NetworKit: An Interactive Tool Suite for High-Performance Network Analysis

Christian L. Staudt∗ Aleksejs Sazonovs† Henning Meyerhenke‡

Abstract
We introduce NetworKit, an open-source software package for high-performance analysis of large complex networks. Complex networks are equally attractive and challenging targets for data mining, and novel algorithmic solutions, including parallelization, are required to handle data sets containing billions of connections. Our goal for NetworKit is to package results of our algorithm engineering efforts and put them into the hands of domain experts. NetworKit is a hybrid combining the performance of kernels written in C++ with a convenient Python frontend. The package targets shared-memory platforms with OpenMP support. The current feature set includes various analytics kernels such as connected components, diameter, clustering coefficients, community detection, k-core decomposition, degree assortativity and multiple centrality indices, as well as a collection of graph generators. Scaling to massive networks is enabled by techniques such as parallel and sampling-based approximation algorithms. In comparison with related software, we propose NetworKit as a package geared towards large networks and satisfying three important criteria: High performance, interactive workflows and integration into an ecosystem of tested tools for data analysis and scientific computation.

Keywords: complex networks, network analysis, network science, parallel graph algorithms, data analysis software

1 Motivation
Complex networks are heterogeneous relational datasets appearing in very different domains but sharing certain structural characteristics. Social (e.g. a friendship graph), technical (e.g. the internet or electric power grids), biological (e.g. protein interaction networks) or informational (e.g. the world wide web) networks are only a few examples for the large variety of phenomena modeled this way [14, 10]. Accordingly, network analysis methods are quickly becoming pervasive in science, technology and society: Node ranking by centrality is the basis for modern web search [36], community detection methods find applications on social media sites [21] or in cancer research [27], and tracking social influence through networks is equally interesting to sociologists [20] and advertisers. It seems plausible that network analysis is going to yield groundbreaking insights in the future as our theoretical understanding and our computing capabilities increase: Consider for example that the human brain at the neuronal scale is a complex network on the order of $10^{10}$ nodes and $10^{14}$ edges, whose mapping and analysis is still beyond current technology. Already on the more modest scale of $10^6$ to $10^{10}$ edges, complex networks challenge current algorithms and available implementations. It is evident that the need to process such networks within a data analysis workflow requires parallel processing whenever possible. With NetworKit, we intend to push the boundaries of what can be done interactively on a shared-memory parallel computer, also by users without in-depth programming skills. The tools we provide make it easy to characterize large networks and are geared towards and informed by network science research.

In this work we give an introduction to the toolkit and describe it in terms of algorithm and software engineering aspects. We also discuss important algorithmic features (including considerations for efficient implementations) such as network analytics kernels and generators, give example use cases and evaluate the performance of its kernels experimentally. Our experiments show that NetworKit is capable of quickly processing large-scale networks for a variety of analytics kernels. It does so with a lower memory footprint and in a more reliable manner compared to closely related tools. We recommend NetworKit for the comprehensive structural analysis of large complex networks in the range of $10^5$ to $10^{10}$ edges.

2 Methods

2.1 Design Goals. There is a variety of software packages which provide graph algorithms in general and network analysis capabilities in particular (see Section 5 for a comparison to related packages). However, NetworKit aims to balance a specific combination of

∗ Karlsruhe Institute of Technology (KIT), Germany, christian.staudt@kit.edu
† University of St Andrews, United Kingdom, as245@st-andrews.ac.uk
‡ Karlsruhe Institute of Technology (KIT), Germany, meyerhenke@kit.edu
strengths. Our software is designed to stand out with respect to three areas:

Performance. Algorithms and data structures are selected and implemented with high performance and parallelism in mind. Some implementations are among the fastest in published research. For example, community detection in a 3.3 billion edge web graph can be performed on a 16-core server with hyperthreading in less than five minutes [43].

Interface. Networks are as diverse as the series of questions we might ask of them – e.g., what is the largest connected component, what are the most central nodes in it and how do they connect to each other? A practical tool for network analysis should therefore provide modular functions which do not restrict the user to predefined workflows. In this respect we take inspiration from software like R, MATLAB and Mathematica, as well as a variety of Python packages. An interactive shell, which the Python language provides, meets these requirements. While NetworKit works with the standard Python 3 interpreter, combining it with the IPython Notebook [37] allows us to integrate it into a fully fledged computing environment for scientific workflows. It is also easy to set up and control a remote compute server.

