

Algorithms and Drawings for Mixed Linear Layouts of Graphs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp de Col, BSc

Matrikelnummer 01528238

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

Mitwirkung: Fabian Klute, M.Sc.

Wien, 28. April 2019

Philipp de Col

Martin Nöllenburg

Algorithms and Drawings for Mixed Linear Layouts of Graphs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp de Col, BSc

Registration Number 01528238

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg

Assistance: Fabian Klute, M.Sc.

Vienna, 28th April, 2019

Philipp de Col

Martin Nöllenburg

Erklärung zur Verfassung der Arbeit

Philipp de Col, BSc
Laudongasse 55/6, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. April 2019

Philipp de Col

Danksagung

Ich danke meinem Betreuer Martin Nöllenburg sowie Fabian Klute für die Unterstützung bei meiner Diplomarbeit. Mit ihrem Wissen und ihrer Motivation halfen sie mir während unseren regelmäßigen Treffen mit hilfreichen Rückmeldungen und Erkenntnissen weiter und gaben mir die Freiheit, meine eigenen Ideen zu verfolgen. Die Zusammenarbeit mit ihnen machte diese Arbeit zu einer interessanten und angenehmen Aufgabe und ich konnte in dieser Zeit viel von ihnen lernen.

Ich möchte auch meinen Freunden und meiner Familie, vor allem meinen Eltern, danken, die mich über all die Jahre hinweg unterstützt haben. Ohne sie wäre diese Leistung nicht möglich gewesen.

Acknowledgements

I would like to thank my thesis advisor Martin Nöllenburg as well as Fabian Klute for the support of my diploma thesis. With their knowledge and motivation, they provided guidance during our regular meetings with helpful feedback and insights while still allowing me the freedom to pursue my own ideas. Working together with them made this thesis an exciting and enjoyable task, and I was able to learn a lot from them during this time.

I would also like to express my gratitude to my friends and family, foremost my parents, that supported me through all the years. This accomplishment would not have been possible without them.

Kurzfassung

Ein gemischtes lineares Layout eines Graphen ist eine totale Ordnung seiner Knoten, so dass die Kanten als Elemente betrachtet werden können, die in dieser Reihenfolge mit den beiden Datenstrukturen eines Stapelspeichers und einer Warteschlange verarbeitet werden können. Layouts, die nur Stapelspeicher verwenden, werden als Bucheinbettungen bezeichnet, was ein weit erforschtes Thema ist und auch reine Warteschlangenlayouts finden viel Beachtung. Die Kombination dieser beiden Konzepte zur gleichen Zeit wird jedoch seltener verwendet. Die Literatur beschreibt viele Algorithmen, Eigenschaften und Ergebnisse für Layouts, die entweder aus Stapelspeichern oder Warteschlangen bestehen, aber oft fehlen die passenden Gegenstücke für gemischte Layouts. In dieser Diplomarbeit stellen wir neue Ergebnisse in den Bereichen Komplexität, Heuristiken und Zeichnungen für gemischte Layouts vor, die es vorher noch nicht gegeben hat oder die bestehende übertreffen. Um dies zu erreichen, haben wir zum einen bereits bestehende Methoden für reine Layouts wiederverwendet und zum anderen neue Ideen zur Lösung der Probleme eingeführt. Wir erwarten, dass diese Ergebnisse dazu beitragen werden gemischte lineare Layouts besser zu verstehen und es ermöglichen, diese für praktische Zwecke zu nutzen.

Abstract

A mixed linear layout of a graph is a total order of its vertices in a way that the edges can be seen as elements that are processed in this order with a stack and a queue data structure. Layouts that only use stacks are known as book embeddings and is a widely researched topic. Queue layouts also receive much attention. However, the study of these two concepts in combination is less understood. The literature describes many algorithms, properties and results for layouts that consist of either stacks or queues but the adequate counterparts for mixed layouts are often missing. In this thesis, we introduce new results for computational hardness, heuristics and drawings for mixed linear layouts that have not existed before or outperform existing ones. To achieve this, we reused existing methods of linear layouts as well as introduced new ideas to tackle the problems. We expect that these results will help to understand mixed linear layouts better and use them for practical purposes.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Linear layouts of graphs	1
1.2 Motivation	2
1.3 Goals and methodology	3
1.4 Related work	4
1.5 Structure of the thesis	6
2 Preliminaries	7
3 Theory	13
3.1 Complexity	13
3.2 Algorithms	23
3.3 Planar bipartite graphs	24
4 Design and evaluation of heuristics	27
4.1 Existing heuristics	28
4.2 Data structure heuristic	30
4.3 Experiments	32
4.4 Optimization	44
4.5 Summary of experiments	47
5 Drawings	49
5.1 Circular drawings	49
5.2 Linear drawings	53
5.3 Linear cylindric drawings	56
6 Conclusion	59
6.1 Summary	59
	xv

6.2 Future Work	60
List of Figures	63
List of Tables	65
List of Algorithms	67
Bibliography	69
Appendix	75

Introduction

In this thesis, we study the mixed linear layouts of graphs. We start with an introduction to linear layouts and explain the concept of a mixed linear layout that consists of a stack and a queue page. We continue with the motivation for studying such layouts and then set the goals for this thesis and describe the methods that we use to achieve them. After that, we introduce related work that has been done on this subject and finally, this introduction will conclude with an outline of the following chapters.

1.1 Linear layouts of graphs

A graph is an abstract structure that models a set of objects and the relationships among these objects. This concept makes a graph a powerful and flexible tool that can represent many physical and abstract structures in the real world such as computer networks, employee hierarchies or streets on a map. Formally such a graph $G = (V, E)$ is a tuple that consists of a set of vertices V (the objects) and a set of edges E (the relationships) where each edge is a pair of two vertices that signifies that those two vertices have a relation.

A linear layout of a graph is a drawing of the graph where all the vertices are drawn on an imaginary straight line that cuts the two-dimensional-space into two half-planes. The edges are drawn as simple curves on such a half-plane which can be seen as the page of a book while the vertices on the line can be seen as the spine of a book. When drawing a graph in a linear layout, it is often desired that the drawing has either none or as few as possible crossings of the edges. This can be achieved by choosing a good order of vertices and, if there is more than one page available, by a good assignment of the edges to the different pages. A page without any crossings is called a stack page. The problem of minimizing and eliminating crossings is an extensive research topic [Oll73] [BK79] [Yan89] [DW04] [BKZ15].

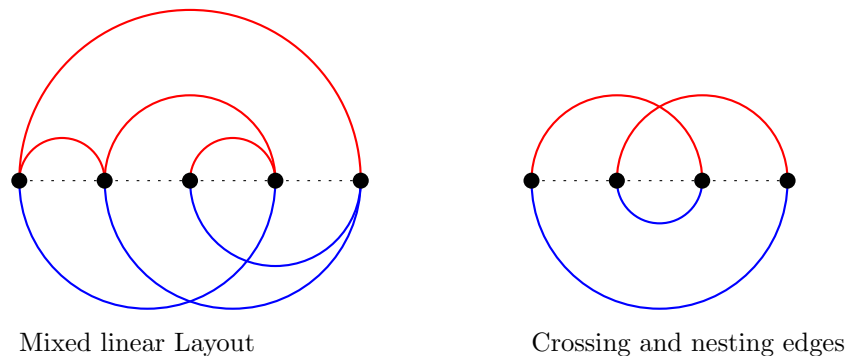


Figure 1.1: The left image is an example of a mixed linear layout. The stack page is shown on the upper half with red edges, and the queue page is shown on the lower half with blue edges. On the right image are the two forbidden patterns. The red stack edges cross each other, and the blue queue edges are nested.

In contrast to the stack there is a dual concept called queue [HLR92] [HR92]. In a queue layout, the edges are allowed to cross, but they are not allowed to nest so that one edge lies strictly inside the area that is enclosed by another edge. A mixed linear layout is a layout of a graph that has at least one stack page and at least one queue page. For this thesis, we mainly focus on a 1-stack 1-queue layout. Therefore the problem of finding such a mixed layout can be formulated as the following question. Given a graph, does a vertex order and an assignment of the edges either to the stack or to the queue page exist so that the graph can be drawn in a linear layout without crossing edges on the stack page and nested edges on the queue page? An example of such a mixed layout can be seen in Figure 1.1 on the left image. The right image shows two edges that are crossing on the upper half and two edges that are nesting on the lower half.

1.2 Motivation

Dujmovic and Wood [DW04] list a wide range of applications for stack and queue layouts. Those applications include sorting permutations, fault-tolerant VLSI design, complexity theory, graph drawing, parallel process schedule, and many others. While stack and queue layouts separately are a widely researched topic, both of these concepts combined receive somewhat less attention.

For example in graph drawing a typical way to draw two stack pages is in a circular layout or as an arc diagram as it can be seen in Figure 1.2. In the circular layout, the vertices are drawn in the same order as in the linear layout. Both pages can be drawn inside the circle with different types of edges. It is also possible that only one page is drawn inside the circle with the edges as straight lines and the second page on the outer side of the circle where the edges are arches which are routed around the circle. Such a drawing is crossing-free and arguably is an easily readable drawing of a graph. The downside of such a drawing is that it is only crossing-free if the graph has a two

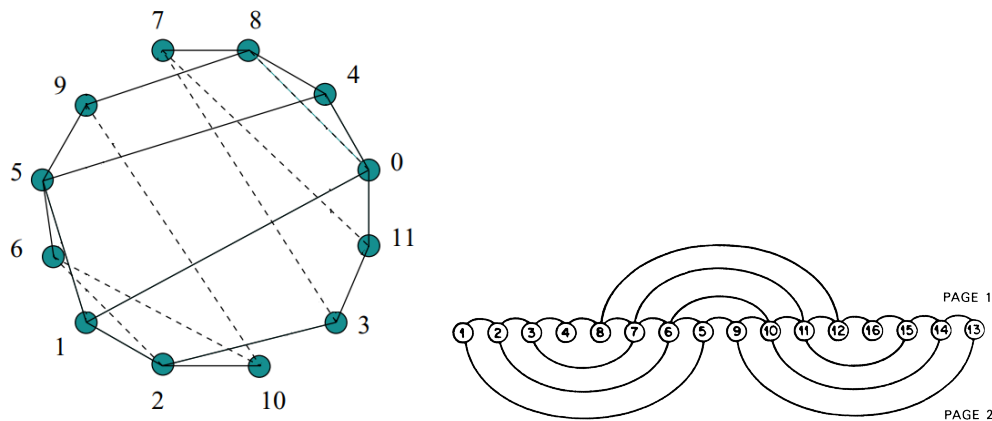


Figure 1.2: The drawing on the left is an example of two stack pages that are drawn in a circular layout from He et al. [HSM07]. The drawing on the right is an example of an arc diagram of two stack pages from Chung et al. [CLR87].

stack linear layout. If this is not the case, then one might accept the crossings or choose another drawing style. Since there exist graphs that do not have a 2-stack layout but do have a 1-stack 1-queue layout, choosing a drawing style for such a mixed layout might be another option.

Another application that combines both concepts is permutation sorting, where input data is sorted by pushing them sequentially into stacks or queues and pushing them to an output in the right order at the right point in time. For our case with one stack and one queue, we can sort the incoming data by choosing to push them to either the stack or the queue and after every newly received data we have the choice to push an arbitrary amount of data from both the stack and the queue to the output stream. Note that not every permutation is possible with such a simple system. In the general case, it is possible to build larger more powerful networks of such data structures and others like a dequeue that are placed in parallel or in series to allow permutations that are otherwise not possible. One can imagine such a system as a railroad switchyard where a train with different wagons arrives and the single detached wagons need to be driven through the switchyard in such a way that they will be sorted in a specific permutation when they leave the switchyard [Tar72].

1.3 Goals and methodology

With this thesis, we want to understand mixed linear layouts better. The aim of this work is many-sided, and we try to focus on several aspects. We cover theoretical aspects and algorithms, examine heuristics, and we also present different ways to draw graphs in a mixed linear layout. The first step of this thesis is to collect the related work on this topic. There is some work about mixed layouts, which serves as a base of this thesis and

substantial work on stack and queue layouts on its own that contains relevant and useful information.

In the theoretical part, we try to provide a better understanding of the properties of a mixed layout. Computational complexity and NP-hardness is one crucial aspect where some results already exist for some types of mixed layouts and where we can prove new results for some others. We also provide methods and algorithms to compute such layouts and check if such a layout exists for a given graph or validate the correctness of a given layout. Pupyrev [Pup17] conjectured that every planar bipartite graph admits a 1-stack 1-queue layout and we take a look on this conjecture.

Since finding a linear layout is, in general, an NP-hard problem we address heuristics to create layouts with preferably as few as possible crossings and nestings. To achieve this goal, we take a look at existing heuristics for book drawings and try to reuse those for the mixed layouts. Therefore we implement them and run experiments to identify the best heuristics. We also develop a new heuristic that is specially designed for mixed layouts and compare it to the existing heuristics.

In the graph drawing part, the focus is on obtaining a good drawing of a graph that has a mixed layout. Linear layouts with two pages are often drawn in a circular style. For such a drawing it would be especially interesting to assess how the properties of a queue layout can be used to obtain a good looking and readable drawing. We identify which ideas can be used for mixed layouts too and develop new drawing styles.

1.4 Related work

Stack layouts and queue layouts are extensively researched topics. These concepts were introduced by Ollmann [Oll73], Heath et al. [HLR92] and Heath and Rosenberg [HR92]. For many common classes of graphs, their respective number of pages which are needed in order to be drawn in such a layout is known or bounded.

Outerplanar graphs have a stack number of one [BK79]. Such a graph is shown in Figure 1.3 and this is precisely the graph class that fits on a single stack page. Graphs have a stack number of two if they are a subgraph of a planar Hamiltonian graph [BK79]. Therefore, this includes every graph that is planar and has a Hamiltonian path. Recognizing such graphs is an NP-complete problem [CLR87] and therefore deciding whether or not a graph has an embedding on two stack pages is in general already NP-hard. Since every planar bipartite graph is subhamiltonian, they can also be embedded on two stack pages [Ove98]. 2-tree graphs, which are maximal series-parallel graphs, also have a stack number of two [RM95]. For every planar graph we know that four stack pages are sufficient [Yan89] but it is not known if it is possible to embed this graph class on only three pages. There exist planar graphs that need three pages, but not a single was found so far where three pages were not sufficient, and therefore it is still unknown if the stack number of planar graphs is three or four [BKZ15].

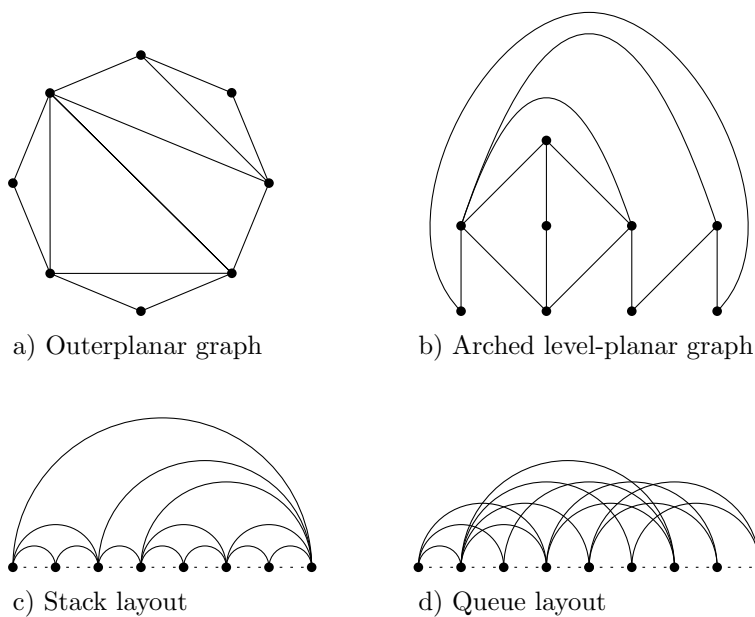


Figure 1.3: On the left side a) shows an example of an outerplanar graph which fits exactly on one stack page which is shown below in c). The vertex order for the stack page can be obtained by taking the order on which the vertices appear on the outer face. On the right side b) is an example of an arched level-planar graph which fits exactly on one queue page which is shown in d). The vertex order can be obtained by placing all consecutive levels next to each other.

For many graph classes, the queue number is also known or bounded. Arched level-planar graphs are the graph class that have a queue number of exactly one [HLR92]. Recognizing such graphs is an NP-complete problem [HR92]. An example of an arched level-planar graph is shown in Figure 1.3. Since trees are level-planar they have also a queue number of one. Outerplanar graphs have a queue number of two, and series-parallel graphs have a queue number of at most three. The queue number of planar 3-trees is bounded between four and five [ABG⁺18]. The queue number of planar graphs with n vertices has been improved over the years from $\mathcal{O}(\sqrt{n})$ to $\mathcal{O}(\log^2 n)$ [DFP13] to $\mathcal{O}(\log n)$ [Duj15]. Recently there has been a lot of progress on this subject, and it has been proven that planar graphs with a bounded degree have a constant queue number [BFG⁺18] [DMW19], and even a constant upper bound for planar graph was found [DJM⁺19]. Wood [Woo05] studied queue layouts of graph products and powers.

Besides these basic bounds for simple graph classes, many more bounds are known for more general graphs. Dujmović and Frati [DF18] provided bounds for the stack number and the queue number for general graphs with layered separators. The queue number is bounded to the path-width [Woo02] and to the tree-width [Wie17] of a graph. An Overview of more upper bounds for the stack and queue number of different graph families can be found in [DW04]. Subdivisions of graphs are a very powerful tool for linear layouts.

Every graph has a 3-stack, 2-queue and 1-stack 1-queue subdivision [DW05].

Mixed layouts are less studied compared to stack or queue layouts. Heath et al. [HLR92] proved that every graph which has a stack number of one can be drawn on two queue pages and vice versa that every graph with a queue number of one can be drawn on two stack pages. Heath and Rosenberg conjectured that each planar graph admits a 1-stack 1-queue layout [HR92]. This conjecture was recently disproved by Pupyrev [Pup17]. Pupyrev also states that mixed layouts are more powerful than stack or queue layouts on their own. He tested all maximal planar graphs up to 18 vertices, and all of these graphs admit a 1-stack 1-queue layout whereas the smallest graph that could not be drawn with a 2-stack layout has 11 vertices and the smallest graph that could not be drawn with a 2-queue layout has 14 vertices. Therefore it seems that the mixture of these two structures allows more graphs to be drawn than any of these structures by themselves with the same number of pages.

While the focus of this thesis lies on 1-stack 1-queue layouts, other mixed layouts with more than one page of the same type are also of great interest. Auer [AG11] [Aue14] [ABB⁺18] studied graphs which can be drawn on a cylinder's surface and processed by a dequeue data structure. These dequeues have a 2-stack 1-queue layout where there are some restrictions between the queue and the two stacks.

For problems that are related to scheduling, parallel processing or other workflows, where some actions need to be done before others, a directed acyclic graph is often better suited, and they have been studied by Heath et al. [HPT99] and Heath and Pemmaraju [HP99].

1.5 Structure of the thesis

We will start with the preliminaries in Chapter 2 where we explain basic terminologies and methods that will be used throughout this thesis. In Chapter 3 we will establish the theoretical part where we will present results for computational complexity and hardness of finding mixed linear layouts as well as some algorithms. Since finding mixed layouts is often NP-hard we will over heuristic approaches to find such layouts in Chapter 4. In Chapter 5 we will focus on the practical aspect and introduce some drawing styles and conventions that can be used to draw such layouts. Finally, we will conclude this thesis with a conclusion and an outlook of future work in Chapter 6.

Preliminaries

In this chapter, we define some terms, statements and algorithms that are used in the following chapters. A *graph* $G = (V, E)$ consists of a set of *vertices* $V = \{v_1, \dots, v_n\}$ and a set of pairs of these vertices $E = \{e_1, \dots, e_m\}$ with $e_j = \{u, v\}$ called the *edges*. Throughout this thesis, a graph will always be a simple undirected graph. Therefore a graph will not have multiple edges between the same two vertices and no edge that is a loop between the same vertex.

