Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis Computer Science

Implementation of an online framework to compute linear layouts of a graph

Julia Männecke

Datum

Reviewer

Prof. Dr. Michael Kaufmann
Algorithms Research Group
Universität Tübingen

Advisor

Dr. Michaelis Bekos
Algorithms Research Group
Universität Tübingen

Männecke, Julia:
Implementation of an online framework to compute linear layouts of a graph
Bachelor Thesis Computer Science
Eberhard Karls Universität Tübingen
Thesis period: 16.05 - 15.09.19

Abstract

In theoretical Computer Science graph theory is an essential subject of research, and even for real-world problems graphs are of great importance as they provide the basis of many things such as simulation of electricity grids, neuronal networks, ...? Within graph theory linear layouts the subject of linear layouts offers many possibilities for research.
A linear layout is defined by the order of vertices in which those lay on the spine of the graph, as well as a subdivision of the edges to a number of subsets. These subsets stand for half planes arranged around the spine, on which the corresponding edges reside. The question to be answered is, in how many of these half planes a graph can be embedded under certain circumstances.
This thesis describes the development of an online framework to create graphs and compute some linear layouts of those.

Zusammenfassung

In der theoretischen Informatik stellt die Graphentheorie einen essentiellen Teil der Forschung, und für viele Probleme des Alltags sind Graphen von großer Wichtigkeit, da sie die Basis für Dinge wie Stromnetze, ...? Innerhalb der Graphentheorie bietet das Feld der linearen Layouts ein dankbares Forschungsthema.
Ein lineares Layout definiert sich durch die Ordnung der Knoten des Graphen, in der diese auf der zentralen Gerade des Graphen liegen, sowie die Zuordnung der Kanten des Graphen zu einer Anzahl von Teilmengen. Diese Teilmengen stehen für die Halbebenen, welche um die Zentralachse angeordnet sind und auf welchen sich die entsprechenden Kanten befinden. Die Frage, die es zu beantworten gilt, ist auf wie vielen dieser Halbebenen ein Graph dargestellt werden kann, wenn bestimmte Voraussetzungen eingehalten werden sollen.
Diese Arbeit beschreibt die Entwicklung einer webbasierten Benutzeroberfläche um eben diese Graphen zu erstellen und zugehörige lineare Layouts zu berechnen.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

In graph theory there are several / a few methods of drawing and analyzing graphs.
One of those is linear layouts, where all vertices are placed on one line in some order and all edges are drawn as half circles above and below this line.
The interesting point of these linear layouts then is to determine an order of the vertices in which the edges connecting these are laid out in a special way.
I go more into detail about the different constraints in the following chapter.
As much as the field of linear layouts is an interesting topic it is also quite tiresome to find a linear layout with a particular attribute by hand, since the researcher would have to either strategically determine for each edge where it should be placed to obtain the desired attribute or iterate through a good portion of the permutations of the vertices, to find this attribute by chance / accident.
For this exact reason researchers found different ways to automate the computation of these linear layouts REFERENCE.
A few years ago the Arbeitsbereich / Lehrstuhl für Algorithmen of University of Tübingen proposed an approach to automation by formulating the desired layout as a SAT-formula REFERENCE and passing it to a SAT-Solver. While this was a notable achievement to the field it still needed a lot of manual work to convert a graph into the SAT-Formula

Chapter 2

Preliminaries

This chapter explains the fundamentals of graph theory and introduces definitions and notations that are used in this thesis.

2.1 Basics

Formally a graph $G = (V, E)$ consists of a finite set of vertices V and a finite set of edges $E \subset V \times V$, where each edge e_i connects two vertices $v_j, v_k \in V$. We state that $n := |V|$ and $m := |E|$.
A *path* p is a series of edges $(e_1, e_2, ..., e_m), e_1, ..., e_m \in E$. The same path can also be described as a sequence of vertices $(v_1, v_2, ..., v_k)$, where for example e_1 connects v_1, v_2.

2.1.1 Attributes of a graph

Directed or undirected

A graph can be either *directed* or *undirected*.
In a *directed* graph all edges $e \in E$ are described as an ordered pair of vertices $e = (u, v)$, where the first vertex is referred to the source and the second is referred to as the target vertex.
In an *undirected* graph each edge is defined by an unordered pair of vertices $e = u, v$ with no definit source or target.

Weighted or unweighted

A graph can have *weighted* or *unweighted* edges.
If a graph is defined as *weighted* it's definition includes a function $w : E \to \mathbb{R}$ which assigns a weight to each edge $e \in E$.

Complete

A *complete* graph contains an edge e for every pair of vertices $v, u \in V$. *Complete* graphs are denoted as K_n where n is the amount of vertices.
For an *undirected* graph with n vertices there exist exactly $\frac{n^2-n}{2}$ edges.

Connected

A graph is called *connected*, if for every two vertices $v_i, v_j \in V$ there is a path from v_i to v_j, meaning there are no unreachable vertices.

Acyclic

An *acyclic* graph has no cycles, which means that there exists no path $(v_0, v_1, ..., v_k)$ where $v_0 = v_k$, that is to say that for every path in the graph the start vertex and end vertex can not be the same.

Bipartite

A graph is called *bipartite* if its vertices V can be divided in two subsets $V_0, V_1 \subset V$ such that the intersecting set is empty, $V_0 \cap V_1 = \emptyset$, the union of the set is the complete set of vertices, $V_0 \cup V_1 = V$, and each edge $(u, v) \in E$ connects one vertice from V_0 with one vertice from V_1, $v \in V_0$ and $u \in V_1$.

Trees and forests

A *tree* is a *connected*, *acyclic* graph.
A *forest* is an graph consisting of several tree graphs, meaning it is not *connected* but each connected subgraph is *acyclic.*

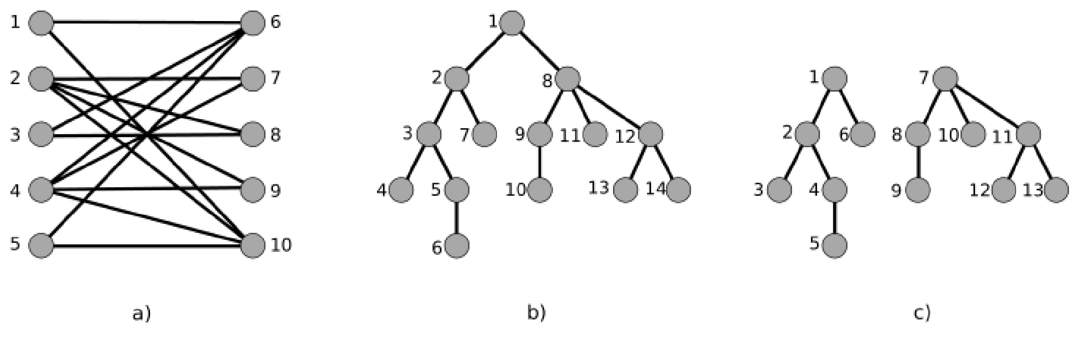


Figure 2.1: a) A bipartite graph, b) a tree, c) a forest

Planar

A graph is *planar* if it can be drawn in a way that no two edges $e_i, e_j \in E$ cross.
A graph is *maximal planar* if no edge can be added to the graph without loosing the planarity. K_4 is the biggest *complete* graph that is planar.
The drawing of a graph is referred to as *plane* if no edges cross. Note that a graph can be *planar* but a particular embedding of the graph is not necessarily *plane*.