Integration. As a Python module, NetworKit enables seamless integration with Python libraries for scientific computing and data analysis, e.g. pandas for data frame processing and analytics, matplotlib for plotting or numpy and scipy for numerical and scientific computing. For certain tasks, we provide interfaces to specialized external tools, e.g. Gephi for graph visualization.

2.2 Architecture. In order to achieve the design goals described above, we implement NetworKit as a two-layer hybrid of high-performance code written in C++ with an interface and additional functionality written in Python. NetworKit is distributed as a Python package, ready to be used interactively from a Python shell, which is the main usage scenario we envision. The code can be used as a library for application programming as well, either at the Python or C++ level. Throughout the project we use object-oriented and functional concepts. Shared-memory parallelism is realized with OpenMP, providing loop parallelization and synchronization constructs while abstracting away the details of thread creation and handling. The roughly 35 000 lines of C++ code include core implementations and unit tests. As illustrated in Figure 1, connecting these native implementations to the Python world is enabled by the Cython toolchain [7]. Currently we use Cython to integrate native code by compiling it into a Python extension module. The Python layer comprises about 4000 lines of code.

![Figure 1: NetworKit architecture overview](image)

2.3 General Framework Features. As the central data structure, the NetworKit.Graph class implements a directed or undirected, optionally weighted graph using an adjacency array data structure with \(O(n+m)\) memory requirement for a graph with \(n\) nodes and \(m\) edges. A node is represented by a 64 bit integer index, and an edge is identified by a pair of nodes. Optionally, edges can be indexed as well. This approach enables a lean graph data structure, while also allowing arbitrary node and edge attributes to be stored in any container addressable by indices. Our API aims for an intuitive and concise formulation of graph algorithms on both the C++ and Python layer. In general, a combinatorial view on graphs – representing edges as tuples of nodes – is used. However, a recent addition to NetworKit is an algebraic interface that enables the implementation of graph algorithms in terms of the adjacency matrix, while transparently using the same graph data structure. While some algorithms may benefit from different data layouts, this lean, general-purpose representation has proven competitive for writing performant implementations.

Large complex networks are difficult to inspect visually. Our package has basic visualization/graph drawing capabilities, but delegates graph visualization to Gephi, a network analysis application specialized on visualization features, via a convenient interface.

3 Analytics

The following describes the core set of network analysis algorithms implemented in NetworKit. In addition, NetworKit also includes a collection of basic graph...
algorithms, such as breadth-first and depth-first search, Dijkstra’s algorithm for shortest paths or code for computing approximate maximum weight matchings.

3.1 Degree Distribution. The distribution of degrees, i.e., number of connections per node, plays an important role in characterizing a network: Empirically observed complex networks tend to show a heavy-tailed degree distribution which follow a power-law with a characteristic exponent: \( p(k) \sim k^{-\gamma} \). The powerlaw Python module, developed by Alstott et al. [3], employs statistical methods from \([13, 28]\) to determine if a probability distribution fits a power law. The module is distributed with NetworKit.

3.2 Degree Assortativity. Generally, a network shows assortative mixing with respect to a certain property \( P \) if nodes with similar values for \( P \) tend to be connected to each other. Degree assortativity measures this correlation with respect to node degree, which can point to important aspects such as a hierarchical network composition. Its strength is often expressed as degree assortativity coefficient \( r \), which lies in the range \(-1 \leq r \leq 1\). We implement Newman’s formulation \([33] \) with \( O(m) \) time and constant memory requirements.

3.3 Diameter. The diameter of a graph is the maximum length of a shortest path between any two nodes. A surprising observation about the diameter of complex networks is often referred to as the small world phenomenon \([32] \): The diameter tends to be very small and is often constant or even shrinks with network growth. We use the iFUB algorithm \([15] \) both for the exact computation as well as an estimation of a lower and upper bound on the diameter. iFub has a worst case complexity of \( O(nm) \) but has shown excellent scalability on complex networks, where it often converges on the exact value in short time.

