[1]:

Full Access to the Functionality of the OGDF with Python's Ease of Use

The Open Graph Drawing Framework (OGDF) is a C++ library containing a vast amount of algorithms and data structures for automatic graph drawing. However, while powerful the library is not easily accessible for new users and the nature of C++ makes implementing even simple algorithms cumbersome to non-experts. The `odgf-python` project remedies these problems by making the full OGDF available from Python. This greatly reduces the overhead and complications when using the OGDF for the first time and unlocks a large ecosystem of tools, for example the interactive computing Notebooks provided by Project Jupyter. This is possible because the `ogdf-python` library aims to be an extensible building block, not a closed system.

Features

- *No C++ skills needed*: The full OGDF API is available from Python.
- *Rapid Prototyping*: Python needs less boilerplate and allows more idiomatic constructions, without the need to configure and compile anything.
- *Iterative execution*: Jupyter Notebooks allow individual lines of code to be adapted and re-run, retaining all previous variable values.
- *Inline results*: Graphs are displayed right next to the code that generates them.
- *Interactive graph exploration*: The inline display allows interactive zooming and panning to easily explore the graph.

Use-cases

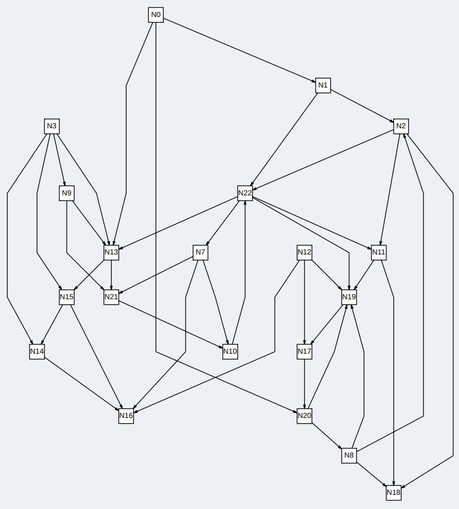
- iterative development of new graph algorithms
- step-by-step debugging of implemented algorithms
- visual editing of in-memory graphs and variables
- interactive visualization of algorithms for teaching
- flexible user interfaces for domain-specific application

[2]:

```python
from ogdf_python import *

cppinclude("ogdf/basic/graph_generators.h")
cppinclude("ogdf/layered/SugiyamaLayout.h")



G = ogdf.Graph()
GA = ogdf.GraphAttributes(G)
ogdf.randomPlanarTriconnectedGraph(G, 10, 20)

for n in G.nodes:
    GA.label[n] = "N%s" % n.index()

SL = ogdf.SugiyamaLayout()
SL.call(GA)
GA # display inline!
```

```cpp
using namespace ogdf;

#include <ogdf/basic/graph_generators.h>
#include <ogdf/layered/SugiyamaLayout.h>
#include <ogdf/fileformats/GraphIO.h>

int main() {
  Graph G;
  GraphAttributes GA(G);
  randomPlanarTriconnectedGraph(G, 10, 20);

  for (node n : G.nodes)
    GA.label[n] = "N" + string(n.index());

  SugiyamaLayout SL;
  SL.call(GA);
  GraphIO::write(GA, "output.svg");

  return 0;
}
```

[2]:



[3]:

Easy Installation - all Batteries included!

`ogdf-python` comes with a prebuilt version of the OGDF that works on Linux, macOS and in the Windows Subsystem for Linux. Just install the packages with `pip` and you are ready to go:

```
pip3 install ogdf-python ogdf-wheel notebook
jupyter notebook # start an interactive notebook
```

[4]:

Focus on the Algorithm, not the Language

Developing algorithms in C++ can be cumbersome. New projects require non-trivial set-up, and distinct code-compile-execute iterations make it hard to incrementally develop a new algorithm. This is because they separate the code from its results and there is no easy way to debug and visually analyze the current state of the program. All these issues have been resolved within the Python ecosystem, and `ogdf-python` now makes these solutions available for graph drawing use-cases. Thanks to this, implementing known as well as new algorithms becomes much easier.

[5]:

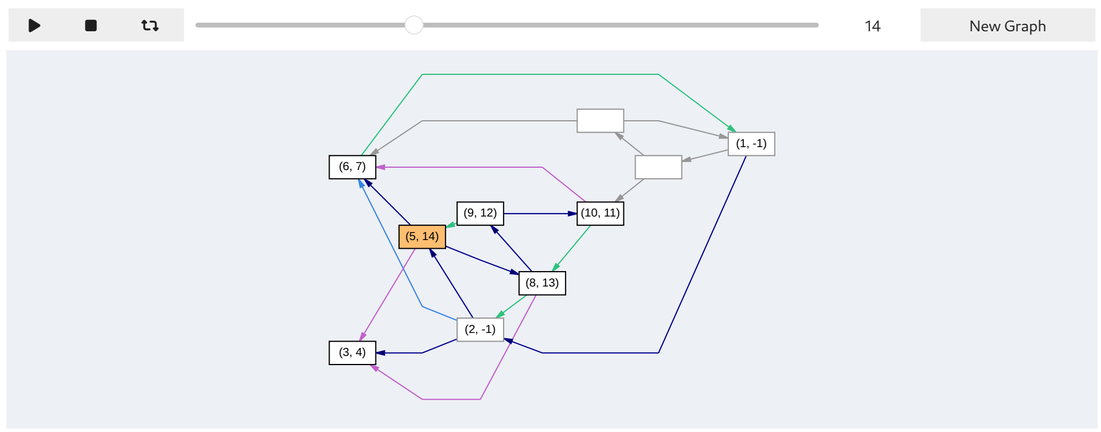
```python
discovery = ogdf.NodeArray[int](G, -1)
finish = ogdf.NodeArray[int](G, -1)
predecessor = ogdf.NodeArray[ogdf.node](G, nullptr)

time = 0

def dfs_visit(u):
    global time

    time += 1
    discovery[u] = time

    for adj in u.adjEntries:
        v = adj.twinNode()
        if adj.isSource() and discovery[v] < 0:
            predecessor[v] = u
            dfs_visit(v)

    time += 1
    finish[u] = time

for n in G.nodes:
    if discovery[n] < 0:
        dfs_visit(n)
```