A *vertex order* \prec is a strict total order of the vertices V . We denote that a vertex u is in the order before another vertex v by $u \prec v$. A strict total order has the properties that it is transitive and trichotomous. The transitivity ensures that if $u \prec v$ and $v \prec w$ it holds that $u \prec w$. Trichotomous means that for all $u, v \in V$ it holds that either $u \prec v$, $v \prec u$ or $u = v$ is true. A vertex u is called the *predecessor* of a vertex v if $u \prec v$ and there exists no vertex w such that $u \prec w \prec v$. Similarly a vertex u is called the *successor* of a vertex v if $v \prec u$ and there exists no vertex w such that $v \prec w \prec u$. Therefore every vertex has a single successor (except the last one) and a single predecessor (except the first one) and the position of every vertex is comparable to every other vertex.

We denote the *start vertex* or *left vertex* of an edge e by $L(e)$ and the *end vertex* or *right vertex* by $R(e)$ with respect to a vertex order \prec . Therefore $L(e) \prec R(e)$ holds for every edge. The use of the words left and right make sense if we imagine that the vertices are placed on a horizontal line with the first vertex in \prec on the left side on the line and the last vertex in \prec on the right side on the line. We define the *edge length* of an edge as the number of the successors that are needed to traverse starting at $L(e)$ until we reach $R(e)$. Therefore for an edge between a vertex v and its successor u the edge length is one and for an edge between the first and the last vertex in the vertex order for a graph with n vertices, the edge length is $n - 1$.

Let e, f be two edges of G . Up to renaming the vertices we say that two edges e and f *cross* each other with respect to a vertex order \prec if $L(e) \prec L(f) \prec R(e) \prec R(f)$. We

call such a relation between two edges a *crossing*. If $L(e) \prec L(f) \prec R(f) \prec R(e)$ we say that e *nects* f and f is *nested* by e . We call such a relation between two edges a *nesting*. Finally e and f are *disjoint* if $L(e) \prec R(e) \prec L(f) \prec R(f)$. Therefore for every two pairs of edges it holds that these two edges are either crossing, nesting, disjoint or they are sharing a common vertex in which case it is not possible that they cross or nest each other.

A *stack page* or *queue page* is a set of edges $E' \subseteq E$ such that no two edges in this set are crossing or nesting respectively. A *conflict* is either a crossing on a stack page or a nesting on a queue page. The total number of conflicts is the sum of all crossings and nestings. A *linear layout* of a graph $G = (V, E)$ with the vertex order \prec and with s stack pages and q queue pages is an assignment of the edges E to the pages so that each edge is assigned to a single page. We call a layout with only stack pages ($s > 0, q = 0$) *stack (linear) layout*, a layout with only queue page ($s = 0, q > 0$) *queue (linear) layout*. If there are both types of pages available ($s > 0, q > 0$) we call it a *mixed (linear) layout*. Especially for mixed layouts it is often needed to specify the exact number of pages and for this, we use the notation of s -stack q -queue (linear) layout. The maximum number of edges that can be assigned to a page for a graph with n vertices is exactly $2n - 3$. This is because outerplanar and arched level-planar graphs are exactly the graphs that fit on a stack or a queue page respectively. For both graph classes this is the maximum number of edges that they can have.

A *k-rainbow* is a set of edges E' such that $L(e_1) \prec L(e_2) \prec \dots \prec L(e_k) \prec R(e_k) \prec \dots \prec R(e_2) \prec R(e_1)$. Therefore, a k -rainbow is a set of k pairwise nesting edges. On a queue layout, such a rainbow needs k different pages but it can be placed on a single stack page. A *k-twist* is a set of edges E' such that $L(e_1) \prec L(e_2) \prec \dots \prec L(e_k) \prec R(e_1) \prec R(e_2) \prec \dots \prec R(e_k)$. In contrary to a rainbow a k -twist is a set of k pairwise crossing edges. On a stack layout, such a twist needs k different pages but it can be placed on a single queue page. Therefore twists and rainbows highlight the dual relationship between stacks and queues. On a stack page we need to have nested edges that form rainbows whereas we are not allowed to have twists. On the contrary on a queue page we need to have crossing edges that form twists whereas we are not allowed to have rainbows.

A *drawing* Γ of a graph is a mapping of the vertices to two-dimensional points $\Gamma : V \rightarrow \mathbb{R}^2$ and a mapping of the edges to curves between two points $\Gamma : E \rightarrow \text{curve in } \mathbb{R}^2$ such that the curve starts at one vertex of the edge and ends at the other vertex.

Until this point, we have used the name stack and queue pages without explaining why we use the terms stack and queue and what the connection is between these data structures and the edges on a page. In computer science, a stack is a collection of elements that allows two operations: Adding an element to the collection (push) and removing an element from the collection (pop). A stack executes these operations in a ‘‘Last In First Out’’ (LIFO) manner, meaning that the most recently added element has to be removed first. A queue is also a collection of elements that allows to add (enqueue) and remove (dequeue) elements but in a ‘‘First In First Out’’ (FIFO) manner, meaning that the oldest element has to be removed first.

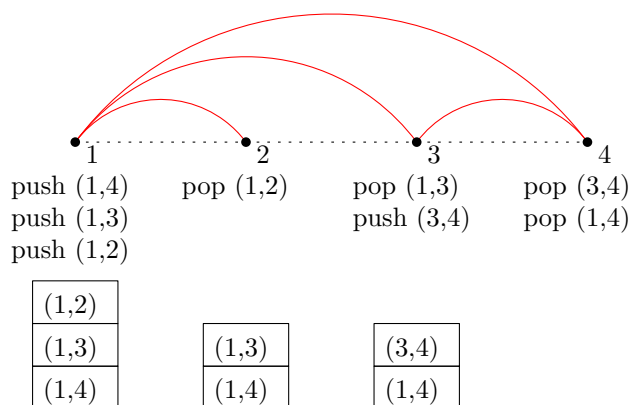


Figure 2.1: Validating a stack page with the help of a stack. The actions at each vertex and the state of the stack are shown below the respective vertex. The current state of the stack is displayed on the bottom of the figure where the uppermost box is the top of the stack. It is possible to execute the actions in the given sequence and therefore it is a valid stack page without crossings.

Algorithm 2.1: validateStackPage

Data: List of vertices V , edges E

Result: *True* if E is a valid stack page without a crossing with respect to the order of the vertices in V and otherwise *False*

```

1 stack ← Stack()
2 foreach  $v$  in  $V$  do
3    $startEdges$  ← edges that start at  $v$  ordered by longest first
4    $endEdges$  ← edges that end at  $v$  ordered by shortest first
5   foreach  $e$  in  $endEdges$  do
6     if  $e \neq stack.pop()$  then
7       return False
8   foreach  $e$  in  $startEdges$  do
9      $stack.push(e)$ 
10 return True

```

A linear layout with the vertex order \prec can be seen as an abstract way how elements should be processed. If we iterate over all vertices in \prec , each vertex can be seen as a point in time where some actions need to be started, and some other actions need to be completed. The edges represent these actions. If an edge starts at a vertex, then this corresponding action should be started, and the edge is added to a data structure. If an edge ends at a vertex, then the corresponding action needs to be finished, and the edge has to be removed from a data structure. If and only if no edges cross on a page, we can use a stack as a data structure to process the edges and if and only if no edges nest on a page, we can use a queue as data structure without violating the respective

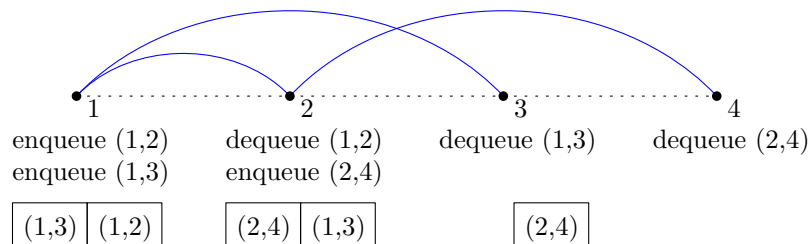


Figure 2.2: Validating a queue page with the help of a queue. The action at each vertex and the state of the queue are shown below the respective vertex. The current state of the queue is displayed on the bottom of the figure where the rightmost box is the head of the queue. It is possible to execute the actions in the given sequence and therefore it is a valid queue page with no nested edge.

LIFO and FIFO constraints. If there is a single crossing or a single nesting, then it is not possible to process the edges respectively with a stack or a queue. Therefore given a graph $G = (V, E)$ with the vertex order \prec and an assignment of all of the edges E into two disjoint subsets $S \cap Q = \emptyset, S \cup Q = E$, then we can validate that no edges cross on the stack page S and that no edges nest on the queue page Q with the help of a stack and queue as data structures.

Figure 2.1 shows an example of how this process can look like and what actions need to be executed at every vertex. We start at the first vertex in the vertex order which is labelled as 1. On a stack page, the longest edge incident to 1 is added to the data structure first. This is in contrast to a queue, where we would need to add the shortest edge first. Therefore we add the edges $(1, 4)$, $(1, 3)$ and $(1, 2)$ to the stack and all actions for the first vertex are done. At vertex 2 we need to remove $(1, 2)$ which is on the top of the stack. At vertex 3 where edges need to be removed and added we start with removing edges. At the last vertex, we can remove all remaining edges. The LIFO principle was not violated in this process. Therefore it is a valid stack page with no crossing edges. A pseudo-code of this procedure is given in Algorithm 2.1.

In a similar way that we can validate a stack page, we can also validate a queue page. An example is shown in Figure 2.2. Again we start at the first vertex in the order which is labelled with 1, but this time we add the edges the other way around starting with the shortest edge so that they can be removed later in the correct order when they need to leave the queue. For a queue it does not make a difference if we first enqueue or dequeue the edges of a vertex and therefore the order of operations at the second vertex does not matter. However we maintain the convention used before concerning stacks to first remove all edges before adding new ones. The remaining edges at vertex 3 and 4 can be dequeued without violating the FIFO principle of a queue. Therefore we have a valid queue page without a nested edge. The pseudo code for this process is given in Algorithm 2.2.

Algorithm 2.2: validateQueuePage

Data: List of vertices V , edges E

Result: *True* if E is a valid queue page without a nesting with respect to the order of the vertices in V and otherwise *False*

```
1 queue  $\leftarrow$  Queue()
2 foreach  $v$  in  $V$  do
3   | startEdges  $\leftarrow$  edges that start at  $v$  ordered by shortest first
4   | endEdges  $\leftarrow$  edges that end at  $v$  ordered by longest first
5   | foreach  $e$  in endEdges do
6   |   | if  $e \neq$  queue.dequeue() then
7   |   |   | return False
8   |   | foreach  $e$  in startEdges do
9   |   |   | queue.enqueue(e)
10 return True
```

CHAPTER 3

Theory

In this chapter, we have a look at the theoretical part of mixed layouts. First, we show complexity and NP-hardness results for layouts with various amount of pages and with and without a fixed vertex order. After that, we introduce algorithms for 1-stack 1-queue layouts. Finally, discuss a conjecture of Pupyrev [Pup17] in which he suggests that every planar bipartite graph has a 1-stack 1-queue layout.

3.1 Complexity

The difficulty of determining if a graph permits a mixed linear layout depends strongly on the number of stack and queue pages of the layout. In the following, we study the complexity of this problem in the case that the vertex order is fixed or without a given vertex order.

3.1.1 Fixed vertex order

The problem of finding a mixed layout with a fixed vertex order can be formulated as follows. Given a simple graph G , an order of the vertices \prec , and two integers $s, q \in \mathbb{N}_0$ does a s -stack q -queue layout exist for G ? An overview of the results for different s, q can be seen in Table 3.1.

For an $s = 1, q = 0$ the problem is trivial and can be solved in linear time. Since all edges are on a single stack page, it only needs to be checked if any two edges are crossing and there are various methods to do so. For example, this can be done by pushing and popping edges to a stack as described in Chapter 2 or with any state of the art algorithm that is used to count crossings in book embeddings. One such algorithm is described by Six and Tollis [ST06] which they call *CountAllCrossings*. This algorithm takes $\mathcal{O}(m + \chi)$ time where m is the number of edges and χ is the total number of crossings. Since we

Table 3.1: Complexity classes and NP-hardness for linear layouts with a fixed vertex order. The three dots denote that the previous class or hardness continues forever into this direction. The star symbol denotes new results in this thesis.

	0-stack	1-stack	2-stack	3-stack	4-stack	5-stack
0-queue		P	P	?	NP-hard	...
1-queue	P	P			NP-hard*	...*
2-queue	P				NP-hard*	...*
3-queue*	...*

only need to check if there is at least one crossing, the algorithm can be adapted to stop after the first crossings is found which results in a worst-case running time of $\mathcal{O}(m)$.

For $s = 0, q \geq 1$ the problem can be solved in polynomial time no matter the number of queue pages. Assume all edges would be on a single page and there is no rainbow with more than q edges, then G permits a q -queue layout on the given vertex order [HR92]. The opposite is also true. If there is a rainbow with more than q edges then it is impossible to have a q -queue layout because every edge of a rainbow needs to be assigned to a different page otherwise two edges would be nested.

For the embedding with two stack pages $s = 2, q = 0$ this can be solved in linear time by planarity testing [Pat13]. For the planarity testing, the edges are added step by step from the first vertex in the vertex order to the last vertex and checked if this can be done in a planar way.

For $s = 1, q = 1$ the problem can be solved in polynomial time by formulating the problem as an instance of 2-SAT as described in Section 3.2. Creating the 2-SAT formula needs quadratic time to find all conflicts between all pairs of edges and 2-SAT itself can be solved in linear time.

For book embeddings with $s = 3, q = 0$ it is unclear whether the problem is NP-hard or solvable in polynomial time. To the best of our knowledge no NP-hardness proof exists. Finding a book embedding can be done by finding a colouring of the edges in the circle graph. If no two edges of the same colour cross then all edges of one colour can be put on a page without crossings. Unger [Ung92] claimed that it could be done in polynomial time for three colours without describing the algorithm in detail. This circle colour problem is proven to be NP-hard for four or more colours [Ung88]. Therefore for $s \geq 4, q = 0$ finding a linear layout with a fixed vertex order is also NP-hard.

To the best of our knowledge, this is a complete description of all results in the literature. We show two theorems that prove that by either increasing s or q for an already NP-hard problem the new problem is also NP-hard.

Theorem 1. *Let Π be an NP-hard decision problem that takes as input a graph $G = (V, E)$ and a linear order of the vertices \prec and asks if an assignment of the edges to s stack*

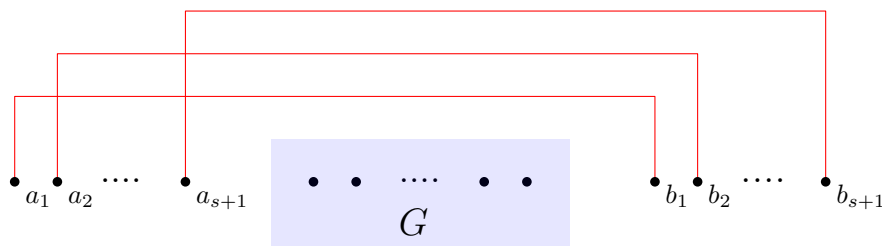


Figure 3.1: The construction of Π' which illustrates that by increasing the number of queue pages by one for an NP-hard linear layout problem with a fixed vertex order the new problem is also NP-hard.

pages and q queue pages exists, where $s, q \in \mathbb{N}$ such that it is a valid s -stack q -queue linear layout. Then the problem Π' with s stack pages and $q + 1$ queue pages is also an NP-hard problem.

Proof. We prove this theorem with a reduction from Π to Π' . The idea of this reduction is to add vertices and edges to the graph in such a way that the additional queue page of Π' contains an edge that prevents all other edges of E from being assigned there, and therefore all edges of E must be assigned to s stack and q queue pages.

Let $G = (V, E), \prec$ be an arbitrary instance of Π . We construct a new set of vertices $V' = V \cup V_a \cup V_b$ where $V_a = \{a_1, a_2, \dots, a_{s+1}\}$ and $V_b = \{b_1, b_2, \dots, b_{s+1}\}$. We also construct a new set of edges $E' = E \cup E_{ab}$ where $E_{ab} = \{(a_1, b_1), (a_2, b_2), \dots, (a_{s+1}, b_{s+1})\}$. Finally, we construct a vertex order \prec' as follows. For every two vertices $u, v \in V'$ we set $u \prec' v$ if one of the following statements is true:

- 1) $u_i, v_j \in V_a$ and $i < j$
- 2) $u \in V_a, v \in V$
- 3) $u, v \in V$ and $u \prec v$
- 4) $u \in V, v \in V_b$
- 5) $u_i, v_j \in V_b$ and $i < j$

An illustration of this construction can be seen in Figure 3.1. We show that Π' with the input $(G' = (V', E'), \prec')$ is a yes instance if and only if $G = (V, E), \prec$ is a yes instance of Π .

If G with \prec has a valid mixed linear layout on s stack and q queue pages, then we can place all the new edges E_{ab} that we added between the two sets of vertices V_a and V_b on the new queue page. Since the edges in E_{ab} form an $s + 1$ -twist, all of the edges are crossing each other and no edge is nested. Therefore it is possible to place all of them on a single queue page. If G' with \prec' has a valid mixed linear layout on s stack and $q + 1$ queue pages, then we can find mixed layout for G with \prec on s stack and q queue pages

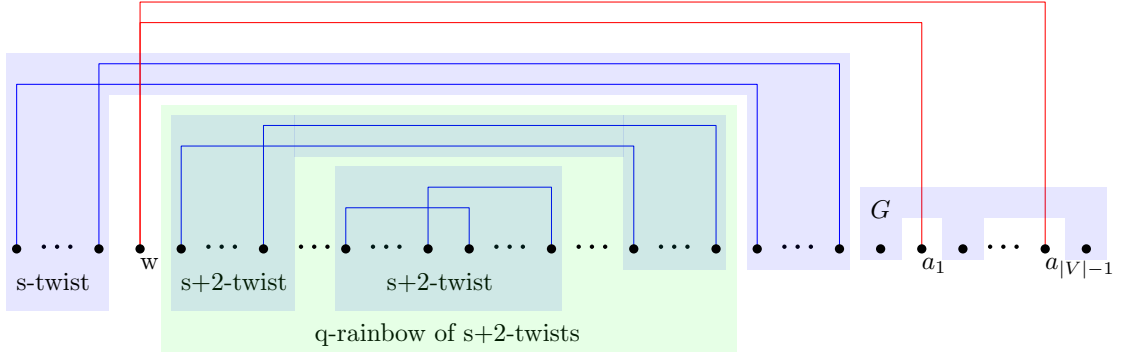


Figure 3.2: The construction of Π' which illustrates that by increasing the number of stack pages by one for an NP-hard mixed linear layout problem with a fixed vertex order the new problem is also NP-hard.

by removing V_a and V_b with the adjacent edges E_{ab} . This leaves the remaining edges assigned to no more than q queue pages.

Since all of the edges in E_{ab} are crossing each other and we have in total $s + 1$ edges and only s stack pages we need to put at least one of these edges on a queue page. Also, because every edge in E_{ab} is nesting all edges in E , not a single edge in E can be assigned to the same queue page. Therefore if no valid linear layout exists for Π with G, \prec it is also not possible to find a valid linear layout for Π' with G', \prec' . \square

Theorem 2. *Let Π be an NP-hard decision problem that takes as input a graph $G = (V, E)$ and a linear order of the vertices \prec and asks if an assignment of the edges to s stack pages and q queue pages exists, where $s, q \in \mathbb{N}$ such that it is a valid s -stack q -queue linear layout. Then the problem Π' with $s + 1$ stack pages and q queue pages is also an NP-hard problem.*

Proof. Similar to the proof for Theorem 1 we show a reduction from Π to Π' where we add additional vertices and edges in a way that some edges are forced to be assigned to the new stack page that prevent any other edge from E to be assigned there too. This time we need a slightly more complex construction, but proving that this reduction is correct is as simple as before. The final construction is illustrated in Figure 3.2.