3.4 Clustering Coefficients. Clustering coefficients are key figures for the amount of transitivity in networks, i.e. the tendency of edges to form between indirect neighbor nodes \([32] \). The global clustering coefficient is the ratio of the number of triangles in a network versus the number of triads (paths of length 2). This ratio is typically high in social networks, whose generative processes tend to close triangles. In contrast, the clustering coefficient is close to 0 for random graphs. The average local clustering coefficient computes the fraction above locally for each node and averages then over all nodes. A straightforward calculation of the clustering coefficient with a node iterator requires \( O(nd_{\text{max}}^2) \) time, where \( d_{\text{max}} \) is the maximum degree. As this might be prohibitive for large graphs, NetworKit also implements a sampling approximation algorithm \([41] \), whose constant time complexity is independent of graph size.

3.5 Components and Cores. Components and cores are related concepts for subdivideing a network: All nodes in a connected component are reachable from each other. A typical pattern in real-world complex networks is the emergence of a giant connected component, accompanied by a large number of very small components. We compute connected components in linear time using breadth-first search. Core decomposition allows a more fine-grained subdivision of the node set according to connectedness. \( k \)-cores result from successively peeling away nodes of degree \( k \). It also categorizes nodes according to the highest-order core in which they are contained, assigning a core number to each node. The kernel runs in \( O(m) \) time, matching other implementations \([6] \).

3.6 Centrality. Centrality refers to the relative importance of a node within a network. We distribute efficient implementations for betweenness, eigenvector centrality and PageRank. Betweenness centrality expresses the concept that a node is important if it lies on many shortest paths between nodes in the network. We implement the well-known algorithm by Brandes \([11] \), which requires \( O(nm + n^2 \log n) \) time and \( O(n + m) \) space for weighted graphs. For unweighted graphs, running time is reduced to \( O(nm) \). Since this is still practically infeasible for the large data sets we target, NetworKit includes also two parallelized implementations of recent approximation algorithms: One \([40] \) gives a probabilistic guarantee that the error is at most an additive constant, while the other \([23] \) uses an approach without such guarantees. Our current research extends the former approach to dynamic graph processing \([8] \). Eigenvector centrality and its variant PageRank \([36] \) assign relative importance to nodes according to their connections, incorporating the idea that edges to high-scoring nodes contribute more. Both variants are implemented in NetworKit based on parallel power iteration.

3.7 Community Detection. Community detection is the task of identifying groups of nodes in the network which are significantly more densely connected among each other than to the rest of nodes. It is a data mining problem where various definitions of the structure to be discovered – the community – exist. This fuzzy task can be turned into a well-defined though NP-hard optimization problem by using community quality measures, first and foremost modularity \([24] \). We approach community detection from the perspective of modularity maximization and engineer parallel heuristics
which deliver a good tradeoff between modularity and running time [44, 43]. The PLP algorithm implements community detection by label propagation [39], which extracts communities from a labelling of the node set. A node adopts the most frequent label in its neighborhood until densely connected groups agree on a common label. Each iteration takes \( O(m) \) time, and the algorithm has been empirically shown to reach a stable solution in only a few iterations. The Lowun method for community detection [9] can be classified as a locally greedy, bottom-up multilevel algorithm. It combines local node moves with multilevel coarsening to optimize modularity. In our version PLM, node moves and coarsening are performed in parallel, and optional refinement phases (PLMR) can be added. In extensive experiments, we compared our community detection methods to other current efforts in the field [43]. We recommend the PLM and PLMR algorithms as the default choice for modularity-driven community detection in large networks. For very large networks in the range of billions of edges, PLP delivers a better time to solution, albeit with a quality loss. In recent work we adapted these algorithms to perform dynamic community detection, i.e., update a partition according to changes in the graph. Our algorithms have been thoroughly evaluated and optimized, making community detection one of the areas in which NetworKit distributes novel, state-of-the-art algorithmic tools.

4 Network Generators

Generative network models aim to explain how networks form and evolve specific structural features. Such models and their implementations as generators have at least two important uses: On the one hand, algorithm or software engineers want generators for synthetic datasets which can be arbitrarily scaled and parameterized and produce graphs which resemble the real application data. On the other hand, network scientists employ models to increase their understanding of network phenomena. NetworKit provides a versatile collection of graph generators for this purpose. In the simple probabilistic Erdős-Rényi model [35] edges are created with a uniform probability. We include an efficient implementation as described in [5]. Variations of Erdős-Rényi, the planted partition model and the stochastic blockmodel are useful for generating graphs which have distinctive dense areas with sparse connections between them (i.e., communities). The Barabasi-Albert model [2] implements a preferential attachment process which results in a power-law degree distribution. The Recursive Matrix (R-MAT) model [12] was proposed to recreate properties including a power-law degree distribution, the small-world property and self-similarity. Design goals also include few parameters and high generation speed. The Chung-Lu model [1] is a random graph model which aims to replicate a given degree distribution. Given a degree sequence, edges are created with a probability of

