NetworKit Development Guide

This text is meant to provide some guidelines for the ongoing development of the project. It is meant for core developers as well as occasional contributors.

The following text assumes some basic familiarity with the Mercurial version control software. It is not a Mercurial tutorial, because you will find a good one at hginit.com. Rather, it explains concepts and workflows for the development of this project.

If you want to contribute, you should to consider the technical report on NetworKit to get familiar with the architecture.

If you use NetworKit in your research publications, please cite the mentioned techincal report or the specific algorithm. A list of publications is available on the website.

How to contribute

Report bugs

For the time being, bugs should be reported by sending a report to the mailing list. Please provide a minimal example so that others can reproduce that bug.

Fork NetworKit

Feel free to fork NetworKit on algohub and start contributing by fixing bugs or taking care of the issues at kanboard.iti.kit.edu. New and missing features are welcomed aswell.

Repositories

The NetworKit main development repository is at http://algohub.iti.kit.edu/parco/NetworKit/NetworKit. Access to this repository is provided on request.

algohub.iti.kit.edu (an installation of RhodeCode) makes it easy to create and manage forks. Forking is distinct from branching and creates a new repository with a new address, its own access control etc. A fork contains all branches of its parent.

Project Tracker (Kanboard)

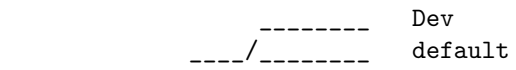
At kanboard.iti.kit.edu we maintain a project task tracker to coordinate development and releases. An account is given on request, please ask on the mailing list. Tasks are moved from left to right to through the columns:

- `Backlog`: improvement ideas, some day maybe, "nice to have"
- `ToDo`: scheduled improvements
- `Work in progress`
- `To Review`: requesting peer review
- `Ready for Release`

There is the possibility to create "swim lanes" for different releases.

Branches

Currently, the two most important branches of NetworKit are `Dev` and `default`.



As the name says, `default` is the branch which you are on if you do not switch. It is therefore the release branch, containing code which is ready for use. Unless you are a core developer preparing a release or fixing an urgent bug, you do not make changes to `default`.

`Dev` is the development branch and most of the development of new features happens in this branch. This is also where new releases are being prepared. When pushing into this branch, think about whether your code is ready for the core development team to work with and will be suitable for a release in the foreseeable future.

It can be appropriate to create additional branches for projects, features, developer teams etc. Creation of branches should be coordinated with the core development team. For this purpose, post to the mailing list.

Tags

A tag is nothing more than a "symbolic name" for a revision. In NetworKit tags are used to mark release versions in the `default` branch, with a `MAJOR.MINOR` version name scheme.

Workflows

This section describes how to work with branches and forks in different scenarios.

Using NetworKit

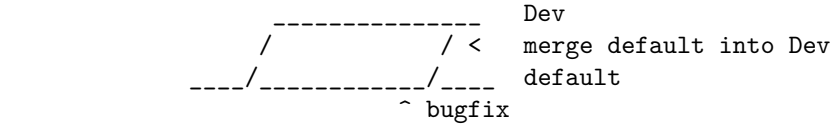
If you want to build and use NetworKit and do not plan to contribute changes, simply clone the repository. By default, you will be on the `default` branch, which represents the current release. Follow the setup instructions in the `Readme`.

Core Development

This section describes workflows for the core development team.

Bugfixes

Bugfixes are changes that should be immediately visible to users of NetworKit, such as solutions for urgent errors or improvements of the `Readme` document. In this case, make the changes in the `default` branch and commit. Then switch to the `Dev` branch and merge the `default` branch back into `Dev`.

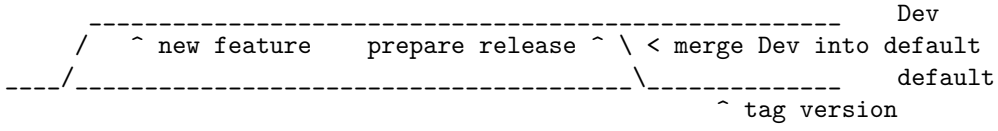


Example:

```
hg up default
...
hg com -m "fixed bug xyz"
hg up Dev
hg merge default
hg com -m "backmerge bugfix xyz"
```

Releasing New Features

When new features should be released, the `Dev` branch is merged into the `default` branch. Additional testing and cleanup is performed before that happens. The new major or minor release is then tagged with a version number.



Example:

```
hg up Dev
hg com -m "ready for release X.Y"
hg up default
hg merge Dev
hg com -m "release X.Y"
```

Multiple heads in multiple branches

If remote changes have happened in multiple branches and you pull them, these branch will have multiple heads. Merging now needs to happen for each of the affected branches before you can push. Switch to each branch and perform a merge as usual. As an alternative to merging, you may try the `rebase` extension.