In our new vertex order \prec' we keep the original order of the vertices in V . Therefore for all vertices $u, v \in V$ it holds that $u \prec' v$ if $u \prec v$. Then we create a new set of vertices $V_a = \{a_1, a_2, \dots, a_{|V|-1}\}$ and we place one of these vertices between every vertex in \prec' . Therefore the order \prec' contains alternating vertices of the sets V and V_a . We then add a vertex w and a set of edges $E_a = \{(w, a_1), (w, a_2), \dots, (w, a_{|V|-1})\}$ and place w before any other vertex that is currently in the order such that $w \prec' u$ for all $u \in V \cup V_a$. The goal of the rest of this reduction is to force all edges in E_a to be assigned to the same stack page. This prevents any edge in E from being assigned to this stack page because between

every two vertices in V there is one vertex of V_a and a crossing would be guaranteed. Therefore, only s stack pages are left for the edges in E .

To prevent any of the edges in E_a to be assigned to a queue page we need to ensure that there is q -rainbow between the vertices w and a_1 . However, since the edges of a rainbow can be assigned without a problem to a single stack page, we need to ensure that each edge of the rainbow is assigned to a queue page. Therefore we construct a q -rainbow with $2q$ new vertices and then replace every pair of vertices of this rainbow that is connected with an edge with an $s + 2$ -twist where each twist introduces $2(s + 2)$ new vertices. We call this construction a q -rainbow of $s + 2$ -twists. Let V_b be the set of vertices and E_b the set of edges of this construction. It holds that $w \prec' u \prec' v$ for all $u \in V_b$ and for all $v \in V$. Since we have only $s + 1$ stack pages available there is at least one edge per twists that need to be assigned to a queue page. Moreover, since we have q such twists that are all nested this construction forces at least one edge to be assigned to every available queue page. Therefore no edge in E_a can be assigned to any queue page.

In the last step of this reduction we want to ensure that all edges in E_a are assigned to the same stack page. To achieve this we add a new set of $2s$ vertices V_c with s edges E_c that form a s -twist. Let $V_{c1} \subset V_c$ be the set of left vertices and $V_{c2} \subset V_c$ the set of right vertices of this s -twist such that $L(e) \in V_{c1}$ and $R(e) \in V_{c2}$ for all $e \in E_c$. We want to place V_{c1} in the vertex order before w and V_{c2} between the q -rainbow of $s + 2$ -twists and the vertices of V . This can be formally expressed as the following. For every two vertices u and v we set $u \prec' v$ if:

- 1) $u \in V_{c1}$ and $v = w$
- 2) $u \in V_b, v \in V_{c2}$
- 3) $u \in V_{c2}, v \in V$

If G with \prec has a valid mixed linear layout on s stack and q queue pages then we can place all new vertices and edges in a way that we have a valid mixed linear layout for G' with \prec' on $s + 1$ stack and q queue pages. We place all edges E_b of the q -rainbow of $s + 2$ -twists on the q available queue pages. All edges of a twist are assigned to the same queue page, and each twist is assigned to another queue page. Since all vertices of this rainbow are placed in the vertex order \prec' before the vertices of V and therefore they are all disjoint this assignment that does not affect the edges E at all. Then we place each edge in E_c of the s -twist on an own stack page. Again all vertices in V_c are placed in \prec' before the vertices of V . Finally, we can place all edges of E_a to the stack page with number $s + 1$. Therefore if G with \prec has a valid mixed linear layout, we can have same order \prec for the vertices of V and the same assignment of the edges E on s stack and q queue pages, and we have a valid linear layout.

If G' with \prec' has a valid mixed linear layout on $s + 1$ stack and q queue pages, then we can find mixed layout for G with \prec on s stack and q queue pages by removing all the vertices and edges that were added in this reduction. This leaves the remaining edges

Table 3.2: Complexity classes and NP-hardness for linear layouts without a given vertex order. The three dots denote that the previous hardness continues forever into this direction. The star symbol denotes new results in this thesis.

	0-stack	1-stack	2-stack	3-stack
0-queue		P	NP-hard	...
1-queue	NP-hard		NP-hard*	
2-queue	NP-hard*			

assigned to no more than s stack pages because all the edges in E_a were assigned to the same stack page which is then an empty page.

If G with \prec does not have a valid mixed linear layout for s stack and q queue pages, then it is also not possible for G' with \prec' to have one on $s + 1$ stack and q queue pages. All the edges in E_a need to be on a stack page because they would nest the edges E_b of the q -rainbow of $s + 2$ -twists and there are only q queue pages available. Also, all of them need to be on the same stack page because otherwise, they would cross edges E_c in the k -twist. Moreover, because between every vertex in V there is a vertex $a \in V_a$ with an edge (w, a) , it is not possible for an edge in E to be assigned to the same stack page as the edges of E_a because they would cross. Therefore all edges of E must be assigned to s stack and q queue pages. If this is not possible for Π with G and \prec it is also not possible for Π' with G' and \prec' to have a valid linear layout. \square

3.1.2 Free vertex order

Now that we have seen the complexity of finding linear layouts with a fixed vertex order we move on to finding such layouts where the vertex is free to choose. The problem can be formulated as follows. Given a simple graph G and two integers $s, q \in \mathbb{N}_0$ does a valid s -stack q -queue layout exist for G with some vertex order \prec ? An overview of the results for different s, q can be seen in Table 3.2.

For $s = 1, q = 0$ this can be done in linear time. Outerplanar graphs are precisely the graph class that fit on a single stack page. Therefore a 1-stack book embedding is possible if and only if the graph is outerplanar [BK79]. Recognising such a graph can be done in linear time [Wie86]. Construction the vertex order is done by taking the exact order of the vertices in which they lie on the outer face.

For $s = 2, q = 0$ we already face an NP-hard problem. A 2-stack book embedding is possible if and only if the graph is planar and subhamiltonian [BK79]. The process of deciding if a graph is Hamiltonian is an NP-complete problem [GJT76]. Therefore deciding if a graph has a 2-stack linear layout is also NP-complete [CLR87].

Finally for $s = 0, q = 1$ this is also an NP-complete problem. Arched level-planar graphs are precisely the graph class that fit on a queue page. Therefore a graph permits a

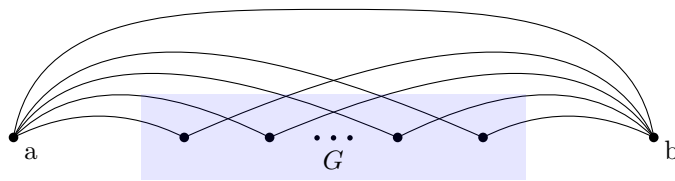


Figure 3.3: The construction of Π' that illustrates that deciding if a linear layout with two queue pages exists is NP-hard.

1-queue layout if and only if it is in this graph class. Recognizing arched level-planar graphs is an NP-complete problem [HR92].

The comparison between the difficulty to find layouts with and without a given vertex order highlights another interesting relationship between stack and queues that was already noticed by Heath and Rosenberg [HR92]. Queues appear to be simpler than stacks when the order is fixed. Queue layouts can be found in polynomial time regardless of the number of queue pages while the problem is for stack layouts NP-hard for at least four or more pages. In contrast with a free vertex order, stacks appear to be simpler than queues since it is possible to find a 1-stack layout in polynomial time whereas the same problem for a 1-queue layout is already NP-hard.

To the best of our knowledge there are no results in the literature for $s = 0, q = 2$ and $s = 2, q = 1$ and we show NP-hardness for these problems with the following two theorems.

Theorem 3. *Deciding whether or not a graph $G = (V, E)$ has a 2-queue linear layout is NP-hard.*

Proof. For $q = 1$ this has been proven by Heath and Rosenberg [HR92]. For $q = 2$ we take a problem Π with one queue page and reduce this to a problem Π' with two queue pages. Let $G = (V, E)$ be an arbitrary instance of Π . We add two new vertices a, b and add an edge from every vertex in V to both of the new vertices and between the two vertices itself such that $E_a = \{(u, v) \mid \forall u \in \{a, b\} \forall v \in E\} \cup \{(a, b)\}$. Then we can construct a graph $G' = (V', E')$ where $V' = V \cup \{a, b\}$ and $E' = E \cup E_a$ which is the input for Π' . This construction can be seen in Figure 3.3. Note that the vertex order is free and it is not forced that a and b are placed on both ends of the order.

If G has a valid linear layout on one queue page we can place a on the first position and b on the last position in the vertex order, as it is shown in Figure 3.3 and assign all edges of E_a to a single queue page. None of the edges in E_a are nested and G' has a valid linear layout on two queue pages.

If G has no valid linear layout on one queue page, that means that for every possible vertex order there must exist a set of edges $E_r = \{e_1, e_2\} \subseteq E$ such that these edges form a 2-rainbow. Let e_1 be the edges that nests e_2 . If there would exist at least one vertex order without such a set of edges, then G would have a valid linear layout. A complete

graph with n vertices has a queue number of $\lfloor n/2 \rfloor$ [HR92]. The complete graph with four vertices K_4 is the smallest graph that needs two queue pages. If G is a K_4 then G' is a K_6 which has a queue number of three. Therefore we can assume that G has at least five vertices because otherwise, it is trivial.

If a and b are placed in the vertex order such that $L(e_1) \prec a \prec R(e_1)$ and $L(e_1) \prec b \prec R(e_1)$ it is easy to see, that we get a 3-rainbow. Either the edge (a, b) nests e_2 or (up to renaming a and b) $(a, L(e_2))$ and $(b, R(e_2))$ are nested. Therefore a 3-rainbow of either $e_1, (a, b), e_2$ or $e_1, (a, L(e_2)), (b, R(e_2))$ is unavoidable.

Since by placing both a and b outside of the rainbow E_r , the edge (a, b) would nest e_1 . Because of this, the only remaining case that we need to verify is if one of the two vertices is placed outside of the rainbow E_r and the other one is placed somewhere inside E_r . Let a be the vertex outside and b the vertex inside. Since a has an edge that stretches over the whole vertex order to some vertex $v \in V$ such that v has no successor, the remaining minimum four vertices of V and b are not allowed to have single nesting because this would result in a 3-rainbow. Since b has edges to every vertex in V this restricts the possible remaining edges drastically. Therefore, every remaining edge must either cover b or be adjacent to v . There can be at most two different vertices that can have edges that cover b because otherwise, those edges would nest among themselves. However, such a graph with the remaining edges where each edge is adjacent to one of two vertices always has a 1-queue layout, and therefore G must have a 1-queue layout.

If G' has a valid linear layout on two queue pages, then we can find mixed layout for G on one queue page. Either a and b are placed on both ends of the vertex order and then removing all of the edges E_a immediately yields a one queue layout for G which has the same vertex order for the vertices V . Otherwise, if one of the vertices a, b is not placed on an end of the vertex order we have already seen that the edges of G are restricted in a way that makes it trivial to find a one queue layout.

□

Theorem 4. *Deciding whether or not a graph $G = (V, E)$ has a 2-stack 1-queue linear layout is NP-hard.*

Proof. Let Π be the decision problem that asks if a graph has a 2-stack linear layout. We prove this theorem by a reduction from Π to the problem Π' that asks if a graph has a 2-stack 1-queue linear layout. Let $G = (V, E)$ be an arbitrary but connected instance of Π . If G is not a connected graph, then it would be possible to calculate a linear layout for each independent part and chain the results together. Similar to the proof of Theorem 3 we add edges and vertices to this graph to create a new graph $G' = (V', E')$ that is an instance of Π' . The idea of this reduction is again that one of the new edges is forced to nest all edges in E such that G' is a yes instance if and only if G is also a yes instance.

The K_8 is the largest complete graph that admits a 2-stack 1-queue layout. An example of such a layout can be seen in Figure 3.4. There are many slightly different ways how

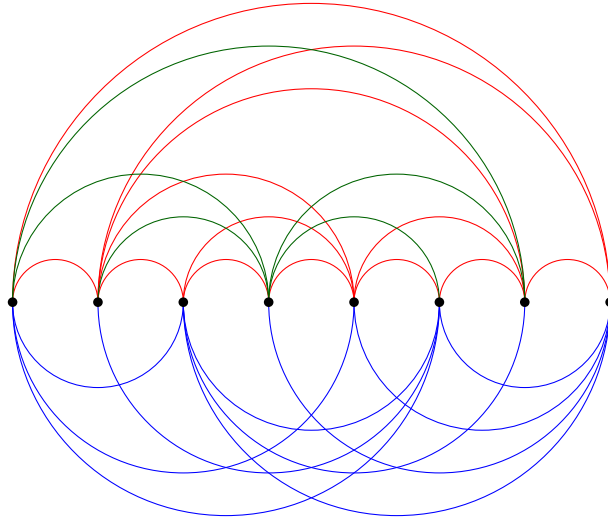


Figure 3.4: The K_8 is the largest complete graph that fits on a 2-stack 1-queue layout. The edges on the stack pages are shown in red and green while the edges on the queue page are shown in blue.

the edges can be assigned to the three pages, but with an exhaustive search where we computed every possible way, we found the Lemma 1 to be always true.

Lemma 1. *For every 2-stack 1-queue layout of a K_8 some edges are forced to be assigned to specific pages. Since the order does not matter for complete graphs, we can without a loss of generality assume that the vertices are in an ascending order from 1 to 8. The edge $(1, 8)$ is assigned to a stack page and we call this page the first stack page. Either $(1, 7)$ or $(2, 8)$ is assigned to the second stack page. Therefore either the vertex 2 or 7 is not covered by an edge on the second stack page. The edges $(1, 3)$ and $(6, 8)$ are always assigned to the queue page.*

Observation 1. *For any graph that has a stack layout, we can freely choose the first vertex in the order. This is true because for any stack layout it is possible to move the first vertex to the last position and it is still a valid stack layout. The same is true for the converse operation of moving the last vertex to the first position and it is still a valid stack layout. These operations can be repeated indefinitely, and therefore any vertex can be in the first position.*

A graphical representation of Observation 1 would be to assume that the vertices are drawn on the boundary of a circle and the edges of a page are either drawn inside or outside of the circle. This circle can be cut at any place between two vertices and the resulting line after the cut will be a valid vertex order.

We construct G' from G as following. Due to Observation 1, we pick an arbitrary vertex $a \in V$, add seven new vertices and add edges so that these eight vertices form a K_8 . We

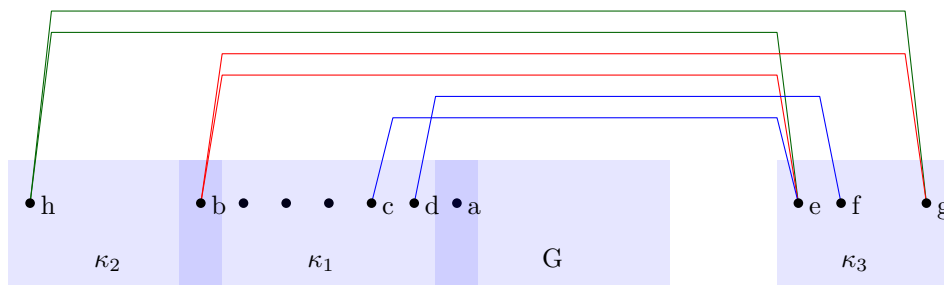


Figure 3.5: The construction of the reduction that illustrates that finding a 2-stack 1-queue layout is NP-hard.

call this complete graph κ_1 . Then we pick a vertex $b \in \kappa_1, b \neq a$ and add seven new vertices and edges such that this construction forms also a K_8 and we call it κ_2 . Then we construct the last K_8 by introduction eight new vertices and the respective edges which we call κ_3 . We choose two vertices $c, d \in \kappa_1, c \neq a, c \neq b, d \neq a, d \neq b$ and three vertices $e, f, g \in \kappa_3$ and add the edges $E_1 = \{(c, e), (d, f)\}$. Finally we choose a vertex $h \in \kappa_2, h \neq b$ and add the edges $E_2 = \{(b, e), (b, g)\}, E_3 = \{(h, e), (h, g)\}$.

An illustration of this construction where G' is a yes instance if G is also a yes instance can be seen in Figure 3.5. All the edges of E can be assigned to two stack pages. The edges E_1 is assigned to the queue page while E_2 can be assigned to one stack page and E_3 to the other stack page. The remaining edges of the three complete graphs $\kappa_1, \kappa_2, \kappa_3$ can be assigned to the three pages as it is shown in Figure 3.4.

If G' has a valid linear layout on two stack and one queue pages, then we can find mixed layout for G on two stack pages. Since none of the edges E can be assigned to the queue page because they would be nested, all of these edges are already assigned to two stack pages anyway. Therefore the same page assignment with the same vertex order results in a two stack layout of G .

Finally, we show that if G has no linear layout on 2-stack pages, then G' cannot have a linear layout on 2-stack and 1-queue pages. Without loss of generality we assume the vertices to be ordered in the direction that we argue. Of course, it is possible to have the whole vertex order reversed but the arguments can be made in the same way for the other direction.

It is important to notice that for a K_8 there are only three vertices that can have an edge on a stack page to a vertex that is not part of the K_8 . The first and the last vertex in the order are two of them. Due to Lemma 1 either the second or next to last vertex is the only vertex that can have such an edge on the second stack page. Since we added for κ_1 edges to four different vertices, there must be one which is covered by edges of both stack pages and therefore forced to be on the queue page. Because of Lemma 1 there is always one vertex that can have such an edge to a vertex that is not part of the K_8 without nesting another queue edges in this K_8 . For κ_1 this must be one of the two edges of E_1 .

Every algorithm that computes a vertex order for our construction needs to place all vertices of a K_8 next to each other in the vertex order without any other vertex in between. A K_8 can not partially overlap with another K_8 because there would be crossings and also a K_8 cannot be placed between two vertices of another K_8 because there would be nestings. Therefore in the vertex order \prec' it holds without loss of generality that $c_2 \prec' c_1 \prec' a$ where $c_2 \in \kappa_2, c_1 \in \kappa_1$.

The next step to show is that κ_1 and κ_2 cannot be placed between a and another vertex $v \in V$. If this would be the case, then we would also need to place κ_3 between these vertices because there are edges on all three pages that connect κ_3 with κ_1 and κ_2 . Then one of the queue edges in E_1 would nest edges in κ_2 .

Since we now have established the placement of the vertices in κ_1, κ_2 and V it is left to show that there is only one place where we can put κ_3 . The only suitable placement is after the vertices of V such that $c_2 \prec' c_1 \prec' v \prec' c_3$ where $c_2 \in \kappa_1, c_1 \in \kappa_1, v \in V, c_3 \in \kappa_3$ as it is shown in Figure 3.5. We cannot place $c_3 \prec' c_2$ because one of the queue edges in E_1 would nest edges in κ_2 . We also cannot place κ_3 somewhere between two vertices of V because κ_3 has edges on both stack pages to the vertices h and b . In order that these edges on the stack page do not cross edges in E , it would be necessary that at least one edge in E is assigned to the queue page, but this is not possible since this edge would nest edges of κ_3 . Therefore the only way to place the vertices (except for the reversed version) is shown in Figure 3.5 and all of the edges in E must be assigned to the two stack pages since an edge on the queue page would be nested. Therefore if G is a no instance of Π , then G' must be a no instance of Π' .

□

3.2 Algorithms

In this section, we present algorithms to find mixed layouts for graphs with a fixed vertex order if such a layout exists.

Finding a mixed layout for a graph can be difficult because there is no graph class known that coincides with a 1-stack 1-queue layout in the sense that every graph in this class admits such a mixed layout and every 1-stack 1-queue layout contained this class. For 1-stack it is outer-planar graphs, and for 1-queue layouts, it is arched level-planar graphs. For a 1-stack 1-queue layout such a characterisation is missing.