\[
p(u, v) = \frac{\deg(u) \cdot \deg(v)}{\sum_k \deg(k)},
\]

which recreates the degree sequence in expectation. The model has been shown to have similar capabilities as the R-MAT model [38]. For a given realizable degree sequence, the algorithm of Havel and Hakimi [26] generates a graph with exactly this degree sequence. Unlike the Chung-Lu model, the generative process promotes the formation of closed triangles, leading to a higher (and possibly more realistic) clustering coefficient. The PubWeb generator implements a geometry-driven model to create community structure [22]. Recent work yielded a parallel geometric graph generator that combines the unit disk graph model with hyperbolic geometry, producing large networks with power-law degree distribution and high clustering [45], which will be distributed with an upcoming release.

5 Comparison to Related Software

Recent years have seen a proliferation of graph processing and network analysis tools which vary widely in terms of target platform, user interface, scalability and feature set. We therefore locate NetworKit relative to these efforts. Although the boundaries are not sharp, we would like to separate network analysis toolkits from general purpose graph frameworks (e.g., Boost Graph Library and JUNG [34]), which are less focused on studying empirical networks. As closest in terms of architecture, functionality and target use cases, we see NetworkX [25], igraph [16] and graph-tool. All of them are distributed as Python modules, provide a broad feature set for network analysis workflows, and have active user communities. NetworkX is a mature toolkit and the de-facto standard for the analysis of small to medium networks in a Python environment, but not suitable for massive networks due to its pure Python implementations. Due to the similar interface, users of NetworkX are likely to move easily to NetworKit for larger networks. igraph and graph-tool address this scalability issue also by implementing core data structures and algorithms in C or C++. graph-tool builds on the Boost Graph Library and parallelizes some kernels using OpenMP. These similarities make those packages ideal candidates for an experimental comparison with NetworKit (see Section 6.2). Related efforts from the algorithm engineering community are KDT [31] (built on an algebraic, distributed parallel backend), GraphCT [17] (focused on massive multithreading architectures such as the Cray XMT), STINGER (a dynamic graph data structure with some analysis capabilities) [18] and Ligra [42] (a recent shared-memory parallel library). They offer high performance through native, parallel implementations of certain kernels. However, to characterize a complex
network in practice we need a substantial set of analytics which those frameworks currently do not provide. Furthermore, the SNAP [29] network analysis package has also recently adopted the hybrid approach of C++ core and Python interface. GUI applications like Gephi [4] and Pajek have a stronger focus on visual network exploration and less on performance on massive datasets. Distributed computing solutions like GraphLab [30] become relevant for massive graph applications, though as we argue, shared-memory multicore machines go a long way. A more detailed discussion of related software is provided in the Supplementary Material. It is our hope that all of these diverse but overlapping lines of development will cross-pollinate and lead to network analysis tools with robust performance and methodological foundations. Through further development, NetworKit will continue to set itself apart with a feature set geared towards large-scale network analysis and by including novel algorithmic approaches.

6 Experimental Evaluation

6.1 Performance Benchmark.

Fig. 2 shows results of a benchmark of the most important analytics kernels in NetworKit. Our platform is a shared-memory server with 256 GB RAM and 2x8 Intel(R) Xeon(R) E5-2680 cores (32 threads due to hyperthreading) at 2.7 GHz. The algorithms were applied to a diverse set of 14 real-world networks in the size range from 7k to 260M edges, including web graphs, social networks, connectome data and internet topology networks (see Table 1 for a description). Kernels with quadratic running time (like Betweenness) were restricted to the subset of the 4 smallest networks. Speed is measured in edges per second to relate the size of the instance to the time required for a solution, and the box plots illustrate the range of processing rates achieved (blue dots are outliers). CommunityDetectionLP and CommunityDetectionLM call the PLP and PLM algorithms, respectively. Clustering coefficient here refers to the average local coefficient, and BetweennessApprox to the method with bounded error [40]. The benchmark illustrates that a set of efficient linear-time kernels, including ConnectedComponents, the community detectors, PageRank, CoreDecomposition and ClusteringCoefficient, scales robustly to networks in the order of 10^9 edges. The iFub algorithm demonstrates its surprisingly good performance on complex networks, moving diameter calculation effectively into the class of linear-time kernels. Algorithms like BFS and ConnectedComponents actually scan every edge at a rate of 10^7 to 10^8 edges per second. Higher throughput is so far only achieved by algorithms that scan node attributes (DegreeAssortativity), and sampling-based approaches like clustering coefficient approximation. For sublinear algorithms such as the latter, scalability is in practice only constrained by the available memory. Betweenness calculation remains very time-consuming in spite of parallelization, but approximate results with an error bound of 0.1 can be obtained two order of magnitudes faster. Fig. 3 breaks processing rates of the Diameter kernel down to the particular instances, in decreasing order of size, illustrating that performance is often strongly dependent on the specific structure of complex networks.

