A generic widget library for Rapid-Prototyping of Graph Algorithms in Jupyter Notebooks

Bachelor Thesis of

Andreas Strobl

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

Reviewers:	Prof. Dr. Ignaz Rutter
Advisors:	Simon Fink

Time Period: 7th November 2021 – 7th February 2022

Statement of Authorship

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

Passau, February 3, 2022

Abstract

The Open Graph Drawing Framework (OGDF) is a C++ library containing algorithms and data structures for automatic graph drawing. However, due to C++ and its difficulties to set up, it is hard to use for rapid-prototyping. In this work, we try and solve this problem by creating a custom widget for Jupyter Notebooks, which gives the OGDF an interactive graphical interface. Jupyter Notebooks provides us with the ability to write multiple consecutive code cells and present the results underneath the code with the help of widgets. This widget is built onto the existing Python-package ogdf-python. Even without further optimization, this widget works up to 10000 nodes. Additionally, we provide demo applications which showcase the abilities of the widget in different contexts, e.g.: a graph editor, an algorithm debugger and an interactive way of teaching graph algorithms.

Deutsche Zusammenfassung

Das Open Graph Drawing Framework (OGDF) ist eine C++-Bibliothek, die Algorithmen und Datenstrukturen für das automatische Zeichnen von Graphen enthält. Aufgrund von C++ und seiner schwierigen Einrichtung ist es jedoch schwer für Rapid-Prototyping zu verwenden. In dieser Arbeit versuchen wir, dieses Problem zu lösen, indem wir ein benutzerdefiniertes Widget für Jupyter Notebooks erstellen, welches dem OGDF eine interaktive grafische Schnittstelle bietet. Jupyter Notebooks bietet uns die Möglichkeit, mehrere aufeinanderfolgende Codezellen zu schreiben und die Ergebnisse unterhalb des Codes mit Hilfe von Widgets zu präsentieren. Dieses Widget basiert auf dem breits bestehenden Python-Paket ogdf-python. Auch ohne weitere Optimierung funktioniert dieses Widget bis zu 10000 Knoten. Zusätzlich stellen wir Demo-Anwendungen zur Verfügung, die die Fähigkeiten des Widgets in verschiedenen Kontexten demonstrieren, z.B.: einen Graphen-Editor, einen Algorithmus-Debugger und eine interaktive Möglichkeit, Graph-Algorithmen zu unterrichten.

Contents

1. Introduction

The Open Graph Drawing Framework (OGDF) is a C++ library containing algorithms and data structures for automatic graph drawing. However, due to C++ and its difficulties to set up, it is hard to use for rapid-prototyping. To try and solve this problem, this work creates a widget for Jupyter Notebooks, which aims to make rapid-prototyping accessible and faster to achieve. To enable rapid-prototyping, we need to reduce the ten steps[1] it takes to set up a C++ project down to a few clicks. To verify our work, we define use cases which we try to solve in demo applications, e.g. a graph editor.

Since we want to use Jupyter Notebooks, this work is based on ogdf-python, which is a Python-package that generates Python bindings for the OGDF. It uses cppyy to generate those bindings, allowing us to work with the OGDF from Python. This already solved some of the overhead problems that existed with the OGDF. However, there are still problems remaining, especially not having an interactive representation of a graph.

To solve the remaining problems with the OGDF we developed a custom widget for Jupyter Notebooks. Jupyter Notebooks provides multiple consecutive code cells and presents the results underneath the cells. The variables are global and will be stored between execution. Creating a custom widget allows us to have full control over the visual representation of the graph. It also allows us to make the graph interactive by using an already existing JavaScript library to visualize the graph. We will be designing use cases that are aimed to solve the problems that the OGDF has. The goal is to create demo applications which are used to verify that the designed widget really solved the overhead problems that we stated.

The thesis starts by stating all the problems that currently arise when using the OGDF and elaborates on our approach for a solution. In Chapter 2 we define three different use cases from which we derive requirements and features. In Chapter 3, we explain the different technologies that were used and how they are communicating with each other. Then we explain in detail how the front- and backend of the widget works before explaining how the source code is organized and how the widget is installed. Chapter 4 explains the features stated in Chapter 2 and how they were realized together with the implementation. Chapter 5 picks up the use cases from Chapter 2 and shows how the desired functionality of the use cases might be implemented. It also takes a look at the performance of the widget.

[1]https://docs.microsoft.com/en-us/cpp/windows/walkthrough-creating-a-standard-cpp-program-cpp?view=msvc-170

2. Use Cases

This chapter is about the current problems that occur when working with the OGDF and the possible solution this work provides. Exemplary, three use cases are examined, which are used to derive requirements and necessary features and thus guide this work. The use cases will also be used later to verify the solution developed in this thesis.

2.1 Current Problems

Working with the OGDF currently is a lot more tedious than it needs to be. The main problem is debugging and looking at a graph while an algorithm we designed is executed, because there is no simple way to visualize the current state of a graph during execution. A really common way for programmers is to use a debugger, which lets them halt a program at any given point and examine its variables' states, in this case the current state of the graph. This is also possible when using the OGDF, but for a human, it becomes really hard to visualize the graph when only seeing text and e.g. positions of a node. Therefore, a common way of debugging currently is to use the OGDF's built-in functionality of exporting the graph as an image. This can be used in every step of an algorithm, resulting in numerous images. Their main problem is that they are static and cannot be interacted with. After running an algorithm, one has to manually go through the images and try to detect a bug. Also, a big downside is that we are not really able to visualize some internal data. The only way to do that is by writing the data into the nodes' and links' label. Additionally, there is no synchronization between the pictures and the internal data. Since we cannot put everything into the nodes' or links' label, there is no way to map the graph's appearance back to the internal data used inside the OGDF. Besides that, we will not have the chance to see internal data step by step, where we could decide to change it while the algorithm is running to test some debug idea we might have. This also means that we can only really debug at points where we thought that it might make sense to dump a picture of the graph. We do not have the comfort of a normal debugger giving us the freedom to freely navigate through our code at runtime. This being said, it becomes obvious why debugging algorithms with the OGDF is not very convenient.

Another problem with the OGDF is manually editing graphs. The OGDF's only possibility to edit a graph is by either making those changes through code, or by manually changing files exported by the OGDF and reimporting them. However, there are external tools where we can export our graph to and then reimport it to the OGDF. Unfortunately, none of these programs can provide a lossless import and export. That's why we sometimes need to make certain changes through code, which is tedious.

Lastly, it is unfortunately not possible to make an algorithm interactive. It would be great for a student to learn new algorithms by experimenting with them step by step and even have the possibility to manipulate internal values to better understand an algorithm. The OGDF also has a hard time showcasing algorithms in a very good way. As stated above, it is only possible to watch an algorithm step by step if we generate an image every step of the algorithm. Therefore, it tends to not be suitable for educational purposes because this process is very static.

2.2 Solution Approach

We quickly realized that an approach close to the data science workflow [ZMW20] would be perfect for solving all these problems. In data science, it is popular to publish documents created with Jupyter Notebooks, because they provide an easy-to-use and interactive environment.[1] The created notebooks also work as an integrated development environment (IDE). They hand the user a flexible way to combine images, code, plots and comments. They provide an environment where code can be executed, and its results are instantly presented.

