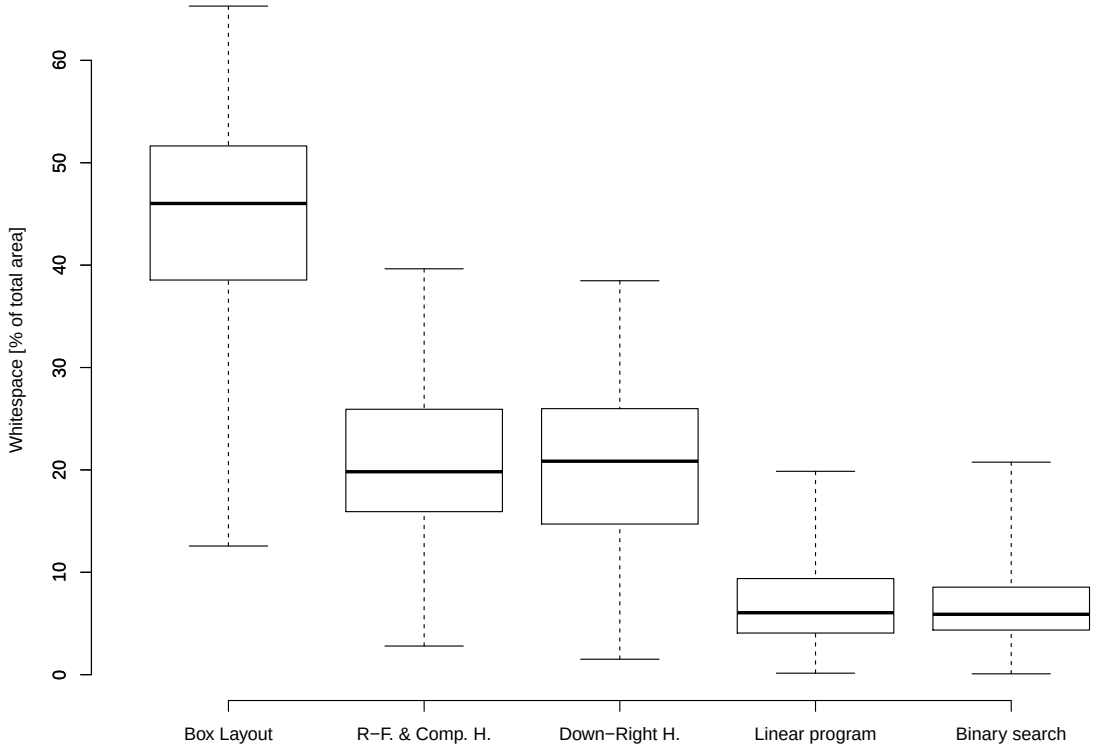


Figure 5.5. Whitespace distribution of algorithms executed on the test instances containing 10 rectangles.

one another. However, the upper quartile is higher for the linear program, ultimately making it the worse algorithm for these test instances. This also demonstrated by the means of 7.065% for the linear program and 6.787% for binary search.

Overall, the results suggest that the Row-Filling and Compaction Heuristic is the best performing heuristic in terms of whitespace.

Special Case

This section explores the presented algorithms regarding the special case, which was defined as having one big rectangle surrounded by multiple same-sized smaller ones. The corresponding test instances were laid out by the applicable algorithms: Box Layout, Row-Filling and Compaction Heuristic, Down-Right Heuristic, binary search and the algorithm for the special case. No results for the linear program were produced since its execution time surpassed a realistic length of time.

Table 5.2 demonstrates the means of the aspect ratios of the laid out drawings. Binary search produces almost optimal results regarding the desired aspect ratio. Box Layout performs the worst. Out of the remaining three, the Down-Right Heuristic produces drawings

Table 5.2. Means of the aspect ratio of algorithms executed on the special case test instances. The desired aspect ratio is 1.3.

Algorithm	Mean
Box Layout	1.843
Down-Right Heuristic	1.186
Row-Filling and Compaction Heuristic	1.177
Binary search	1.291
Special Case Heuristic	1.429

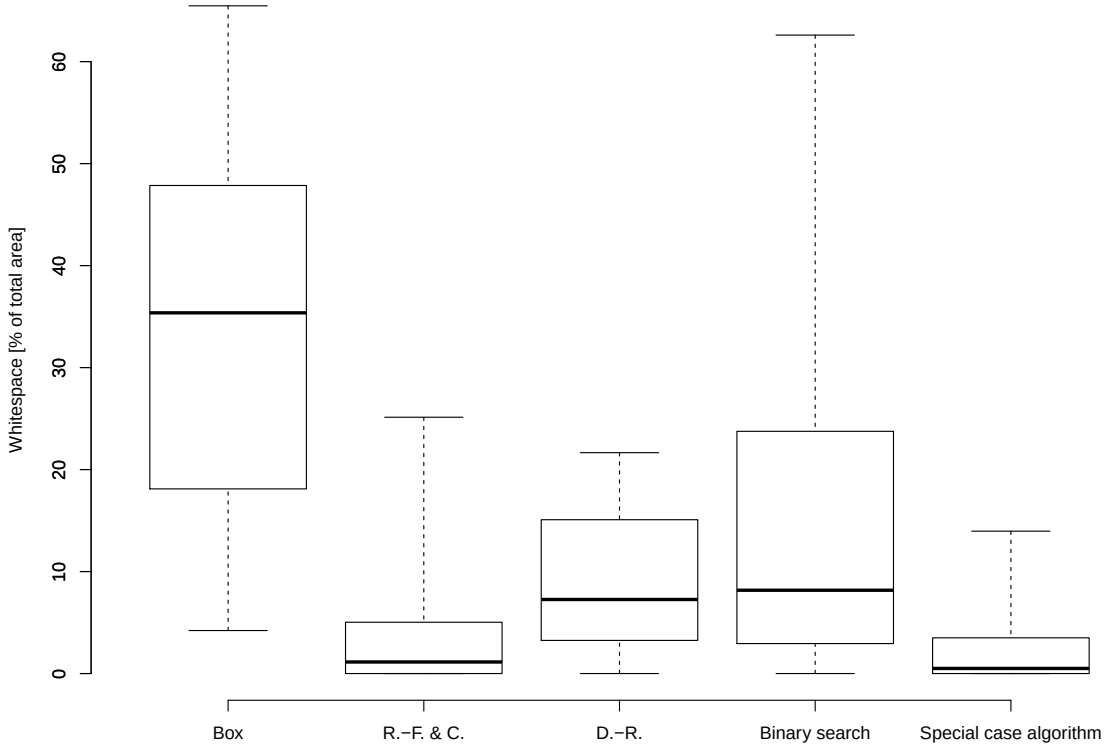


Figure 5.6. Whitespace distribution of algorithms executed on the special case test instances.

with aspect ratios closest to the desired aspect ratio, closely followed by the Row-Filling and Compaction Heuristic.

Figure 5.6 illustrates the box plots of the data gained by the algorithms that were used on the test instances simulating the special case. Similar to previous results, Box Layout produces the worst values. Close to its upper whisker is the upper whisker of binary search. The Down-Right Heuristic performs similarly to binary search regarding its median and lower quartile, but performs better in its upper whisker and quartile. However, binary search

produces better results regarding the aspect ratio, yielding a mean of 1.291 versus 1.186 produced by the Down-Right Heuristic. The best results were produced by the Row-Filling and Compaction Heuristic and the algorithm for the special case. The former proves to have a greater distribution since its upper whisker and quartile surpass the upper whisker and quartile of the special case algorithm. Hence, the special case algorithm performs slightly better.

Overall, the Special Case Heuristic and Row-Filling and Compaction Heuristic provide the best results regarding the special case.

5.1.3 Discussion

The results for the different metrics indicate which algorithm working on all test instances is superior: the Row-Filling and Compaction Heuristic. It performs equally to the Down-Right Heuristic regarding scale measure, but performs better regarding whitespace. It is only topped by the Down-Right Heuristic in terms of aspect ratio. However, an explanation for the worse performance regarding the aspect ratio might be the following: the initial packing of the Row-Filling and Compaction Heuristic is arranged according to a bounding box that has been approximated by the Down-Right Heuristic. The Down-Right Heuristic already provides a tighter packing than Box Layout according to Figure 5.4. When calculating the initial packing, the rectangles are placed in rows as long as the width of the bounding box allows. The compaction and row-filling phases cause the height of the drawing to recede. According to Figure 5.4, the rectangles are packed even tighter with the Row-Filling and Compaction Heuristic. Consequentially, the height of the drawing is smaller than the height of the approximated bounding box, which leads to a bigger aspect ratio than anticipated by the Down-Right Heuristic. Apparently, there is a trade-off between aspect ratio and whitespace for the two heuristics since the Down-Right Heuristic performs better regarding aspect ratio and the Row-Filling and Compaction Heuristic performs better regarding whitespace.