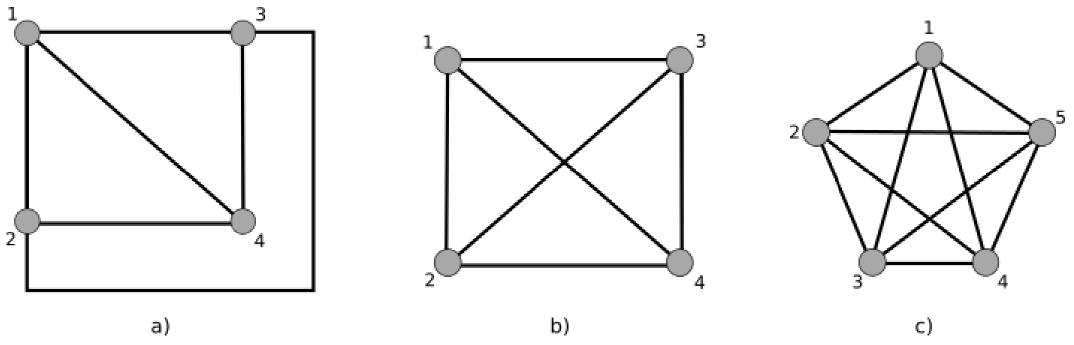


Figure 2.2: a) A planar and plane graph, b) a planar but not plane graph, c) a neither plane nor planar graph

2.2 Linear layouts

A linear layout of a graph is a layout in which all the vertices V are positioned along a line called the *spine*.
The order in which these vertices are placed on the spine, is described by a bijective function

$$\sigma : V \mapsto \{1, ..., n\}$$

. This function defines a relation where the following statements hold:

- Antisymmetry: if $\sigma(v_0) \leq sigma(v_1)$ and $\sigma(v_1) \leq \sigma(v_1)$ then $v_0 = v_1$
- Transitivity: if $\sigma(v_0) \leq \sigma(v_1)$ and $\sigma(v_1) \leq \sigma(v_2)$ then $\sigma(v_0) \leq \sigma(v_2)$
- Connexity: either $\sigma(v_0) \leq \sigma(v_1)$ or $\sigma(v_1) \leq \sigma(v_1)$ is true

These relation properties make the relation a *total order* and the set of vertices a *linearly ordered set*.
The edges of the graph are sorted into p disjoint subsets $E_p \subset E$ by the surjective function

$$\pi : E \rightarrow \{1, .., p\}$$

making it a partition of E.
In the linear layout each subset represents one half-plane, delimited by the spine, on which all the edges will be placed. In order to make it possible to draw these graphs in a 2D setting each half-plane is usually colored differently which is why the subsets are sometimes also referred to as *colors*.
The following sections explain which requirements the order of vertices and the assignment of edges follow.

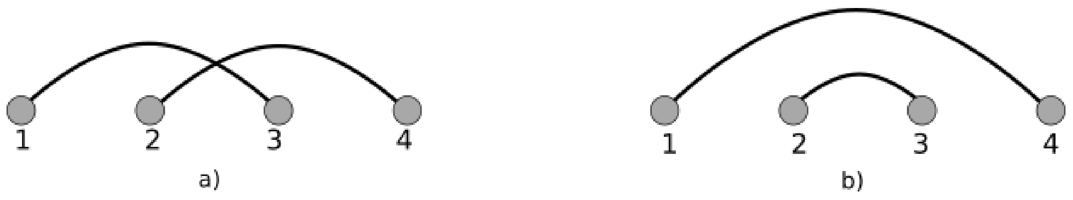


Figure 2.3: a) crossing edges, b) nesting edges

2.2.1 Stack layouts

For a stack layout $\mathcal{E}(G, p)$, sometimes also referred to as a *book embedding*, the vertices V need to be ordered in a way that now two edges e_i, e_j assinged to the same subset $\pi(e_i) = \pi(e_j)$ cross, that is to say each each half plane contains a *planar* subgraph. The subsets of E_p in a stack layout are calles *pages*.

The *book thickness* or *stack number* of a graph defines the minimal number of pages that is needed to embed the graph.
In 1968 Prof. Dr. Mihalis Yannakakis [?] was able to prove that every planar graph admits to a stack layout with a *book thickness* $p \leq 4$. Prof. Yannakakis also sketched the construction of a graph whose book thickness $p = 4$. However, his sketch was never completed and to this day researchers have not been able to find a graph that requires four pages.

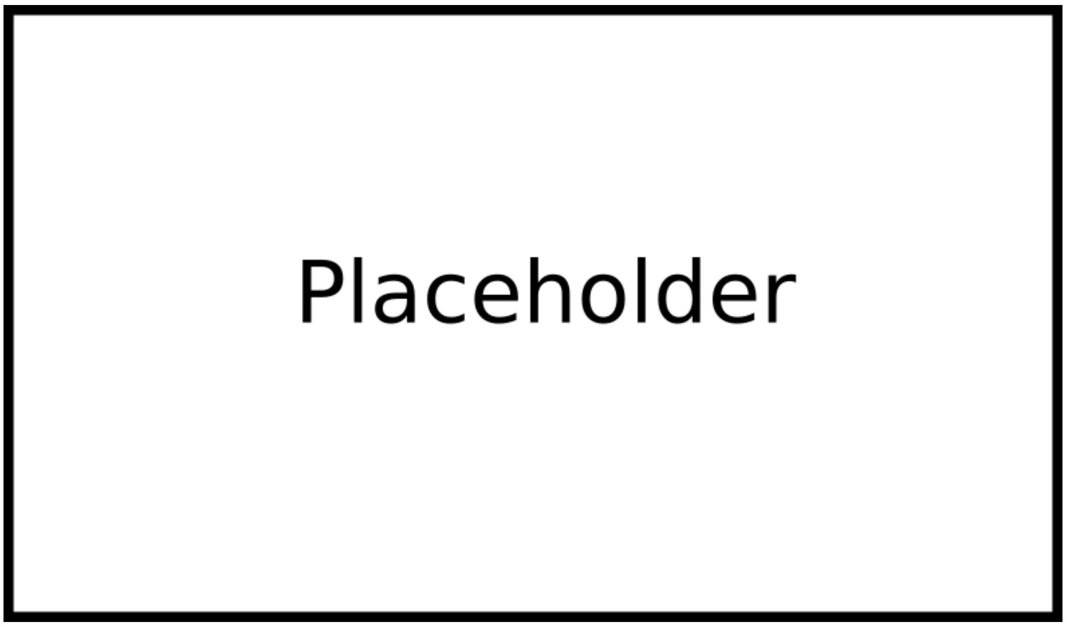


Figure 2.4: K_4 and the corresponding stack layout

2.2.2 Queue layouts

For a queue layout $\mathcal{Q}(G, q)$ the order of the vertices is chosen so that no two edges e_i, e_j where $\pi(e_i) = \pi(e_j)$ are nested. Two edges e_i, e_j are nested if both endpoints of edge e_i are between both endpoints of edge e_j, as shown in Figure 2.3. This property is not violated if edges e_i, e_j share the same source or target vertex.
Similar to the *book thickness* in stack layouts a queue layout has a *queue number* that states in how many queues the graph can be embedded in minimally.

2.2.3 Further restrictions

In addition to either being a stack or a queue, each half plane can be restricted further to have a special graph structure:

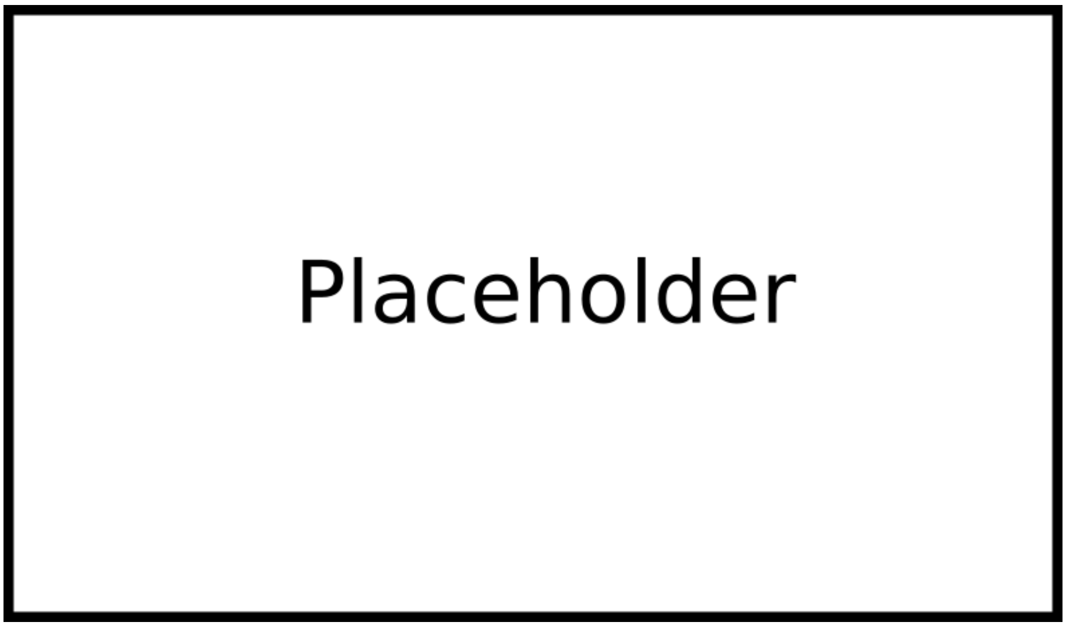


Figure 2.5: K_4 and the corresponding queue layout

Tree or forest subgraphs

A half plane or page of the layout has to be in the structure of a *tree* or *forest*, meaning that it is *acyclic* and in the case of a *tree* also connected.