Determining if a graph $G = (V, E)$ with the vertex order \prec permits a 1-stack 1-queue layout and creating such a layout if it exists can be done efficiently by transferring this problem into an instance of 2-SAT. Testing for satisfiability as well as finding a truth assignment if it exists for an instance of 2-SAT can be done in linear time as Aspvall et al. [APT79] showed. To do so, we create for every edge $e \in E$ a boolean variable e' . If e' is set to *false*, then this means that the edge is assigned to the queue page and if it is set to *true*, then it is assigned to the stack page. Then for every pair of edges $(e, f) \in E \times E$ if e and f would cross each other if both were assigned to the stack page, we create the

clause $(\neg e' \vee \neg f')$. Therefore, to satisfy the formula either e' or f' must be *false* and therefore at least one of the edges must be assigned to the queue page. Similarly, if e and f would nest we create the clause $(e' \vee f')$ to ensure that at least one of the edges is assigned to the stack page.

Since we can formulate our mixed linear layout problem as an instance of 2-SAT, other algorithms that are used for 2-SAT can help us with these layouts. For example, Lewin et al. [LLZ02] described a method to approximate the MAX-2-SAT problem with an approximation ration of at least 0.94. Therefore, for graphs that do not permit a 1-stack 1-queue layout and therefore do not have a satisfying assignment for the 2-SAT formula, we can at least approximate a good solution that is close to the minimum number of conflicts. Since every unsatisfied clause in the 2-SAT formula is causing exactly one conflict, the number of unsatisfied clauses equals the number of conflicts. Due to this, the solution provided by Lewin et al. is indeed quite close to an optimal solution.

3.3 Planar bipartite graphs

In a recent paper, Pupyrev [Pup17] proved that there are planar graphs that do not admit a 1-stack 1-queue layout and that every planar graph has a 1-stack 1-queue subdivision with one division vertex per edge. Since he noticed in his counterexamples that faces of degree three were an important factor for non-embeddability and a subdivision with one vertex per edges is bipartite, he made the following conjecture:

Conjecture 1 ([Pup17]). *Every bipartite planar graph admits a mixed 1-stack 1-queue layout.*

This conjecture is a current and valuable problem to base this thesis on. In the following section, we give a lower bound on the size of planar bipartite graphs that admit such a mixed layout and also list forbidden bipartite subgraphs.

In order to find either a counterexample for this conjecture or a lower bound for the minimum number of vertices that always admit a 1-stack 1-queue layout, we tested all edge maximal planar bipartite graphs from 4 to 19 vertices. We generated the planar bipartite graphs for our test with the program Plantri 5.0 described in [BM99]. Plantri allows a user to quickly create a large set of isomorphism free graphs of various planar graph classes. Therefore we were able to generate all planar bipartite graphs for a given number of vertices. In his paper, Pupyrev also referred to a tool that he created which allows testing if a graph admits a specific linear layout by creating a respective SAT formula that has a solution if the graph admits such a layout. We used this tool and a modern SAT solver to test the graphs that we generated with Plantri. We did not use the 2-SAT formula approach because it was much slower to test the permutations of the vertex order than having one bigger SAT formula that tests for all vertex orders at once. We run the tests on multiple Linux (Ubuntu 16.04.6 LTS) machines where each has two Intel Xeon E5540 (2.53 GHz Quad Core) processors and 24GB RAM.

Table 3.3: All maximal planar bipartite graphs up to 19 vertices admit a 1-stack 1-queue linear layout

Vertices	Graphs	1-stack 1-queue layouts
4	1	1
5	1	1
6	2	2
7	3	3
8	9	9
9	18	18
10	62	62
11	198	198
12	803	803
13	3.378	3.378
14	15.882	15.882
15	77.185	77.185
16	393.075	393.075
17	2.049.974	2.049.974
18	10.938.182	10.938.182
19	59.312.272	59.312.272

Table 3.3 shows that all planar bipartite graphs up to 19 vertices admit a 1-stack 1-queue layout. This result is not very surprising since Pupyrev showed in the same paper in a similar way that all planar graphs up to 18 vertices permit a 1-stack 1-queue layout. The exponential increase of the possible number of graphs and the rising time that was needed to solve the SAT formulas made it difficult to test for graphs with 20 or more vertices exhaustively.

For a complete bipartite graph $K_{a,b}$ it is also interesting to see which of them allow a 1-stack 1-queue mixed layout. For $a = 3$ it is always possible to have such a layout. Two of the three vertices need to be placed on the ends of the vertex order, and the adjacent edges are assigned to the queue page. The third vertex can be placed anywhere, and those edges are assigned to the stack page. The $K_{4,4}$ is the largest complete bipartite graph for that we could compute a mixed layout for.

Design and evaluation of heuristics

The goal of this chapter is to develop and test heuristics that can be used to calculate mixed layouts that have a low number of conflicts. The number of conflicts is the sum of all crossings on the stack pages plus the sum of all nestings on the queue pages. We focus mainly on 1-stack 1-queue layouts, but in our experiments, we also include tests where the number of stack and queue pages reaches up to five.

To the best of our knowledge, no specific heuristics for mixed layouts exist yet, but for stack layouts, there has been much work done for the so-called book embeddings or book drawings. Naturally, it is an interesting question if we can reuse the same principles and apply them for the mixed layouts. Usually, construction heuristics consist of two steps. First, creating a vertex order and then creating an order of the edges in which they are assigned greedily to the currently best page. Some heuristics do both steps at once so that when a vertex is placed on the spine, then the adjacent edges are immediately assigned to pages. After that, optimisation heuristics can be applied that try to improve a given layout step by step by minimising conflicts. A good overview of such heuristics and extensive experiments about the best heuristics for various graph classes for book drawings was published by Klawitter [Kla16] and Klawitter et al. [KMN17]. We believe that the work of Klawitter contains state-of-the-art heuristics and we use those for the mixed layouts too. We test how some of the best heuristics for book embeddings perform for mixed layouts, and we introduce a new page assignment heuristic which is compared to the existing heuristics.

Until now we did not allow any crossings on stack pages and no nesting on queue pages as it was defined in Chapter 2. In this chapter, we allow such conflicts because the heuristics try to minimise the total number of conflicts and they do not try to find the minimum number of pages such that there is no conflict at all.

4.1 Existing heuristics

Vertex order heuristics compute an order of the vertices based on different criteria without yet assigning the edges to pages. The following heuristics deliver good results for stack layouts, and we include them for the experiments on the mixed layouts.

Random depth-first search (*randDFS*): The first vertex is picked randomly, and at every step of the depth-first search the next vertex to go into depth is chosen randomly. The order in which the vertices are visited is the vertex order [BSC⁺08].

Smallest degree depth-first search (*smlDgrDFS*): Similar to *randDFS* but the first vertex is randomly chosen by one of the vertices with the smallest degree of the graph, and at every step, the next vertex to go into depth is the one with the smallest degree. He and Sýkora [HS04] introduced this idea.

Random breadth-first search (*randBFS*): The first vertex for the breadth-first search is picked randomly, and the order of the vertices on each layer is also chosen randomly. The vertex order is the order in which the vertices are visited [SSS13].

Tree-based breadth-first search (*treeBFS*): With the help of a breadth-first search, a spanning tree of the graph can be created. Trees have a stack number of one, and such a vertex order seems a good starting point because it allows all edges of the spanning tree to be assigned to one stack page without crossings. This heuristic was introduced by Klawitter [Kla16].

Greedy connectivity-based (*conGreedy*): Baur and Brandes [BB04] introduced the idea of this heuristic. At every step, it picks the vertex with the most placed neighbours. In case of a tie, one of the vertices with the least unplaced neighbours is chosen. This vertex is placed at the beginning or the end of the spine where it adds the fewest crossings with closed edges. A *closed edge* is an edge where both vertices are already placed on the spine. Klawitter adapted this heuristic so that every position in the spine is tested and we use this adapted version for our tests. For the mixed layouts, we further adapted this heuristic so that we count not only crossings but also nestings and the sum of these conflicts is crucial.

All the depth-first search and breadth-first search based heuristics that we have presented here have an asymptotic running time of $\mathcal{O}(m + n)$. The original connectivity-based heuristic from Baur and Brandes can be implemented in $\mathcal{O}((m + n) \log n)$ while the adaptations of Klawitter raise this to $\mathcal{O}(m^2n)$.

Page assignment heuristics are taking a previously calculated vertex order that can be created by a vertex order heuristic, and assign all of the edges to the pages. Naturally, such an assignment should produce as few conflicts as possible. Since the vertex order is fixed by now, the task of finding a good page assignment for stack layouts is similar to the fixed linear crossings number problem (FLCPN). It is precisely the same as the FLCPN if we have two stack pages. Therefore, techniques and algorithms that are developed for the FLCPN can be used here such as described by Cimikowski [Cim02] [Cim06]. Note that the FLCPN is an NP-hard optimisation problem for two stack pages [MNKF90].

Edge length (*eLen*): The edges are ordered by decreasing length with respect to the linear vertex order. Since long edges are more likely to cause more conflicts, this heuristic aims to assign these edges before the shorter ones [Cim02]. Therefore, given an order $1 \prec 2 \prec \dots \prec n$ the edge $(1, n)$ would be considered the first edge to be assigned to a page. An edge $(a, a + 1)$ would be the shortest possible edge and would be assigned as one of the last edges.

Ceil-floor (*ceilFloor*): Similar to *eLen* the edges are ordered by decreasing length but this heuristic takes a circular layout into account instead of a linear layout [KRSZ02]. Therefore, the edge $(1, n)$ would be one of the shortest possible edges since the vertex order loops from the end to the start. The circular length for an edge (a, b) is defined as $\min(|a - b|, n - |a - b|)$. The idea of this heuristic is that for stack pages the longest edges that span over most of the vertex order are usually causing only a few crossings. The longest possible edge $(1, n)$ can never cross any other edge.

For both *eLen* and *ceilFloor*, finding the edge order takes $\mathcal{O}(m \log m)$ time to sort the edges and $\mathcal{O}(m^2)$ to distribute the edges. The number of pages does not influence the running time because every edge that gets assigned needs to be compared with all already assigned edges and the number of different pages is therefore not significant.

We have presented heuristics that either compute a vertex order or assign edges to pages for a given order. Now we take a look at complete heuristics that compute both together. Such a complete heuristic can either be a combination of a vertex order and a page assignment heuristic, or an entirely independent heuristic on its own.

Combining a vertex order and a page assignment heuristic is a straight forward and also a flexible process. They are just executed after another, and this allows a versatile combination of the different concepts. As we will see in the experiments in Section 4.3 some combinations deliver good results for some graph classes while they perform worse for others. Therefore, choosing the right combination for a given graph is crucial.

Greedy connectivity-based plus (*conGreedy+*): Creating a full heuristic often uses very similar concepts. Usually, a vertex order is computed step by step, and once both vertices of an edge are placed, then the edge is greedily assigned to a page where it has the least amount of conflicts. Due to the immediate edge assignment, such a full heuristic can produce a different vertex order if the assigned edges are considered. This is the case for a heuristic that Klawitter [Kla16] introduced which he called *conGreedy+*. It is based on *conGreedy* that we have seen before, but instead of just computing a vertex order it also greedily assigns the edges. The worst case running time of this algorithm is $\mathcal{O}(m^2n)$. In the experiments that Klawitter did this heuristic was generally successful for stack layouts and therefore we also include this heuristic for our experiments for the mixed layouts. Since *conGreedy+* produces a different vertex order than *conGreedy*, we also use *conGreedy+* as only a vertex order heuristic. This is done by just dropping the page assignment and using another heuristic to compute a new page assignment.

4.2 Data structure heuristic

Data structure (*dataStructure*): In an attempt to create a new page assignment heuristic for linear layouts we came up with our own idea of using the data structures stack and queue to help to keep track of conflicts and possible future conflicts as well. The other page assignment heuristics that are described above order the edges in some way and then they are put to the page where they add the least conflicts. This approach neglects edges that have not been assigned yet. The *dataStructure* heuristic was designed to also consider the open and unassigned edges while efficiently processing the edges to still run in $\mathcal{O}(n^2)$ time.

In the following, we describe how this algorithm works for a 1-stack 1-queue layout and a pseudo code of this is shown in Algorithm 4.1. In Chapter 2 we have seen how we can use a stack and a queue to validate that a given page has no crossings or no nestings. We can use a similar idea for a heuristic that aims to minimise conflicts in a mixed layout. In this case, it is allowed to remove edges if they are not on top of the stack or in front of the queue, but doing so will result in conflicts. The vertices are processed one after another in the order of a precomputed vertex order. Each edge that starts at a vertex is put on the stack and in the queue. We put the edges on both data structures because we do not want to decide yet on which page we will finally place them. We want to see where in the data structure the edge will end up when we need to remove it. This allows us to postpone the page assignment decision to a later point in time when we have more information available.

Once an edge ends at a vertex, we need to decide on which page we place it. At this point, we have more information about the edge at hand. We know how many edges are above the edge on the stack and in front of it in the queue. Each such edge will be a definite conflict if we put both them on the same page. Therefore, we can increase a crossing counter (or a nesting counter) for each edge on top of the edge (or in front of the edge) in case we decide to place the edge on the stack (or queue). Every edge on every data structure has an individual counter to keep track of the possible conflicts.

By processing the closing edges before the opening edges of a vertex and considering that edges which are adjacent to the same vertex can never cross or nest each other, we reach to the following state. Every time we need to decide if we want to assign an edge to the stack or the queue, we already know how many crossings or nestings this adds to the layout by checking the crossing counter and the nesting counter of the edge. We can also estimate all potential crossings and nestings of the edge that will be added in the future by counting the edges that are on top of it on the stack or in front of it in the queue. Then we can use all of this information to decide if it is better to place the edge on the stack or the queue page. The advantage of *dataStructure* compared to the other page assignment heuristics is that we can potentially assign the edge to a page that adds more conflicts now but likely avoids even more conflicts in the future. The other page assignment heuristics are just greedily assigning an edge to the page where it adds the least conflicts right now.

Algorithm 4.1: dataStructure

Data: Ordered list of vertices V , edges E
Result: Partitioning of E into two disjoint sets S and Q

```

1 stack ← Stack()
2 queue ← Queue()
3 foreach  $v$  in  $V$  do
4   startEdges ← opening edges at  $v$ 
5   endEdges ← closing edges at  $v$ 
6   foreach  $e$  in endEdges do
7     potentialCrossings ← count edges in stack without endEdges above  $e$ 
8     potentialNestings ← count edges in queue without endEdges before  $e$ 
9     crossings ← crossing counter of  $e$ 
10    nestings ← nesting counter of  $e$ 
11    Decide if  $e$  should be added to  $S$  or  $Q$ 
        // e.g. if  $crossings + potentialCrossings * 0.5 \leq$ 
        //  $nestings * potentialNestings * 0.5$  then  $S$  else  $Q$ 
12    if  $e$  was added to  $S$  then
13      Increase crossing counter by one for each edge in stack without
        endEdges above  $e$ 
14    else
15      Increase nesting counter by one for each edge in queue without
        endEdges before  $e$ 
16    Remove  $e$  from stack and queue
17  Add startEdges to stack and queue

```

In all our experiments we based this decision on the following formula. We estimated the final number of crossing by multiplying the potential crossings with 0.5 because there is naively a 50% chance that the edge lands on either page and added the definite number of crossings to it. Then we estimated the final number of nestings similarly, and if the estimated final number of crossings was less or equal to the estimated final number of nestings, we added the edge to the stack page and otherwise the to queue page. In the experiments with more than two pages, we use instead of 0.5 the value of one over the total number of pages because this also seems like a very naive way to estimate the probability that the edge ends up on this page. While this approach for the page assignment decision is quite simple, it is also easily possible to replace it with a more complex one that might deliver better results for specific graph classes. One approach would be to change the weights or even introduce new weights in the decision procedure.

Until now we described how *dataStructure* works on 1-stack 1-queue layouts. Adapting this heuristic to calculate linear layouts for a variable amount of pages is quite straight forward, and even stack or queue layouts are possible. Each page needs a separate data structure, each edge needs a conflict counter for each page, and the page assignment

decision part needs to find the best page for a variable number of pages. This makes it also possible to compare *dataStructure* to other heuristics that were either designed or often used for stack layouts. These comparisons are made in the following Section 4.3 where the results of the experiments are shown.

4.3 Experiments

In this section, we test the heuristics that we introduced before on various frequently seen graph classes and determine which heuristic or combination of heuristics yield the best results. For each test, we use a total of 19 different heuristics. These are the six vertex order heuristics (*randDFS*, *smlDgrDFS*, *randBFS*, *treeBFS*, *conGreedy* and *conGreedy+*) each combined with the three page assignment heuristics (*eLen*, *ceilFloor*, *dataStructure*) which results in a list of 18 heuristics that is completed by *conGreedy+*. Remember that we use *conGreedy+* as complete heuristic as well as only as a vertex order heuristic. It is also noteworthy that Klawitter tested some heuristics successfully for book drawings that we did not include in our tests. Those heuristics are specially designed for stack pages and usually try to partition the edges in preferably planar sets. These approaches naturally did not work well for the queue pages.

The experiments in the next subsections are mostly structured in the same way. For each graph class, we recall the fundamental properties of this class such as the edge density, stack number, queue number or the currently best-known bounds on those numbers. Since we are mainly interested in 1-stack 1-queue linear layouts, there is a line chart with the eight best heuristics on this layout over various sizes of the graphs and a box plot diagram of the three best heuristics on the largest graph size. After that, we have a look at the best heuristics for a variable amount of pages ranging from zero to five stack pages and also from zero to five queue pages. For the stack layouts, we compare the results with the work of Klawitter [Kla16] and Klawitter et al. [KMN17].

4.3.1 Planar graphs

The first set of our experiments is based on connected planar graphs. Such graphs with n vertices have up to $3n - 6$ edges, and the stack number is at most four [Yan89]. It is currently unknown if this bound is tight or if three pages are enough for all planar graphs because no planar graph is known that cannot be embedded within three pages [BKZ15]. If the graph is subhamiltonian, the stack number is two [BK79]. For the queue number, it was for a long time unknown if it is bounded by a constant number and very recently it has been shown that such a bound exists [DJM⁺19].

We generated our graphs by randomly creating n points in the two-dimensional space and then calculating the Delaunay triangulation of these points. Such a process generates nearly maximal planar graphs because the outer face is the only face that can be bounded by more than three edges. All other faces are bounded by precisely three edges. For every test, we generated 200 such graphs of the required size of n vertices.

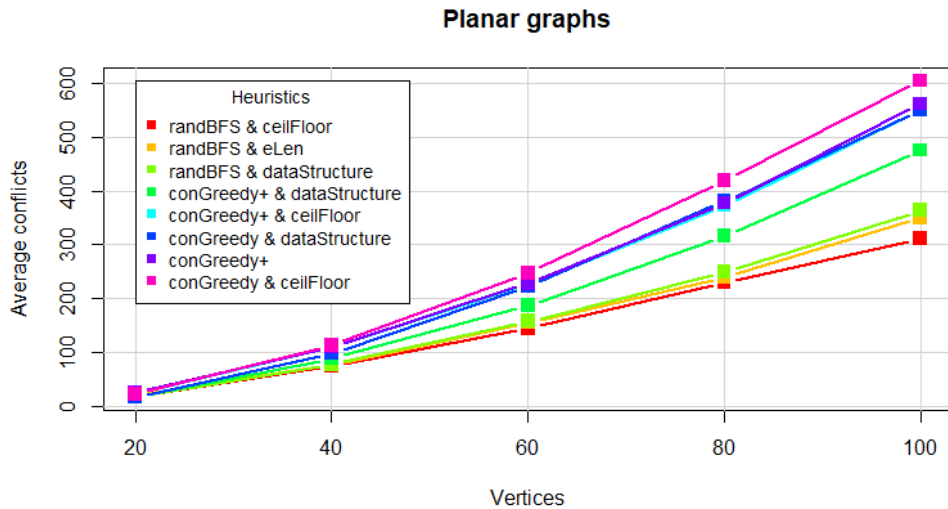


Figure 4.1: The eight best heuristics for a 1-stack 1-queue layout on planar graphs.