Therefore, we wanted to provide a widget for Jupyter Notebooks, which adds the same comfort for working with graphs. The basis of this work is the OGDF and ogdf-python, which offers Python bindings for the OGDF using cppyy (more information can be read in Chapter 4). The Python-package ogdf-python already provides a visual representation, but its representation is too static to reasonably work with. Consequently, the widget needs to provide an interactive graphical representation of a graph given from the OGDF. This solves the problem of having the visualization disconnected from the data. Inside the notebook, the widget is an inline representation of the graph, and it provides interactivity. It essentially uses the same method that data scientists [WMBO19] used for a long time (e.g. time series data) but for graphs. It also allows for manipulating data while working with the graph, since Jupyter Notebooks works incrementally on the data in the notebook. Jupyter Notebooks supplies the user with a representation of the graph, while working on their code. Another feature, provided by Python, is that we are able to halt the algorithm at any point during its execution. With the provided representation, we can see the current state of the graph, which makes debugging easier and faster [KM04, Gri08]. It is also possible to manipulate values while executing the algorithm to see its changed behavior. This provides almost all functionality a normal debugger has.

Since the widget is interactive, we can also solve the problem of graph editing by simply using the provided tools to build our own OGDF graph editor. This graph editor works lossless because it manipulates the OGDF's data itself and no export or import to external programs is required.

The last problem stated above was that showing and learning new algorithms is barely possible with the OGDF. Since the widget will provide some sort of interactivity, it should be possible to create notebooks where we can interact and change the algorithms' behavior. It should also provide a visual way to see the changes that the algorithm made. Therefore, it is a very powerful tool which can be used in education to better explain the algorithm. It can also be used as an addition to scientific papers, simply to provide an implementation we can easily experiment with and therefore better understand the theory behind an algorithm[2].

[1]https://medium.com/@ODSC/why-you-should-be-using-jupyter-notebooks-ea2e568c59f2
[2]https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/

2.3 Use Cases

In this section we name and explain three different use cases which will help us to later identify requirements and features for this widget. These use cases will also be used later to evaluate if the widget provides a sufficient solution.

Graph editor

With the widget, we should be able to build a graph editor. With the graph editor, we should be able to move around the graph and interact with the nodes and links. Interacting includes adding, deleting, moving or coloring nodes or links. We should also be able to move, add or delete bends. With a graph editor that works directly with the OGDF as its data structure, we can ensure lossless graph editing. This allows us to modify existing graph instances with a graphical interface improving the workflow, because it is easier to work on a visual representation, than on a textual representation. This will obviously work for empty graph instances as well, therefore allowing us to create our own graphs from nothing. The features used in this use case can also be combined with other use cases to, for example, change a graph's structure (e.g. deleting a link), while an algorithm is executed on it and then observe the algorithms' changed behavior.

Visualizing algorithms

The widget should provide a platform that enables us to present algorithms. This can be used by professors who are teaching algorithms. They can implement the algorithm and then provide their students with the notebook. Trying out the algorithm for themselves, and not simply watching a fixed example, will help the students to better understand how the given algorithm is working. To show algorithms, it is important to visualize user defined data and also make internal data accessible, to change if needed. It therefore should be possible to implement the Push-Relabel-Algorithm, which is a maximum-flow algorithm. Since this algorithm has two main operations (push and relabel) it is possible to implement it interactively, i.e. performing a push operation if the user clicks on a link and performing a relabel-operation once he clicks a node.

Algorithm debugging

With the provided features of the widget, it should be possible to significantly improve the debugging process. Debugging and visualizing an algorithm are very similar. The main difference is that when debugging an algorithm, one can find the error while looking at the algorithm and assuming that it is not running correctly. Another difference is the amount of effort one needs to put in to visually see the necessary data. Therefore, this use case should prove that the debugging process is greatly simplified with the widget. To show this, we create a graph and show how one can easily see internal data with little effort. We also show how fast and easy it is to apply a simple layout to the graph without having to figure out how the OGDF's layouts work.

2.4 Requirements

In this section, we list the requirements that derive from the use cases that are listed above. These requirements will guide the implementation process and help to verify the widget.

Before stating the requirements, we first have to distinguish between two different types of users. There is the user who is going to use this widget as a tool to create a notebook, e.g. a professor who wants to teach his students an algorithm interactively and therefore uses the widget in a notebook. In the following section, this user will be referred to as a developer. On the other hand, we have the user that actually interacts with the widget and therefore is not concerned about anything that has to do with the code that created the features he uses. In the following section, this user will be referred to as a user. In the previous example, this user would be a student of the professor who receives the notebook to learn the functionality of the algorithm using the notebook without even looking at the code.

- **Interactive** The widget needs to be interactive, meaning the developer should have a way to detect interaction with the visualized graph. There needs to be feedback for the developer when the visualization has been interacted with. It should also provide the ability to navigate around a graph, to see bigger graphs with more detail.
- **Simple use case coverage** Since every stated use case needs to be covered, the widget has to be as simple to use as possible. For a developer who wants to create notebooks with the widget, there has to be as little overhead as possible. This being said, the widget needs to provide the necessary tools to make the developers' work as easy as possible.
- **Generic for future use cases** Being generic is required for the widget because we do not want to be limited to the use cases stated above. These use cases only serve the purpose of showcasing a broad range of things the widget should be capable of. It should be possible to hand the widget different types of graphs that will then all be displayed correctly, and there should hardly be any limitation to what the widget is capable of. Therefore, having the possibility to create our own functionality via code is crucial.
- **Accurate graph representation** Accurately representing the graph is really important, since it is the only thing that the user will see. For that reason, it has to be an exact representation of the given graphs. Here it is also important to monitor changes to the graph in real-time and change its representation accordingly. It will also be beneficial to display said changes in an obvious way.

2.5 Features

In this section, we break the specified requirements down to the features and partially explain how they work. For more detailed information, look at Chapter 4.

First, let's have a look what features derive from the **Interactive** requirement:

- **Moving around the graph** The user should be able to zoom into the graph and easily move around with his mouse. This will be a great feature, especially for huge graphs, because when he is able to zoom he can look at sections of the graph in great detail without having to isolate them into separate graphs.
- **Click callbacks** Having click callbacks on the links, nodes and the background will be great for customization. It allows the developer to program his own functionality upon interaction with the graph is detected.
- **Move Links/Nodes** Moving the nodes around could easily be done with code. But moving nodes around the actual depiction will have the benefit of actually seeing the changes in real-time. For links, it needs to be possible to move their bends around.

Now, let's focus on the requirement **Simple use case coverage**.

- **Helper functions** Providing the developer with helper functions will simplify using this widget. There are many things that will often be needed, like converting between the screen position and OGDF's internal coordinates. For a developer, figuring this conversion out would be tedious. This can easily be avoided by providing helper functions for things that will be needed by a developer.

- **Overall usability** The widget should provide a high level of usability, to make it simple to use. This can be achieved by sending out easy to understand errors, if an input is not the way it is anticipated. We can also achieve usability by making it as simple as possible to look at a graph. The widget should offer the option to look at the graph without any overhead, but for that reason restrict interactivity.
- **Provide predefined layouts** Another way to achieve usability is by providing the developer with ways to display their graphs without needing to know anything about OGDF's layout algorithms. Therefore, the widget should provide the option to simply select a layout for the graph without the need to write any extra code. There needs to be an option to simply activate either a simple OGDF layout or an externally calculated layout.

Next, we have a look at the requirement **Generic for future use cases**.

- **Only provide tools** Only providing the tools and not fixing functionality in a restricted way is needed to guarantee the flexibility of this widget. This can be achieved by not having predefined functionality when e.g. a node is interacted with, we just inform the developer. Then the developer can use this and build his desired functionality.
- **Built-in functionality has to be activated** Following up on the feature stated above, built-in functionality has to be activated to be used. The widget provides functionality which is deactivated by default. This ensures that no unwanted behavior is accidentally activated upon interacting with the widget.

Lastly, we elaborate the requirement **Accurate graph representation**.