Contributions

Users of NetworKit are welcome to contribute their modifications. New features must be added to the `Dev` branch, not the `default` branch. We recommend the following workflow:

1. create a fork of the main repository
2. switch to the `Dev` branch
3. make and commit your changes while being on the `Dev` branch
4. send a pull request to the main repository

Branching Cheat Sheet

- list all available branches: `hg branches`
- check on which branch you are: `hg branch`
- see heads (most recent commits) of all branches: `hg head`
- see tip (most recent commits) of the branch you are currently working on: `hg tip`
- switch to a specific branch: `hg update <branchname>`
- start a new branch: `hg branch <branchname>`
- merge `branchY` into `branchX`: `hg update branchX`, then `hg merge branchY`

Conventions

The following general conventions apply to all NetworKit developers.

Versioning

- Before you commit, make sure your code compiles and run the unit tests. Never push code which breaks the build for others.
- Commit regularly and often to your local repository.
- Use meaningful commit messages.
- Get the newest changes from the repository regularly and merge them into your local repository.
- Make sure that you merged correctly and did not break other people's work.
- Push correct code early if possible. Merging is easier if all developers are up to date.
- Never `push --force` to the main repository.

Unit Tests and Testing

Every new feature must be covered by a unit test. Omitting unit tests makes it very likely that your feature will break silently as the project develops, leading to unneccessary work in tracing back the source of the error.

Unit tests for the C++ part of NetworKit are based on the `googletest` library. For more information read the googletest primer. The Python test framework currently relies on `nose` to collect the tests.

- Each source folder contains a `test` folder with `googletest` classes. Create the unit tests for each feature in the appropriate `test/*GTest` class by adding a `TEST_F` function.
- Prefix standard unit tests with `test` and experimental feature tests with `try`. A `test*` must pass when pushed to the main repository, a `try*` is allowed to fail.
- Keep the running time of test functions to the minimum needed for testing functionality. Testing should be fast, long-running unit tests look like infinite loops.
- If the unit test requires a data set, add the file to the `input/` folder. Only small data sets (a few kilobytes maximum) are acceptable in the repository.
- Any output files produced by unit tests must be written to the `output/` folder.

To build and run the tests you need the gtest library. Assuming, gtest is successfully installed and you add the paths to your build.conf, the unit tests should be compiled with:

```
scons --optimize=Dbg --target=Tests
```

To verify that the code was built correctly: Run all unit tests with

```
./NetworKit-Tests-Dbg --tests/-t
```

Performance tests will be selected with

```
./NetworKit-Tests-Dbg --benchmarks/-b
```

while experimental tests are called with

```
./NetworKit-Tests-Dbg --trials/-e
```

To run only specific unit tests, you can also add a filter expression, e. g.:

```
./NetworKit-Tests-Dbg --gtest_filter=*PartitionGTest*/-f*PartitionGTest*
```

initiates unit tests only for the Partition data structure.

For the **Python** unit tests, run:

```
python3 setup.py test [--cpp-tests/-c]
```

This command will compile the _NetworKit extension and then run all test cases on the Python layer. If you append `--cpp-tests/-c`, the unit tests of the c++ side will be compiled and run before the Python test cases.

Test-driven development

If you implement a new feature for NetworKit, we encourage you to adapt your development process to test driven development. This means that you start with a one or ideally several test-cases for your feature and then write the feature for the test case(s). If your feature is mostly implemented in C++, you should write your test cases there. If you expose your feature to Python, you should also write a test case for the extension module on the Python layer. The same applies for features in Pyton.

Code Style

- Compiler warnings are likely to turn into future errors. Try to fix them as soon as they appear.
- Read some code to get used to the code style and try to adopt it.
- Document classes, methods and attributes in Doxygen style.
- Use the `count` and `index` integer types for non-negative integer quantities and indices.
- In most cases, objects are passed by reference. New objects are stack-allocated and returned by value. Avoid pointers and `new` where possible.
- Use the `override` keyword to indicate that a method overrides a virtual method in the superclass.
- In Python, indent using tabs, not spaces.

Algorithm interface and class hierarchy

We use the possibilities provided through inheritance to generalize the common behaviour of algorithm implementations:

- Data and paramters should be passed in the constructor.
- A void run()-method that takes no parameter triggers the execution.
- To retrieve the result(s), getter-functions() may be defined.

The `Algorithm` base class also defines a few other other functions to query whether the algorithm can be run in parallel or to retrieve a string representation.

There may be more levels in the class hierarchy between an algorithm implementation and the base class, e.g. a single-source shortest-path class `SSSP` that generalizes the behaviour of BFS and Dijkstra implementations or the `Centrality` base class. When implementing new features or algorithms, make sure to adapt to the existing class hierarchies. The least thing to do is to inherit from the `Algorithm` base class. Changes to existing interfaces or suggestions for new interfaces should be discussed through the mailing list.

Exposing C++ Code to Python

Assuming the unit tests for the new feature you implemented are correct and successful, you need to make your features available to Python in order to use it. NetworKit uses Cython to bridge C++ and Python. All of this bridge code is contained in the Cython code file `src/python/_Networkit.pyx`. The content is automatically translated into C++ and then compiled to a Python extension module.