Dispersible subgraphs

A *dispersible* subgraph is a graph in which no two edges share one endpoint. This kind of graph is also called a *matching*.

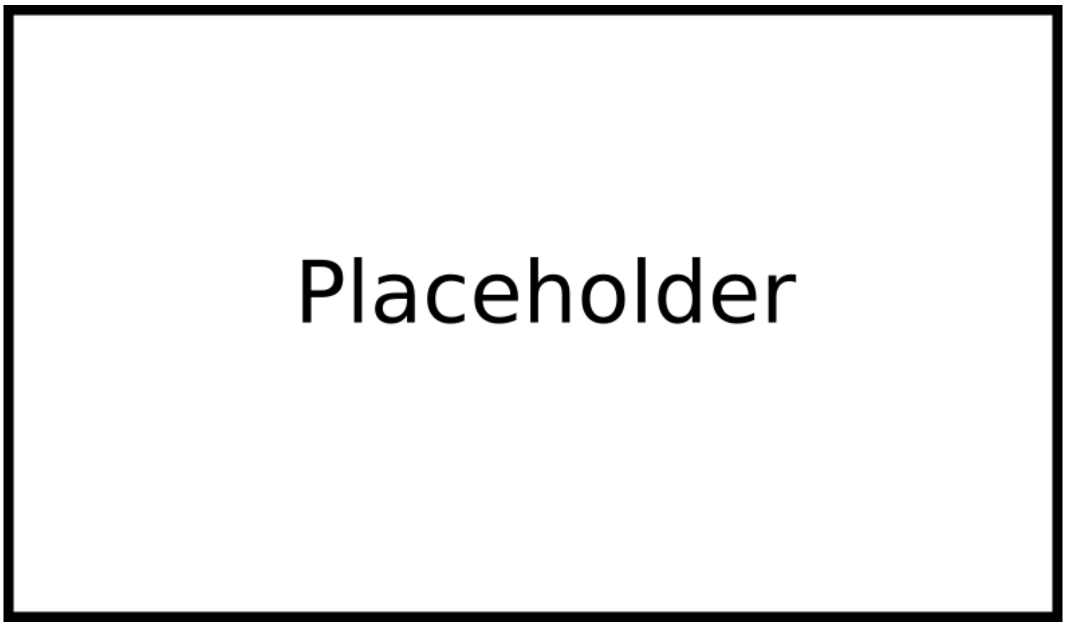


Figure 2.6: Dispersible Subgraphs

2.3 Boolean satisfiability problem

In 2015 the Algorithms Research Group of University of Tübingen proposed a new method to compute linear layouts automatically [?], by formulating the problems as SAT instances which can then be solved by a SAT solver in reasonable time. The following sections provide the theory behind SAT-formulas and a rough explanation of how the properties of a linear layout are translated.

2.3.1 SAT problem

SAT is short for satisfiability and means the *Boolean satisfiability problem*, which is the problem of determining whether there exists an assignment of truth values that satisfies a Boolean formula in *conjunctive normal form.* This problem is np-complete.
A Boolean formula consists of variables and operators ($\wedge$ for AND, $\vee$ for OR, $\neg$ for negation) which may be organized by parantheses. A formula is satisfiable if there exists an assignment of truth variables **true** and **false** to the variables, such that the formula evaluates to **true**.
Instances in SAT are in *conjunctive normal form.* This means that the formula is a conjunction of clauses (or literals), where each clause is a disjunction of literals. That is to say the variables in a clause are connected by OR ($\vee$) and the clauses are connected by AND ($\wedge$). The opposite of CNF would be the

disjunctive normal form, where the formula is disjunction of clauses which are are conjunction of variables.

$$\neg A \wedge (B \vee \neg C \vee D) \wedge (A \vee \neg D)$$

$$\begin{aligned} A &= \text{false} \\ B &= \text{true} \\ C &= \text{false} \\ D &= \text{false} \end{aligned}$$

Figure 2.7: A formula in conjunctive normal form and an assignment of truth variables that satisfies the formula

2.3.2 The linear layout as a SAT instance

Let $G = (V, E)$ be the graph and $\mathcal{F}(G, p)$ the logic formula that describes the problem in *conjunctive normal form.*
For each pair of vertices $v_i, v_j \in V$

Chapter 3

Used technologies

In this chapter the implemenation will be explained in terms of the used technologies and software libaries.

3.1 Languages

3.1.1 HTML

HTML, short for *Hypertext Markup Language* is the standard markup language used for displaying content in a web browser, currently in it's fifth major version (HTML 5) [?, ?].
Each *HTML 5* document has to start with the declaration

<!DOCTYPE html>

This instructs the web browser to interpret the contents as *HTML* and also specifies the version of *HTML*, since this decaration varied for the former versions.
A HTML document consists of HTML elements that structure the web page. Each element is introduced by an opening tag (e.g. <div>, <img>) and besides a few exceptions each element has to be ended by a closing tag (e.g. </div>). These tags have to be organized in a tree structure with the <html>-tag as the unique root of the tree, followed by its two children, the <header>-element, where the meta info is located and the <body>-element, where the content of the website is specified. Each tag can be assigned attributes such as a class, id, name and specific attributes like a standard value or a source, as seen in Figure 3.1.
HTML 5 provides the user with an abundance of predefined elements such as the aforementioned divs, which are box containers providing the general structure of a page as well as headlines, paragraphs, tables, elements of forms

and tags for embedding external sources, like images and audio.
Because *HTML* creates static websites the *Document Object Model* (DOM) was developed as an interface between markup languages and scripting languages, e.g. *JavaScript*. The DOM requires the user to assign ids, names or classes to the elements in order to adress them.
The standardisation of *HTML* (as well as *CSS*) is maintained by the *World Wide Web Consortium (W3C)*. This project follows the HTML5-standard.

```
<!Doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Sample code</title>
  <link rel="stylesheet" type="text/css" href="example2.css"/>
</head>
  <body>
    <div class="contents">
      <h1 id="headline">Sample code</h1>
      <p> This is a paragraph.
      <p> How to include images: <p><img id="img" src="image.png">
      <form>
        <p> <input id="textInput" type="text">
        <p> <button id="button">Send</button>
      </form>
    </div>
    <script src="example3.js"></script>
  </body>
</html>
```

Figure 3.1: Examplecode in HTML

3.1.2 CSS

Cascading Style Sheets, almost always referred to as *CSS*, is a language to define the representation of HTML elements. It is, amongst HTML and Javascript, one of the main technologies used in todays web development.
Usually styling rules are specified in the header of a html document, either as a reference to a *.css-file* or inside the <style>-tag. Which *HTML* element is affected by each rule is down to the selector that precedes the rule. Selectors can be a multitude of things, first of all ids and classes or tag names, as well as different relations between elements, such as successors, descendants or states of the element such as *disabled* or *hover* (see in Figure 3.2).
Styling rules can also be defined as *Inline Styles*, meaning that the rule is specified within the tag. This makes a simple and fast way to change appearances

without defining a class or id to the element.

```
body {
  background-color: #FFFFFF;
  font-family: arial;
}

.contents {
  font-size: 20pt;
  height: 80%;
  text-align: center;
}

#headline {
  font-size: 5em;
}

#headline:hover {
  color: red;
}

.contents>form {
  border: 1px solid blue;
}
```

Figure 3.2: CSS example code

3.1.3 XML and graphML

XML

XML is a markup language similar to *HTML*. It is used to encode documents of all sorts in a way that is readable for machines and humans alike. It's standards are also monitored by the *W3C*.
The type of an *XML* document does not need to be declared but can be by

<?xml version="1.0" encoding="UTF-8"? >

An XML document consists of *markup* and *content*.
Usually *markup* parts of the document start with "<" and end with ">", what makes them similar to *HTML* elements, but they also may be started with "&" and ended with ";". The syntax allows opening tags e.g. "<item>" which have to be at some point afterwards ended by a closing tag "<\item>", or one lined tags as "<item \>". The tags are again organized in a tree structure, allowing only one unique root of the tree.
Anything that is not *markup* is *content*.