- **Displaying all visual attributes** Displaying all of OGDF's graph attributes is key, because these are the main things the user will see and interact with. Therefore, displaying position, color, label, stroke, size, etc. of a node or link is required.
- **Add/Delete Node/Links** Detecting changes like addition or deletion of a graph-element can be achieved by using OGDF's `GraphObserver`, which notifies us when this sort of change happens.
- **Update Nodes/Links** For changes that cannot be detected by the `GraphObserver`, e.g. changing position of a node, changing color of a node, etc. the developer needs to have the possibility to manually notify the widget. The widget will then update the necessary graph-elements.
- **Animate changes** Animating changes will have a great effect on the user experience, because it will help to comprehend changes that happened. If we e.g. change the position of many nodes that are not easily distinguishable, it can be confusing to see which node went where. Therefore, animating such changes to see a node move from position A to B will help to better understand changes.

3. Tech Stack

In this chapter, we present all the technologies that were used to design the widget. We also have a closer look at some particular properties that are used to produce the widget. Finally, we take a look at how the source code is organized and how the widget is installed and build.

3.1 Communication Between Technologies

This section will concentrate on the different technologies used to create this widget and how they interact with each other. We will start by stating the base technologies and keep on adding onto that, until all used technologies and their benefits are stated.

The Open Graph Drawing Framework (OGDF) [CGJ+12] is a C++ library containing algorithms and data structures for automatic graph drawing. Its algorithms and data structures are very powerful, but unfortunately working with the framework can at times create a lot of overhead. The main problem is that the visual part and the data underlying it are disconnected. It therefore can be hard to map plotted pictures back to stages of an algorithm. This disconnection between actual internal data and a visual representation makes it harder to debug algorithms. Since the OGDF is written in C++, it has the same overhead as any other C++ library. Just setting up a new project to quickly try out an idea is tedious because of compiler problems, library paths, the main method or the make files. It therefore completely eliminates the possibility of rapid-prototyping because C++ is not the right language to quickly try out a simple idea.

To reduce this overhead, we need to take a look at cppyy [LD16]. The Python-package cppyy is an automatic Python-C++ bindings generator for calling C++ in Python. It provides an almost completely automatically generated Python interface to any given C++ library. In our case, we can use it to generate Python bindings for the OGDF. This solves the problems that the language C++ brings with itself, and it should also reduce the time to set up a project for quick ideas.

We get further useful benefits from cppyy by introducing Jupyter Notebooks [KRKP+16, PG07]. Jupyter Notebooks provides an easy-to-use, interactive environment. These notebooks also work as an integrated development environment (IDE). Jupyter Notebooks hands the user a flexible way to combine images, code, plots and comments. Jupyter Notebooks provides multiple consecutive code cells and presents the results underneath the cells. The variables are global and stored between execution. The order in which cells are executed is not relevant, neither is the amount of times a single cell is executed. This is great for rapid-prototyping, since we can incrementally work on data, which provides a better workflow than starting all over again every time we make a change. Jupyter Notebook runs a server which talks to a kernel to execute the Python code in the notebooks. The results of the executed code are represented with the notebook inside the browser using HTML and JavaScript. The technology described up until here was already given by ogdf-python, which displays a static SVG image of a graph created by the OGDF inside Jupyter Notebooks. This already improves the workflow with the OGDF by a lot. Now it is possible to write code in Python and easily display it with Jupyter Notebooks.

At this point, the remaining aspect that should be improved is the interactivity of the presented graph, since it currently only displays a static SVG. The first technology used to solve this problem is ipywidgets[1] which is used to create custom widgets for Jupyter Notebook. A widget inside Jupyter Notebook is an easy-to-use software component, mostly existing to solve a specific problem. In our case, this problem is that we do not have an interactive representation of a graph. This provides us with the ability to create an interactive frontend with JavaScript for the browser. It also allows us to control the backend functionality which is running on a kernel inside a server of a Jupyter Notebook instance. It hands more control to those who create notebooks and reduces the work for those who use them. The framework ipywidgets internally works with traitlets[2] on the Python side and with backbone.js[3] on the JavaScript side. The traitlets are data types, specified on the Python side, which are automatically synchronized to the JavaScript side by backbone.js. The traitlets and backbone.js use a web socket-based communication channel (comm) to synchronize the data. This channel can also be used to send custom messages from the backend to the frontend if anything changes in the data and vice-versa.

With the previously stated technologies, the only thing missing is to render an interactive representation of the graph. We no longer use the static SVG provided by the OGDF or any part of OGDF's rendering. We mimic the representation with the help of our final technology d3.js [BOH11]. The JavaScript library d3.js helps to visualize data, in this case on an SVG. It provides features like zoom capabilities or easy-to-implement drag functionality. With d3.js, we are also able to animate changes in the graph and provide a live force-layout for the graph. In our application d3.js is responsible for the whole frontend including detecting click events. These can be sent to the backend using the comm, which enables us to implement the desired interactivity that one might need.

3.2 Models

front- and backend

The widget is split into front- and backend. The backend (see Listing 3.1) is programmed in Python and the frontend (see Listing 3.2) is programmed in JavaScript. Ipywidgets handles the instantiating and mapping between both as long as their properties have the same values.

[1]https://ipywidgets.readthedocs.io/en/latest/
[2]https://traitlets.readthedocs.io/en/stable/
[3]https://backbonejs.org/

```python
@widgets.register
class Widget(widgets.DOMWidget):
    # Name of the widget view class in frontend
    _view_name = Unicode('WidgetView').tag(sync=True)
    # Name of the widget model class in frontend
    _model_name = Unicode('WidgetModel').tag(sync=True)
    # Name of the frontend module containing widget view
    _view_module = Unicode('ogdf-python-widget').tag(sync=True)
    # Name of the frontend module containing widget model
    _model_module = Unicode('ogdf-python-widget').tag(sync=True)
    # Version of the frontend module containing widget view
    _view_module_version = Unicode('^0.1.0').tag(sync=True)
    # Version of the frontend module containing widget model
    _model_module_version = Unicode('^0.1.0').tag(sync=True)
    ...
```

Listing 3.1: Backend Python code for widget instantiating.

```javascript
let WidgetModel = widgets.DOMWidgetModel.extend({
    defaults: _.extend(widgets.DOMWidgetModel.prototype.defaults(), {
        _model_name: 'WidgetModel',
        _view_name: 'WidgetView',
        _model_module: 'ogdf-python-widget',
        _view_module: 'ogdf-python-widget',
        _model_module_version: '0.1.0',
        _view_module_version: '0.1.0',
        ...
    })
});

let WidgetView = widgets.DOMWidgetView.extend({
    ...
});

module.exports = {
    WidgetModel: WidgetModel,
    WidgetView: WidgetView
};
```

Listing 3.2: Frontend JavaScript code for widget instantiating.

```python
class Widget(widgets.DOMWidget):
    def __init__(self): #constructor
        super().__init__() #call parent constructor
        self.on_msg(self.handle_msg) #bind incoming msg to method
        self.send(...) #send message to frontend

    def handle_msg(self, *args): #handles messages
        msg = args[1] #2nd positional parameter is the message
        ...
```

Listing 3.3: Backend Python code for custom messages.

```
let WidgetView = widgets.DOMWidgetView.extend({
    initialize: function (parameters) { //override constructor
        //call parent constructor
        WidgetView.__super__.initialize.call(this, parameters);
        //bind incoming msg to function
        this.model.on('msg:custom', this.handle_msg.bind(this));
        //send msg to backend
        this.send(...)
    },

    handle_msg: function (msg) { //handles messages
        ...
    }
}
```

Listing 3.4: Frontend JavaScript code for custom messages.

custom messages

Custom messages play an important role for this widget, since the synchronization of model properties is not powerful enough to support the communication, that is needed between the front- and backend. Therefore, using the comm channel for custom messages proved to be the best way to implement the remaining communication that is needed for the widget. To send and receive messages on the Python side (see Listing 3.3), we have to bind a method to receive the messages and use the already existing method to send a message. The message object that is sent and received is a Python Dictionary.