Cython syntax is a superset of Python that knows about static type declarations and other things from the C/C++ world. The best way to getting used to it is working on examples. Take the most common case of exposing a C++ class as a Python class. Care for the following example that exposes the class `NetworKit::Dijkstra`:

```
cdef extern from "cpp/graph/Dijkstra.h":
    cdef cppclass _Dijkstra "NetworKit::Dijkstra"(_SSSP):
        _Dijkstra(_Graph G, node source, bool storePaths, bool storeStack, node target) except +
```

The code above exposes the C++ class definition to Cython - but not yet to Python. First of all, Cython needs to know which C++ declarations to use so the the first line directs Cython to place an `#include` statement. The second line defines a class that is only accessible in the Cython world. Our convention is that the name of the new class is the name of the referenced C++ class with a prepended underscore to avoid namespace conflicts. What follows is the "real" C++ name of the class. After that, the declarations of the methods you want to make available for Python are needed. The `except +` statement is necessary for exceptions thrown by the C++ code to be rethrown as Python exceptions rather than causing a crash. Also, take care that the Cython declarations match the declarations from the referenced header file.

```
cdef extern from "cpp/graph/SSSP.h":
    cdef cppclass _SSSP "NetworKit::SSSP"(_Algorithm):
        _SSSP(_Graph G, node source, bool storePaths, bool storeStack, node target) except +
        vector[edgeweight] getDistances(bool moveOut) except +
        [...]

cdef class SSSP(Algorithm):
    """ Base class for single source shortest path algorithms. """

    cdef Graph _G

    def __init__(self, *args, **namedargs):
        if type(self) == SSSP:
            raise RuntimeError("Error, you may not use SSSP directly, use a sub-class instead")

    def __dealloc__(self):
        self._G = None # just to be sure the graph is deleted

    def getDistances(self, moveOut=True):
        """
        Returns a vector of weighted distances from the source node, i.e. the
        length of the shortest path from the source node to any other node.

        Returns
        -------
        vector
            The weighted distances from the source node to any other node in the graph.
        """
        return (<_SSSP*>(self._this)).getDistances(moveOut)
    [...]
```

We mirror the class hierarchy of the C++ world also in Cython and Python. This also saves some boiler plate wrapping code as the functions shared by Dijkstra and BFS only need to be wrapped through SSSP.

```
cdef class Dijkstra(SSSP):
    """ Dijkstra's SSSP algorithm.
    Returns list of weighted distances from node source, i.e. the length of the shortest path from s
    any other node.

    Dijkstra(G, source, [storePaths], [storeStack], target)

    Creates Dijkstra for `G` and source node `source`.

    Parameters
    ----------
    G : Graph
        The graph.
    source : node
        The source node.
    storePaths : bool
        store paths and number of paths?
    storeStack : bool
        maintain a stack of nodes in order of decreasing distance?
    target : node
        target node. Search ends when target node is reached. t is set to None by default.
    """
    def __cinit__(self, Graph G, source, storePaths=True, storeStack=False, node target=none):
        self._G = G
        self._this = new _Dijkstra(G._this, source, storePaths, storeStack, target)
```

For the class to be accessible from the Python world, you need to define a Python wrapper class which delegates method calls to the native class. The Python class variable `_this` holds a pointer to an instance of the native class. Please note that the parameters are now Python objects. Method wrappers take these Python objects as parameters and pass the internal native objects to the actuall C++ method call. The constructor of such a wrapper class is called `__cinit__`, and it creates an instance of the native object.

The docstring between the triple quotation marks can be accessed through Python's `help(...)` function and are the main documentation of NetworKit. Always provide at least a short and precise docstring so the user can get in idea of the functionality of the class. For C++ types available to Python and further examples, see through the `_NetworKit.pyx`-file. The whole process has certainly some intricacies, e.g. some tricks are needed to avoid memory waste when passing around large objects such as graphs. When in doubt, look at examples of similar classes already exposed. Listen to the Cython compiler - coming from C++, its error messages are in general pleasantly human-readable.

Make algorithms interruptable with CTRL+C/SIGINT

When an algorithms takes too long to produce a result, it can be interrupted with a SIGINT signal triggered by CTRL+C. When triggering from the Python shell while the runtime is in the C++ domain, execution is aborted and even terminates the Python shell. Therefor, we implemented a signal handler infrastructure in C++ that raises a special exception instead of aborting. When implementing an algorithm, it is strongly encouraged to integrate the signal handler into the implementation. There are many examples of how to use it, e.g. `networkit/cpp/centrality/Betweenness.cpp` or `networkit/cpp/community/PartitionFragmentation.cpp`

Contact

To discuss important changes to NetworKit, use the mailing list (`networkit-dev@ira.uka.de`).

Further Reading

- hginit.com
- Working with named branches
- Managing releases and branchy development
- Cython Documentation