Figure 2: Processing rates of NetworKit analytics kernels

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>powergrid</td>
<td>4941</td>
<td>6594</td>
</tr>
<tr>
<td>fb-Caltech36</td>
<td>social</td>
<td>769</td>
<td>16656</td>
</tr>
<tr>
<td>PGPgiantcompo</td>
<td>social</td>
<td>10680</td>
<td>24316</td>
</tr>
<tr>
<td>as-22july06</td>
<td>internet</td>
<td>22963</td>
<td>48436</td>
</tr>
<tr>
<td>fb-Smith60</td>
<td>social</td>
<td>2970</td>
<td>97133</td>
</tr>
<tr>
<td>fb-MIT8</td>
<td>social</td>
<td>6440</td>
<td>251252</td>
</tr>
<tr>
<td>caidaRouterLevel</td>
<td>internet</td>
<td>192244</td>
<td>609066</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>social</td>
<td>299067</td>
<td>977076</td>
</tr>
<tr>
<td>fb-Texas84</td>
<td>social</td>
<td>36371</td>
<td>1590655</td>
</tr>
<tr>
<td>in-2004</td>
<td>web</td>
<td>1382908</td>
<td>13591473</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>citations</td>
<td>6009555</td>
<td>16518948</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>social</td>
<td>4847571</td>
<td>43369619</td>
</tr>
<tr>
<td>con-fiber_big</td>
<td>brain</td>
<td>591428</td>
<td>46374120</td>
</tr>
<tr>
<td>uk-2002</td>
<td>web</td>
<td>18520486</td>
<td>261787258</td>
</tr>
</tbody>
</table>

Table 1: Networks used for performance benchmark
6.2 Comparative Benchmark. NetworKit, igraph and graph-tool rely on the same hybrid architecture of C/C++ implementations with a Python interface and implement igraph uses non-parallel C code while graph-tool also features parallelism. We benchmarked all packages in comparison on our 16-core parallel platform and present the measured performance in Fig. 4. For comparison, the performance of some of NetworkX’s Python-based kernels is shown in Fig. 5. Various issues complicated the benchmarking: In our benchmark, NetworKit was the only framework that could robustly run the set of fast kernels (excluding the expensive betweenness, its approximation, and exact clustering coefficients) on the full set of networks in less than an hour. Not all packages implement the same set of kernels in a comparable way – we compared where we could find an equivalent. igraph’s PageRank crashed on our instances and is therefore not listed, and its community detection code, while very fast on smaller instances, did not terminate in a nightly run on large networks. graph-tool’s BFS implementation was significantly slow compared to the performance of its other kernels and could therefore not be run on the larger instances. We therefore restricted the benchmark set for the comparison to the 8 smaller networks (see Table 1). This allows us to assess relative performance, but the results are somewhat distorted since some acceleration techniques such as parallelism pay off primarily on large instances. igraph, graph-tool and NetworKit are all capable of similar high processing rates which can surpass NetworKit’s rates, but the performance varies strongly among the implementations of different kernels. Another scalability factor is the memory footprint of the graph data structure. NetworKit provides a lean implementation in which the 260M edges of the uk-2002 web graph occupy only 9 GB, compared with igraph (93GB) and graph-tool (14GB). After indexing the edges, NetworKit requires 11 GB for the graph. Our code also enables fast I/O: loading the web graph from a GML file took 4 minutes, 25 minutes and 29 minutes, respectively.

7 Example Use Cases.

In the following, we present possible workflows and use cases, highlighting the capabilities of NetworKit as an interactive tool and a library.