On the JavaScript side (see Listing 3.4), this is more difficult, since the callback does not yet exist. Therefore, the constructor needs to be overwritten. We can bind a function to the custom message callback, which gets called upon receiving a message. Sending a message works exactly like on the Python side. The message object that is sent and received is a JSON object.

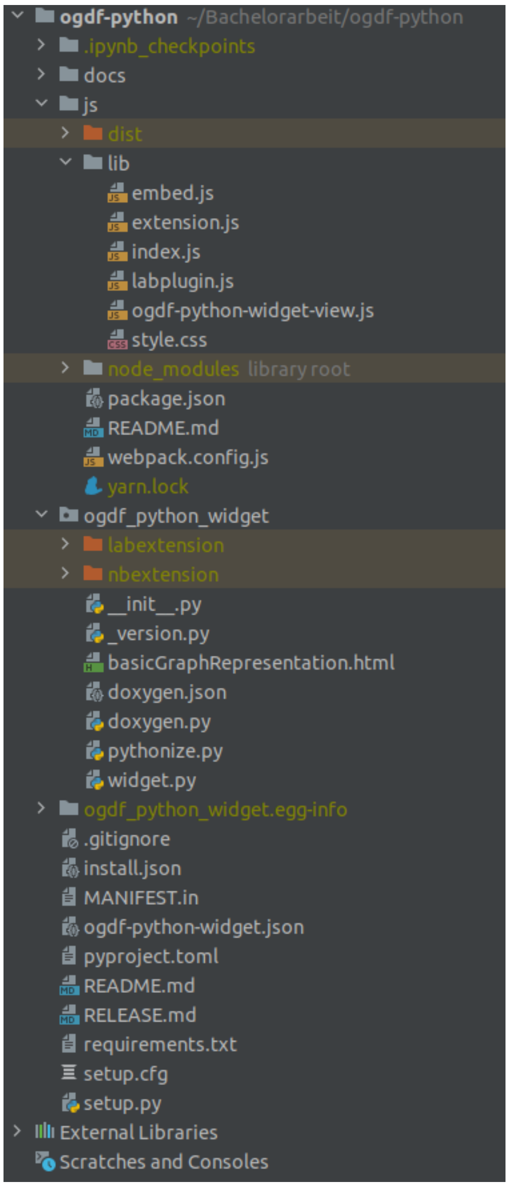
Traitlets and backbone.js

As stated earlier, Traitlets and backbone.js are responsible for syncing data between the front- and backend. This mechanism is used as follows. First, we need to declare the property in the backend, e.g.: `width = Integer(960).tag(sync=True)`. Here we declared a property named `width`, being an `Integer` with the default value of 960. The tag is needed to make sure the property is always synced to the backend. In the frontend, we can use the provided callback from backbone.js to detect changes to the property by using: `this.model.on('change:width', this.svgSizeChanged, this)`. This will call the given method (`this.svgSizeChanged`) if the property `width` changes in the backend. In this method, we can get the value of the property by using `this.model.get('width')`.

Since there is no need in this widget to sync any changes from the front- to the backend using this mechanism, we can ignore this case.

3.3 Source Code Organization

In this section, we are going to talk about the folder structure (see Figure 3.1) of the whole project and how it is built.



```
ogdf-python ~/Bachelorarbeit/ogdf-python
  .ipynb_checkpoints
  docs
  js
    dist
    lib
      embed.js
      extension.js
      index.js
      labplugin.js
      ogdf-python-widget-view.js
      style.css
    node_modules library root
    package.json
    README.md
    webpack.config.js
    yarn.lock
  ogdf_python_widget
    labextension
    nbextension
    __init__.py
    _version.py
    basicGraphRepresentation.html
    doxygen.json
    doxygen.py
    pythonize.py
    widget.py
  ogdf_python_widget.egg-info
  .gitignore
  install.json
  MANIFEST.in
  ogdf-python-widget.json
  pyproject.toml
  README.md
  RELEASE.md
  requirements.txt
  setup.cfg
  setup.py
External Libraries
Scratches and Consoles
```

Figure 3.1: Source code organization of the project.

3.3.1 Project Structure

- **/docs** This folder contains all demos and tests for the widget.
- **/js**
  - **/lib/index.js** entry point to widget library
  - **/lib/embed.js** webpack entry point for standalone HTML page
  - **/lib/extension.js** entry point for classic notebook for loading widget and lists d3.js dependency
  - **/lib/labplugin.js** entry point for jupyter lab
  - **/lib/ogdf-python-widget-view.js** frontend source code for widget
  - **/lib/style.css** contains all the set styling for the widget
  - **/package.json** npm package definition for frontend

  - **/webpack.config.js** configures bundler entry points and outputs

- **/ogdf_python_widget**
  - **/labextension and /nbextension** output folder for built frontend (generated using: `yarn run build`)
  - **/___init___.py** allows ogdf_python_widget to be imported
  - **/_version.py** contains version for front- and backend
  - **/basicGraphRepresentation.html** represents graph without installing the widget in a non-interactive way
  - **/doxygen.json** provided by ogdf-python
  - **/doxygen.py** provided by ogdf-python
  - **/pythonize.py** provided by ogdf-python
  - **/widget.py** source file containing the widget
- **/install.json** contains data for setup.py like packageName etc.
- **/ogdf-python-widget.json** enables Jupyter to load extension
- **/pyproject.toml** lists dependencies to build project
- **/requirements.txt** lists dependencies for runtime
- **/setup.cfg** marks Python package as universal wheel
- **/setup.py** installation script for backend

3.3.2 Installation

```
git clone https://github.com/N-Coder/ogdf-python
cd ogdf-python/
git checkout even_better_visualization
pip install .
jupyter nbextension install --py --symlink --overwrite --sys-prefix ogdf_python_widget
jupyter nbextension enable --py --sys-prefix ogdf_python_widget
cd docs/examples/
jupyter notebook
```

3.3.3 Building the Widget

To rebuild the widget after changes in the frontend, the command `yarn run build` needs to be executed in the directory `/ogdf-python/js`. This is going to rebuild the frontend of the widget into the output folders `/labextension` and `/nbextension`. For changes on the frontend to take effect, one needs to refresh the browser tab in which jupyter notebook uses the widget. After changes in the backend code were made, the command `pip install .` needs to be executed. For the changes to take effect, the kernel needs to be restarted.

4. Features and their Implementation

This chapter presents the interface of the custom widget and takes a look at the implementation details for every feature this widget provides.