GraphML

GraphML[1] is a widely used format to encode graphs and their drawings, that is based upon *XML*. It was initiated by the *Graph Drawing Community* **??**. It's root is required to be "<graphML >" and the defining tags are "<node >" and "<edge>" but a *GraphML* document can be extended to fit the users needs.
The *yFiles for HTML* library provides means to encode and decode graphs in GraphML. For this project the *GraphML* code was extended to also hold informations about the pages of each graph, (see in Section 4.2.4) and the constraints imposed on the graph (see in Section 4.2.4).

3.1.4 Javascript

Javascript [**?**, **?**, **?**] is a script language developed for client side programming of websites, although today it is also possible to implement server sided applications e.g. with *JavaScript* and *Node.js*[2]. In a majority of today's websites *JavaScript* components can be found. Its main application is to add flexibility and interactiveness to the otherwise static html elements. This is achieved by accessing the elements with the aforementioned *DOM* and adding

- event listeners: An event listener is triggered by the specified event, for example hovering over an element, clicking on an element or using the keyboard
- reading and writing: the value of an html input element such as a text area, checkbox or select menu can be read and evaluated at any time or even manipulated. The content of an element can also be read and changed.
- changing appearances: the appearance of an html element can be changed to the same extend *CSS* can.

In addition to the usual datatypes (*boolean*, *number*, *string* and *null*) *JavaScript* yields *object literals* which are customizable by the developer to hold values of any form. An object literal is enclosed by curly brackets and holds key value pairs defining the object, see for example the code in Figure 3.3.

JSON

JSON (JavaScript Object Notation) is a format for exchanging structured data. As the intent of *JSON* was to be able to send data from *JavaScript*

[1]`http://graphml.graphdrawing.org/`
[2]`https://nodejs.org/en/about/`

```
var graph = {
  nodes: 11,
  edges: 27,
};

document.getElementById("button").addEventListener("mouseover", function() {
  document.getElementById("button").style.border = "1px solid red";
})

document.getElementById("button").addEventListener("click", function() {
  var string =  "This graph has " + graph.nodes.toString() +
          " nodes and " + graph.edges.toString() + " edges"
  alert(string)
})
```

Figure 3.3: JavaScript example code

to a server application, a *JSON*-object has the same structure as *JavaScript* object literals. Today *JSON* is a language-independent format. *JavaScript*, among other languages, provides methods to easily encode and decode objects in *JSON*.

3.2 Software libraries

3.2.1 yFiles for HTML

yFiles for HTML[3] is a software library by the company yWorks, who specializes in software solutions for visualizing graphs, diagrams and networks for various platforms. The software library is written in Javascript and compatible with all modern web browsers without the need of any additions. It offers support for interactive user interfaces to edit and view graphs, starting at basic interactions up to complex algorithms to analyze graphs.
The documentation for *yFiles for HTML*[4] is accompanied by a multitude of demo applications to demonstrate the various possibilities.
For this implementation the version *yFiles for HTML 2.1.0.6* was used.

3.2.2 jQuery

The javascript library *jQuery*[5] is by far the most used javascript extension. *jQuery* provides shorter, easier to use and read syntax for the same function-

[3]https://www.yworks.com/products/yfiles-for-html
[4]https://docs.yworks.com/yfileshtml/#/dguide/introduction
[5]https://jquery.com/

alities as JavaScript.
It especially simplifies the usage of the document object model, since it shortens the needed code significantly, as seen in Figure 3.4. This implementation uses version *jQuery 1.12.4*. The GUI uses plugins such as *jQuery Tag-it!*[6] and *ColorPick.js*[7] which are based on *jQuery*.

```
var graph = {
  nodes: 11,
  edges: 27,
};

$("#button").click(function() {
  $("#button").style.border = "1px solid red";
})

$("#button").click(function() {
  var string =  "This graph has " + graph.nodes.toString() +
          " nodes and " + graph.edges.toString() + " edges"
  alert(string)
})
```

Figure 3.4: jQuery example code

3.2.3 jQuery UI

jQuery UI extends *jQuery* by a collection of modern, free to use widgets and effects for a wide range of website types. Each module can be used independently as the design is neutral and fits into most environments. This project uses the buttons, checkboxes and selectmenus as well as dialogs provided by *jQuery UI*.

[6]http://aehlke.github.io/tag-it/
[7]https://github.com/philzet/ColorPick.js

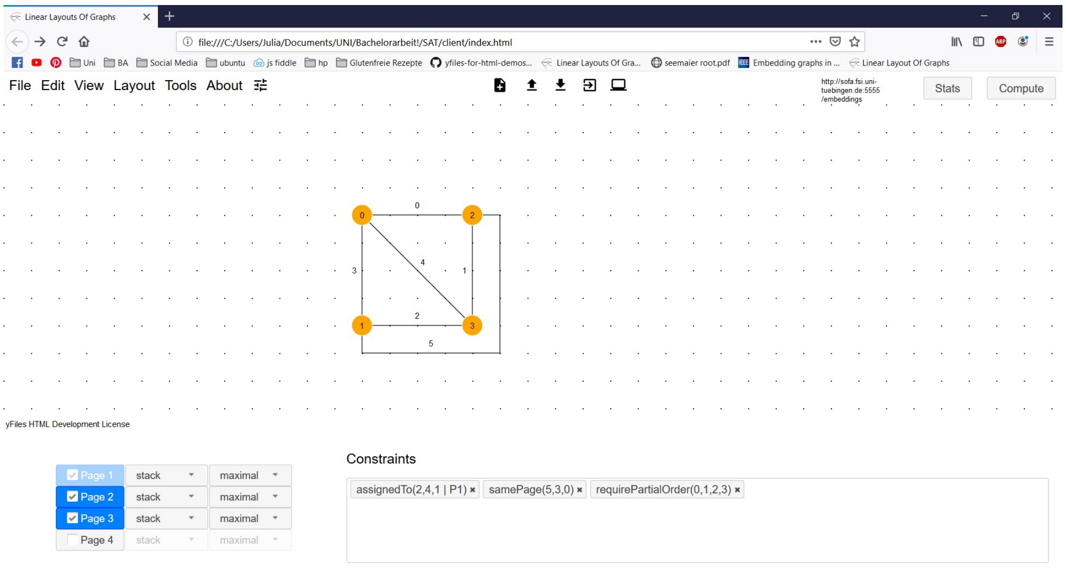
Chapter 4

Implementation

The following chapter explains how the user interface is structured and which options it holds for the user, after describing the intention of this project.

4.1 Motivation and intent

As mentioned in the introduction, figuring out a linear layout with certain properties by hand is a time consuming, error-prone task. This is why efforts have been made to automate this process. The discovery that SAT formulas could do this quite efficiently [?] was a big achievement.
Up to now the graphs had to be transformed into a textual representation in order to then be translated tp a SAT instance and passed to the solver. The intent of this project was therefore to develop an editor where a graph can be created as a drawing, then be passed to a translating routine and a solver. After receiving the solver result, the embedding should be displayed again in a visually appealing way.
Furthermore it was an ambition to check a graph for specific linear layouts, since randomized SAT-sovers produce random linear layouts. Thankfully this can be easily achieved by expanding the SAT instance by clauses that lead the SAT-solver in the desired direction. Therefore the editor needed an option to impose such constraints upon the future linear layout.
It was decided that the editor should be webbased in order achieve broad availability, since then nothing more than a fairly modern browser is needed.


Linear Layouts Of Graphs
file:///C:/Users/Julia/Documents/UNI/Bachelorarbeit!/SAT/client/index.html
File
Edit
View
Layout
Tools
About
http://sofa.fsi.uni-tuebingen.de:5555/embeddings
Stats
Compute
0
1
2
3
4
5
yFiles HTML Development License

✓ Page 1	stack	maximal
✓ Page 2	stack	maximal
✓ Page 3	stack	maximal
Page 4	stack	maximal

Constraints

assignedTo(2,4,1 | P1) samePage(5,3,0) requirePartialOrder(0,1,2,3)

Figure 4.1: Overview over the user interface

4.2 Graph Editor

4.2.1 Overwiew

The user interface of this tool consists of three main areas: the toolbar on top, the interactive graph editor in the middle and the configuration panel, where the user can specify the linear layout he or she wishes to compute.