Networks at a Glance. NetworKit uses many of the analytics kernels described above to provide an
overview of important structural features of a network. The `properties.overview` function prints a profile of the data set in tabular form. For typical networks with several thousands of edges, this output is produced almost instantly. The following example tests the scalability of the function by giving an overview of the `uk-2007-05` web graph, a network of more than 100 million pages and 3.3 billion hyperlinks from the `.uk` domain. We see that the network is sparse, highly clustered, has one giant connected component among many small ones and a diameter that is low relative to its size. Its degree distribution is highly skewed and follows a power law with exponent $\gamma \approx 2.4$, and degrees of neighbors are correlated negatively. The network has a modular structure in which several hundred thousand communities can be identified. The full analysis required less than 100 GB of memory and took 7 hours and 11 minutes on our test platform, time which is unequally distributed on the various kernels. For example, diameter estimation takes 16 minutes, and another parallel community detection algorithm (PLP) processes the graph in 1 min and 20 seconds (at 40M edges/s).

![Network Properties: uk-2007-05](image)

**Network Properties: uk-2007-05**

<table>
<thead>
<tr>
<th>Basic Properties</th>
<th>------------------</th>
<th>-----------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes, edges</td>
<td>105896555, 3301876564</td>
<td></td>
</tr>
<tr>
<td>directed?</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>weighted?</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>isolated nodes</td>
<td>742603</td>
<td></td>
</tr>
<tr>
<td>self-loops</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>density</td>
<td>0.000001</td>
<td></td>
</tr>
<tr>
<td>clustering coefficient</td>
<td>0.736643</td>
<td></td>
</tr>
<tr>
<td>max. core number</td>
<td>5704</td>
<td></td>
</tr>
<tr>
<td>connected components</td>
<td>756936</td>
<td></td>
</tr>
<tr>
<td>size of largest component</td>
<td>104288749</td>
<td></td>
</tr>
<tr>
<td>estimated diameter range</td>
<td>(112, 122)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node Degree Properties</th>
<th>------------------</th>
<th>-----------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>min./max. degree</td>
<td>(0, 975419)</td>
<td></td>
</tr>
<tr>
<td>avg. degree</td>
<td>62.360415</td>
<td></td>
</tr>
<tr>
<td>power law?, likelihood, gamma</td>
<td>True, 597.0328, 2.3996</td>
<td></td>
</tr>
<tr>
<td>degree assortativity</td>
<td>-0.1045</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Community Structure</th>
<th>------------------</th>
<th>-----------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>community detection (PLM)</td>
<td>communities 762775</td>
<td></td>
</tr>
<tr>
<td></td>
<td>modularity 0.996244</td>
<td></td>
</tr>
</tbody>
</table>

**Example Pipeline.** A thesis by Flick [19] provides an early example of NetworKit as a component in an application-specific data mining pipeline (Fig. 6), which performs analysis of protein-interaction (PPI) networks. The pipeline implements a preprocessing stage in Python, in which networks are compiled from heterogeneous datasets containing interaction data as well as expression data about the occurrence of proteins in different cell types. During the network analysis stage, preprocessed networks are retrieved from a database, and NetworKit is called via the Python frontend. The C++ core has been extended to enable more efficient analysis of tissue-specific PPI networks, by implementing in-place filtering of the network to the subgraphs of proteins that occur in given cell types. Finally, statistical analysis and visualization is applied to the network analysis data. The system is close to how we envision NetworKit as a high-performance algorithmic component in a real-world data analysis scenario, and we therefore place emphasis on the toolkit being easily scriptable and extensible.

![Figure 6: PPI network analysis pipeline with NetworKit as central component](image)

### 8 Conclusion

Among solutions for large-scale graph analytics, distributed computing frameworks are often prominently named. However, graphs arising in many data analysis scenarios are not bigger than the billions of edges that fit into a conventional main memory and can therefore be processed far more efficiently in a shared-memory parallel model (Shun and Blelloch [42] make a similar case for shared-memory graph processing). With NetworKit, we provide a package of network analytics kernels, network generators and utility software to explore and characterize large relational datasets on typical parallel workstations. Techniques that allow NetworKit to scale to large networks include both efficient C++ implementations and appropriate algorithmic methods (parallelism, graph sampling and others). The interface provided by our Python module allows domain experts to focus on data analysis workflows instead of the intricacies of programming. The open-source package is under continuous
development and will be amended by optimizations and new features informed by algorithm engineering research as well as developments in network science. Packages which also follow the hybrid approach of C/C++ kernels and Python interface achieve similar high performance, but specific figures vary among frameworks and kernels. Our experimental study showed that NetworKit is capable of quickly processing large-scale networks for a variety of analytics kernels in a reliable manner, whereas the other two frameworks in the study could not always solve all test cases. To achieve high performance overall, NetworKit uses a combination of algorithmic acceleration techniques, efficient implementations and low memory footprint. We recommend NetworKit for the comprehensive structural analysis of large complex networks in the range of $10^5$ to $10^{10}$ edges. Specialized programming skills are not required, though users familiar with the Python ecosystem of data analysis tools will appreciate the possibility to seamlessly integrate our toolkit.