Note that the Down-Right Heuristic was evaluated with the shift of rectangles placed near the most recently placed rectangles enabled. As stated in Section 3.2.1, disabling the shift will produce more whitespace. Since Section 5.1.2 has demonstrated that the Down-Right Heuristic with shifts enabled performs worse than the Row-Filling and Compaction Heuristic, it seems reasonably safe to assume that the Down-Right Heuristic with shifts disabled performs even worse regarding the quantitative metric whitespace.

It seems remarkable that the Row-Filling and Compaction Heuristic dominates the Box Layout regarding whitespace so clearly, even though the Row-Filling and Compaction Heuristic faces more restrictions. The Box Layout does not follow the order restriction at all.

In terms of scale measure and whitespace, binary search and linear program outperform the heuristics. Unfortunately, they cannot be considered an acceptable solution for the given problem since they produce layouts with whitespace openings that cannot be eradicated as demonstrated in Sections 3.3 and 3.4 and their execution time is too long to find solutions in interactive scenarios for instances with more than 10 and 15 items per instance respectively.

Regarding the special case, binary search performs remarkably poor in terms of whitespace.

An explanation for this conspicuousness is given by the goal and design of the algorithm itself. The bounds are reduced strictly considering the desired aspect ratio. Hence, resulting drawings will most likely have an aspect ratio that is very close to the desired aspect ratio. This is also demonstrated by the algorithm's produced mean of 1.291 as mentioned in Section 5.1.2. Thus, binary search places the smaller rectangles further away from the large rectangle in order to meet the aspect ratio, which ultimately results in whitespace.

Moreover, the heuristic for the special case performs well. However, it cannot be used in the use case given by the motivation of this thesis since the smaller rectangles are not guaranteed to have the same size. Fortunately, the Row-Filling and Compaction performs just as well and can be used to handle the special case.

All things considered, the Row-Filling and Compaction Heuristic is the best performing algorithm, especially when comparing it to Box Layout. It is also viable for the special case as it does not produce a lot of whitespace and produces on average aspect ratio values that are close to the desired aspect ratio.

5.2 Performance Evaluation

An algorithm can be assessed by more than quantitative results. Chapter 1 defined requirements for the final drawing that cannot be measured quantitatively, such as the possibility to expand nodes to eliminate all whitespace openings inside the drawing. Moreover, the execution time of the algorithms is crucial, since the application as explained in Chapter 1 needs the results of an algorithm in real-time.

This section considers only those algorithms that have been found to be able to solve test instances with 20 or more contained elements: Box Layout, Down-Right Heuristic, and Row-Filling and Compaction Heuristic.

5.2.1 Drawings

If the shift of rectangles is allowed, the Down-Right Heuristic can produce abnormalities, which cause the algorithm to not produce a viable solution to the given problem. This was described in Section 3.2.1. Figure 5.7 illustrates an excerpt of a drawing produced by the Down-Right Heuristic applied to one of the test instances described in Section 5.1.1. The example shows how the algorithm can place rectangles such that they leave openings that cannot be eliminated. This issue is caused by the shift of rectangles placed near the most recently placed rectangle. The shift's intention is to eliminate whitespace. Chapter 1 lists a uniform rectangular border of the drawing with no whitespace in between the rectangles as a requirement. Thus, the Down-Right Heuristic with shifts enabled cannot be considered a solution to the given problem.

Figures A.3 and A.4 in the appendix provide illustrations of two drawings given by the Down-Right Heuristic. The first drawing was produced with the shift of rectangles enabled, whereas the later had that option disabled. It is clear that disabling the shift produces

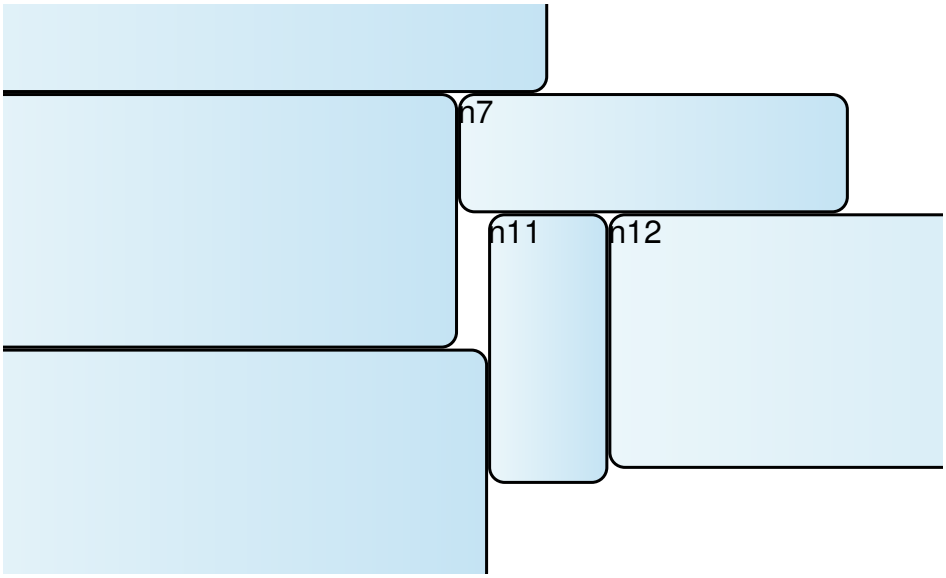


Figure 5.7. Excerpt of a layout produced by the Down-Right Heuristic. Rectangles form an opening that cannot be closed by expanding rectangles.

remarkably more whitespace. The order constraint was introduced in order to structure the packing in a way that makes it easy to follow the ordering of the rectangles in the packing as a viewer. However, it is hard to follow said order in both packings. In Figure A.4, this is due to the large whitespace areas.

A problem that might occur in drawings produced by the Row-Filling and Compaction Heuristic is illustrated in Figure 5.8. Arranging the rectangles in the last row beside each other would produce less whitespace. The design of the algorithm allows this to happen as the two phases are executed separately. The rows are compacted before they are filled with subsequent rectangles or stacks. Hence, it is possible for the highest stack of a row containing the highest rectangle to be assigned to a preceding row after compactions had already been made in the yielding row. After the row-filling phase, the highest stack of the yielding row might be a stack containing multiple rectangles as it is the case in Figure 5.8. If this occurs in any row but the last one, it is less of an issue since the row-filling phase will adequately fill the row with stacks and rectangles from the subsequent row. However, the row-filling phase never assigns stacks or rectangles to the last row as there are no subsequent rows. Hence, this problem occurs and causes unwanted whitespace. However, placing the rectangles in this example next to each other would further decrease the drawing's height, which would lead to an even higher aspect ratio.

5.2.2 Runtime

This section examines the execution time of the two remaining algorithms: Box Layout and Row-Filling and Compaction Heuristic. Both have proven to be able to solve instances with

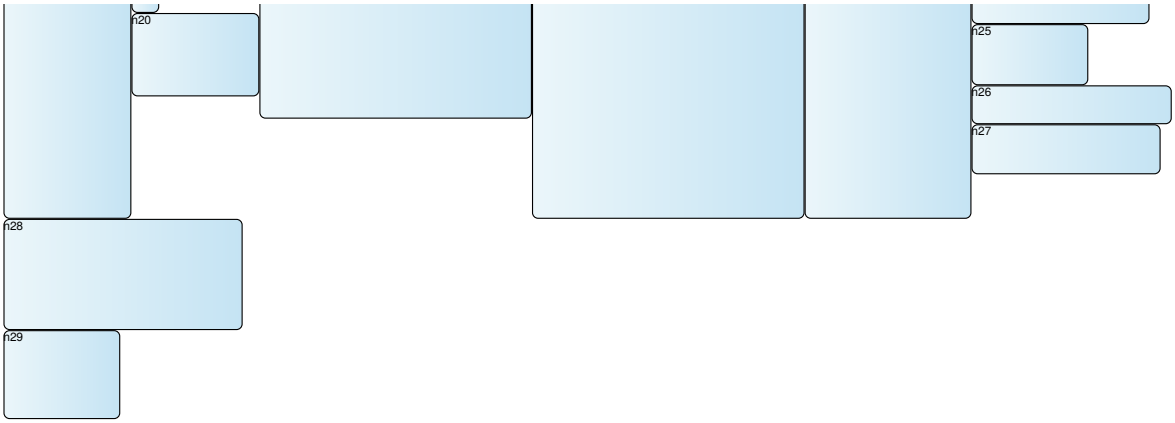


Figure 5.8. Excerpt of a layout produced by the Row-Filling and Compaction Heuristic. A compaction was made that is not wanted in the final product. Arranging the rectangles *n28* and *n29* beside each other would produce considerably less whitespace.

up to 100 elements in near real-time. Unfortunately, GrAna does not offer a comprehensive execution time analysis.