4.2.2 Graph Editor

The graph editor is the center area of the user interface. It displays a grid by default which can be disabled.

User Interaction

Nodes are created by clicking on the canvas. Edges can only be created between two existing nodes, by dragging a line from the source node to the target. By default, double edges are forbidden but can be allowed by the user through the **tools** submenu.
On creation, each node and edge is assigned a label, counting from 0. These labels can be changed by the user.
Bends in the edges can be achieved by releasing the mouse button wherever

the bend should be. In big graphs this enables a tidier drawing.
If the grid is visible the graph elements encourage certain positions on the grid by clipping to the dots of the grid when moved, but still every element can be placed freely on the canvas.
The elements of a graph can be selected and then copied, cut, pasted and deleted. Edges can only be copied when duplicate edges are allowed or when at least one of the corresponding nodes is selected, too.

Identifier of elements

To compute the linear layout of the graph it is necessary to have a unique id for each element of the graph, so the solver can distinguish the different nodes and edges.
On creation every node and edge is assigned a label, which is displayed in the graph editor and can be changed by the user without restriction. This means that labels are not unique and therefore can not be used to identify elements.
Invisible for the user each node and edge also is assigned a tag, a feature provided by the *yFiles for HTMl* library. These tags can not be changed by the user and are designed to be unique. Nodes get the next free number as a tag, edges get the tag "a-(0)-b", where a is the tag of the source node and b is the tag of the target node. The *(0)* in the middle of each edge tag is necessary if the user chooses to allow duplicate edges. A duplicate edge would then be assigned the tag "a-(1)-b", ensuring the uniqueness.

4.2.3 Toolbar

On the left section of the toolbar the user can choose from submenus, the options being a *file*, *edit*, *view*, *layout* as well as a link to the about information of this page and a button to toggle the visibility of the configuration panel.

The file submenu

🗋	Clears the canvas and configuration panel to create an entirely new graph
⭳	Save the current graph into the users local file system
⭱	Load a graph from the users local file system
⇥	Export the graph to either a .png- or .pdf-file
💻	Change the server to which the graph should be passed on computation. This is a feature for people who use this tool frequently and prefer to install a local copy of the server. The current server setting is displayed in the top right corner, next to the stats button.

The edit submenu

↶	Erases the last changes on the canvas
↷	Redoes the last undone changes on the canvas
⧉	Copies the current selection of elements, also achievable by ”ctrl + c”
📋	Pastes formerly cut or copied items to the canvas, also achievable by ”ctrl + v”
✂	Cuts the current selection of elements, also achievable by ”ctrl + x”
⬚	Selects all elements currently on the canvas, also achievable by ”ctrl + a”
×	Deletes the currently selected elements, also achievable by ”del” key
● ● ●	This set of buttons changes which elements of the graph should be selected when a rectangle is dragged over the canvas. The first means all elements are selected, the second means only nodes are selected, the last means that only edges are selected. This is convenient when a constraint needs to be imposed on a large group of nodes or edges.

The view submenu

🔍	Zooms into the canvas
🔍	Zooms out of the canvas
⊡	Focusses the graph in the center of the canvas
▦	Toggles visibility of the grid.

The layout submenu

The layout submenu holds several algorithms to transform the graph. The most widely used layouts for graphs were used for this, containing *hierarchic*, *organic*, *orthogonal*, *circular*, *tree*, *balloon* and *radial* layouts.

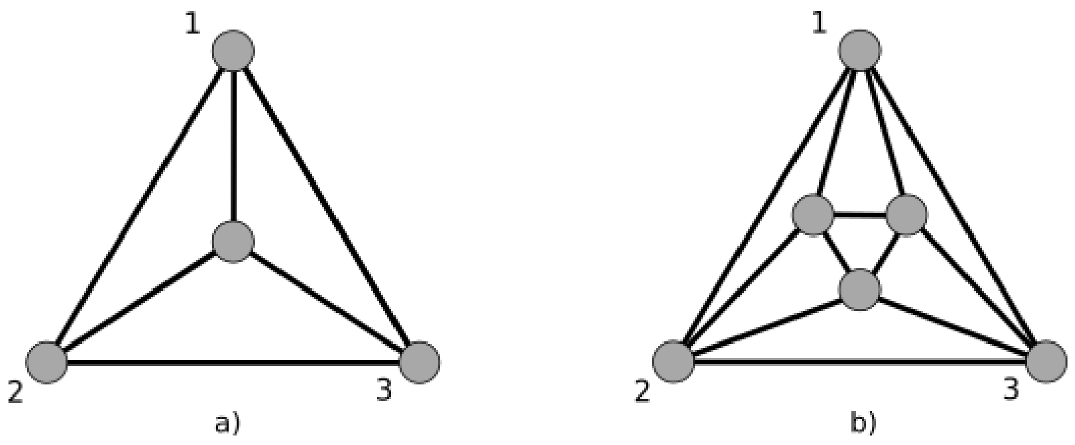


Figure 4.2: Stellation of a triangular face

The tools submenu

Toggles, whether nodes are resizable or not

Toggles, whether duplicate edges are allowed or not

When no nodes are selected, this stellates every face of the graph. That means a new node is placed in the center each face of the graph and edges connect the new node to each node of the face. When one or more nodes are selected, the new node is connected to each of the selected node (see Figure 4.2)

When less than three nodes are selected, three nodes are inserted into every face of the graph that is bounded by three edges. The new nodes are connected to themselves and the three nodes of the face, as to be seen in Figure 4.2

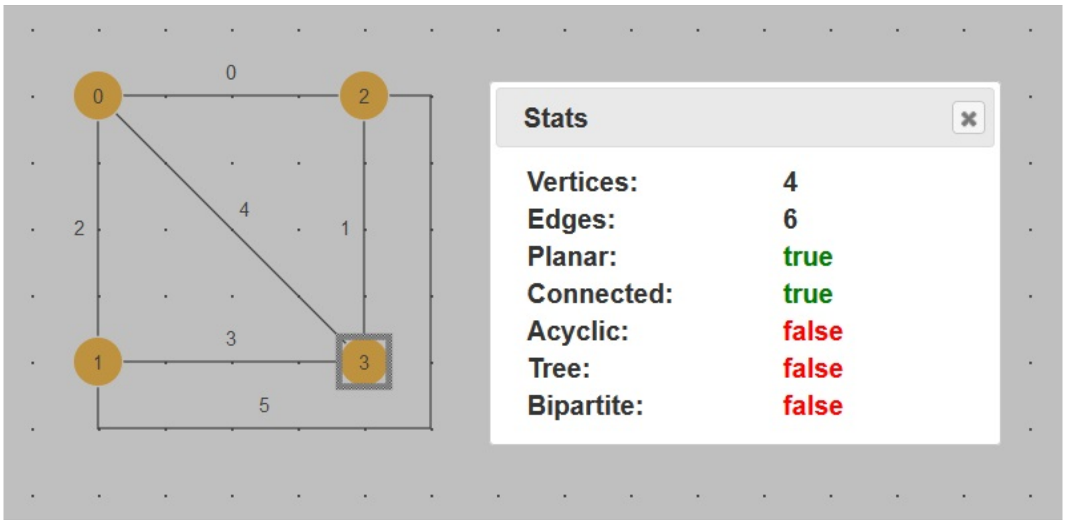
If no edge is selected, every above every edge a new node is created, which is connected to the two end nodes of the edge. If a set of edges is selected, this only applies to the selection

Stats

The stats panel gives information of the current graph. First of all it states how many vertices and edges the graph contains. Furthermore it provides information, whether a graph is *planar*, *connected*, *acyclic*, a *tree* and a *forest*. The algorithms for these properties are provided by *yFiles for html*.

Compute

After clicking on the *compute* button, a dialog opens to ask the user whether she would like to proceed to computing the linear layout as it is defined right



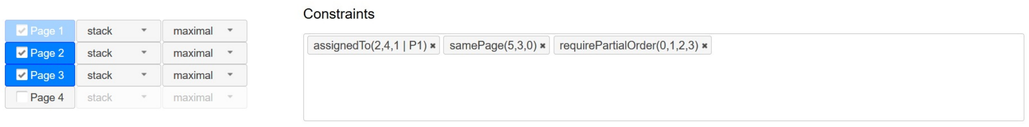
Stats

Vertices:	4
Edges:	6
Planar:	true
Connected:	true
Acyclic:	false
Tree:	false
Bipartite:	false

Figure 4.3: Stats of a graph

now. The user then can choose to first save the graph, proceed or cancel.
If the computation is wished, the graph including its constraints and page settings are sent to the server with an ajax-request. Since the request is processed asynchronously by the server, it returns the id of the future embedding right away. This response triggers a redirection to the second page with the newly acquired id as a hash parameter.
If the server returns an error, the error message is displayed in a dialog and the user is not redirected.