Installation and Support. NetworKit is free software licensed under the permissive MIT License for the least amount of friction with respect to sharing and reuse of code in a scientific setting. We would like to encourage usage and contributions by a diverse community, including data mining users and algorithm engineers. The package source and additional resources can be obtained from http://networkit.iti.kit.edu. The included documentation provides support for setting up and getting started with NetworKit. It also includes a tutorial on basic network analytic tasks. The module networkit is also installable via the Python package manager `pip`. Optionally, the Python layer can be omitted by building a subset of functionality as a C++ application.

History and Roadmap. NetworKit started as a collection of community detection algorithms written in C++, first released in March 2013. The current release is version 3.3 of August 2014, and we will release an update with additional features and improvements later this year. Features of upcoming releases are likely to focus on multiple areas: Sparsification and filtering of networks to reduce computational work or enhance properties; algorithms for layouting and visualizing large complex networks; tools for the analysis of dynamic graphs; as well as general usability improvements in response to applications.

Acknowledgements. This work was partially supported by the project Parallel Analysis of Dynamic Networks – Algorithm Engineering of Efficient Combinatorial and Numerical Methods, which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg. A. S. acknowledges support by the RISE program of the German Academic Exchange Service (DAAD).

We thank Maximilian Vogel for co-maintaining the software. We also thank Lukas Barth, Miriam Beddig, Elisabetta Bergamini, Stefan Bertsch, Pratistha Bhattacharai, Andreas Bilke, Simon Bischof, Guido Brückner, Patrick Flick, Michael Hamann, Lukas Hartmann, Daniel Hoske, Gerd Lindner, Moritz von Looz, Yassine Marrakchi, Marcel Radermacher, Klara Reichard, Marvin Ritter, Florian Weber, Michael Wegner and Jörg Weisbarth for contributing code.

References


A Supplementary Material

A.1 Detailed Comparison to Related Software.

In the following, we compare NetworKit to a wider set of software packages with a possibly similar scope of application. Network analytics packages offer a collection of analysis kernels through a specific user interface. We selected a number of software packages which we see as close to NetworKit in one aspect or the other, and compare them with respect to design goals, usability, feature set and other criteria. Although the boundaries to graph frameworks (e.g. Boost Graph Library) are often blurred, we do not consider them sufficiently similar. Table 2 shows a comparison of the feature sets.

NetworkX [25] is a feature-rich Python package for network analysis whose development started in 2002. It is considered the de-facto standard for the analysis of small to medium networks in a Python environment. For these reasons, NetworKit aims for compatibility with NetworkX through functions for the conversion of graph object. NetworkX has a large feature set and provides a highly flexible graph data structure, but its applicability to large graphs is limited. igraph [16] is a C library aimed at creating and manipulating networks with millions of nodes and edges. In addition to the C library, igraph provides a highly flexible graph data structure, but its applicability to large graphs is limited. igraph [16] is a C library aimed at creating and manipulating networks with millions of nodes and edges. In addition to the C library, igraph provides interfaces for Python, R, and Ruby. graph-tool [42] is a graph processing framework for shared-memory machines. Algorithms are written in C++ and expressed in terms of edge maps, vertex maps and vertex subsets. Ligra provides several fundamental graph algorithms and analytic kernels, but its feature set is still limited from a network analysis point of view. Nor does it support interactive network analysis. GraphCT (Graph Characterization Toolkit) [17] is written in C and targets both general multicore systems and the specialized parallel platform Cray XMT. Similar to NetworKit, the graph resides in main memory and is accessed by multiple threads. The defining feature of the Cray XMT architecture is that it uses massive hardware multithreading and fast thread switching to hide memory latency. GraphCT includes a basic collection of network analysis kernels. STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation) [18] is mainly a dynamic graph structure that has been extended over time by several graph and network analysis algorithms. These algorithms can be used within a standalone command line tool or as a library. Targeted at shared-memory platforms including the Cray XMT, STINGER focuses on fast execution. Recently, experimental interfaces to Python and Java have been added. The GraphLab project [30] follows a parallel, distributed and asynchronous model for the implementation of graph algorithms. While GraphLab ships with a number of network analytics kernels, it fills a different niche as a framework for building large-scale distributed applications such as web-based recommender systems.