Hence, this section discusses a small sample of execution times taken from the execution of the two largest test instances that were introduced in Section 5.1.1. Both instances were tested five times for each algorithm, including the expansion of rectangles after calculating the layout. Overall, the means of the execution times are 1.02*ms* for Box Layout and 5.285*ms* for the Row-Filling and Compaction Heuristic. The small sample's purpose is to get an overview of the performance of the algorithms. It clearly demonstrates that Box Layout performs better regarding execution time. However, the Row-Filling and Compaction Heuristic performs better regarding quantitative attributes. An execution time of 0.005*s* for the biggest available test instance implies near real-time execution. The Row-Filling and Compaction Heuristic is based on the Down-Right Heuristic since it provides an approximated bounding box for the algorithm. Basically, the Row-Filling and Compaction Heuristic executes two algorithms, which ultimately leads to an increased execution time.

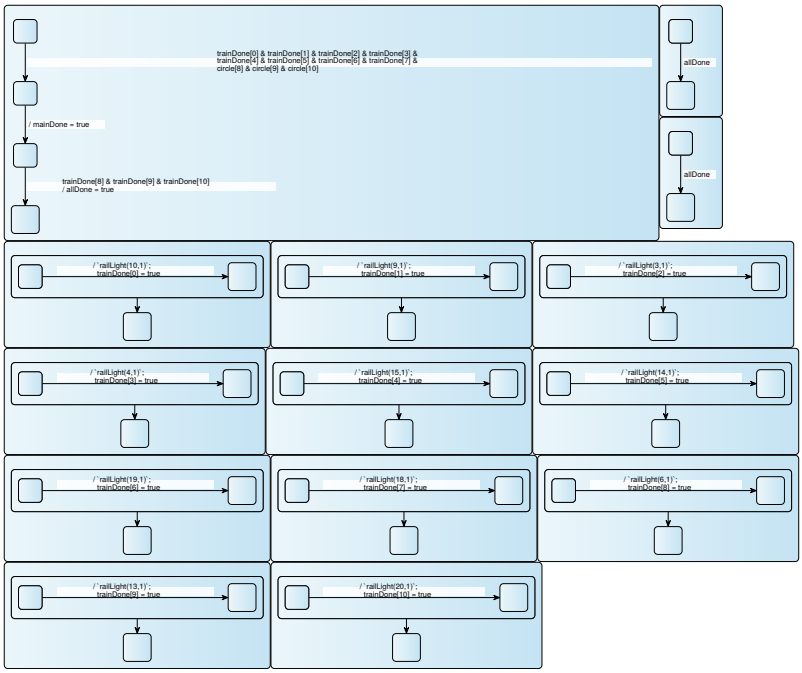
5.3 Real World Application

This section uses two instances taken from a real-life example to compare the Row-Filling and Compaction Heuristic with Box Layout. The examples are taken from a student project on the model railway from 2014, which was advised by the Real-Time and Embedded Systems group at the University of Kiel [The06]. The task was to implement a controller operating the railway model. Beside others, SCCharts were used to develop the controller. In order to test the algorithms, *.elkt* files were extracted from the SCCharts. In the tested instances, the arranged regions are displayed without their respective macro state.

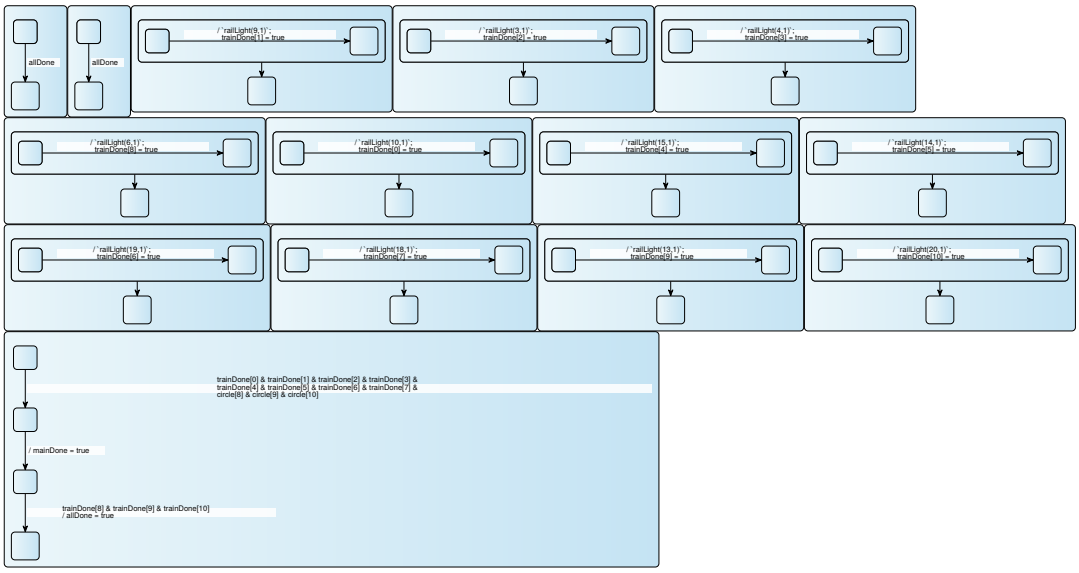
The layout of both instances was calculated with the same desired aspect ratio (1.3) and padding (none). Box Layout was executed without spacing between rectangles. The layout

and presented data are taken from ELK and GrAna.

Figure 5.9a illustrating the Row-Filling and Compaction Heuristic has an area of 1 038 065, aspect ratio of 1.198, and whitespace of 6.864%, whereas Figure 5.9b illustrating Box Layout for the same rectangles has an area of 1 184 361 , aspect ratio of 1.907, and whitespace of 20.119%. The Row-Filling and Compaction Heuristic produced better values for whitespace, aspect ratio, and area. Figure 5.10 has the same values for both layouts, as only a rectangle with big width and small height changed places according to the given order. Again, note that the Row-Filling and Compaction Heuristic observes the ordering of regions, in case of SCCharts given by their textual order in the .sctx-file, whereas the Box Layout does not observe the order.

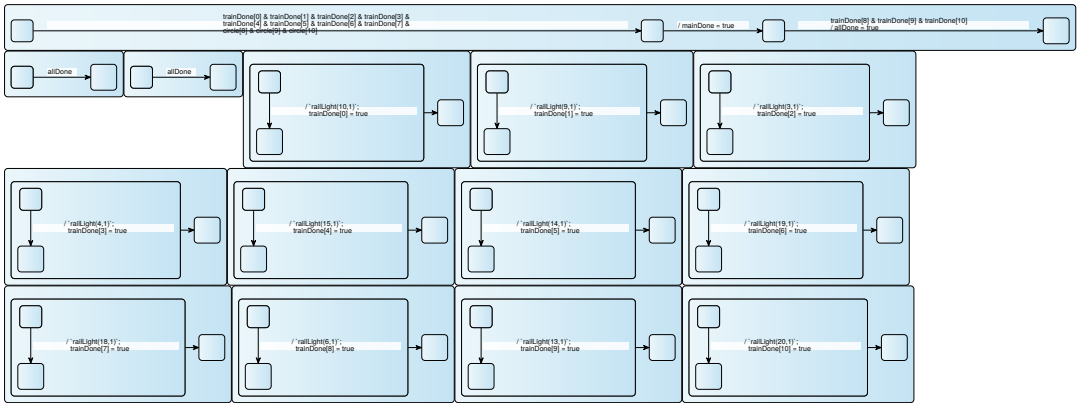


(a) Layout calculated by the Row-Filling and Compaction Heuristic.

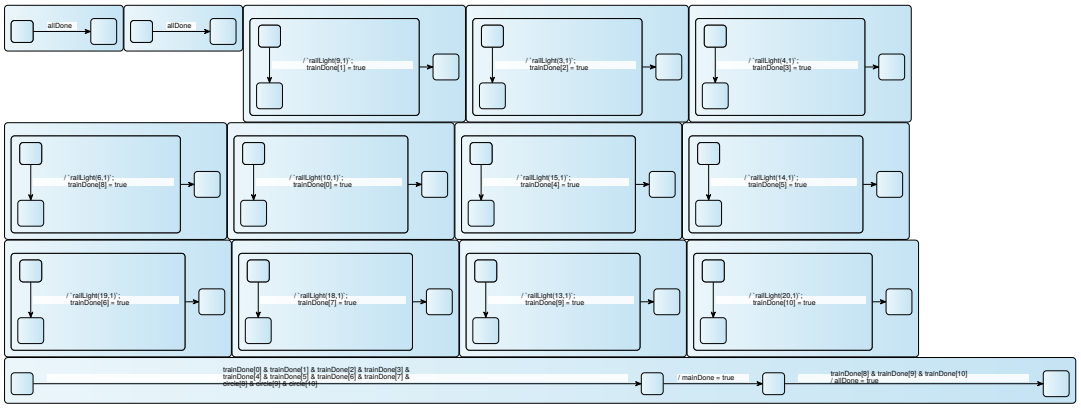


(b) Layout calculated by Box Layout.