[6]:

Build portable Graph User Interfaces

Combining `ogdf-python`'s graph display widget with the UI elements provided by Jupyter's `ipywidgets` allows you to easily build platform-independent user interfaces for interacting with graphs and algorithms. In our example below, this is used to build an animated visualization of the individual steps performed by the depth-first-search algorithm on a graph. This can for example be used for teaching an algorithms course.

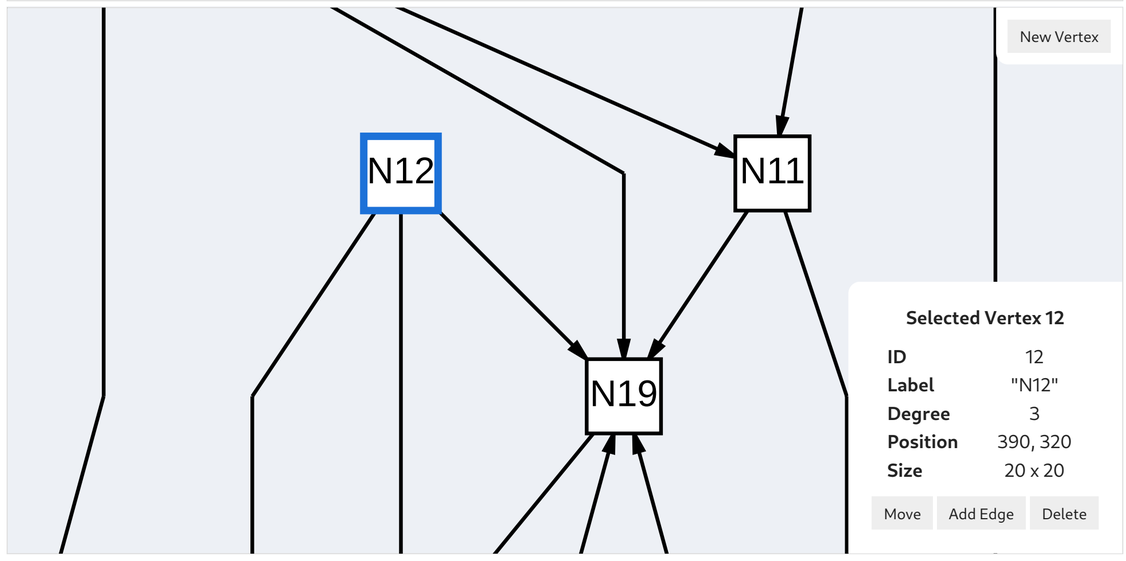
[6]:



[7]:

Editing Graphs Made Easy

The graph display widget can be used as-is to quickly modify the current graph in-memory. At the same time, it can be used as building block for more complex user interfaces, where its interactivity can be fully customized to suit different applications.



[8]:

Seamless Interaction between C++ and Python

Thanks to the underlying `cppyy` library, arbitrary C++ code can be loaded and called from Python. This allows objects, functions and even classes to be created and used in either language.

[9]:

```cpp
%%cpp

// Include C++ Code in your Python Notebook
std::cout << "Hello World from C++!" << std::endl;

ogdf::Graph G; // created in C++
int width = 3, height = 3;
ogdf::gridGraph(G, width, height, true, false);
```

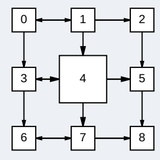
[9]:

```
Hello World from C++!
```

[10]:

```python
from cppyy import gbl as cpp  # namespace with all C++ variables

G = cpp.G  # used from Python
GA = ogdf.GraphAttributes(G, ogdf.GraphAttributes.all)

for n in G.nodes:
    GA.label[n] = str(n.index())
    GA.x[n] = (n.index() % cpp.width) * 50
    GA.y[n] = (n.index() // cpp.height) * 50

middle = G.numberOfNodes() // 2
GA.width[G.nodes[middle]] = 40
GA.height[G.nodes[middle]] = 40

GA
```

[10]:



[11]:

```cpp
%%writefile avg_width.h
// these are the contents of file avg_width.h

// function implemented in C++ for efficiency
int avg_width(const ogdf::GraphAttributes &GA) {
    int sum = 0;
    for (auto n : GA.constGraph().nodes) {
        sum += GA.width(n);
    }
    return sum / GA.constGraph().numberOfNodes();
}
```

[12]:

```python
cppinclude("avg_width.h") # load method from C++ file
print("The node widths are", list(GA.width()))
print("The average width is", cpp.avg_width(GA)) # call your own C++ functions from python
```

[12]:

```
The node widths are [20.0, 20.0, 20.0, 20.0, 40.0, 20.0, 20.0, 20.0, 20.0]
The average width is 22
```