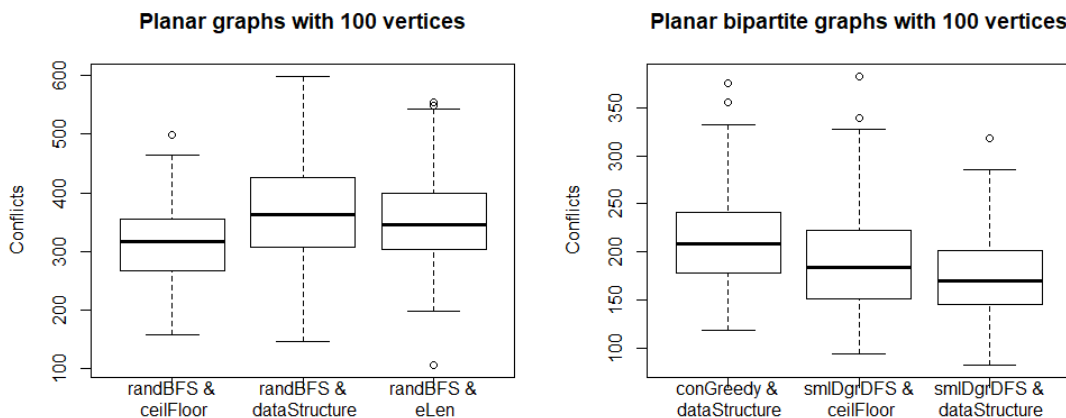







































Figure 4.2: The three best heuristics for planar and planar bipartite graphs in detail.

In Figure 4.1 we can see that the vertex order heuristic has a far more significant impact than the edge assignment heuristic for the planar graphs. The *randBFS* vertex order heuristic delivered way better results than the others regardless of the page assignment heuristic that it was combined. The combinations with *conGreedy+* resulted in about 50% more conflicts, and other heuristics had at least twice as many. Figure 4.2 shows that *eLen* and *ceilFloor* were nearly identically good and *dataStructure* was closely behind those two when they are combined with *randBFS*. We can also see that there is a high variance in the results. The best results had around 150 conflicts for graphs with 100

Table 4.1: The best heuristics for planar graphs for different amounts of stack and queue pages.

	0-S	1-S	2-S	3-S	4-S	5-S
0-Q						
1-Q						
2-Q						
3-Q						
4-Q						
5-Q						

 conGreedy+	 randBFS	 dataStructure	 ceilFloor
 eLen	 conGreedy+	 conGreedy	

vertices while for the same heuristics this can go up to about 500 conflicts.

Table 4.1 shows the results for linear layouts with different amounts of stack and queue pages. Empty spaces in this table indicate that more than one heuristic had on average less than one conflict per graph, and we left this tile out since it seems to be too easy to find a valid layout for so many pages. We can see that *conGreedy+* as only a vertex order heuristic was most successful for layouts with many stack pages. This is consistent with the results of Klawitter, although even better results were made when he combined *conGreedy+* with another page assignment heuristic called ear decomposition. We did not include this heuristic in our tests because it is specially designed for stack layouts and was not usable for queue or mixed layouts. In general, the best combination was *randBFS* together with *eLen*. The *dataStructure* page assignment was the best or close to the best for equally balanced mixed layouts where there is roughly the same number of stack and queue pages. This is a trend that we are also observing in the next experiments. Often the best vertex order heuristic for layouts with mainly stack pages is *conGreedy+*, and *dataStructure* is often the best page assignment heuristic for layouts that are either roughly equal or have only a few pages.

4.3.2 Planar bipartite graphs

The next graph class for which we have a great interest are planar bipartite graphs. They are a commonly seen graph class, and Pupyrev [Pup17] conjectured that they always admit a 1-stack 1-queue linear layout. Therefore, we tested maximal planar bipartite graphs to get a feeling of how well the heuristics can create linear layouts, and how close they come to the conjectured zero conflicts. These graphs have exactly $2n - 4$ edges

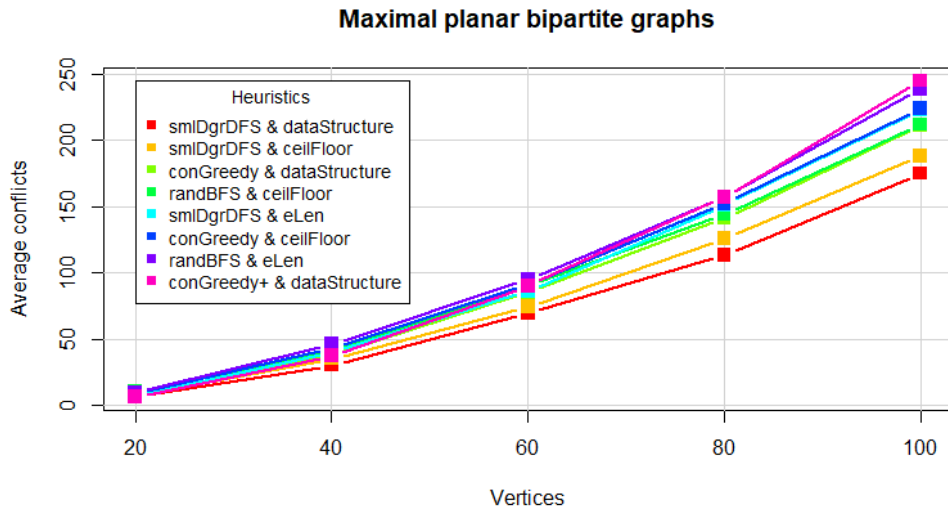


Figure 4.3: The eight best heuristics for a 1-stack 1-queue layout on maximal planar bipartite graphs.



























and are sparser than the planar graphs that we have seen before. Every planar bipartite graph is subhamiltonian, and therefore the edges can be embedded on two stack pages [BK79].






The bipartite graphs that we have generated were all balanced so that both sets of vertices A, B had the same size. Since every planar bipartite graph has a 2-stack embedding, we randomly generated a vertex order of alternating vertices from the sets to ensure that a Hamiltonian path exists. We then randomly selected two vertices of the two sets $a \in A, b \in B$ and added the edge to the graph if it was possible to add the edge to one of the two pages without a crossing. We repeated the process of randomly selecting the vertices until the maximum number of $2n - 4$ edges had been reached. As before we created 200 graphs this way for every test.

In Figure 4.3 it is shown that due to the lesser amount of edges the number of conflicts is lower than before for the planar graphs. The *smlDgrDFS* heuristic combined with *dataStructure* was the best heuristic for this test. In general, the *smlDgrDFS* vertex order heuristics performed much better on these bipartite graphs than before. It also seems that for the page assignment heuristic *dataStructure* and *ceilFloor* are the best to use. The box plot diagram in Figure 4.2 shows that all page assignment heuristics were able to find equally good results around 150-200 conflicts, but in all cases, there are some negative outliers which are far worse than the median results.

Table 4.2 confirms that also for other mixed layouts *smlDgrDFS* seems more successful than before. This is true, especially for stack pages. For layouts with more queue pages than stack pages *randBFS* was still better. In general, all page assignment heuristics

Table 4.2: The best heuristics for maximal planar bipartite graphs for different amounts of stack and queue pages.

	0-S	1-S	2-S	3-S	4-S	5-S
0-Q						
1-Q						
2-Q						
3-Q						
4-Q						
5-Q						

	ceilFloor		smlDgrDFS		dataStructure		eLen
	randBFS						

were about as good as the others, and while *dataStructure* and *ceilFloor* were the winners for some layouts, *eLen* slightly better most of the time.

4.3.3 2-trees

2-tree graphs are another interesting graph class that is worth considering. They have the same number of edges as the maximal planar bipartite graphs that we have tested before, namely $2n - 4$ edges for n vertices, and therefore it is a good way to compare the heuristics on different classes with the same density. We have here again very sparse graphs that are in contrast to the bipartite graphs very structured. They admit the same stack number as the planar bipartite graph which is two, and the queue number is at most three.

We generated the graphs by first creating a 3-clique. Then from all possible 2-cliques (which are all two vertices that are connected with an edge) we have randomly chosen one and added a new vertex with new edges to all members of this cliques. We repeated the process of randomly choosing such 2-cliques from the current graph and adding vertices until the graph had n vertices in total. For every test, we generated 200 new graphs this way.

The structuredness has a moderate impact on the ranking of the heuristics, and this can be seen in Figure 4.4. The *smlDgrDFS* is still the best to use and *conGreedy+* has gained some ranks. However, the impact on the number of conflicts is enormous. All heuristics delivered far better results and for some as *smlDgrDFS* the number of conflicts nearly halves. Figure 4.5 shows that in the best cases for graphs with 100 vertices the results

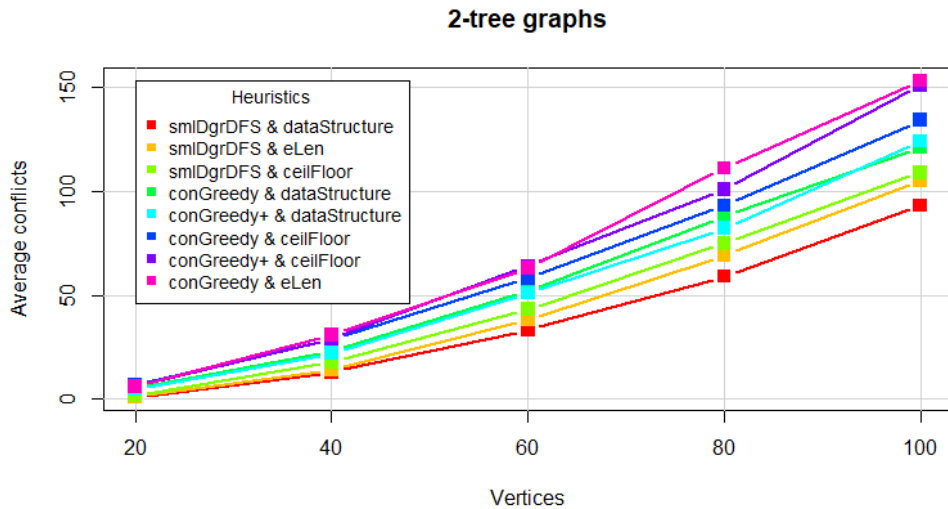


Figure 4.4: The eight best heuristics for a 1-stack 1-queue layout on 2-tree graphs.

were less than 50 conflicts. On the contrary, for the bipartite graphs with the same size, the best instances still had over 100 conflicts.

The improvements in the conflicts are also visible in Table 4.3. Missing boxes in this table indicate that more than one heuristic was able to compute linear layouts with less than one conflict on average. Four pages were sufficient for 4-stack and 3-stack 1-queue layouts while five pages were sufficient for 2-stack and 3-queue layouts to achieve conflict-free embeddings regularly. Since the stack number of 2-trees is lower than the queue number, it is not surprising that it is easier for the heuristics if they have more stack pages available. Due to the structuredness, it is visible in this table that *conGreedy+* as vertex order heuristic was more successful than on the bipartite graphs before where we have not seen it as one of the best for any layout.

4.3.4 3-trees

The next graph class that we test are 3-trees. They have the same connection to planar graphs as the 2-tree graphs have with the bipartite planar graphs that we have seen before. 3-trees with n vertices have exactly $3n - 6$ edges which are the same as for maximal planar graphs. The exact queue number is not known yet, but it is bound between four and five [ABG⁺18].

We generated the graphs the same ways as we did with the 2-trees but with larger cliques. Therefore, we started with a 4-clique and then from all possible 3-cliques, we added a new vertex with new edges to all members of this cliques and repeated this process until the graph had the desired size.

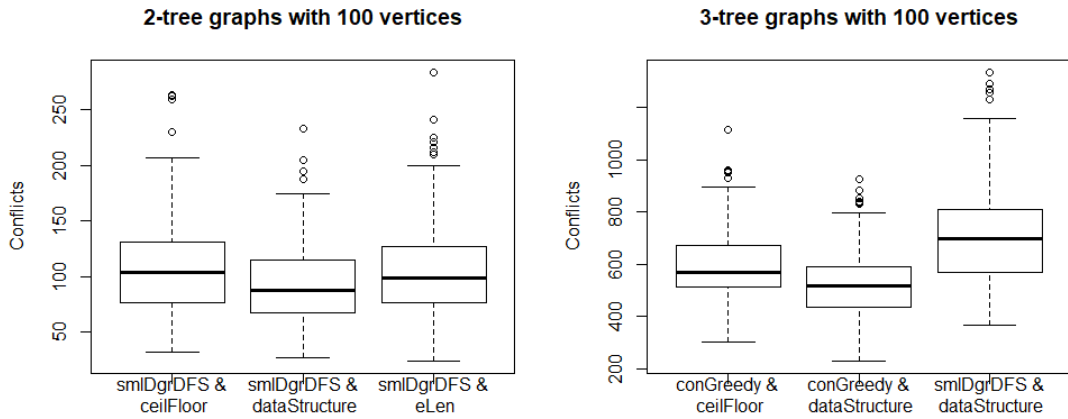


Figure 4.5: The three best heuristics for k-tree graphs in detail.

Table 4.3: The best heuristics for 2-tree graphs for different amounts of stack and queue pages.

	0-S	1-S	2-S	3-S	4-S	5-S
0-Q						
1-Q						
2-Q						
3-Q						
4-Q						
5-Q						

dataStructure conGreedy+ eLen smlDgrDFS
 conGreedy ceilFloor

Compared to planar graphs, the heuristics that are shown in Figure 4.6 are again able to compute layouts with fewer conflicts, and the difference between each heuristic has been significantly reduced.

Again, it seems that the structuredness of the graph favours *conGreedy* which is the best vertex order heuristic for this graph class. Also, *dataStructure* is the best page assignment heuristic no matter which vertex order has been computed before. Compared to the planar graphs we see here a lot more conflicts. While the results for the 2-tree

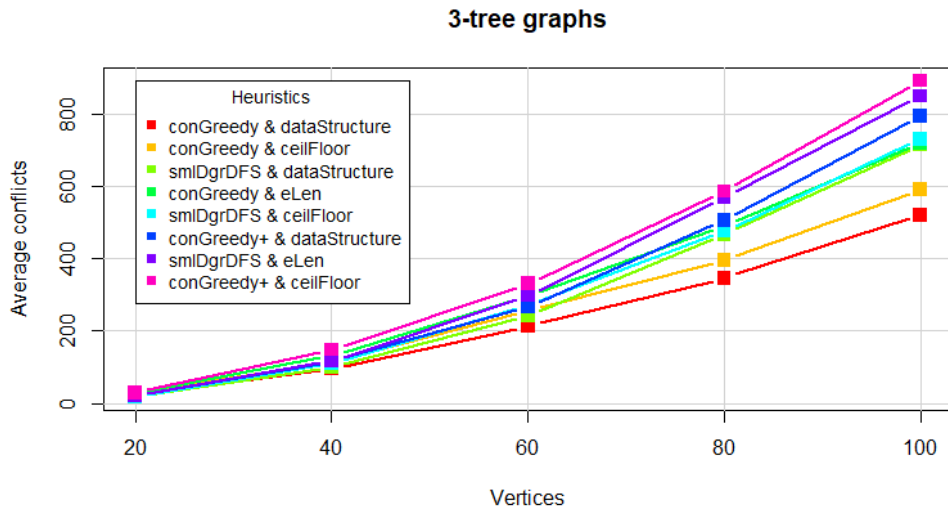


Figure 4.6: The eight best heuristics for a 1-stack 1-queue layout on 3-tree graphs.

graphs were better than for the bipartite graphs, the results got worse for the 3-tree graphs compared to the planar graphs. Structuredness and planarity are two significant factors for the quality of the solution.

The boxplot diagram in Figure 4.5 shows again that there is considerable variance in the results even for the same heuristic. The best results for the best combination of *conGreedy* and *dataStructure* was able to find solutions with close to 200 conflicts which is less than one conflict per edge. In the worst cases, the solutions had four to five times as many conflicts.













































Table 4.4 shows the results for other combinations of mixed layouts that indicate that a connectivity approach for the vertex order is the best one to use. For stack or queue layouts *conGreedy+* gave the best order, and for mixed layouts, *conGreedy* stood out. As for most test that we had seen before, *dataStructure* was again the best page assignment heuristic for 1-stack 1-queue layouts while in most other cases it was *eLen*.






4.3.5 Random graphs with $3n$ edges

The next graph class that we test are random graphs with three times as many edges as vertices. These graphs have nearly the same number of edges as the planar and 3-tree graphs that we have seen before. This makes it possible to compare graphs with a particular structure to completely random graphs.

We generated our graphs by first creating all vertices. Then we uniformly at random selected two vertices and added an edge between those two vertices to the graph if this edge did not already exist until we reached the desired number of edges. When the

Table 4.4: The best heuristics for 3-tree graphs for different amounts of stack and queue pages.

	0-S	1-S	2-S	3-S	4-S	5-S
0-Q						
1-Q		 	 	 	 	
2-Q	 	 	 	 	 	
3-Q	 	 	 	 		
4-Q	 	 	 			
5-Q	 	 	 			

 dataStructure
 eLen
 conGreedy+
 conGreedy
 ceilFloor

generated graph had exactly $3n$ edges, we did a check to see if the graph was connected. Finding a linear layout for disconnected components is easier because a layout can be computed for each component separately and composed together without creating any conflicts between edges of two different components. That is the reason why we did not want to have disconnected graphs for this test. Therefore, we removed such graphs and started the whole generation process again until we had 200 connected graphs.

It is not surprising that the heuristics performed worse on random graphs than on planar graphs or 3-trees, but there is quite a considerable increase in the number of conflicts. Figure 4.7 and Figure 4.8 are indicating that the best results have about three times more conflicts than on the 3-trees and about six times more than on planar graphs. While the best heuristics are about the same as for the 3-trees, it is very clear that planarity and structuredness helps a lot when trying to find a good layout. In case of such random graphs, it almost certainly ends up with many edges that span over a large part of the vertex order and such edges naturally cause a lot of crossings or nestings. It seems that the vertex order heuristics cannot prevent such longer edges as well as before.

For the mixed layouts that are presented in Table 4.5 we make an interesting observation. The page assignment *ceilFloor* was better than *eLen* in most cases. This is something that we have not seen for the previous graph classes yet. The reason for this is likely that longer edges that span over a bigger part of the vertex order are assigned to pages at a later point in time due to the circular definition of the edge length in *ceilFloor*. When assigning all long edges first, a greedy heuristic places some of those on the queue page because they immediately cause more crossings than nestings with the longer edges. However, later when the short edges are assigned these long edges on a queue page are

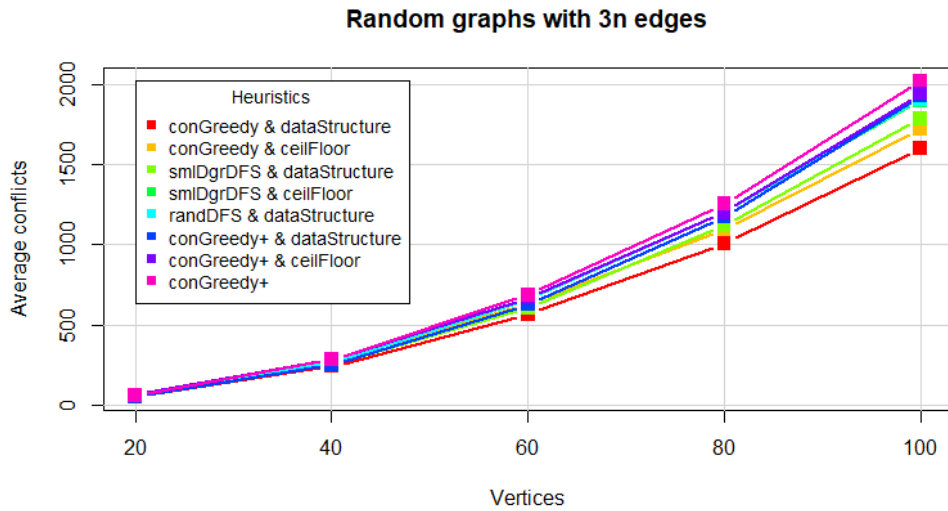


Figure 4.7: The eight best heuristics for a 1-stack 1-queue layout on random graphs with $3n$ edges.