Figure 5.9. An example taken from the railway model controller implemented by students in 2014.



(a) Layout calculated by the Row-Filling and Compaction Heuristic.



(b) Layout calculated by Box Layout.

Figure 5.10. Another example taken from the railway model controller implemented by students in 2014.

Chapter 6

Conclusion

This chapter summarizes the work that I did in this thesis and draws conclusions. Finally, suggestions for future work are presented.

6.1 Summary

This thesis described and formalized a layout problem produced by the Box Layout algorithm in Chapter 1 and tried to solve it by introducing several algorithms in Chapter 3. Said Box Layout algorithm produces a lot of unnecessary whitespace, especially when placing one big rectangle surrounded by multiple smaller ones. Even though ELK specializes in automatic graph layout, the given problem cannot be easily solved to optimality as it is a rectangle packing problem with multiple restrictions at its core. Additionally, this thesis introduced and formalized another restriction for the final layout in Section 3.1: an order restriction for rectangles to improve the legibility and mental map for SCCharts developers. Chapter 3 introduced an algorithm tailored to a slightly altered special case, two heuristics for fast and easy placement of any set of given rectangles, and two algorithms based on linear programs to solve the problem almost optimally.

The algorithm for the special case as described in Section 3.2.3 aims to evenly distribute smaller rectangles around a big one. The two other heuristics are introduced in Section 3.2.1 and Section 3.2.2. The former is called *Down-Right Heuristic* and focuses on sequentially placing rectangles below or to the right of preceding rectangles, whereas the latter is called *Row-Filling and Compaction Heuristic* and organizes the rectangles in stacks, which are assigned to rows. The rows are filled with stacks to meet an estimated bounding box. The stacks are filled and compacted as much as the corresponding row allows. When a row's width fails to meet the estimated bounding box after a compaction, said row is filled with stacks from a subsequent row. Hence, a compact packing of rectangles is found. Lastly, a formulation for a minimization problem was introduced in Section 3.3, which was implemented using IBM's CPLEX. Based on the previous formulation, a binary search algorithm was presented in Section 3.4. It calculates dimensions minding the aspect ratio and passes these dimensions as a restriction to a linear program, which, if possible, finds a layout.

The algorithms were implemented as a plug-in for ELK as described in Chapter 4. A specific algorithm can be executed through the properties the implementation provides.

Chapter 5 evaluated the implemented algorithms. The implementations were tested on 350 randomly generated instances containing up to one hundred rectangles. They were

examined for whitespace, aspect ratio, and the scale measure. The algorithms based on linear programs turned out to be too slow to realistically solve reasonably sized instances. Out of the remaining algorithms, the Row-Filling and Compaction algorithm performed best regarding whitespace and scale measure, but the Down-Right Heuristic performed slightly better in producing the desired aspect ratio. However, the Down-Right Heuristic, linear program and binary search turned out not to be viable algorithms for the given problem since they create whitespace openings that cannot be closed by expanding rectangles. Lastly, a small sample provided the observation that the Box Layout has a faster execution time than the Row-Filling and Compaction Heuristic.

As the Row-Filling and Compaction Heuristic was found to be the superior algorithm presented in this thesis, it will be enabled by default in the implementation as described in Chapter 4.

6.2 Future Work

The rectangular packing problem has been proven to be NP-complete as stated in Chapter 1. It often remains in that complexity class when restricting the problem [MS11]. The problem in this thesis has more restrictions than the general rectangular packing problem, such as achieving the desired aspect ratio of the packing and respecting an order of the rectangles in the drawing area. Future work could investigate whether the restrictions as formulated in this thesis cause the problem to remain NP-complete.

Generally, future work could include finding ways to speed up the proposed algorithms. As a concrete example, possible shifts are calculated and executed after every step in the two main phases of the Row-Filling and Compaction Heuristic, since the algorithm has no way to determine whether there are still more steps ahead. It would be interesting to investigate or prove that the shifting calculations can be reduced to either doing them at the end of a phase or after all improvements are done. Another example is given by the search of vertical or horizontal constraints when shifting a rectangle during the Down-Right Heuristic. The current approach searches through all previously placed rectangles. With the help of sophisticated data structures, it might be possible to speed up the search to constant time.

As stated in Section 5.1, the Down-Right Heuristic was evaluated with shifting of rectangles enabled because Section 3.2.1 had already stated that disabling the shift will cause more whitespace. Future work could evaluate the Down-Right Heuristic with shifts disabled for quantitative results that are comparable to the results produced in Section 5.1.

Section 5.2.1 stated that packings produced by the Down-Right Heuristic make it hard to follow the given order. This observation could be reinforced by a qualitative evaluation. Since the Down-Right Heuristic strictly follows the definition of an order given by Equation 3.1.2, it would be interesting to further investigate the order definitions as introduced in Section 3.1. Future work could include a detailed analysis of the order definitions that focuses on the trade-off between whitespace and legibility and whether ordering constraints help maintaining the natural reading order and mental map of a viewer. Moreover, there might be other possible

definitions of what it means to follow an order. Future work could propose such definitions and examine them.

Since the Row-Filling and Compaction Heuristic depends on a bounding box approximation, it is possible to look into the Down-Right Heuristic for improvements or to find a new approximating algorithm overall. The improved version should produce a narrower bounding box, since the Row-Filling and Compaction Heuristic packs the rectangles very closely. This causes the algorithm to not fill the bounding box in its height, which yields aspect ratios that are bigger than desired. This was demonstrated in Chapter 5. Another possible improvement could be to decrease the execution time of the approximating algorithm since two algorithms are executed when using the Row-Filling and Compaction Heuristic.

Alternatively, an approach could be to approximate the width by which the bounding box should be reduced before executing the Row-Filling and Compaction Heuristic. The goal of reducing the approximated width is to produce a higher and narrower packing, which results in a smaller aspect ratio that is closer to the desired aspect ratio.

Since the Row-Filling and Compaction Heuristic fills rows with rectangles and stacks from subsequent rows, the bottom row will always provide stacks and rectangles without taking any. This can cause it to not fill the bounding box width, which ultimately leads to unused whitespace. It might be possible to assess the whitespace in the last row and estimate by what amount the bounding box should be reduced to fill said whitespace. Then, another iteration of the Row-Filling and Compaction Heuristic with the altered dimensions is executed. The idea of this approach is to minimize the whitespace even more. This approach would naturally increase the execution time.

Lastly, the issue of drawings produced by the Row-Filling and Compaction Heuristic as described in Section 5.2.1 can likely be solved by executing the phases of the algorithm on a single row until no more improvements can be made on said row. This way, unwanted compactions are not made since the highest stack of a yielding row may be assigned to a preceding row before compactions can be made in the yielding row. Due to time constraints, this issue was not tackled by this thesis and is proposed for future work.