4.2.4 The configuration panel



✓ Page 1	stack	maximal
✓ Page 2	stack	maximal
✓ Page 3	stack	maximal
Page 4	stack	maximal

Constraints

assignedTo(2,4,1 | P1) samePage(5,3,0) requirePartialOrder(0,1,2,3)

Figure 4.4: Configuration panel

Page properties

In this panel the user can specify on how many pages the graph should be attempted to be embedded, by checking or unchecking the corresponding checkboxes.
The type and layout of each (checked) page can be chosen by two select menues.

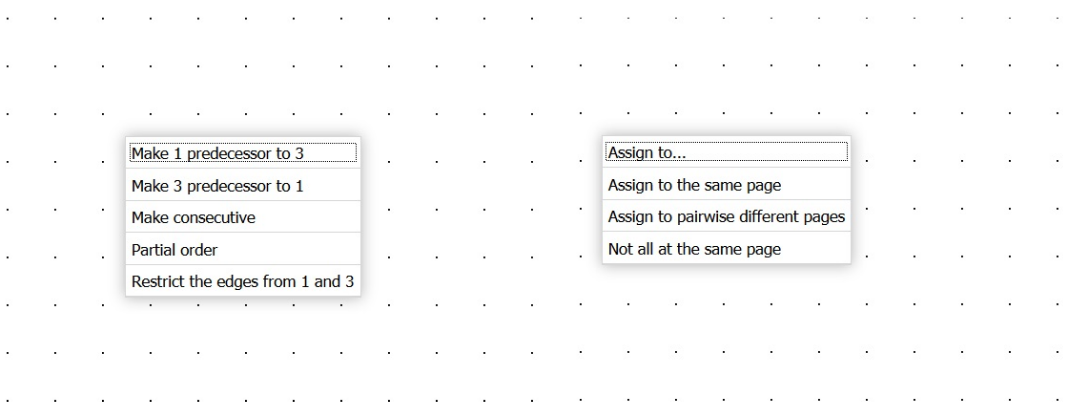


Figure 4.5: The available constraints in context menues

For types, the options *stack* and *queue* are available, for layouts *maximal* (meaning unrestricted), *tree*, *forest* and *matching* (meaning a dispersable embedding) are possible.

Constraints

The key feature of this project is the possibility to lso impose constraints on the linear layout. This is needed because the SAT-Solver produces solutions to the formula on random and sometimes a researcher might want to check whether a layout with specific properties exists. For an example see the chapter 5.
The constraints are based upon a constraint class which yields a subclass for each available constraint.
Constraints are created via the context menu that opens whenever a selection of nodes or edges is right clicked, see the context menu in Figure 4.5. During the runtime all active constraints are saved in an array and displayed as so called tags in the *Tag-It*[1] plugin, which is configured so the tags can be deleted but not edited.
If the user choses to delete an element from the graph the constraints corresponding to this element are also deleted, after reminding the user of this and giving the opportunity to cancel the deletion.

Restrict the linear order

To restrict the linear order of the vertices, the user can impose the following constraints:

1. **Predecessor** With this constraint the user can specify a relative order of

[1]`http://aehlke.github.io/tag-it/`

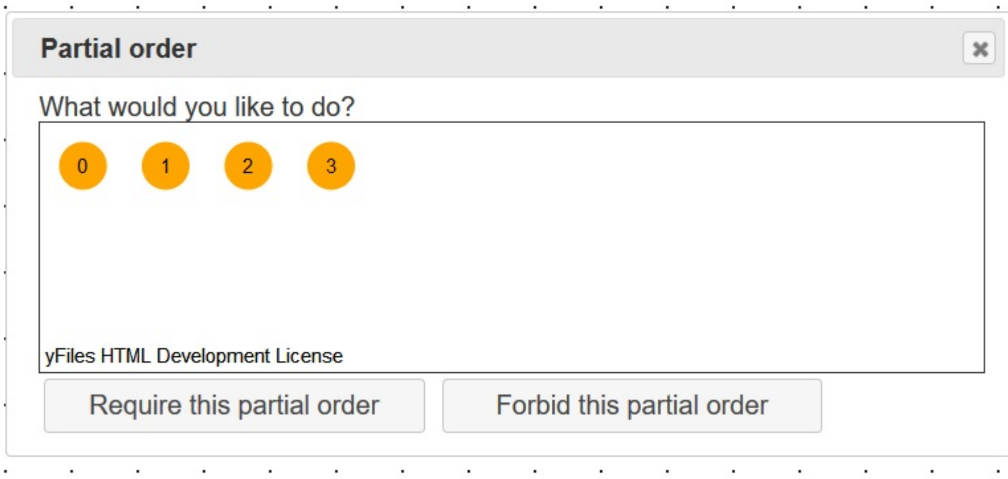

Partial order
What would you like to do?
0
1
2
3
yFiles HTML Development License
Require this partial order
Forbid this partial order

Figure 4.6: Partial order dialog

the nodes by defining one node as the predecessor of another. It implicitly also provides successorship, by reversing the predecessor relation.

2. **Consecutivity** With this constraint two nodes can be required to be consecutive in the linear layout. It does not restrain the order of the nodes further.

3. **Partial order** When a sequence of at least two nodes is selected, an absolute partial order of these nodes can be either required or forbidden, meaning the exact order in which the nodes have been selected, as to be seen in Figure 4.6. If the nodes are not selected individually the order is determined by the time of creation. For easier usage the order in question is displayed in a dialog.

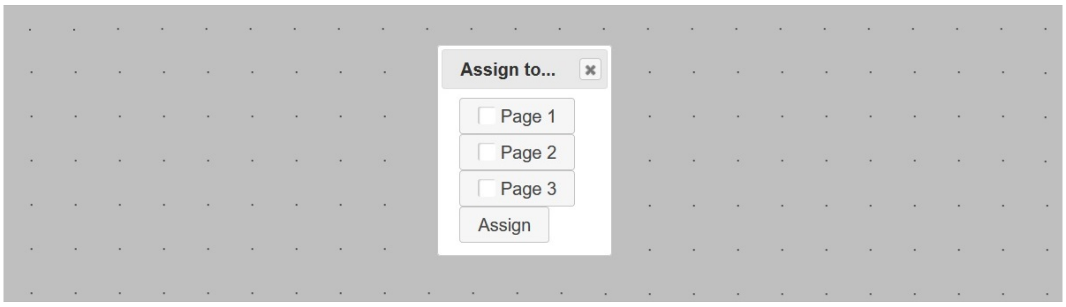


Figure 4.7: Edge assignment dialog

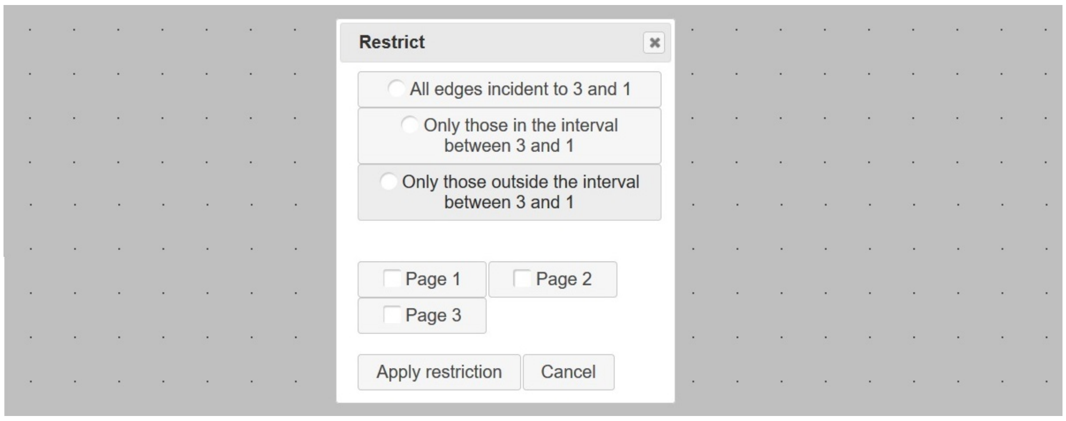

Restrict
All edges incident to 3 and 1
Only those in the interval between 3 and 1
Only those outside the interval between 3 and 1
Page 1
Page 2
Page 3
Apply restriction
Cancel

Figure 4.8: Incident edges dialog

Restrict the placement of edges

1. **Assign edges to certain pages** When at one or more edges are selected, right clicking and selecting "Assign to.." opens a dialog, where the user can choose on which pages of the embedding these edges can be located. The set of selected edges is not necessarily on the same page, if more than one page is selected.