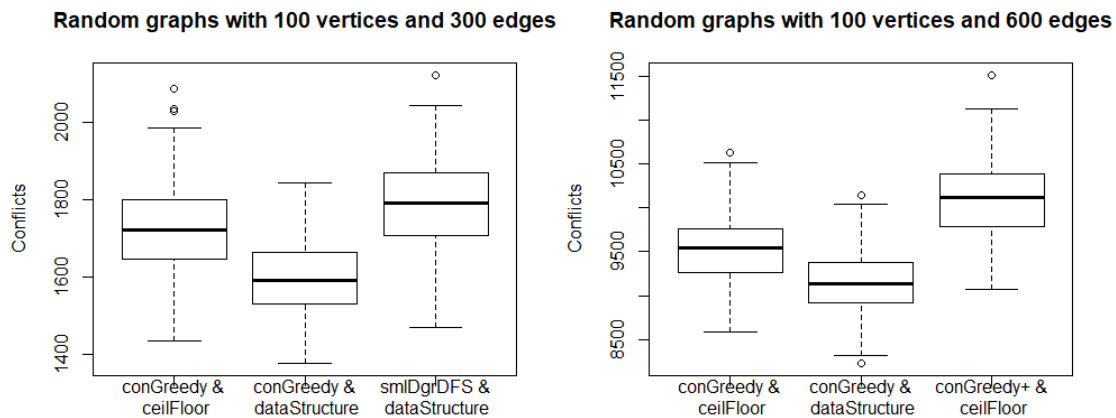


Figure 4.8: The three best heuristics for random graphs with $3n$ and $6n$ edges in detail.

easily causing nestings. Other than that we can see that *dataStructure* was again the best page assignment heuristic for 1-stack 1-queue layouts. The vertex order heuristics *randBFS* was the best for layouts which more queue pages, *conGreedy+* excelled the others for layouts with more stack pages and there is a thin line of mixed layouts where *conGreedy* was better than the others.

Table 4.5: The best heuristics for random graphs with $3n$ edges for different amounts of stack and queue pages.

	0-S	1-S	2-S	3-S	4-S	5-S
0-Q						
1-Q						
2-Q						
3-Q						
4-Q						
5-Q						

	conGreedy+		randBFS		dataStructure		ceilFloor
	conGreedy		eLen				

4.3.6 Random graphs with $6n$ edges

Since we have seen many rather sparse graphs, we start to test the heuristics on more dense graphs now. Therefore, the next graph class that we test are again random graphs but this time with six times as many edges as vertices. Here we have twice as many edges as for the $3n$ random graphs and the planar graphs that we have seen before.

We generated the graphs in the same ways as we generated the $3n$ random graphs. Again we made sure that all graphs were connected and we restarted the creation process if, after adding all $6n$ edges, the graph was not connected. As for all other tests before we created again 200 graphs for every graph size that we tested.

In Figure 4.9 shows that the number of conflicts raised drastically due to the higher density compared with the $3n$ random graphs. In the previous experiment, the best heuristics delivered on average results with about 1600 conflicts. Now we already reached the 10000 conflict mark. Figure 4.8 shows that *conGreedy* combined with *dataStructure* was able to find the best results. The results are also fairly stable now ranging from 8500 to 11000 conflicts. Also, the eight best heuristics are now really close together since they all are in the range of 9000 to 11000 conflicts on average over the 200 different graphs that we tested.

In Table 4.6 we can see that *conGreedy+* or *conGreedy* combined with *ceilFloor* performs well on dense graphs for nearly all mixed layouts that we tested. Only in the area of 5-queue layouts, *randBFS* outperformed *conGreedy+*.

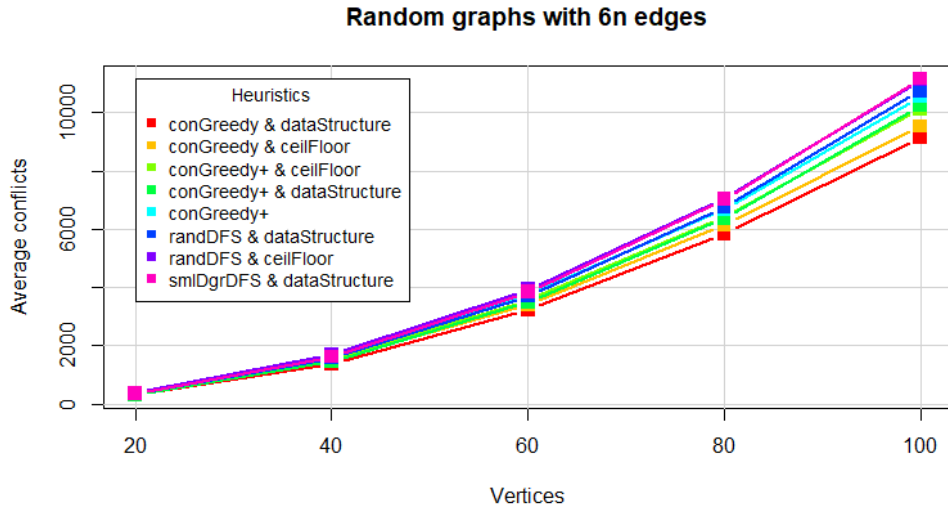


Figure 4.9: The eight best heuristics for a 1-stack 1-queue layout on random graphs with $6n$ edges.

4.3.7 Complete graphs

Finally, we test the heuristics on the densest possible graphs. Complete graphs with n vertices have exactly $\frac{n(n-1)}{2}$ edges. A complete graph with n vertices has a stack number of $\lceil n/2 \rceil$ if $n \geq 4$ [BK79] and a queue number of $\lfloor n/2 \rfloor$ [HR92]. Since the order of the vertices does not matter we just took any order and tested only the page assignment heuristics. Because none of our page assignment heuristics are randomised, the heuristics always compute the same results. Therefore, we needed to run our tests only once for each size of the graph.

Figure 4.10 shows the ranking of the page assignment heuristics for 1-stack 1-queue graphs. The *dataStructure* heuristic was slightly better than *conGreedy+*. Both *eLen* and *ceilFloor* performed far worse.

The slight lead for *dataStructure* for graphs with 50 vertices on the 1-stack 1-queue layouts can be seen too for 3-queue layouts in Table 4.7 which is quite surprising since this is not the case for 2-queue or 2-queue 1-stack layouts. Especially for a larger number of pages *conGreedy+* performs by far the best. It seems that for layouts with a large number of stack pages *eLen* is the best to use. This is also quite surprising because in the test before *eLen* was better for sparse graphs while *ceilFloor* was better for dense graphs. It seems that either there is a density where *eLen* is once again the preferred heuristic or this is due to the unique properties of complete graphs.

Table 4.6: The best heuristics for random graphs with $6n$ edges for different amounts of stack and queue pages.

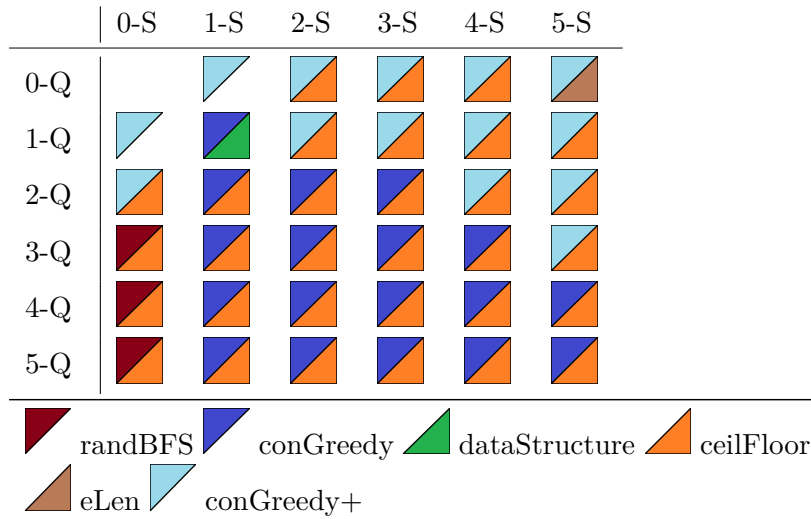
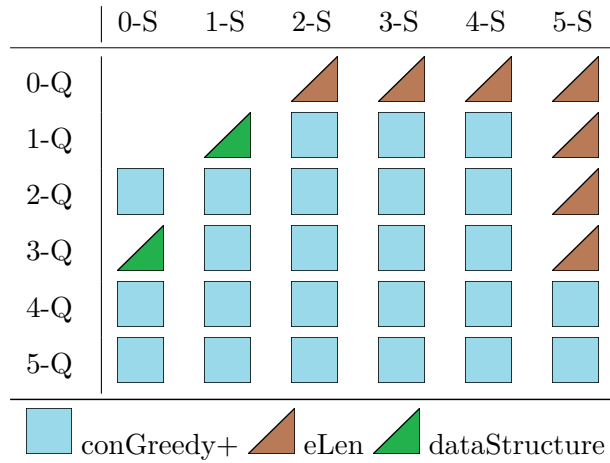


Table 4.7: The best page assignment heuristics for complete graphs for different amounts of stack and queue pages.



4.4 Optimization

In the previous two sections, we have introduced heuristics to compute solutions that should have a reasonable number of conflicts. Those solutions can sometimes be quite good. At other times they can still contain many conflicts that could be resolved to further optimise the quality of the given solution further. Construction heuristics usually

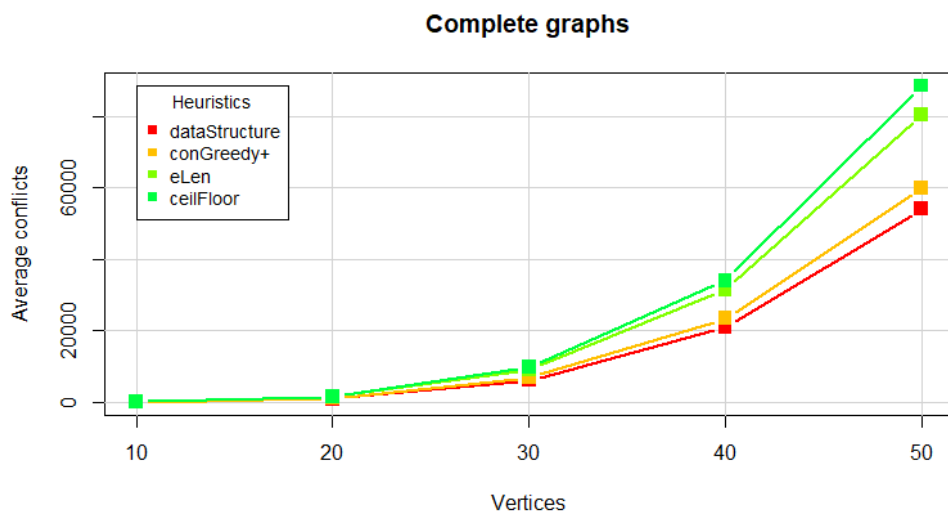


Figure 4.10: The number of conflicts for the four page assignment heuristics on complete graphs.

start with an empty solution and add the elements of the problem step by step until the solution is complete, but they never remove or change already positioned elements. This is the case for all the heuristics that we have presented so far. Once a vertex is added to the spine it is never moved again, and once an edge is assigned to a page, it has to stay on this page. While this approach finds reasonable solutions in a reasonable time, it leaves some space open for further optimisation that can be applied afterwards.

As before, we take methods that have been applied successfully for book drawings and try to reuse these for the mixed layouts. Two very straightforward methods are general enough to be applied directly for queue and stack pages. One method tries to optimise the page assignment by placing an edge to a better page, and the other method optimises the vertex order by placing a vertex to a better place on the spine.

Greedy edge assignment optimisation (*edgeOpt*): This greedy optimisation checks for every edge if it would result in a fewer number of total conflicts of the layout if the edge would be assigned to any other page. If such reassignment is possible, then the edge is assigned to the better page.

Greedy vertex order optimisation (*vertexOpt*): For every vertex, it is checked if the total number of conflicts of the layout can be reduced by placing the vertex to any other place on the spine.

Klawitter [Kla16] proposed and tested several ways how these two basic optimisation steps can be combined such that none of them can find further improvements and a local optimum is reached. One way is to run *edgeOpt* and *vertexOpt* alternately so that if one algorithm finds an improvement, then the optimisation is continued with the other

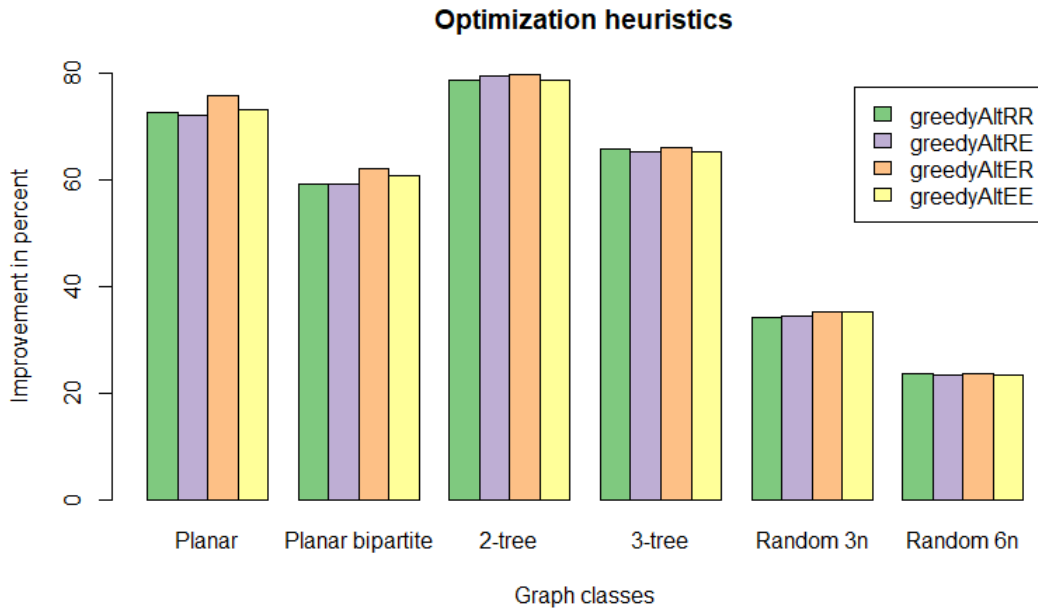


Figure 4.11: Improvements over the initial solution with the four optimization heuristics.

algorithm. Another way is to run one of those algorithms exhaustively until no further optimisation is possible and only then switch over to the other algorithm. This leads to four different ways of how these algorithms can be combined.

Greedy Alternating vertex and edge optimization (*greedyAltRR*): One round of *vertexOpt* alternating with one round of *edgeOpt*.

Greedy Alternating exhaustive edge optimization (*greedyAltRE*): One round of *vertexOpt* alternating with an exhaustive search of *edgeOpt*.

Greedy Alternating exhaustive vertex optimization (*greedyAltER*): An exhaustive search of *vertexOpt* alternating with one round of *edgeOpt*.

Greedy Alternating exhaustive vertex and edge optimization (*greedyAltEE*): An exhaustive search of *vertexOpt* alternating with an exhaustive search of *edgeOpt*.

In our experiments, we saw that both *edgeOpt* and *vertexOpt* could improve the initial calculated solution. Interestingly all of the four alternating combinations seemed to be equally good as it can be seen in Figure 4.11. No combination was able to improve solutions significantly better than the others. The *vertexOpt* algorithm is computationally costlier than *edgeOpt* because, depending on the density of the graph, usually there are more ways to move the vertices to different positions than to assign the edges to a different page. Recalculating the total number of conflicts is also easier for *edgeOpt* because there is only one edge that needs to be considered. For *vertexOpt* all conflicts of

the edges of the moved vertex need to be considered. Therefore, we might recommend using *greedyAltRE* because in our tests it was the fastest combination and the results were equally good as the others.

Beyond the simple versions of *edgeOpt* and *vertexOpt*, it is also possible to use the same ideas for similar optimisation algorithms that have a larger neighbourhood and therefore might be able to escape from a local optimum and deliver better results. For *edgeOpt* instead of reassigning a single edge, it could try to reassign two edges at the same time. Such a pairwise reassignment could for example swap two edges from different pages, which would not be possible in the simple version of *edgeOpt* that we have presented so far. In the same way, instead of moving a single vertex it is possible to move two vertices at the same time for *vertexOpt*. Again this offers a greater space of possibilities to improve the solution. Such an expansion is computationally much more expensive than the basic versions. For larger graphs, the alternating combination of the basic versions can take already a long time. Such larger neighbourhoods might make sense to be applied after the basics versions are not able to improve the solution any more. We have not done any experiments to compare to improvements that can be achieved compared to the simple versions of *edgeOpt* and *vertexOpt*.

4.5 Summary of experiments

We have seen that for the 1-stack 1-queue layouts the vertex order heuristic is far more important than the edge assignment heuristic. For a specific graph class, often there was a single vertex order heuristic that performed the best, regardless of which edge assignment heuristic it was combined with. In general, one could say that *smlDgrDFS* was the best vertex order heuristic for graphs that were either sparse like the bipartite graphs or were regularly structured as the 2-trees. The denser the graph was, the better was the performance of *conGreedy* and *conGreedy+*. The *randDFS* and especially *treeBFS* heuristics were often far worse than the others. The *dataStructure* heuristic that we introduced for the page assignment competed very well against *ceilFloor* and *eLen*.

For mixed layouts from zero to five stack pages and zero to five queue pages, the results were quite interesting. Often we could see that there is a difference if there is a mixed layout which has mainly queue or mainly stack pages. The *randBFS* heuristic was often better for the queue layouts or mixed layouts where there are more queue pages than stack pages better. For the opposite case where there are more stack pages, *smlDgrDFS* was often one of the best. For denser graphs, *conGreedy* and *conGreedy+* outperformed the others while for mixed layout with about an equal number of stack and queue pages we could see *dataStructure* as one of the best page assignment heuristics.

In general, we think that reusing heuristics that are used for book drawings for mixed layouts works well. For a book drawing, it is desired to have short edges since shorter edges result likely in fewer crossings. For queue pages, it is also desired to have rather short but not extremely short edges. Therefore the vertex order heuristics that are based on depth-first search and breadth-first search work well for queue layouts and mixed

layouts too. The greedy page assignment with the longest edges first is also a reasonable strategy for queue layouts and mixed layouts.

The *dataStructure* page assignment heuristic that we introduce with this thesis was the best heuristic for six out of the seven graph classes that we tested for 1-stack 1-queue layouts. Only for the planar graphs *ceilFloor* was better. Also for other mixed layouts, it was sometimes the best and often came close to the best one. Assigning the edges in the linear order in which they appear instead of assigning them by the longest edges first is a disadvantage of this heuristic, but this order is necessary to process them with the data structures. However, it seems that this is sometimes surpassed by the advantage of considering open and not yet assigned edges.



Drawings

In the previous chapters, we have seen the theoretical background of linear layouts, and we had a look at algorithms and heuristics to compute linear layouts. In this chapter, we visualise such layouts. To do so, we present different ways of how we can draw queue pages and how we can draw them together with stack pages. The main focus of this chapter is on 1-stack 1-queue layouts, but it is also possible to use the same ideas for mixed layouts with more than two pages. The reason why the layout is restricted to a maximum of two pages often makes sense because linear layouts are most commonly drawn either in a linear or circular style. In the linear style, the vertices are drawn on a line that separates the drawing into an upper and a lower half, and on each of these halves, the edges of one page can be drawn. In the circular style, the line of the vertices is drawn on the boundary of a circle by connecting both ends of the vertex order, and one page is drawn inside this circle, and the other page is drawn on the outside area of the circle. For layouts with more than two pages, other methods have to be used.