Appendix A

Appendix

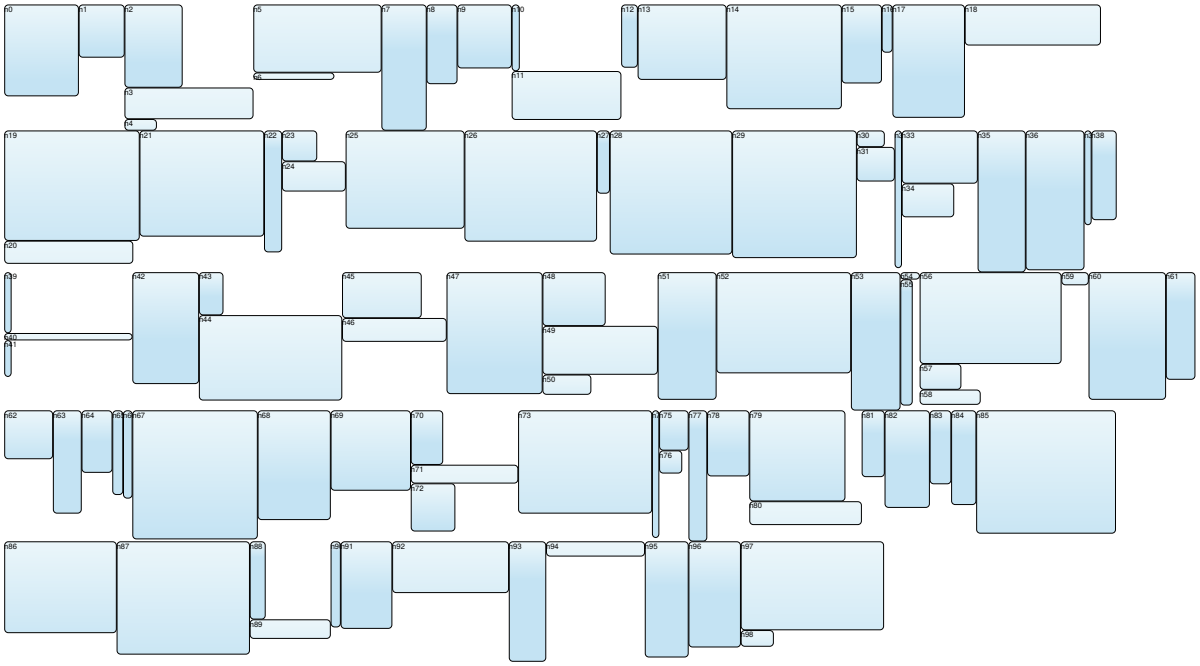


Figure A.1. A final layout produced by the Row-Filling and Compaction Heuristic on the same test instance as Figures A.2, A.3, and A.4. The desired aspect ratio is 1.3. The packing has an area of $1664 \cdot 918 = 1\,527\,552$ with an aspect ratio of 1.812 and 17.057% whitespace.

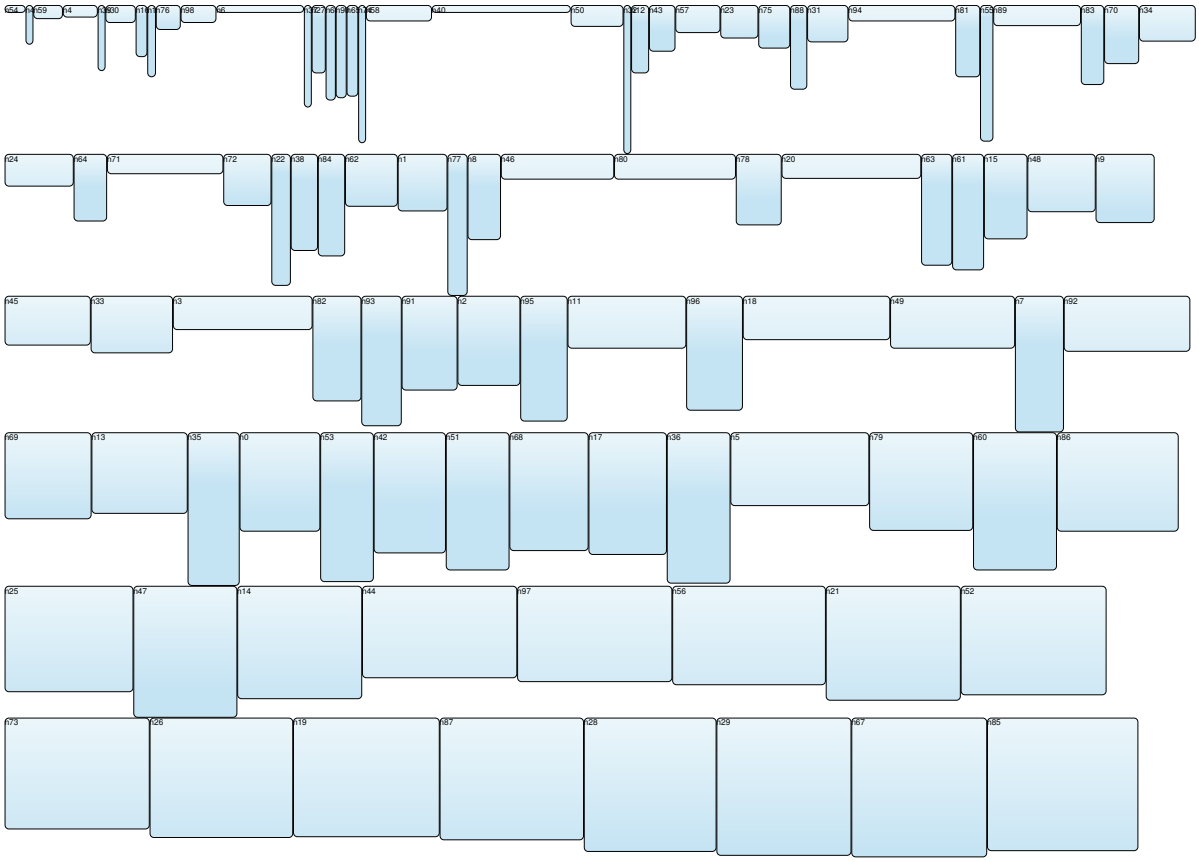


Figure A.2. A final layout produced by Box Layout on the same test instance as Figures A.1, A.3, and A.4. The desired aspect ratio is 1.3. The packing has an area of $1536 \cdot 1099 = 1\,688\,064$ with an aspect ratio of 1.398 and 43.428% whitespace.

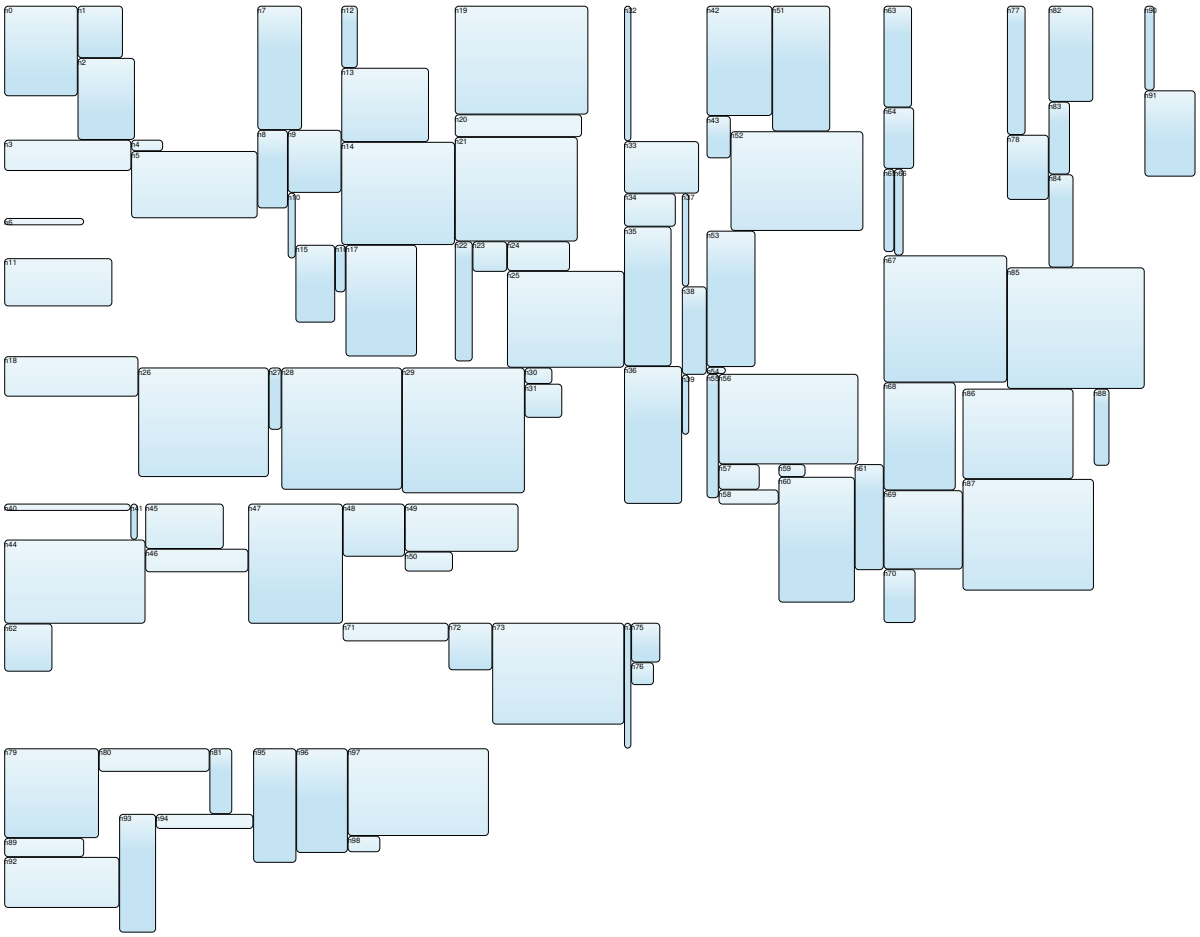


Figure A.3. A final layout produced by the Row-Filling and Compaction Heuristic with shifts enabled on the same test instance as Figures A.1, A.2, and A.4. The desired aspect ratio is 1.3. The packing has an area of $1689 \cdot 1314 = 2\,219\,346$ with an aspect ratio of 1.285 and 39.441% whitespace.