4.1 Interface

In this section, we present the interface of the widget.

```python
class Widget():
    width: int
    height: int
    x_pos: int
    y_pos: int
    zoom: int
    click_thickness: int
    animation_duration: int
    force_config: Dict
    rescale_on_resize: bool

    #on_node_click_callback(node, alt_pressed, ctrl_pressed)
    on_node_click_callback: Callable[[ogdf.node, bool, bool], None]
    #on_link_click_callback(link, alt_pressed, ctrl_pressed)
    on_link_click_callback: Callable[[ogdf.link, bool, bool], None]
    #on_svg_click_callback(x, y, alt_pressed, ctrl_pressed, background_clicked)
    on_svg_click_callback: Callable[[int, int, bool, bool, bool], None]
    #on_bend_clicked_callback(link, bend_index, alt_pressed)
    on_bend_clicked_callback: Callable[[ogdf.link, int, bool], None]
    #on_node_moved_callback(node, x, y)
    on_node_moved_callback: Callable[[ogdf.node, int, int], None]
    #on_bend_moved_callback(link, x, y, bend_index)
    on_bend_moved_callback: Callable[[ogdf.link, int, int, int], None]

    def __init__(self, graph_attributes, debug: bool = ...) -> None: ...
    def set_graph_attributes(self, graph_attributes) -> None: ...
    def update_graph_attributes(self, graph_attributes) -> None: ...
    def refresh_graph(self) -> None: ...
    def start_force_directed(self, charge_force: int = ...,
        force_center_x: int | None = ..., force_center_y: int | None = ...,
        fix_start_position: bool = ...) -> None: ...
    def stop_force_directed(self) -> None: ...
    def get_node_from_id(self, node_id: int): -> ogdf.node ...
    def get_link_from_id(self, link_id: int): -> ogdf.link ...
    def enable_node_movement(self, enable: bool) -> None: ...
    def move_link(self, link) -> None: ...
    def remove_all_bend_movers(self) -> None: ...
    def remove_bend_mover_for_id(self, link_id: int) -> None: ...
    def update_node(self, node, animated: bool = ...) -> None: ...
    def update_link(self, link, animated: bool = ...) -> None: ...
    def update_all_nodes(self, animated: bool = ...) -> None: ...
    def update_all_links(self, animated: bool = ...) -> None: ...
    def download_svg(self, file_name: String = ...) -> None: ...
    def svgcoords_to_graphcoords(self, svg_x: int, svg_y: int) -> Dict: ...
```

4.2 Implementation Details

In this section, we present every feature of the widget and explain how we implemented them respectively.

- **Representing `GraphAttributes`:** When instantiating the widget, a reference of `GraphAttributes` is required. From the `GraphAttributes` the important features like position, color, etc. are extracted and serialized into a `Dict`. This dictionary is then sent towards the frontend with custom messages. There the dictionary arrives in a JSON format and is then visually represented with the help of d3.js. The system also offers a visual representation without instantiating a widget. If there is no need for any interactive representation, an instance of `GraphAttributes` or `Graph` can be displayed by just typing the object name at the end of a Jupyter Notebook cell. This is done by overwriting the internal `_repr_html()`-method. This representation internally accesses the same methods that are used for the widgets' representation.

- **Zooming and moving:** To move and zoom, the SVG is equipped with zoom handlers provided by d3.js. The only custom implemented feature here is that upon initialization, the zoom and position will always cover the whole graph. This is done by finding the bounding box of the graph and then converting it to a position and zoom level so the SVG fits the whole graph. If we want to zoom via code, the variables `x_pos`, `y_pos` and `zoom` are provided. Upon changing them in the backend, they will be automatically synced to the frontend and applied to the SVG. This synchronization also works in the other direction when the SVG is moved and zoomed.

- **Resize SVG:** To resize the SVG, we only need to change the `width` and `height` variables in the backend. This will, similar to the `zoom` etc., be synced to the frontend and will be applied.

- **Downloading SVG:** To start the download of the SVG, the method `download_svg` has to be called. By default, the filename of the SVG will be the current date and time. The filename can be changed by setting the optional parameter `file_name` to the desired filename. The downloaded SVG will be the same SVG as the one currently viewed with the widget, including zoom level and zoom position. After calling the method, the backend will inform the frontend with a custom message to start the download of the SVG. In the frontend, we set the namespaces of the

SVG and create a URL to download it. After that, we artificially click the URL to automatically start the download.

- **Easier click thickness:** Since links between nodes are most likely going to be small, a widget attribute called the click thickness is provided to let the user choose the clickable thickness of a link, to make it easier to click on. This is realized by adding a transparent, thicker path on top of the visible path. This path is responsible for click events. The clicking thickness can be modified by changing its value in the backend variable `click_thickness`. The frontend will listen for changes of this synced variable and will change the thickness of the transparent layer.

- **Force-directed layout:** Compared to the OGDF's static graph layouts, which are calculated once, a force-directed graph layout is a physics simulation which is calculated multiple times a second. It works by assigning forces to nodes and links and then simulating the positions with physics. With a force-directed layout the graph feels more alive, e.g. changing a node's position will most likely affect all other nodes because there are forces between all nodes and links.

  To activate the force-directed layout, the method `start_force_directed(...)` can be called where we can also specify the force strength and the center of that force. It also offers the possibility to set the starting positions of the nodes to a fixed position, which is activated by default. This makes sense, if we already have a layout, and we do not want to have all of our nodes moving instantly. When the force layout is activated, we are also able to move nodes around. Compared to a non-force-directed layout, it works a bit different. If we move a node, it will stay fixed at the dragged position until the node is clicked. Moving bends is not possible, because the force layout does not allow bends. While the simulation is running, the nodes' positions will be synced back to the OGDF approximately four times per second. If we want to deactivate the force-directed layout, we can use the method `stop_force_directed`.

  Internally, all of this works with d3.js's `forceSimulation`. When activated, all links, which are displayed using paths, will get replaced with links displayed as lines. This is necessary because we use paths to display bends, but there are no bends in the force-simulation and the force-simulation only functions with lines. Every tick of the simulation, the positions of the nodes and links are updated. The data is synced to the backend every five ticks and also when the simulation ends. To fix the positions of a node, the `fx` and the `fy` attributes are set.

- **Adding and deleting nodes/links:** Adding and deleting nodes/links was implemented with the help of the OGDF's `GraphObserver`. By implementing the `GraphObserver` the user will get notified if a node/link is deleted or added. This class is implemented in the backend on the Python side and sends custom messages to the frontend containing the necessary data and message-codes. The frontend then removes the node/link from the SVG and updates the data.

- **Moving nodes:** After activating node movement with the `enable_node_movement`-method, one can move the nodes with the help of d3.js's drag system. By adding a drag handler to every single node, the nodes implement the three necessary drag events (`dragStart, drag, dragEnd`). With the help of these events, everything including the label and all appending links will get moved. When the drag ends, the backend will be notified that a node was moved with the help of a callback. The backend then also changes the position on the OGDF side by updating the position of the node in the `GraphAttributes`.

- **Moving links:** To move a link, one needs to activate movement for a link with the given method, `move_link`. When moving a link it is only possible to move the bends since there is no need to move a whole link around because its end-point and start-point will always be dependent on the node they are attached to. After activating the bend movement, a yellow colored circle (`bendMover`) will appear at every bend of the link. Now, we are able to move this bend by moving around the `bendMover`. When we are done with moving the bend, a callback will again be triggered in the backend to let the user know that a bend has been moved. The bends' position is also changed on the OGDF side. If we want to get rid of the `bendMovers` we can turn off the bend movement for a specific link (`remove_bend_mover_for_id(id)`) or for every link with the method `remove_all_bend_movers()`.

- **Click callbacks:** To make the widget more interactive, callbacks for various different click events are provided:
  - **Node click:** D3.js gives us the opportunity to listen for clicks with on-click methods. If we click on a node, this method is called and a message will be sent to the backend, which retrieves the OGDF-object with the received `nodeId` and hands the OGDF-object to the user via a callback.
  - **Link click:** Upon clicking on a link, a callback is triggered in the backend using the same mechanism as described in the node click above. The main difference is that the click is not listened for at the drawn link but at a transparent link lying above the visible link. This system is implemented, because thin links will otherwise be really hard to click. The advantage of using a transparent link above the visible link is that we are able to make it as thick as we want.
  - **SVG click:** Clicking on the SVG itself can also be listened for. Since the callback fires every time the SVG is clicked, this callback contains the parameter `backgroundClicked`, which is true only if we have not clicked on a node or link. This callback also does not need to give back any OGDF-objects. It therefore gives back the x and y coordinate of our clicking position.
  - **Bend-mover click:** Clicking a bend mover will also cause a click event to be called. This callback provides us with the link, that the bend mover is on, the index of the bend and whether the ALT-key was pressed. Unfortunately, it is not possible to check if the CTRL-key was pressed, since the bend mover has drag capabilities and these will not get called as long as we press the CTRL-key. Therefore, it is impossible to detect the CTRL-key.
- **Update nodes/links:** Changing anything in the `GraphAttributes` will not be detected by the widget and therefore has to be updated manually. This is easily done with the methods `update_node(node)` and `update_link(link)` provided by the widget to push changes to individual objects in the frontend without redrawing the whole graph. These methods will export every important `GraphAttribute` of the given node/link to the frontend with a message, and will redraw the node/link there.