5.1 Circular drawings

Circular drawings of graphs are quite commonly seen and not only used for linear layouts. Every graph can be drawn with its vertices placed on the boundary of a circle. Therefore, improving such circular layouts is a general problem where many techniques exist to get a good and readable drawing. The easiest way to draw the edges is in a straight line fashion within the circle but also many other styles exist. For example edge bundling and drawing a crossing-free set of edges on the exterior side of the circle is a way to improve such drawings [GK06]. A paper that focuses more on design and aesthetics than the algorithmic part calls this drawing a connected ring pattern, and it lists some real-world examples [DLR09]. Crossing reduction is an especially essential point for circular drawings and much work has been done on this [BB04] [HS04] [ST13].

For the drawings of the mixed linear layouts, we can reuse the ideas that are already developed for circular layouts as well as taking advantage of the unique properties of the stack and the queue pages. The stack edges are by definition all crossing-free, and this property still holds if they are drawn either on the inside or the outside of a circle. Reducing crossings is the most important aesthetic when it comes to readability and human understandability [Pur97]. Therefore, it seems very natural to draw the stack edges either as straight lines or arches. Finding good drawing styles for the queue page is a more interesting and challenging problem because queue edges usually need to cross other edges in order to avoid being nested. Therefore, optimising the number of crossings is only possible when edges are bundled and bundled crossings are counted instead of the crossings of individual edges, and we need to try to optimise other aesthetics such as symmetry, bends and the angle of crossings. In the following, we present five drawing styles that we have developed that take advantage of the unique properties of a queue and show how they can be used in a mixed layout together with a stack page.

For all of the circular drawings that we present in this chapter, we are using the Goldner–Harary graph as an example. This graph has a book thickness of three because it is non-hamiltonian and maximal planar. It serves as an example for a graph where two stack pages are not sufficient, but it admits a 1-stack 1-queue layout. With 11 vertices and 27 edges, the graph is not too big but still has a good edge density such that we can see the consequence of the different drawing styles.

The first drawing style that we present is generally straight forward. The stack edges are drawn as straight lines within the circle. The inner side of the circle is the better side to draw the edges. On the inside, the distance between any two vertices is smaller, and the edges can be drawn without bends. This puts the stack page in the best possible place with its crossing-free edges as arguably best drawing style. For the queue page, we take advantage of its non-nesting property to gain structured and repeating patterns that allow the reader to follow its edges easier than for randomly placed edges. For each edge e_1 it holds that it must cross all other edges e_2 if $L(e_2) \prec L(e_1) \prec R(e_2) \prec L(e_1)$. Therefore, one idea is to have all such crossings close together and in a 90-degree angle close at the vertex $L(e_1)$. The pattern that we generate for the edges is quite simple. We traverse the vertex order, and every new outgoing edge crosses all edges that cover the starting vertex, and the edge is routed on its own concentric ring until it reaches its end vertex. This simplicity comes with the cost of quite a lot of wasted space because each edge requires a separate level. Therefore, the space that is required for the drawing is bounded by a linear factor of the number of edges on the queue page.

An example of this is shown in Figure 5.1. Since many edges are drawn in parallel and it is hard to follow them individually, we have drawn them with two different colours so that all the outgoing edges of once vertex have the same colour, and this colour alternates for each succeeding vertex. Therefore, following the outgoing edges of a vertex is easier because the reader can follow a bundle of edges of the same colour and does not need to follow an individual edge.

The two colours that we have used for the queue pages makes it easier to follow the edges,

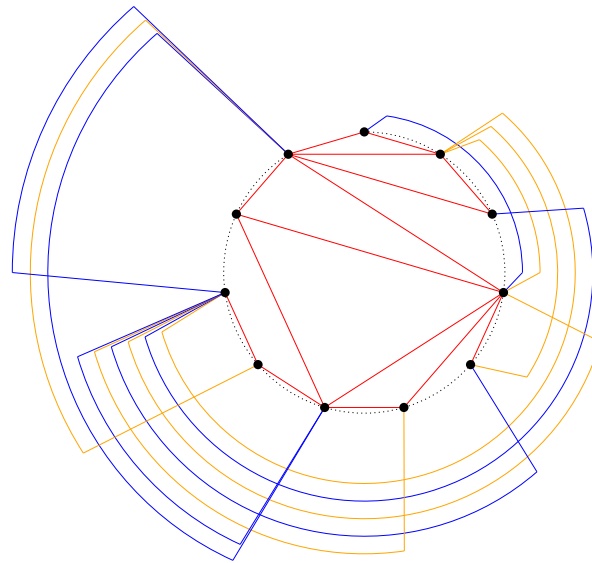


Figure 5.1: A circular drawing in which the stack is drawn on the inside as crossing-free straight line chords, and the queue is drawn on the outside with two different colours in a level oriented way. The simple pattern that we used for the queue page can introduce unnecessary crossings for edges that share a common vertex.

but for larger graphs with more and longer edges, it can still be hard to read the drawing. The next drawing style that we present uses more colours such that no two edges with the same colour are crossing, and it is shown in Figure 5.2. A total number of seven different colours was needed for this graph. On one hand, the use of more colours, can make it easier to follow the edges but on the other hand, it makes the drawing more colourful, and one must choose a good set of colours that fit together. While graph colouring for general graphs is shown to be an NP-complete problem by Karp [Kar72] already in 1972, the colouring of the edges of a queue page can be done in polynomial time. To achieve this, we can iterate over the vertex order and assign all outgoing edges of a vertex the same colour. Since no edges are allowed to nest the colours are not allowed to be used again until the iteration reaches a vertex where the last edge of the respective colour ends. Then, the colour is free to use again. To minimize the number of total colours that are needed for new edges such a freed colour must be used instead of using a colour that has never been used before. A straight forward algorithm to get such a colouring is shown in Algorithm 5.1.

In the previous examples, the edges of the queue page had constantly increasing height and were drawn further and further away from the vertices on the circle which increases the drawing area and edge length that is needed. We now present a technique to optimise the area and the edge length that is needed for a queue page. The edges are routed on concentric rings around the circle as we have seen before. However, this time they are allowed to move to lower levels once there is free space underneath. This occurs if other

Algorithm 5.1: queueColouring

Data: Ordered list of vertices V , edges E
Result: Colouring of the edges in E such that no two edges with the same colour cross on a linear layout with the vertex order V , and with the minimum number of different colours that are needed.

```

1 colours  $\leftarrow$  empty list of positive integers
2 foreach  $v$  in  $V$  do
3   if there is an available number in colours then
4     | colour  $\leftarrow$  lowest available number in colours
5   else
6     | colour  $\leftarrow$  lowest number that is not in colours
7     | Add colour to colours
8   edges  $\leftarrow$  all uncoloured edges of  $v$ 
9   Colour each edge in edges with colour
10   $v_2 \leftarrow$  end vertex of the longest edge in edges with respect to  $V$ 
11  Mark colour as unavailable until the loop reaches  $v_2$ 

```

edges are closed because they reach their end vertex. The space on this level can then be reused for open edges. Figure 5.3 shows such a reuse of levels. The drawing needs less space than before, but the edges have more bends because they often need to be rerouted to lower levels. The space that is required for the drawing is now reduced to a linear factor of the cutwidth of the queue page where the cutwidth is defined as the maximum number of edges that can be intersected by a vertical line between any two vertices.

Another approach to optimising the space and edge length for queue pages is shown in Figure 5.4. This time we do not allow more than two bends as we have in Figure 5.3 but levels can be reused once a concentric ring is free again. This breaks the pattern that we had before that new edges must cross all edges that cover the start vertex, but by doing so also crossings are reduced. The cutwidth of the queue page again bounds the space that is needed for this drawing.

In the previous examples, we have seen that the queue page is much harder to read than the stack page. A compromise between the stack and the queue can be made by putting the queue on the inside of the circle and the stack on the outside. The stack is crossing-free but with longer edges. On the contrary, the edges of the queue are shorter, and since they are harder to read, this reduced length should increase the overall readability of the drawing. An example of such a layout can be seen in Figure 5.5. One aesthetic result of this layout is that now the whole drawing is arranged around circles with the same centre. Not only the vertices but also both sets of edges are drawn as arches around circles with the same centre. This makes the drawing on the first sight more uniform and smooth.

Circular layouts have another advantage for the stack page. There is no necessity that these edges are drawn along the vertex order and it is possible for them to take a shortcut

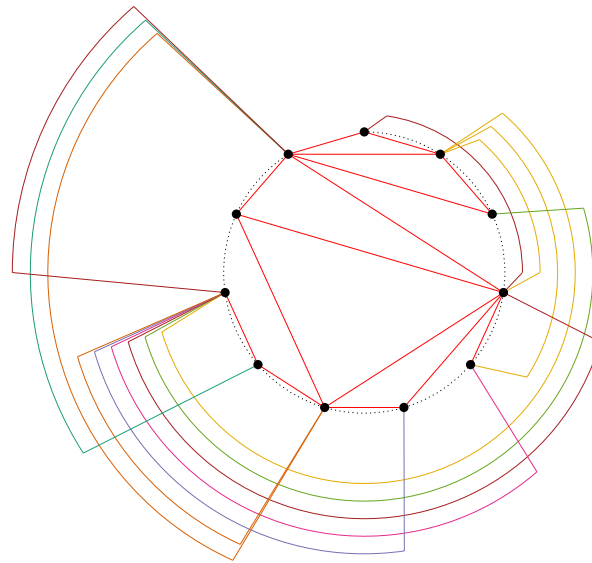


Figure 5.2: A circular drawing in which the stack is drawn on the inside as crossing-free straight line chords, and the queue is drawn on the outside in a level oriented way with as many colours as needed such that no two edges with the same colour cross.

over the gap between the first and the last vertex in the order. For edges where the length is at least $\frac{|V|}{2}$, this can be the shorter way. An extreme case of such a shortcut would be an edge from the first to the last vertex. Instead of being drawn around nearly 360 degrees of the circle it can just connect the two neighbours on a straight line. In Figure 5.5 the vertex order is indicated by the small dotted path which has a gap on top of the circle.

5.2 Linear drawings

Another drawing style that can be used is the arc diagram. This style fits the definition of linear layout because here all vertices are drawn on a straight line and all edges of a page are drawn as arcs on one side of this line. As with the circular layouts, this is used best for a total number of one or two pages. More than two pages can hardly be drawn on a two-dimensional plane without visual cluttering.

Most of the concepts we have seen for the circular layout can be applied as well for the linear layouts. Stack pages can be drawn again without crossings. For queue pages, we can again use the same methods of edge colouring and bundling of crossing that we have seen before. Also, the concept of layers and reusing free layers is applicable. The edges themselves can be drawn as arcs but for the queue page using straight lines with nearly 90-degree bends is also very useful for the readability. Arcs can produce more visual clutter if the crossing are drawn in different and small angles. In Figure 5.6 an example is

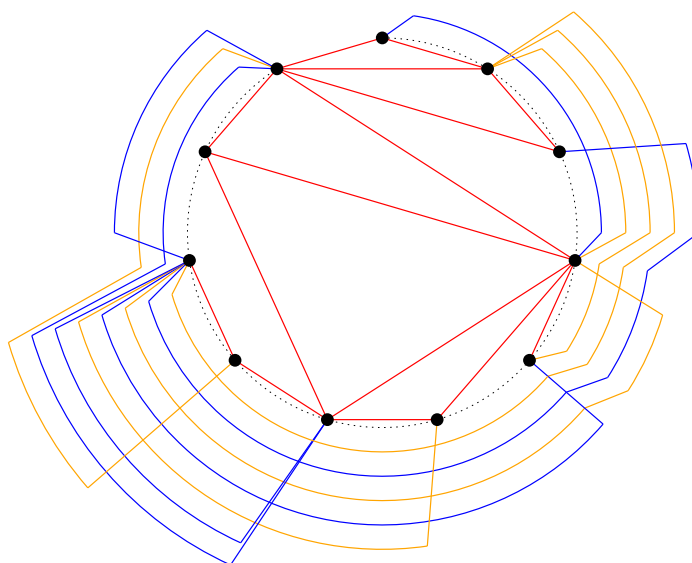


Figure 5.3: A circular drawing in which the stack is drawn on the inside as crossing-free straight line chords, and the queue is drawn on the outside with two different colours in a level oriented way where edges are routed to lower levels if there is free space.

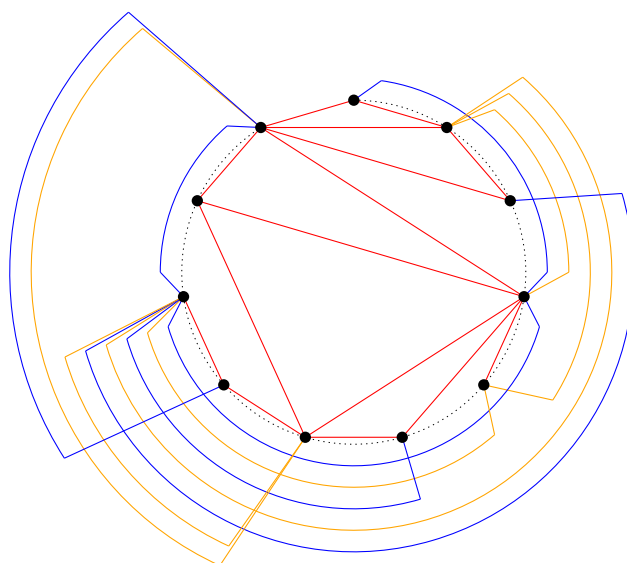


Figure 5.4: A circular drawing in which the stack is drawn on the inside as crossing-free straight line chords, and the queue is drawn on the outside with two different colours in a level oriented way where edges reuse lower levels if there is free space.

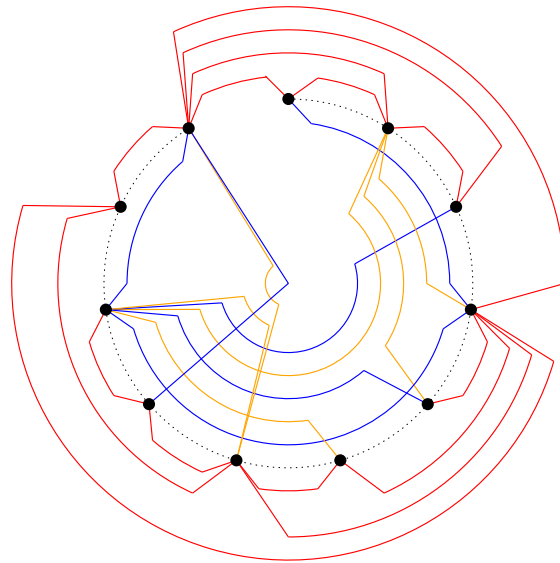


Figure 5.5: A circular drawing in which the stack is drawn on the outside as crossing-free arches, and the queue is drawn on the inside with two different colours in a level oriented way with reusing space.

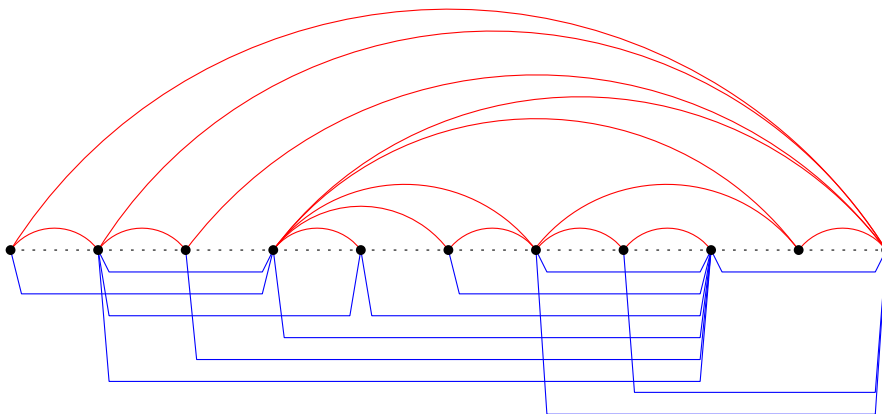


Figure 5.6: A linear layout with with the stack page on the upper half and the queue page on the lower half.

shown with the stack page on the upper half with the edges drawn as arcs and the queue page on the bottom half. It is notable that the linear drawings need more space than the circular drawings, and they are also rectangular shaped instead of a square or circle. Such linear drawings can be drawn in $\mathcal{O}(n) \times \mathcal{O}(m)$ space where $n = |V|$ and $m = |E|$.

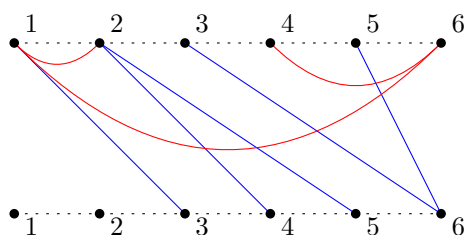


Figure 5.7: An unrolled cylinder drawing of a 1-stack 1-queue layout with the stack edges drawn in red and the queue edges drawn in blue.

5.3 Linear cylindric drawings

In the circular and linear drawings that we had seen before, the edges on the queue page naturally caused many crossings which made it harder to read than the stack pages. We have developed and seen methods on how we can try to improve the readability of the crossings to some extent, for example, by using different colours or edge bundling, but when the graph gets larger or denser, it gets harder to read the graph.

To overcome this problem with the crossing on the queue page Auer et al. [ABB⁺10] [ABB⁺18] developed a drawing style which they named linear cylindric drawings. For such a drawing the vertices of a graph are placed on a horizontal line along the axis of a cylinder in the three-dimensional space, and the edges are drawn on the surface of the cylinder without crossing the line of the vertices. Therefore, the edges can be drawn either as arcs on one of the sides of the line of the vertices or they can wrap around the surface of the cylinder. Since a two-dimensional object is easier to draw and read than a three-dimensional object, the authors then proceed by cutting the cylinder on the line of vertices and level the surface into the two-dimensional space. By doing so, the vertices are doubled and visible on the top and on the bottom of the resulting space, which they call an unrolled cylinder drawing. An example of such a drawing can be seen in Figure 5.7. The blue edges wrap around the cylinder, and the red edges are drawn on one side of the line of vertices.

The advantage of such an unrolled cylinder drawing is that it is possible to draw all queue edges without having any crossing between any two queue edges. A queue page can be drawn on such a cylinder in a planar way if all edges wrap around the cylinder once. Similarly, a stack page can be still drawn in a planar way if the edges are arcs on one side of the vertices. For the stack page, this is not surprising because as we have seen before, it is the same as for book embeddings and the linear drawings. This planar property still applies if we transform a cylindric drawing into an unrolled cylindric drawing. This means that we have indeed a way to draw stack and queue pages in a planar way if we do not mind to have all the vertices twice. Note that in this case the planarity is only given for edges within the same pages. Edges of different pages can cross each other in the resulting drawing.

The unrolled cylinder drawings allow drawing crossing-free queue pages when duplicating

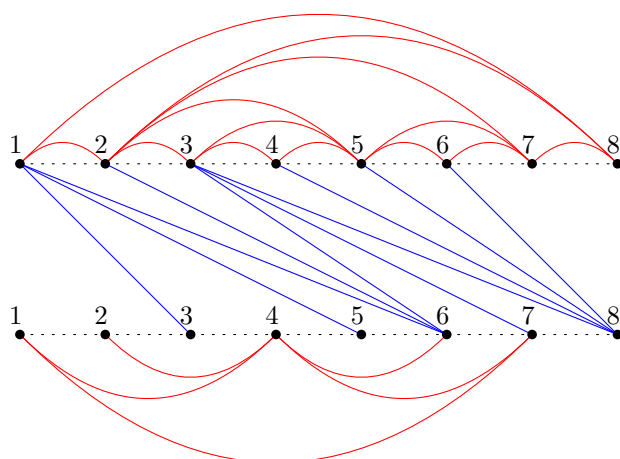


Figure 5.8: A linear cylindric drawing of a complete graph with eight vertices on two stack and one queue pages.

the line of vertices but the crossings between different pages are disturbing. Since our focus is on drawing mixed linear layouts in the best possible way, and we do not really care about the properties of a cylinder, it is already easy to see that we can draw the graph in Figure 5.7 in a planar way by drawing the stack edges on the other side of the first line of vertices. More importantly, this trick of duplicating the vertices allows us to draw mixed layouts with an arbitrary amount of pages without any crossings at all. This idea was also introduced by Auer et al. [ABB⁺10]. If we have an s -stack q -queue layout we can draw the line of vertices $s + q + 1$ times in parallel and draw each page between two of these lines.