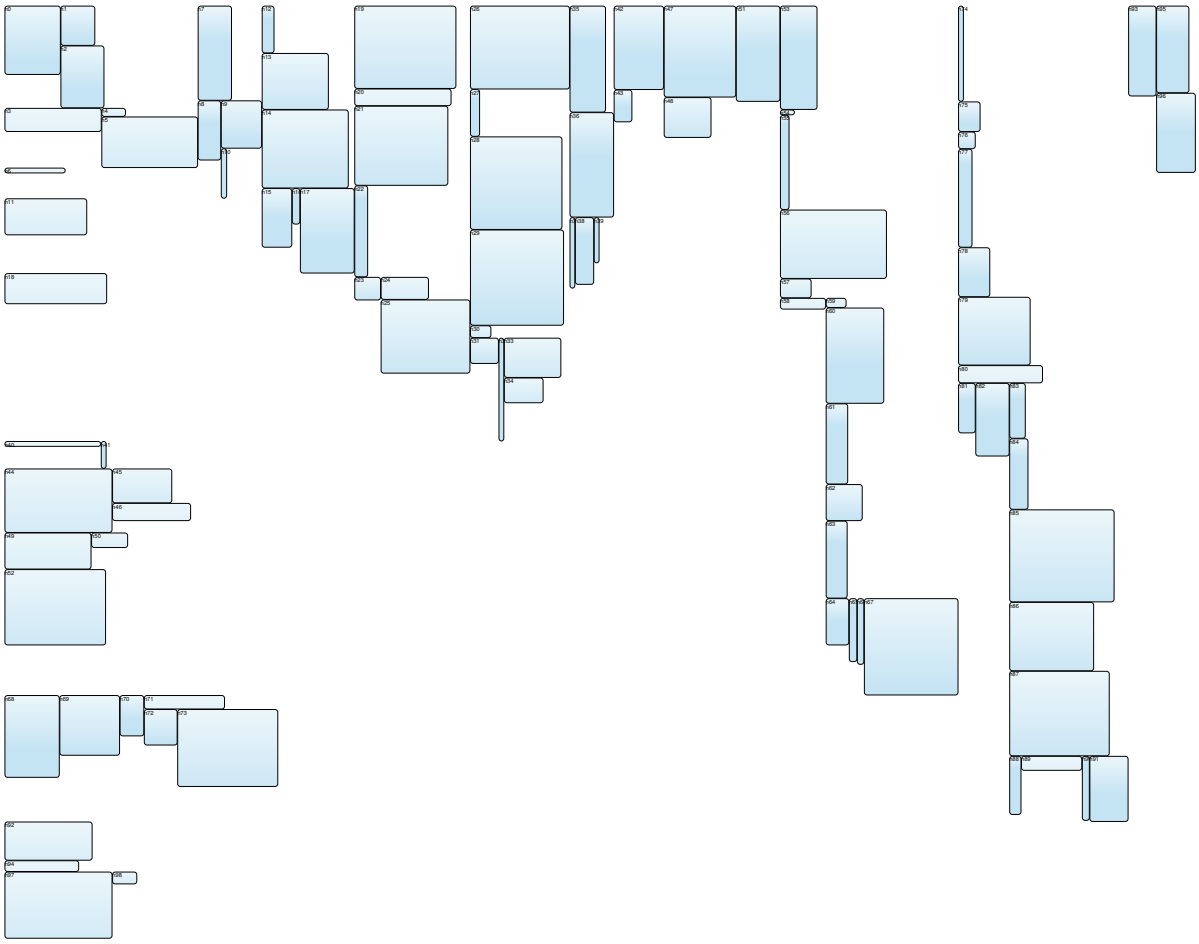


Figure A.4. A final layout produced by the Down-Right Heuristic with shifts disabled on the same test instance as Figures A.1, A.2, and A.3. The desired aspect ratio is 1.3. The packing has an area of $2213 \cdot 1733 = 3\,835\,129$ with an aspect ratio of 1.277 and 64.752% whitespace.

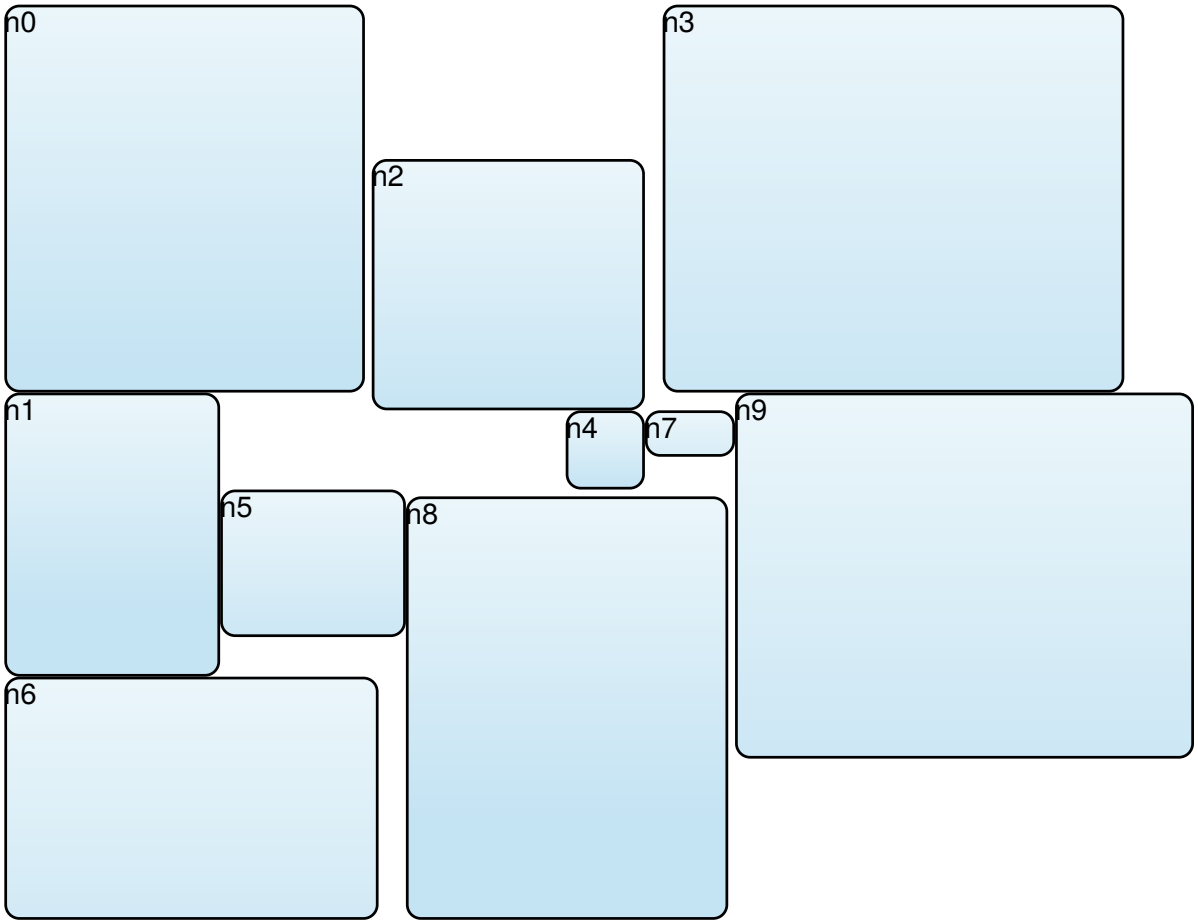


Figure A.5. A final layout produced by the linear program on the same test instance as Figure A.6. The whitespace opening between *n0*, *n1*, *n2*, *n4*, *n5*, and *n8* cannot be closed by enlarging these rectangles. The desired aspect ratio is 1.3. The packing has an area of $435.5 \cdot 335 = 145\,892.5$ with an aspect ratio of 1.3 and 6.234% whitespace.