2. **Assign edges to the same page** Similar to assigning to certain pages, the user can also specify that the selected edges should be placed on the same page.

3. **Assign edges to different pages** As long as the user does not select more edges then there are pages available, the user can assure that all selected edges are on different half planes of the layout by using this constraint.

4. **Assign the edges incident to certain nodes** When the user rightclickes on two or more nodes, she can also constraint all edges incident to these nodes.

If exactly two vertices, say u and v, are selected, the user can choose to restrain all edges incident to these nodes or to restrain the edges that will end either in the interval between u and v or the interval between v and u. If more than two edges are selected these two latter options are not available.

It is important to note that the corresponding constraints do only show up if the selection is exclusively nodes, respectively edges. Otherwise the context menu would be too unwieldy to use efficiently.

4.3 Linear Layouts Viewer

4.3.1 Overview

The viewer is structured similarly to the editor, with a toolbar on top, the viewing area in the middle and a panel on the bottom, where the imposed constraints are displayed and the appearance of the graph can be modified for easier examination.

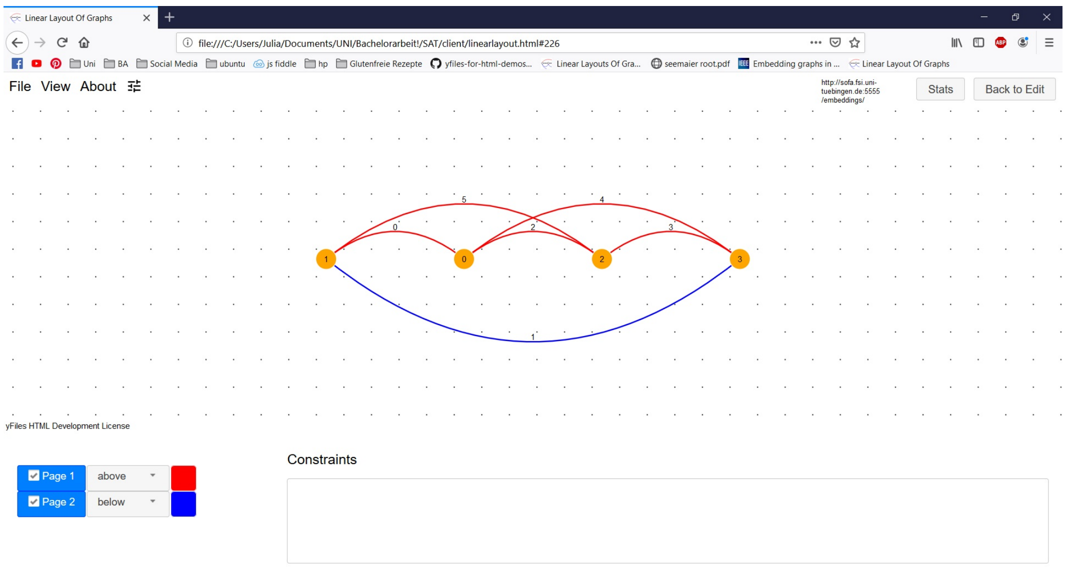
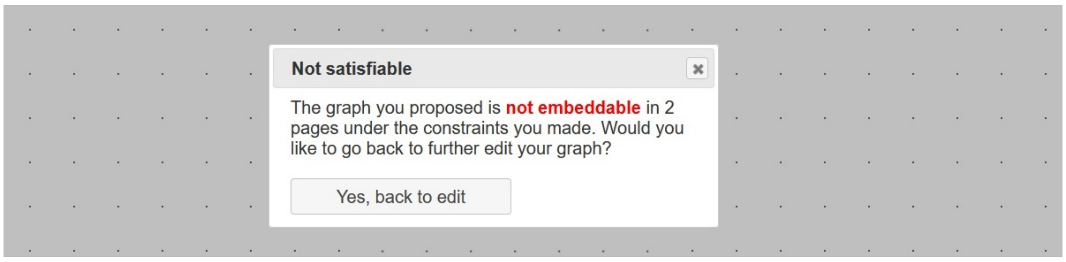

Linear Layout Of Graphs
file:///C:/Users/Julia/Documents/UNI/Bachelorarbeit!/SAT/client/linearlayout.html#226
File
View
About
http://sofa.fsi.uni-tuebingen.de:5555/embeddings/
Stats
Back to Edit
5
4
0
2
3
1
1
0
2
3
1
yFiles HTML Development License
Page 1
above
Page 2
below
Constraints

Figure 4.9: Overview of the viewing page

4.3.2 Linear layouts viewer

As the user accesses the second page, either directly or by redirection from the editor the url is checked for the hash parameter, where the id of the desired embedding should be located. If there is no id specified an error dialog shows up, otherwise an ajax-request is sent to the server.
The response is a JSON-encoded object and contains the following fields [?]


Not satisfiable

The graph you proposed is **not embeddable** in 2 pages under the constraints you made. Would you like to go back to further edit your graph?

Figure 4.10: Dialog when a graph is not embeddable as desired

id	the id of the embedding
graph	the 64-byte encoded graph that was sent to the server
pages	the pages of the embedding as JSON objects in a list, each with the attributes *id*, *type* and *layout*
constraints	a list of constraints as JSON objects. Each constraint object has the attributes ”type”, ”modifier” and ”arguments”.
status	a string, either ”IN_PROGRESS” or ”FINISHED”, determining if the server finished the computation of the embedding
vertex order	a list of strings, where each string is the tag of a vertex. This list is ordered like the vertices need to be ordered along the spine of the linear layout
satisfiable	a boolean, determining whether the graph was embeddable as proposed
rawSolverResult	the string as returned by the SAT solver
created	a timestamp in string form, e.g. *’2019-06-25T09:50:13.122499+00:00’*

After the website received the response from the server it checks the status of the response. Should it be ”IN_PROGRESS”, another request is sent after 5 seconds until the server finished computation. Then as the second step the website checks if the graph was embeddable in the way it was defined, which is stated in the field *satisfiable*. If it is not embeddable a dialog shows up, allowing the user to go back to editing the graph.

If it was indeed embeddable the website proceeds to modify the graph to display the linear layout in the following steps:

1. The graph is decoded and loaded into the viewer

2. The vertices are ordered according to the ”vertex order” field of the responded embedding

3. The edges are divided into arrays representing each half plane of the

graph, according to the ”assignment” field

4. Each edge is assigned the ”Arc Edge Style” provided by *yFiles for HTML*. In order to display the edges correctly one extra step is needed: the position of each arc is determined by the source and target nodes, that is to say edges with a source node right to the target node are displayed beyond the nodes and when the source node is left to the target the edge is displayed above the nodes. As the representation of a linear layout requires all edges of one page to be either above or below the spine, the source and targed nodes of the affected edges have to be switched.

5. For each array representing a page of the embedding, all edges in this array get assigned a color. Each page is positioned aternatingly above and below the spine by changing the height attribute of the affected arcs.

4.3.3 Toolbar

The toolbar of the second page is a thinned out version of the editors toolbar. The **file** submenu does no longer allow to load in graphMLs but still provides the saving and exporting options.