This approach can be improved in a straightforward and obvious way to reduce the copies of vertices that we need. A stack page can be drawn above the very first line and below the very last line so that we do not need an extra copy for those two stack pages. Also, the area between two lines can be used by two stack pages at the same time. Therefore, if we add two stack pages to a drawing, we only need one extra copy of vertices. Figure 5.8 shows a planar drawing of a K_8 on a 2-stack 1-queue layout that needs only one additional copy of the vertices. This is the largest complete graph that fits on three pages. This drawing of the K_8 can be compared to the linear drawing in Figure 3.4 where the edges are partitioned on the same pages but drawn in a linear layout. Note that it becomes much easier to verify that there are no nested edges on the queue page since such edges would cross in this layout. For other drawings, this not immediately recognisable.

Conclusion

In this last chapter, we summarise the work that has been done as well as the results of the previous chapters. Finally, we give an outlook on future work.

6.1 Summary

In Chapter 3 we were dealing with NP-hardness, algorithms and planar bipartite graphs. We collected known results for complexity classes and NP-hardness for stack, queue and mixed layouts with either a fixed or free vertex order. We have seen that the linear layout problems with a fixed vertex order are computationally easier to decide than problems where the order is free. We have proven that if the vertex order is fixed and by adding more pages to an NP-hard linear layout problem that the problem will stay NP-hard. For the linear layout problems where the vertex order is free, we have seen that the 1-stack layout is the only problem that is known to be not NP-hard. Additionally to the known results from the literature that we have listed, we have proven that the 2-stack 1-queue layout and the 2-queue layout is NP-hard.

We have also seen that the problem of finding a 1-stack 1-queue layout with a fixed vertex order can be transformed into an instance of the 2-SAT problem which allows us to find such layouts efficiently if they exist. Finally, we worked on a conjecture of Pupyrev [Pup17] in which he suggests that every planar bipartite graph admits a 1-stack 1-queue layout. Due to an exhaustive computational assisted search, we have seen that every planar bipartite graph with up to 19 vertices admits a 1-stack 1-queue layout.

Chapter 4 contained our heuristic approaches to find mixed layouts with as few as possible conflicts. To achieve this goal we took existing heuristics that were developed for stack layouts and adapted and tested them for mixed layouts. We also introduced a new page assignment heuristic that we called *dataStructure* that can be used for linear layouts. In our tests, we have seen that the heuristics and ideas that are used for stack layouts work

generally well for mixed layouts, too. Often they try to generate a vertex order where the edges are preferably short, and they assign the longest edges first. This approach also makes sense for mixed or queue layouts. The *dataStructure* heuristic that we have developed was on the same level as the other page assignment heuristics and especially for mixed layouts with roughly the same number of stack and queue pages it was often the best page assignment heuristic. For six out of the seven graph classes that we have tested, *dataStructure* was the best heuristic for 1-stack 1-queue layouts.

Besides that, we have seen that connectivity-based vertex order heuristics like *conGreedy* and *conGreedy+* were good for dense graphs while the breadth-first search-based heuristics *randBFS* was good for sparser graphs and for layouts with more queue than stack pages. We have also seen that optimisation heuristics that are applied after these construction heuristics can reduce the number of conflicts significantly. In general, we think that reusing heuristics that are used for book drawings for mixed layouts works very well and the conflicts that we got for the mixed layouts are comparable to stack layouts.

In Chapter 5 we have shown different approaches to draw queue pages and mixed layouts. For layouts with one or two pages, a circular or linear layout might be a natural way to draw them. For stack pages, the edges are straightforward to draw because they can be drawn crossing-free in such layouts. For a queue page, we have developed different approaches on how we can use the properties of a queue to get a reasonable drawing because the edges are usually often crossing each other. We have seen that edge bundling, a colouring of the edges and crossings with nearly 90-degree angles are some strategies to improve the drawing. For layouts with more than two pages we have also shown the so-called linear cylindrical drawing style that duplicates the line of vertices but in return allows a crossing-free drawing even for the queue pages.

6.2 Future Work

We have collected and proven some results for the complexity and NP-hardness of some specific linear layouts, but we were not able to find in the literature or prove it ourselves if the 1-stack 1-queue layout with free vertex order problem is NP-hard or not. We know that it is decidable in polynomial time for 1-stack layouts and we also know that for 1-queue, 2-stack and 2-stack 1-queue it is NP-hard. Especially since the 1-queue layout is NP-hard, we would assume that the 1-stack 1-queue layout is NP-hard, too.

The open conjecture of Pupyrev [Pup17] that suggests that every planar bipartite graph admits a 1-stack 1-queue layout remains open, and an answer to this question would be an exciting result for mixed layouts.

The *dataStructure* page heuristic that we introduced in Chapter 4 delivered promising results. An interesting aspect about this heuristic is that it can calculate the conflicts that are introduced at the point of the edge assignment and possible future conflicts for each edge quite efficiently. Also, the strategy to decide on which page the edge should be assigned can vary. For our experiments, we always used the same strategy for all the tests

that we have presented in this thesis, but we have also tried other strategies where we introduced different weightings for the decision procedure. By carefully fine-tuning the weightings, it was nearly always possible to improve the results of *dataStructure* further for specific graph classes. Therefore, we think it would be interesting to develop and test other decision strategies for *dataStructure* that could be more successful for specific layouts or graph classes.

The drawings that we have presented were chosen purely by our liking. Especially for the circular layouts, we have introduced some new ideas, and we presented arguments on our opinion of the advantages and disadvantages. We have not performed any formal readability testing and user studies, and naturally, it would be interesting if such studies could deliver further insights about the readability of the drawings.

List of Figures

1.1	Mixed layout	2
1.2	Circular layout and arc diagram examples	3
1.3	Outerplanar and arched level-planar graph	5
2.1	Validating a stack page	9
2.2	Validating a queue page	10
3.1	Reduction for fixed vertex order and extra queue page	15
3.2	Reduction for fixed vertex order and extra stack page	16
3.3	Reduction for free vertex order and queue layouts	19
3.4	2-stack 1-queue layout of a K_8	21
3.5	Reduction for free vertex order and 2-stack 1-queue layout	22
4.1	Eight best heuristics for planar graphs	33
4.2	Best three heuristics for planar graphs in detail	33
4.3	Eight best heuristics for planar bipartite graphs	35
4.4	Eight best heuristics for 2-tree graphs	37
4.5	Best three heuristics for k-tree graphs in detail	38
4.6	Eight best heuristics for 3-tree graphs	39
4.7	Eight best heuristics for random graphs with $3n$ edges	41
4.8	Best three heuristics for random graphs detail	41
4.9	Eight best heuristics for random graphs with $6n$ edges	43
4.10	Best page assignment heuristics for complete graphs	45
4.11	Optimization heuristics	46
5.1	Circular drawing with the stack inside and the queue in a level oriented way	51
5.2	Circular drawing with the queue in crossing-free colours	53
5.3	Circular drawing in which the queue reuses lower levels with bends	54
5.4	Circular drawing in which the queue reuses lower levels	54
5.5	Circular drawing with the stack on the outside	55
5.6	Linear layout	55
5.7	Unrolled cylinder drawing	56
5.8	Linear cylindric drawing of a K_8	57

List of Tables

3.1	Complexity classes and hardness for layouts with fixed vertex order	14
3.2	Complexity classes and hardness for layouts with free vertex order	18
3.3	Planar bipartite graphs that admit a 1-stack 1-queue layout	25
4.1	The best heuristics for mixed layouts for planar graphs	34
4.2	The best heuristics for mixed layouts for planar bipartite graphs	36
4.3	The best heuristics for mixed layouts for 2-tree graphs	38
4.4	The best heuristics for mixed layouts for 3-tree graphs	40
4.5	The best heuristics for mixed layouts for random graphs with $3n$ edges . .	42
4.6	The best heuristics for mixed layouts for random graphs with $6n$ edges . .	44
4.7	The best heuristics for mixed layouts for complete graphs	44

List of Algorithms

2.1	Validate a stack page	9
2.2	Validate a queue page	11
4.1	<i>dataStructure</i> page assignment heuristic	31
5.1	Chromatic colouring of a queue page	52

Bibliography

- [ABB⁺10] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Andreas Gleißner. Plane drawings of queue and deque graphs. In Ulrik Brandes and Sabine Cornelsen, editors, *Graph Drawing (GD 2010)*, volume 6502 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2010.
- [ABB⁺18] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Andreas Gleißner. Data structures and their planar graph layouts. *J. Graph Algorithms Appl.*, 22(2):207–237, 2018.
- [ABG⁺18] Jawaherul Md. Alam, Michael A. Bekos, Martin Gronemann, Michael Kaufmann, and Sergey Pupyrev. Queue layouts of planar 3-trees. In Therese C. Biedl and Andreas Kerren, editors, *Graph Drawing and Network Visualization (GD 2018)*, volume 11282 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2018.
- [AG11] Christopher Auer and Andreas Gleißner. Characterizations of deque and queue graphs. In Petr Kolman and Jan Kratochvíl, editors, *Graph-Theoretic Concepts in Computer Science (WG 2011)*, volume 6986 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2011.
- [APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.
- [Aue14] Christopher Auer. *Planar graphs and their duals on cylinder surfaces*. PhD thesis, Universität Passau, 2014.
- [BB04] Michael Baur and Ulrik Brandes. Crossing reduction in circular layouts. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science (WG 2004)*, volume 3353 of *Lecture Notes in Computer Science*, pages 332–343. Springer, 2004.
- [BFG⁺18] Michael A. Bekos, Henry Förster, Martin Gronemann, Tamara Mchedlidze, Fabrizio Montecchiani, Chrysanthi N. Raftopoulou, and Torsten Ueckerdt.

- Planar graphs of bounded degree have constant queue number. *CoRR*, abs/1811.00816, 2018.
- [BK79] Frank Bernhart and Paul C. Kainen. The book thickness of a graph. *J. Comb. Theory, Ser. B*, 27(3):320–331, 1979.
- [BKZ15] Michael A. Bekos, Michael Kaufmann, and Christian Zielke. The book embedding problem from a SAT-solving perspective. In Emilio Di Giacomo and Anna Lubiw, editors, *Graph Drawing and Network Visualization (GD 2015)*, volume 9411 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2015.
- [BM99] Gunnar Brinkmann and Brendan D. McKay. Fast generation of some classes of planar graphs. *Electronic Notes in Discrete Mathematics*, 3:28–31, 1999.
- [BSC⁺08] Richa Bansal, Kamal Srivastava, Shweta Chaurasia, Kirti Varshney, and Nidhi Sharma. An evolutionary algorithm for the 2-page crossing number problem. In *IEEE Congress on Evolutionary Computation (CEC 2008)*, pages 1095–1102. IEEE, 2008.
- [Cim02] Robert J. Cimikowski. Algorithms for the fixed linear crossing number problem. *Discrete Applied Mathematics*, 122(1-3):93–115, 2002.
- [Cim06] Robert J. Cimikowski. An analysis of some linear graph layout heuristics. *J. Heuristics*, 12(3):143–153, 2006.
- [CLR87] Fan Chung, Frank Leighton, and Arnold L. Rosenberg. Embedding graphs in books: A layout problem with applications to VLSI design. *SIAM Journal on Algebraic Discrete Methods*, 8(1):33–58, 1987.
- [DF18] Vida Dujmovic and Fabrizio Frati. Stack and queue layouts via layered separators. *J. Graph Algorithms Appl.*, 22(1):89–99, 2018.
- [DFP13] Giuseppe Di Battista, Fabrizio Frati, and János Pach. On the queue number of planar graphs. *SIAM J. Comput.*, 42(6):2243–2285, 2013.
- [DJM⁺19] Vida Dujmovic, Gwenael Joret, Piotr Micek, Pat Morin, Torsten Ueckerdt, and David R. Wood. Planar graphs have bounded queue-number. *CoRR*, abs/1904.04791, 2019.
- [DLR09] Geoffrey M. Draper, Yarden Livnat, and Richard F. Riesenfeld. A survey of radial methods for information visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(5):759–776, 2009.
- [DMW19] Vida Dujmovic, Pat Morin, and David R. Wood. Queue layouts of graphs with bounded degree and bounded genus. *CoRR*, abs/1901.05594, 2019.

- [Duj15] Vida Dujmovic. Graph layouts via layered separators. *J. Comb. Theory, Ser. B*, 110:79–89, 2015.
- [DW04] Vida Dujmovic and David R. Wood. On linear layouts of graphs. *Discrete Mathematics & Theoretical Computer Science*, 6(2):339–358, 2004.
- [DW05] Vida Dujmovic and David R. Wood. Stacks, queues and tracks: Layouts of graph subdivisions. *Discrete Mathematics & Theoretical Computer Science*, 7(1):155–202, 2005.
- [GJT76] M. R. Garey, David S. Johnson, and Robert Endre Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Comput.*, 5(4):704–714, 1976.
- [GK06] Emden R. Gansner and Yehuda Koren. Improved circular layouts. In Michael Kaufmann and Dorothea Wagner, editors, *Graph Drawing (GD 2006)*, volume 4372 of *Lecture Notes in Computer Science*, pages 386–398. Springer, 2006.
- [HLR92] Lenwood S. Heath, Frank Thomson Leighton, and Arnold L. Rosenberg. Comparing queues and stacks as mechanisms for laying out graphs. *SIAM J. Discrete Math.*, 5(3):398–412, 1992.
- [HP99] Lenwood S. Heath and Sriram V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM J. Comput.*, 28(5):1588–1626, 1999.
- [HPT99] Lenwood S. Heath, Sriram V. Pemmaraju, and Ann N. Trenk. Stack and queue layouts of directed acyclic graphs: Part I. *SIAM J. Comput.*, 28(4):1510–1539, 1999.
- [HR92] Lenwood S. Heath and Arnold L. Rosenberg. Laying out graphs using queues. *SIAM J. Comput.*, 21(5):927–958, 1992.
- [HS04] Hongmei He and Ondrej Šýkora. New circular drawing algorithms. In *Workshop on Information Technologies - Applications and Theory (ITAT)*, 2004.
- [HSM07] Hongmei He, Ondrej Šýkora, and Erkki Mäkinen. Genetic algorithms for the 2-page book drawing problem of graphs. *J. Heuristics*, 13(1):77–93, 2007.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
- [Kla16] Jonathan Klawitter. Algorithms for crossing minimisation in book drawings. Master’s thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2016.

- [KMN17] Jonathan Klawitter, Tamara Mchedlidze, and Martin Nöllenburg. Experimental evaluation of book drawing algorithms. In Fabrizio Frati and Kwan-Liu Ma, editors, *Graph Drawing and Network Visualization (GD 2017)*, volume 10692 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2017.
- [KRSZ02] Nidhi Kapoor, Mark Russell, Ivan Stojmenovic, and Albert Y. Zomaya. A genetic algorithm for finding the pagenumber of interconnection networks. *J. Parallel Distrib. Comput.*, 62(2):267–283, 2002.
- [LLZ02] Michael Lewin, Dror Livnat, and Uri Zwick. Improved rounding techniques for the MAX 2-SAT and MAX DI-CUT problems. In William J. Cook and Andreas S. Schulz, editors, *Integer Programming and Combinatorial Optimization (IPCO 2002)*, volume 2337 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2002.
- [MNKF90] Sumio Masuda, Kazuo Nakajima, Toshinobu Kashiwabara, and Toshio Fujisawa. Crossing minimization in linear embeddings of graphs. *IEEE Trans. Computers*, 39(1):124–127, 1990.
- [Oll73] Tayler Ollmann. On the book thicknesses of various graphs. In *Combinatorics, Graph Theory, and Computing*, number Bd. 4 in *Congressus numerantium*, page 459. Utilitas Mathematica Pub., 1973.
- [Ove98] Shannon Overbay. *Generalized book embeddings*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 1998.
- [Pat13] Maurizio Patrignani. Planarity testing and embedding. In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization*, chapter 1, pages 1–42. Chapman and Hall/CRC, 2013.
- [Pup17] Sergey Pupyrev. Mixed linear layouts of planar graphs. In Fabrizio Frati and Kwan-Liu Ma, editors, *Graph Drawing and Network Visualization (GD 2017)*, volume 10692 of *Lecture Notes in Computer Science*, pages 197–209. Springer, 2017.
- [Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe Di Battista, editor, *Graph Drawing (GD 1997)*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997.
- [RM95] S. Rengarajan and C. E. Veni Madhavan. Stack and queue number of 2-trees. In Ding-Zhu Du and Ming Li, editors, *Computing and Combinatorics (COCOON 1995)*, volume 959 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 1995.
- [SSS13] Dharna Satsangi, Kamal Srivastava, and Gursaran Srivastava. K-page crossing number minimization problem: An evaluation of heuristics and its solution using GESAKP. *Memetic Computing*, 5(4):255–274, 2013.

- [ST06] Janet M. Six and Ioannis G. Tollis. A framework and algorithms for circular drawings of graphs. *J. Discrete Algorithms*, 4(1):25–50, 2006.
- [ST13] Janet M. Six and Ioannis G. Tollis. Circular drawing algorithms. In Roberto Tamassia, editor, *Handbook on Graph Drawing and Visualization*, chapter 9, pages 285–315. Chapman and Hall/CRC, 2013.
- [Tar72] Robert Endre Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19(2):341–346, 1972.
- [Ung88] Walter Unger. On the k-colouring of circle-graphs. In Robert Cori and Martin Wirsing, editors, *Theoretical Aspects of Computer Science (STACS 1988)*, volume 294 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 1988.
- [Ung92] Walter Unger. The complexity of colouring circle graphs (extended abstract). In Alain Finkel and Matthias Jantzen, editors, *Theoretical Aspects of Computer Science (STACS 1992)*, volume 577 of *Lecture Notes in Computer Science*, pages 389–400. Springer, 1992.
- [Wie86] Manfred Wiegers. Recognizing outerplanar graphs in linear time. In Gottfried Tinhofer and Gunther Schmidt, editors, *Graphtheoretic Concepts in Computer Science (WG 1986)*, volume 246 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 1986.
- [Wie17] Veit Wiechert. On the queue-number of graphs with bounded tree-width. *Electr. J. Comb.*, 24(1):P1.65, 2017.
- [Woo02] David R. Wood. Queue layouts, tree-width, and three-dimensional graph drawing. In Manindra Agrawal and Anil Seth, editors, *Foundations of Software Technology and Theoretical Computer Science (FST TCS 2002)*, volume 2556 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2002.
- [Woo05] David R. Wood. Queue layouts of graph products and powers. *Discrete Mathematics & Theoretical Computer Science*, 7(1):255–268, 2005.
- [Yan89] Mihalis Yannakakis. Embedding planar graphs in four pages. *J. Comput. Syst. Sci.*, 38(1):36–67, 1989.

Appendix

We implemented a collection of useful functions that we used throughout this thesis, and we made them publicly available on a GitHub repository ¹. The code is written in Python 3 and contains among other information all the algorithms and experiments that we have presented in this thesis. We used it to generate the graphs and run the heuristics in Chapter 4. It can also be used to create drawings in the styles that we presented in Chapter 5.

¹<https://github.com/pdecol/mixed-linear-layouts>