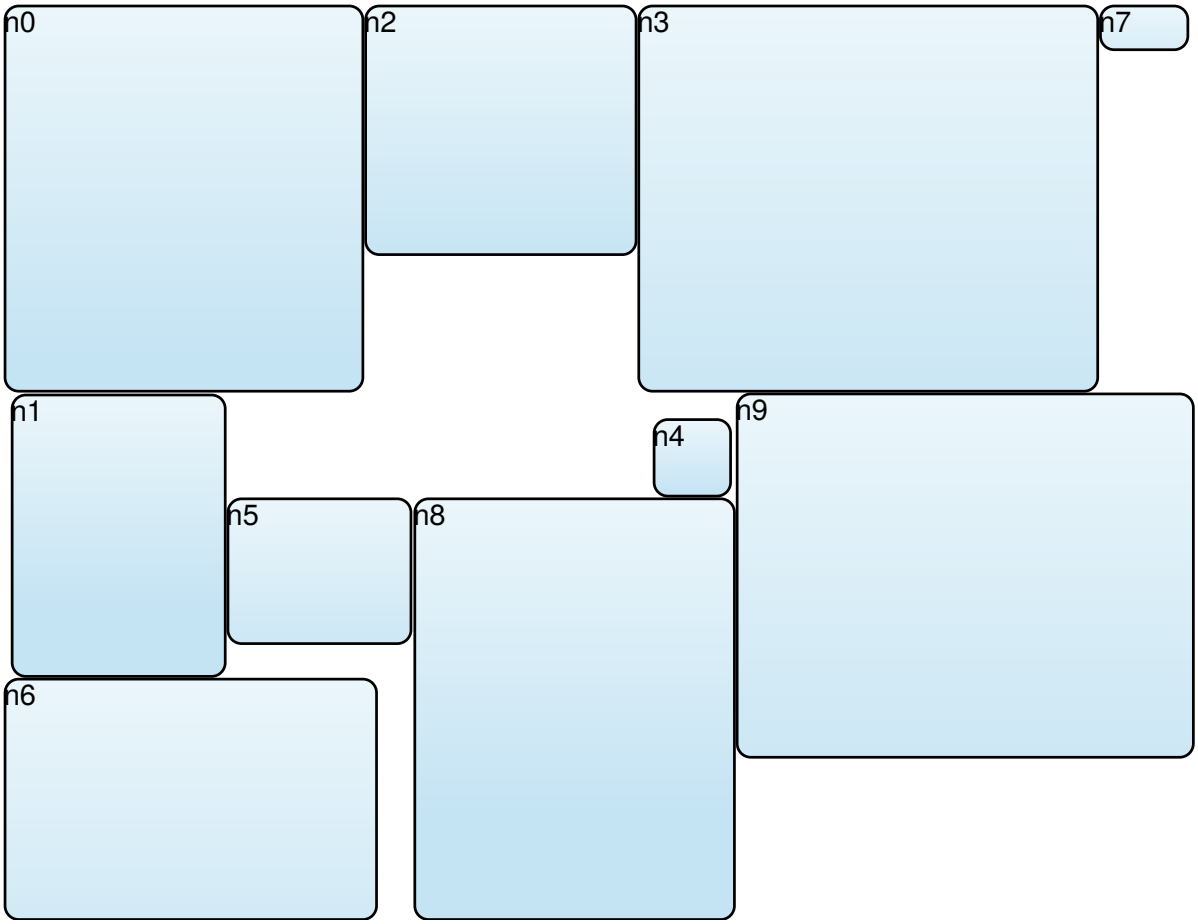


Figure A.6. A final layout produced by binary search on the same test instance as Figure A.5. The whitespace opening between *n0*, *n1*, *n2*, *n4*, *n5*, and *n8* cannot be closed by enlarging these rectangles. The desired aspect ratio is 1.3. The packing has an area of $436.01 \cdot 335.39 = 146\,235.49$ with an aspect ratio of 1.3 and 6.947% whitespace.

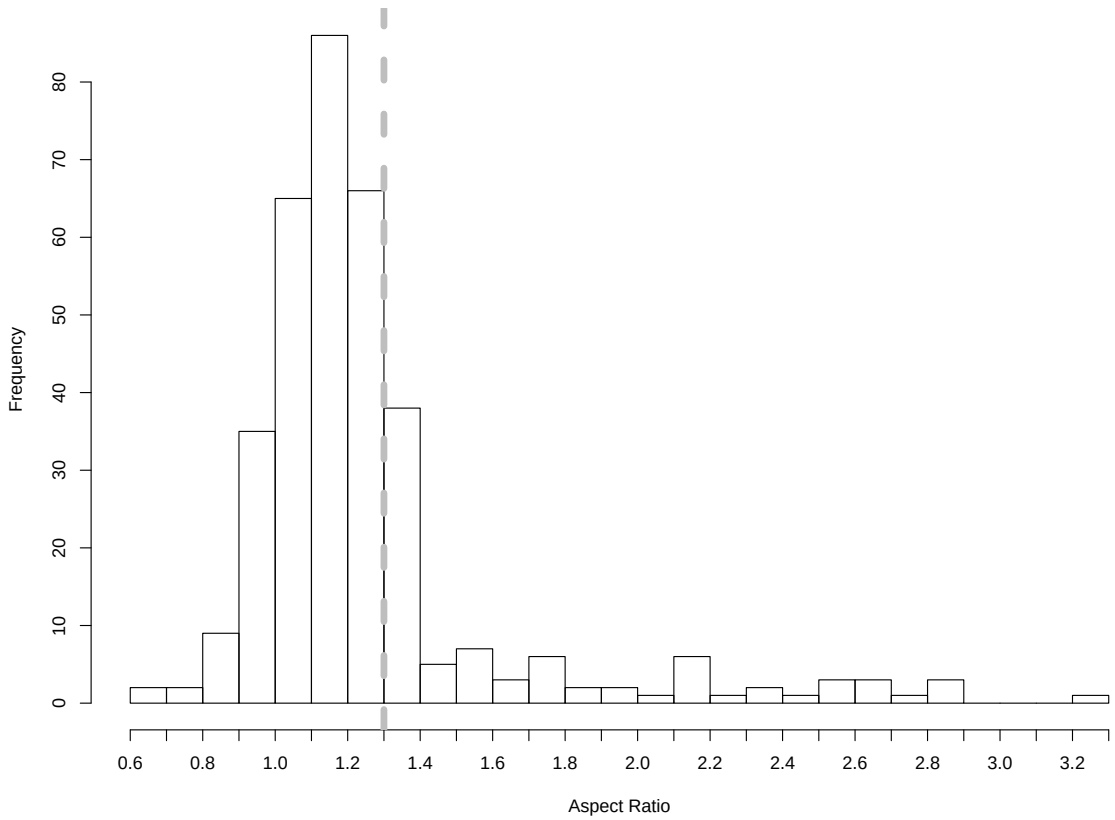


Figure A.7. Histogram of the aspect ratio of Box Layout executed on all test instances. The dotted vertical line indicates the desired aspect ratio.

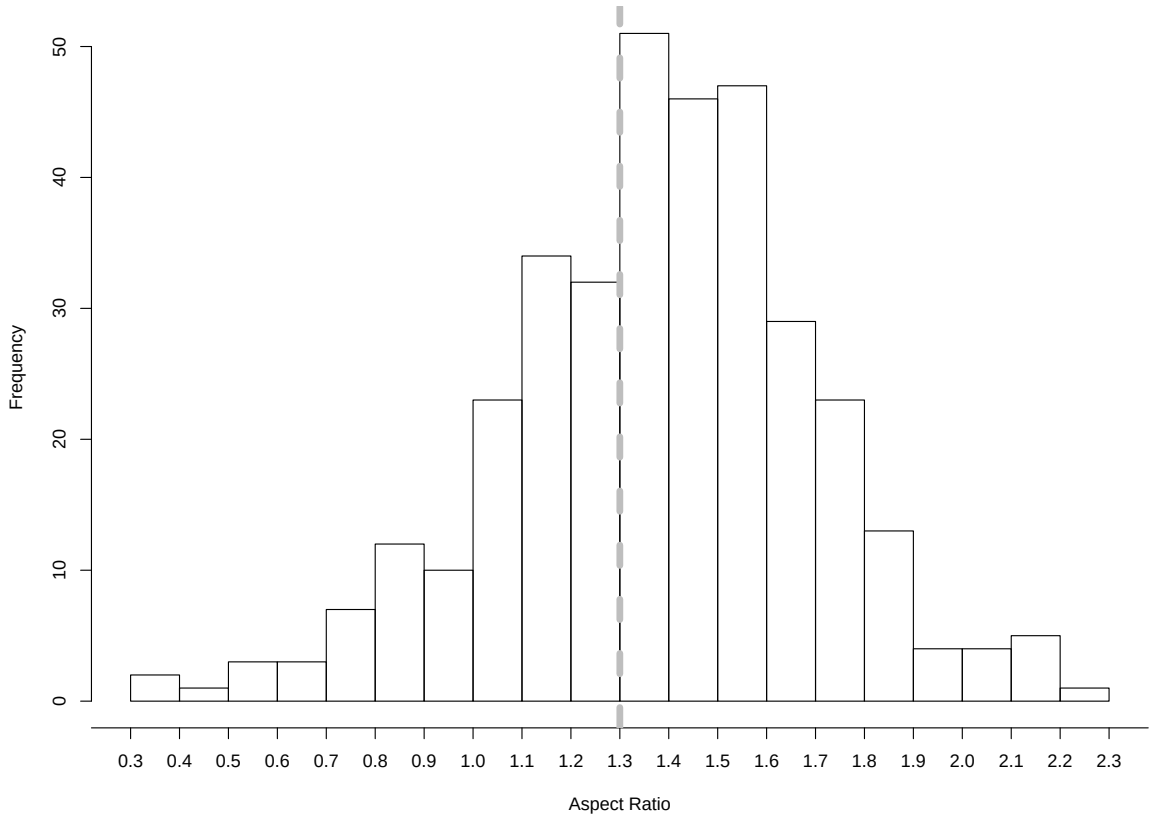


Figure A.8. Histogram of the aspect ratio of the Row-Filling and Compaction Heuristic executed on all test instances. The dotted vertical line indicates the desired aspect ratio.