  By default, these updates will be animated rather than instantly applied. The duration of the animation can be adjusted by changing the variable `animation_duration` in the backend, which defaults to 1000 (one second). Those animations are done with d3.js's transitions. A special feature of these animations is when a link is animated to a link that does not have the same amount of bends. The problem when animating a path with more bends to a path with less, is that the animation cuts the extra bends and just animates the remaining path. When we want to do an animation the other way around, the extra bends will already be rendered at the new location and therefore will not get animated. To solve this problem, extra bends are added in the center between existing bends to the link with fewer bends until both have the same amount. When there is an animation from a lower bend count to a higher bend count, firstly, the starting link is replaced with the link that got artificial bends added to itself, then the animation is carried out as normal. If the animation is from a higher bend count to a lower bend count, the target link will internally be replaced with a link which contains the artificially added bends. Then an animation is done from the starting link to the new target link. When the animation has finished, the target link will be replaced with the original target link.

- **Animate graph layout changes** If the user wants to animate differences between two layouts, the method `update_graph_attributes(graph_attributes)` can be used. It is required, that the currently displayed `GraphAttributes` are referencing the same `Graph`-instance as the ones passed into the method. If this requirement is met, the method will internally change the represented graph layout to the one given by the new `GraphAttributes` and will animate the changes.

5. Demos and Evaluation

This chapter focuses on the implemented demo applications that were described in Chapter 2. These will be used to verify the usability of the widget and whether it is a generic toolbox that makes it easy to implement those use cases.

5.1 Demos

Initialization

To initiate the widget, one has to provide an instance of `GraphAttributes` (GA) for the widget. Once initialized, the widget can easily be displayed by simply letting Jupyter render the widget object (see Figure 5.1).

```
w = Widget(GA)
w
```

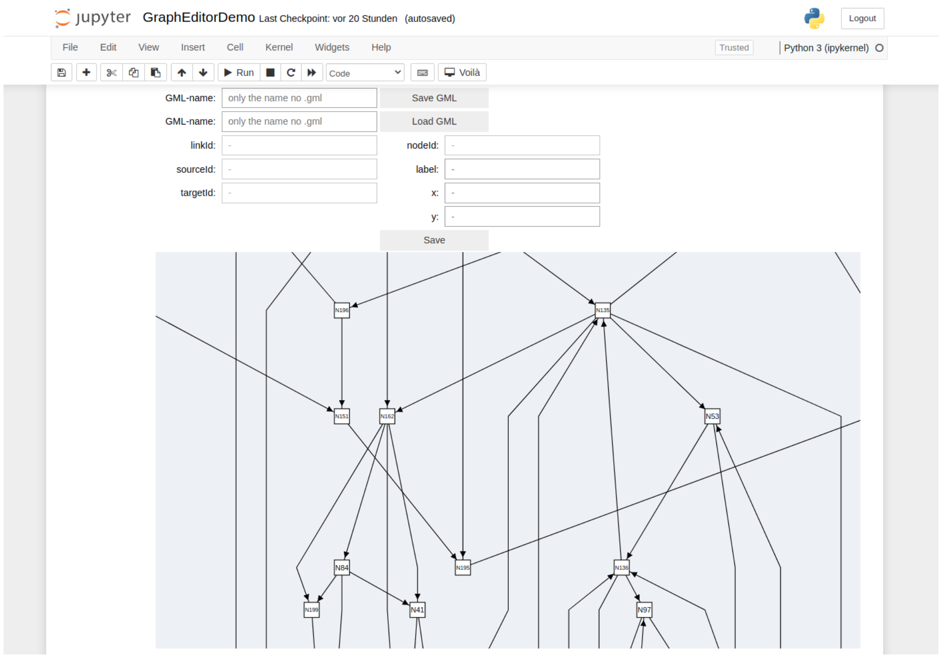


Figure 5.1: The whole Widget.

5.1.1 Graph Editor

```python
def clickSvg(x, y, alt, ctrl, backgroundClicked):
    global w
    if backgroundClicked and ctrl:
        coords = w.svgcoords_to_graphcoords(x, y)
        n = w.graph_attributes.constGraph().newNode()
        w.graph_attributes.x[n] = coords['x']
        w.graph_attributes.y[n] = coords['y']
        w.graph_attributes.label[n] = "N%s" % n.index()
        w.update_node(n, False)

w.on_svg_click_callback = clickSvg
```

Listing 5.1: Adding nodes to a graph.

Adding nodes to a graph (see Listing 5.1)

To add a node by clicking onto the SVG, we use the `on_svg_click_callback`-callback, which we assign to our own method `clickSvg`. Then we check if the background was clicked because we do not want to add a node in case we click on an existing node or link. Then we use the helper-method `svgcoords_to_graphcoords` to convert the SVG coordinates that are provided by the callback to the OGDF's coordinate-space. After that, we create the node inside the OGDF and set its position and label. To notify the widget that a new node has been created, we call `update_node()` with the node passed inside and set the parameter `animation` to `False` to instantly show the node.

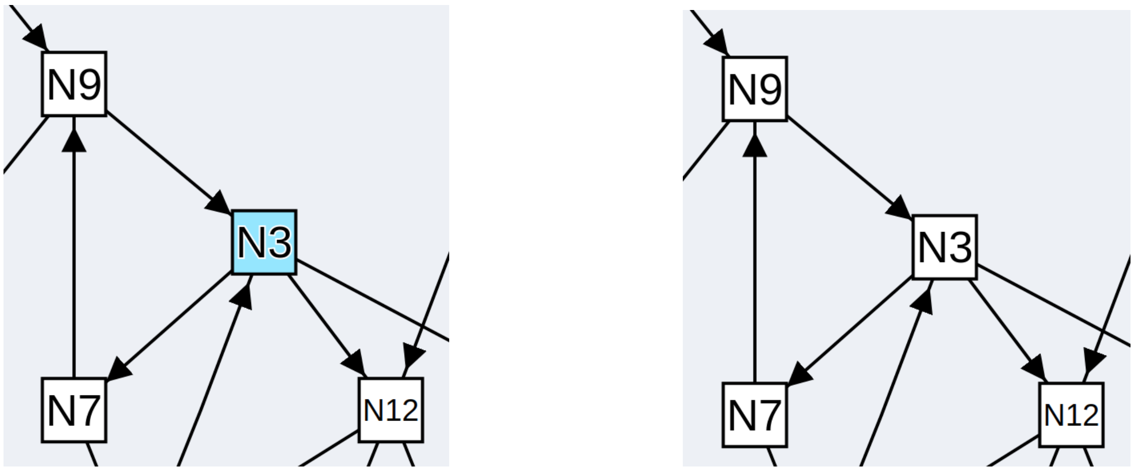


Figure 5.2: Node selected (left) and unselected (right).