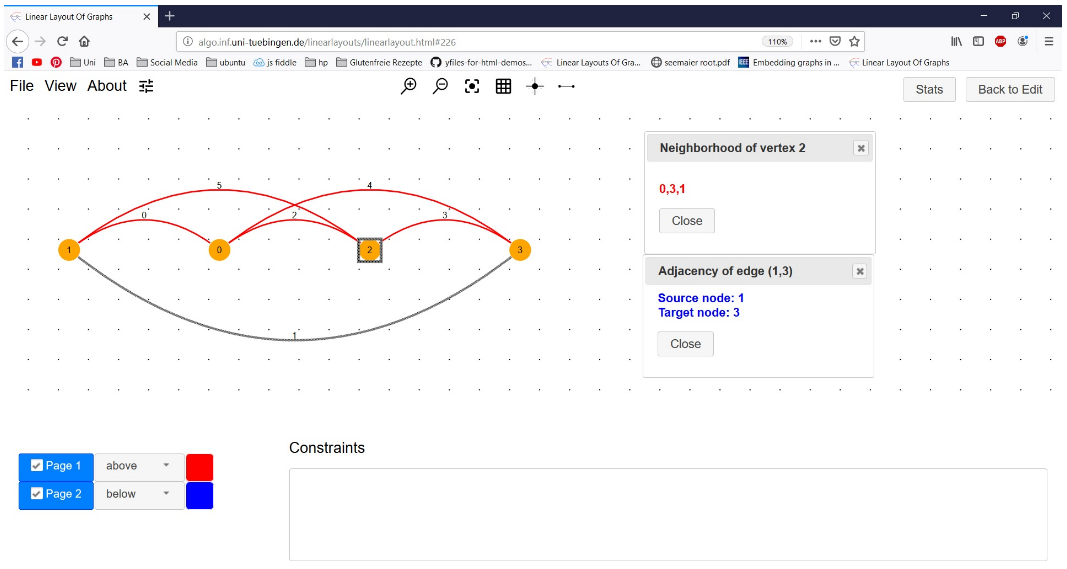
The **view** tab still has the same options as before, so the user can zoom in and out, center the graph on the screen and toggeling the visibility of the grid.
In addition to this it also provides the user with the option to examine the node neighborhood and edge adjacencies in the graph, as explained in **??**.
The *about* dialog can still be accessed on the second page and similar to the first page the user can hide the constraints panel at the bottom enlarge the graph viewing area. The *stats* button is in the same place as before, too.
Instead of a *compute* button, the toolbar contains a *back to edit* button that takes the user back to the editor, to either edit the original layout or the linear layout, which the user can choose in a dialog. This redirection is explained in Section 4.3.3

Neighborhood and adjacency dialogs

NandA] The *node neighborhood* and the *edge adjacency* buttons in the **view** tab of the toolbar, each open a dialog.

- The **node neighborhood** dialog holds information about the node that is currently selected and is only updated as a different, single node is selected. It lists the neighboring nodes and displays them in the color of the edge that connects the selected node to its neighbor.
- The **edge adjacency** dialog tells the user what is the source and end node of the edge currently selected. Similar to the *node neighborhood* it is only updated, when another edge is selected.

These dialogs proved to be useful to observe patterns in large embeddings, since following single edges turned out to be cumbersome.



Neighborhood of vertex 2

0,3,1

Adjacency of edge (1,3)

Source node: 1
Target node: 3

Figure 4.11: Neighborhood and Adjacency

Returning to edit the graph

When the the "Return to edit" button is clicked a dialog as seen in Figure 4.12 opens. The user can choose to either edit the linear layout or the original embedding of the graph. After choosing, a redirection back to the editing page is triggered. The correct redirection is obtained again by adding hash parameters to the url. If the user choses to edit the original layout, the url gets extended by "#or" and the id of the graph otherwise it is "#ll" followed by the id. That way a user can also access both layouts later on and even save them as a bookmark.
The editing page, recognizing if there is a hash parameter set, sends an ajax-request to the server similar to the viewing page. If the user wishes to

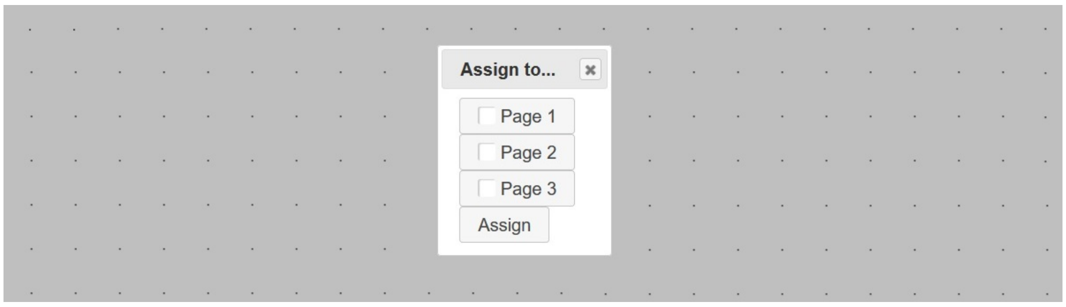


Figure 4.12: Return to edit dialog

edit the linear layout the same processes as described in subsection 4.3.2 are triggered. When the original embedding is to be edited, the graph is loaded into the editor. The edges still get colored in the respective page color, in order to make examination easier, otherwise the graph remains unchanged.

4.3.4 The configuration panel

Constraints

In the bottom part of the page, the constraints defined in the creation of the graph are displayed in a similar fashion to the editing page, except that the user can not delete a constraint. The constraints read in from the server response.

Page representation

For easier observation of the embedding, the graphs representation can be modified. Each page of the embedding can be hidden by unchecking the corresponding checkbox. Furthermore the position and color of each set of edges can be changed, the first by selecting either *above* or *below* in a select menu and the latter with the *Colorpick* plugin[2].

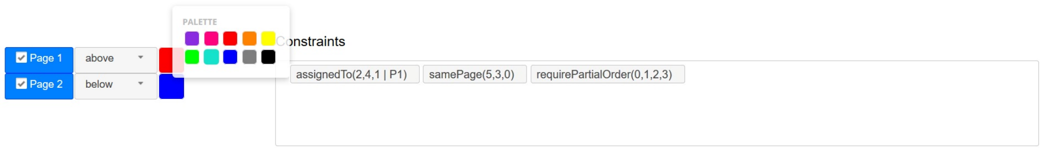


Figure 4.13: Pages and Constraints Panel

[2]https://github.com/philzet/ColorPick.js

Chapter 5

Experiments

This chapter describes the experiments that were done with this tool in the attempt to validate a proof sketch that Prof. Dr. Mihalis Yannakakis submitted [?].
As mentioned in chapter **??**, Prof. Yannakakis discovered that planar graphs need at most 4 pages in a stack embedding, but yet no researcher managed to find a planar graph that actually requires four pages to be embedded. Yannakakis himself gave a sketch on how to construct such a graph in an extended abstract of a symposium contribution, but ...
The proof sketch is made up of three steps, three graphs to be found in order to achieve a graph that is embeddable in 4 pages.

Step 1

In the first step

Step 2

Step 3

Chapter 6

Discussion and Outlook

All in all the project is considered a success. It's main functionalities, the creation of graphs as their drawings, the possibility to impose constraints on those and the presentation of the calculated linear layout are working as expected and help researchers to examine graphs and their linear embeddings. Though, through using the tool regularly and for large graphs, it proved to be inelegant and inefficient in some aspects.
One of the deficiencies that was already eliminated, was that stellating faces of a graph (a process that is used regularly in the research of linear layouts?) turned out to be very cumbersome, considering one had to add a node to each face and then connect this new node to each node delimiting the face. This insight lead to the implementation of the stellation buttons within the tools submenu (see **??**).
The other big issue concernes the creation of constraints. On big graphs, for example the graphs that were proposed by Prof. Dr. Yannakakis [**?**] and explained in **??**, there were at one point 150 constraints to be created. While for us this issue could be resolved through a script and copying the constraints into the already created graphML file, this is not really a solution for everyday usage as it is not user-friendly and very errorprone.
A considered solution was to add the constraints to the copy and paste commands, so when a set of elements is pasted all corresponding constraints should be as well. While this is certainly possible, it might turn out to be rather hindering than helpful. The other idea this behalf was to allow text input to the constraints field, which is actually supported by the *Tag-It* plugin used for this feature. This would make the creation of constraitns faster and more effective. As *Tag-It* even offers the possibility for auto completion, the user wouldn't even have to remember the constraints in their entirety.
This idea will also come in handy in future, since the plan is to add more costraints. Ultimately this will require a new solution for creating the constraints, as the context menu will be overcrowded and cumbersome to use.

List of Abbreviations

BLAST	Basic Local Alignment Search Tool
...	...

Appendix A

Further Tables and Figures

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum	Unterschrift