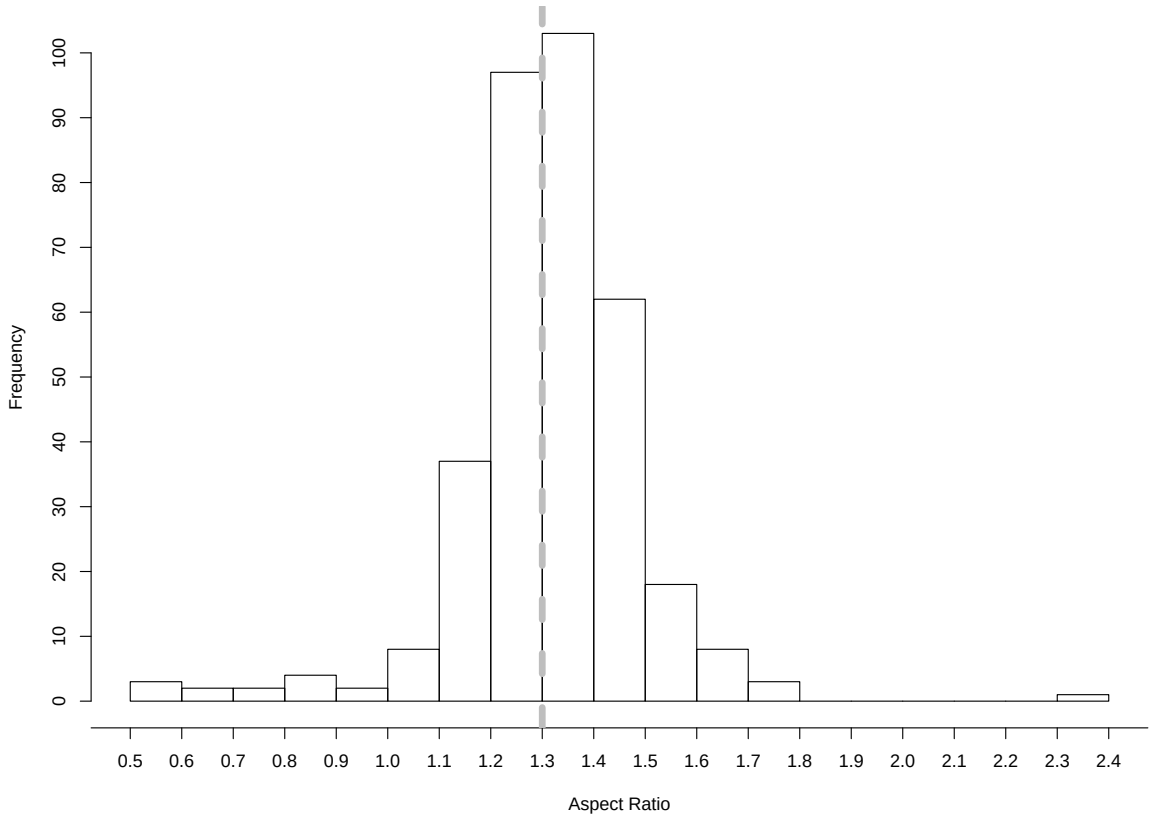


Figure A.9. Histogram of the aspect ratio of the Down-Right Heuristic executed on all test instances. The dotted vertical line indicates the desired aspect ratio.

Bibliography

[And95] Charles André. *Syncharts: a visual representation of reactive behaviors*. Tech. rep. 1995.

[BCR80] Brenda S. Baker, Ed Coffman, and Ronald L. Rivest. "Orthogonal packings in two dimensions". In: *SIAM Journal on Computing*. Vol. 9. Nov. 1980, pp. 846–855. DOI: 10.1137/0209064.

[CMS99] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999. ISBN: 1558605339.

[Cru11] Alan Cruse. *Meaning in language: an introduction to semantics and pragmatics*. Oxford University Press UK, 2011.

[DD92] Kathryn A. Dowsland and William B. Dowsland. "Packing problems". In: *European Journal of Operational Research* 56.1 (1992), pp. 2–14. DOI: 10.1016/0377-2217(92)90288-K.

[DET+94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry: Theory and Applications* 4 (1994), pp. 235–282.

[DET+99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[FDK02] Karlis Freivalds, Ugur Dogrusoz, and Paulis Kikusts. "Disconnected graph layout and the polyomino packing approach". English. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 378–391. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4_30.

[FH09] Hauke Fuhrmann and Reinhard von Hanxleden. *The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage*. http://www.informatik.uni-kiel.de/rtsys/kieler/. 2009.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Long version: Technical Report 1311, Christian-Albrechts-Universität

zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274. Edinburgh, UK: ACM, 2014.

[HIM+98] Kunihiko Hayashi, Michiko Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara. "A layout adjustment problem for disjoint rectangles preserving orthogonal order". In: *Graph Drawing*. Springer. 1998, pp. 183–197.

[Kor03] Richard E. Korf. "Optimal rectangle packing: initial results". In: *Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'03. Trento, Italy: AAAI Press, 2003, pp. 287–295. ISBN: 1-57735-187-8.

[MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. "Layout adjustment and the mental map". In: *Journal of Visual Languages & Computing* 6.2 (1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.

[MS11] Jens Maßberg and Jan Schneider. "Rectangle packing with additional restrictions". In: *Theoretical Computer Science* 412.50 (2011), pp. 6948–6958. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2011.09.007.

[PHG06] Helen C. Purchase, Eve E. Hoggan, and Carsten Görg. "How important is the "mental map"? – an empirical investigation of a dynamic graph layout algorithm". In: *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*. Vol. 4372. LNCS. Springer, 2006, pp. 184–195. ISBN: 978-3-540-70903-9. DOI: 10.1007/978-3-540-70904-6.

[Pur02] Helen C. Purchase. "Metrics for graph drawing aesthetics". In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516.

[RH18] Ulf Rüegg and Reinhard von Hanxleden. "Wrapping layered graphs". In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. To appear. Springer, 2018.

[Rie10] Martin Rieß. "A graph editor for algorithm engineering". Bachelor Thesis. Kiel University, Department of Computer Science, 2010.

[Sch06] Douglas C. Schmidt. "Model-driven engineering". In: *Computer* 39.2 (2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSR16] Christoph Daniel Schulze, Miro Spönemann, and Ulf Rüegg. *Eclipse layout kernel*. https://www.eclipse.org/elk/. 2016.

[Ste97] A. Steinberg. "A strip-packing algorithm with absolute performance bound 2". In: *SIAM J. Comput.* 26.2 (1997), pp. 401–409. ISSN: 0097-5397. DOI: 10.1137/S0097539793255801.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (1981), pp. 109–125.

[TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. "Automatic graph drawing and readability of diagrams". In: *IEEE Transactions on Systems, Man and Cybernetics* 18.1 (1988), pp. 61–79. ISSN: 0018-9472.

[The06] The model railway. *Project homepage*. Group of Real-Time and Embedded Systems, Department of Computer Science, Kiel, Germany. 2006. URL: http://www.informatik.uni-kiel.de/~railway.

[WTZ+17] Lei Wu, Xue Tian, Jixu Zhang, Qi Liu, Wensheng Xiao, and Yaowen Yang. "An improved heuristic algorithm for 2d rectangle packing area minimization problems with central rectangles". In: *Engineering Applications of Artificial Intelligence* 66 (2017), pp. 1–16. ISSN: 0952-1976.