Selecting a node (see Listing 5.2)

To make the user able to click on a node and let it seem like the user has selected that node, we first need to initialize a global variable named `selectedNode`, which stores the selected node reference. Then we define variables which store color values and define a method which will change the color of a given node visibly in the SVG. The extra method and variables are obviously not necessary, but they make it easier to read and change the code. To recognize if the user clicked on a node, we set the `on_node_click_callback`-callback to our custom `clickNode`-method. When a node has been clicked, we make sure that both

```
selectedNode = None
defaultNodeColor = ogdf.Color(255, 255, 255)
selectedColor = ogdf.Color(150, 231, 255)

def changeNodeColor(node, color):
    w.graph_attributes.fillColor[node] = color
    w.update_node(node, False)

def clickNode(node, alt, ctrl):
    global w, selectedNode
    if not alt and not ctrl:
        if selectedNode is not None:
            changeNodeColor(selectedNode, defaultNodeColor)
        changeNodeColor(node, selectedColor)
        selectedNode = node

def clickSvg(x, y, alt, ctrl, backgroundClicked):
    global w, selectedNode
    if backgroundClicked and not ctrl and not alt:
        if selectedNode is not None:
            changeNodeColor(selectedNode, defaultNodeColor)
            selectedNode = None

w.on_node_click_callback = clickNode
w.on_svg_click_callback = clickSvg
```

Listing 5.2: Selecting a node.

the CTRL- and ALT-key were not pressed. After that, we need to check if there is already a node selected. In that case, we change the already selected nodes' color back to the default color. To visually select the clicked node, we change the color of the node passed by the callback to the selected color (see Figure 5.2 left) and set `selectedNode` to the clicked node. Now we can select a node and change the selection, but have no way of deselecting a node. Therefore, we use the `on_svg_click_callback`-callback, which we assign to our own method `clickSvg`. Here we check whether both keys were not pressed and whether the background was clicked. Now we just need to check if there is actually a node selected. If this is the case, we change the color of the currently selected node to the default color (see Figure 5.2 right) and change the `selectedNode` to `None`.

```
def clickNode(node, alt, ctrl):
    global w, selectedNode
    if ctrl and selectedNode is not None:
        w.graph_attributes.constGraph().newEdge(selectedNode, node)
        changeNodeColor(selectedNode, defaultNodeColor)
        selectedNode = None

w.on_node_click_callback = clickNode
```

Listing 5.3: Adding a link.

Adding a link (see Listing 5.3)

To add a link we require the previous code which lets us select a node. The way we add a link is to select a node and if we press the CTRL-key and click on another node, a link forms between the two nodes. To implement this, we first set the node callback to our custom `clickNode`-method. Within the method, we check if the CTRL-key is pressed and if there currently is a node selected. If both of these things are satisfied, we can create a link between the selected node and the clicked node. Adding a link is automatically detected by the widget and therefore does not require an update call. After creating the link, we set the `selectedNodes`' color back to default and with that unselect it.

```python
def clickLink(link, alt, ctrl):
    global w
    if alt and not ctrl:
        w.graph_attributes.constGraph().delEdge(link)

def clickNode(node, alt, ctrl):
    global w
    if alt and not ctrl:
        w.graph_attributes.constGraph().delNode(node)

w.on_link_click_callback = clickLink
w.on_node_click_callback = clickNode
```

Listing 5.4: Deleting a node/link.

Deleting a node/link (see Listing 5.4)

Deleting a node or a link when clicking on it requires us to set the `on_link_click_callback` and the `on_node_click_callback` to our own methods named `clickLink` and `clickNode`. Both methods check if the ALT-key was pressed and then delete the clicked node or link inside the OGDF. Since deletions can be detected by the widget automatically, there is no need to update the node or link.

Moving nodes (see Listing 5.5)

In this example, we showcase how one can create a graphical interface with the help of ipywidgets and the widget. Activating movement for the nodes can simply be done by calling `enable_node_movement(True)`. First, we need to import ipywidgets and create a checkbox with it. Then we create a custom callback method named `on_nodeMoveToggle` and assign it to the checkbox object named `nodeMoveToggle`. To display both the widget and the checkbox at the same Jupyter-cell, we put them into a `VBox` from ipywidgets which renders them vertically aligned. Then we just display the `VBox` instead of just displaying the widget.

```
import ipywidgets as widgets
from ipywidgets import VBox

nodeMoveToggle = widgets.Checkbox(
    value=False,
    description='move Nodes',
    disabled=False,
    indent=False
)

def on_nodeMoveToggle(change):
    global w, nodeMoveToggle
    w.enable_node_movement(nodeMoveToggle.value)

nodeMoveToggle.observe(on_nodeMoveToggle, 'value')

vbox = VBox([nodeMoveToggle, w])
vbox
```

Listing 5.5: Activating and deactivating node movement with ipywidgets.

```
movingLinkIds = []

def clickLink(link, alt, ctrl):
    global w, movingLinkIds,
    if ctrl and not alt:
        if link.index() in movingLinkIds:
            w.remove_bend_mover_for_id(link.index())
            movingLinkIds.remove(link.index())
        else:
            w.move_link(link)
            movingLinkIds.append(link.index())

w.on_link_click_callback = clickLink
```

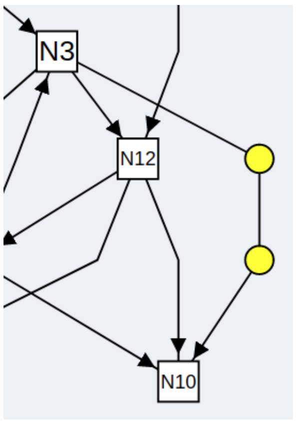


Figure 5.3: The code for enabling bend moving (left) and a screenshot of the bend movers it enables (right).

Moving bends (see Figure 5.3 left)

To move around bends, we simply call `move_link(link)`. In the example below, it is implemented so that we can toggle the bend movement on and off by clicking on a link while holding the CTRL-key. First, we defined a global list which stores the IDs of bends which can currently be moved. Then we set the `on_link_click_callback`-callback to our own `clickLink`-method. In our own method, we check for the CTRL-key. If the CTRL-key is pressed, we check whether the ID of the link is already in our `movingLinkIds`-list. If it is in there, we remove the bend movers (see Figure 5.3 right) by calling `remove_bend_mover_for_id(link.index())` from the widget and remove it from the list. If the ID is not in the list, we add it to the list and call the `move_link(link)`-method.

```python
import ipywidgets as widgets
from ipywidgets import VBox

def create_text_widget(name, disabled = False):
    return widgets.Text(
    value='',
    placeholder='-',
    description=name,
    disabled=disabled
)

node_label_text = create_text_widget('label:')

def update_node_texts():
    global selectedNode
    if selectedNode is not None:
        node_label_text.value = str(w.graph_attributes.label[selectedNode])
    else:
        node_label_text.value = ''

vbox = VBox([node_label_text, w])
vbox
```

Listing 5.6: Visualizing internal data.

5.1.2 Visualizing Algorithms and Graphs

Visualizing internal data (see Listing 5.6)

To visualize internal data, we once again use ipywidgets. Firstly, we use a helper method named `create_text_widgets` to create a text-field (see Figure 5.1), which we name `node_label_text`. Then we define a method called `update_node_text` which should be called every time the selected node changes. This method checks if the selected node is `None`, if so, it sets the value of the text-fields to an empty string. If it is not `None`, it will set the value that is currently inside the OGDF. To display it, we put the text-field and the widget into a `VBox` and let that be displayed.

```
def push(link, reversePush = False):
    ...

def relabel(u):
    ...

w.on_link_click_callback = lambda link, alt, ctrl: push(link, ctrl)
w.on_node_click_callback = lambda node, alt, ctrl: relabel(node)
```

Listing 5.7: Making an algorithm interactive.

Making an algorithm interactive (see Listing 5.7)

In this example, we make the push-relabel-algorithm interactive. To achieve this, we simply set the callback for link clicks to call the `push`-method. We also hand the `ctrl`-attribute, so it is possible to make a reverse push if the CTRL-key is being held. Then we set the node callback to call the `relabel`-method.

```
w.start_force_directed(fix_start_position=False)
w.stop_force_directed()
```

Listing 5.8: Applying and stopping the force-directed layout.

Using force layout (see Listing 5.8)

In this example, we want to show how easy it is to activate the force-directed layout (see Figure 5.4). First, we activate the layout by calling the method `start_force_directed(...)` and let all the possible parameters default except the `fix_start_position`, which we set to `False`. This will enable the force-directed layout to move around freely. To stop the force-directed layout, we simply need to call the method `stop_force_directed()`.

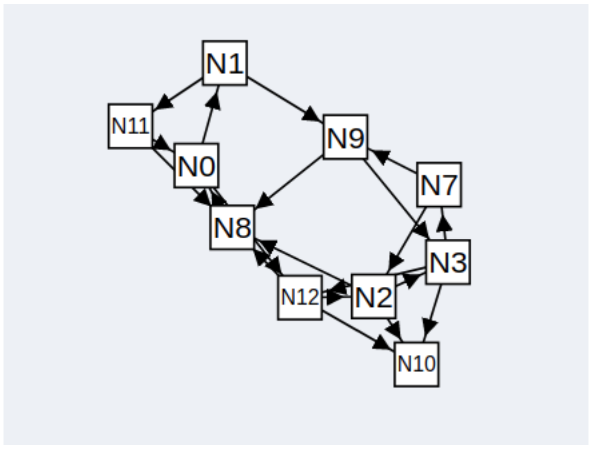


Figure 5.4: Force-directed layout.

5.2 Performance

To test the performance of this widget, we used Firefoxes' development tools to create a runtime analysis and take a look at the frames per second (fps) that this widget produces while moving around.

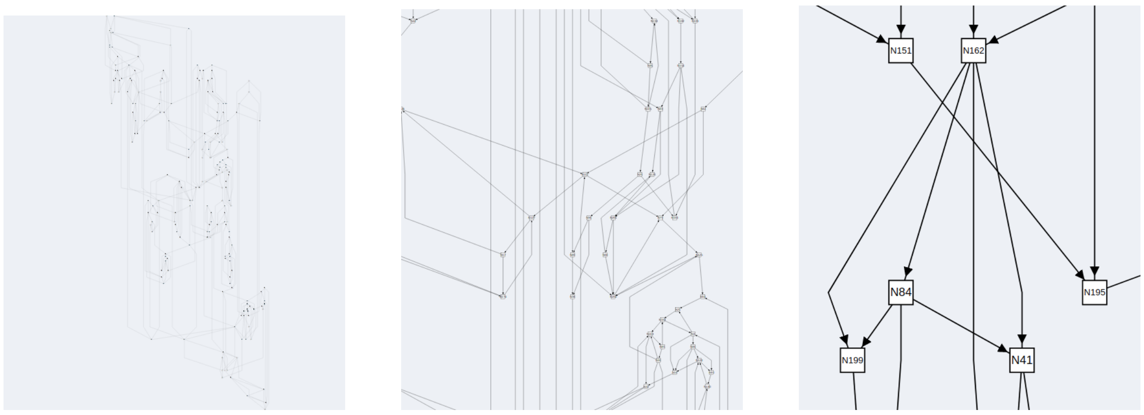


Figure 5.5: Screenshot of zoom level 1 (left), zoom level 2 (middle) and zoom level 3 (right).

The performance test was done using the Firefox Browser (version 96.0) on the operating system Ubuntu (version 20.04.3). The used computer has an Intel® Core™ i7-8550U CPU @ 1.80GHz - Processor and 16 GB of RAM. To test the widgets' performance, we used three different sized graphs. In the following table, they are specified by their nodes amount (N) and their links amount (M). To further run a meaningful evaluation, we also differ between three different zoom levels (see Figure 5.5). At the first zoom level, we are able to see the whole graph. At the second zoom level, we are able to distinguish between nodes and see their links. At the last zoom level, we are able to read the node's label. All of these graphs were drawn using the OGDF's Sugiyama-Layout.

Graph size	zoom level 1 (fps)	zoom level 2 (fps)	zoom level 3 (fps)	animation (fps)	init time
N=200; M=400	60	60	60	40	<500ms
N=2000; M=4000	12	30	45	-	3s
N=10000; M=20000	1	20	30	-	9s

Table 5.1: Performance results

We can observe (see Table 5.1) that the size of the graph has an obvious effect on the performance. But we can also see that the zoom level plays a big role in the performance. We can therefore conclude that the widget is still very usable with graphs that have around 10000 nodes, provided that the widget is used at a reasonable zoom level. The time for the widget's initialization is also acceptable, considering the size of the graph. Animating the changes between different graph attributes is only reasonable up to 200 nodes and 400 links. With more nodes and links, we can observe that the animation of the links and nodes are not carried out at the same time but one after the other. This leads to long waiting times and the animations do not serve a purpose if they are not carried out at the same time.

6. Conclusion

In this thesis, we identified problems regarding the current workflow with the OGDF and provided a possible solution for them. Firstly, we stated that an approach close to the approach of data scientists based upon Jupyter Notebooks would be ideal. Therefore, we needed to design a widget for Jupyter Notebooks. We started by defining use cases which described the desired functionality to solve the stated problems. From those, we derived requirements, which then were split up into more detailed features. After the list of features was completed, we took a look at the technologies that would be needed to implement a widget capable of satisfying these features. We described every technology used in detail and explained how they are supposed to work together. After that, we started to take a look at every feature and explained it with insights to their respective implementation. Lastly, we took a look at the demo applications that were built to test and verify the widgets' functionalities. There, we took code snippets from the demos to show how easy it is to implement certain functionalities with the widget. To complete the verification of the widget, we also tested its performance. It turned out that the widget works with graphs up to 10000 nodes even without further optimization.

We showed that this widget can massively improve rapid-prototyping when working with the OGDF. In the future, it may be interesting to cover more of the OGDF, e.g. cluster graphs. Since this widget is designed to be as generic as possible, it will be interesting to see which further use cases will be enabled. There are also possibilities to convert notebooks which are designed for Jupyter Notebooks to a standalone version, which could result in standalone graph editors for the OGDF and many more standalone applications. Furthermore, it will be interesting to see how this widget will affect rapid-prototyping with the OGDF and how it will impact the general way the OGDF is used.

Bibliography

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[CGJ+12] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. *The Open Graph Drawing Framework (OGDF).* CRC Press, 2012. to appear.

[Gri08] Volker Grimm. Visual debugging: A way of analyzing, understanding and communicating bottom-up simulation models in ecology. *Nat. Resour. Model.*, 15(1):23–38, June 2008.

[KM04] Amy J Ko and Brad A Myers. Designing the whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, April 2004. ACM.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter notebooks ? a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[LD16] Wim T. L. P. Lavrijsen and Aditi Dutta. High-performance python-c++ bindings with pypy and cling. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16, pages 27–35, Piscataway, NJ, USA, 2016. IEEE Press.

[PG07] Fernando Perez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science Engineering*, 9(3):21–29, 2007.

[WMBO19] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. How data scientists use computational notebooks for real-time collaboration. *Proc. ACM Hum. Comput. Interact.*, 3(CSCW):1–30, November 2019.

[ZMW20] Amy X Zhang, Michael Muller, and Dakuo Wang. How do data science workers collaborate? roles, workflows, and tools. *Proc. ACM Hum. Comput. Interact.*, 4(CSCW1):1–23, May 2020.