The Knowledge Discovery Toolbox (KDT) [31] is a project with a focus similar to NetworKit: It provides high-performance kernels through an interactive Python interface. The underlying graph representation is algebraic. KDT is layered on top of Combinatorial BLAS, a linear algebra library which provides sparse matrix classes and operations. It is written in C++ and uses MPI for distributed memory parallelism. KDT currently has a limited feature set. Ligra [42] is a graph processing framework for shared-memory machines. Algorithms are written in C++ and expressed in terms of edge maps, vertex maps and vertex subsets. Ligra provides several fundamental graph algorithms and analytic kernels, but its feature set is still limited from a network analysis point of view. Nor does it support interactive network analysis. GraphCT (Graph Characterization Toolkit) [17] is written in C and targets both general multicore systems and the specialized parallel platform Cray XMT. Similar to NetworKit, the graph resides in main memory and is accessed by multiple threads. The defining feature of the Cray XMT architecture is that it uses massive hardware multithreading and fast thread switching to hide memory latency. GraphCT includes a basic collection of network analysis kernels. STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation) [18] is mainly a dynamic graph structure that has been extended over time by several graph and network analysis algorithms. These algorithms can be used within a standalone command line tool or as a library. Targeted at shared-memory platforms including the Cray XMT, STINGER focuses on fast execution. Recently, experimental interfaces to Python and Java have been added. The GraphLab project [30] follows a parallel, distributed and asynchronous model for the implementation of graph algorithms. While GraphLab ships with a number of network analytics kernels, it fills a different niche as a framework for building large-scale distributed applications such as web-based recommender systems.
<table>
<thead>
<tr>
<th>Feature Group</th>
<th>Feature</th>
<th>Sub-Feature</th>
<th>NetworKit</th>
<th>NetworkX</th>
<th>igraph</th>
<th>graph-tool</th>
<th>Gephi</th>
<th>JUNG</th>
<th>Pajek</th>
<th>KDT</th>
<th>STINGER</th>
<th>Graphtool</th>
<th>GraphLab</th>
<th>Ligra</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>API</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>interactive</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Data Structures</td>
<td>graph (undirected/directed/ weighted)</td>
<td></td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
</tr>
<tr>
<td></td>
<td>hypergraph</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Algorithms</td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>connected components</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>search (BFS/DFS)</td>
<td></td>
<td>●●●●</td>
<td>●●</td>
<td>●●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>weighted shortest path</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>degree power law estimation</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>degree assortativity</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>core decomposition</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>diameter (ex/ap)</td>
<td></td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
</tr>
<tr>
<td></td>
<td>betweenness (ex/ap)</td>
<td></td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
</tr>
<tr>
<td></td>
<td>closeness (ex/ap)</td>
<td></td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
</tr>
<tr>
<td></td>
<td>eigenvector/page rank</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>clustering coefficients (ex/ap)</td>
<td></td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
<td>●●●●</td>
</tr>
<tr>
<td></td>
<td>community detection</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>modularity</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>matching</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>independent set</td>
<td></td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>flows</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Generators</td>
<td>Erdős-Rényi</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Barabási-Albert</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>R-MAT</td>
<td></td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Watts-Strogatz</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Chung-Lu</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Havel-Hakimi</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Hyperbolic Unit-Disk</td>
<td></td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 2: Feature matrix of network/graph analysis software (Legend: ● present, ○ upcoming, o missing)
(a) Betweenness

(b) Betweenness approximation

(c) Breadth-first search

(d) Average local clustering coefficient

(e) Approximation of average local clustering coefficient

(f) Community detection with PLM

(g) Community detection with PLP

(h) Connected components

(i) Core decomposition

(j) Degree assortativity

Figure 7: NetworKit benchmark results broken down by network instance
Figure 8: NetworKit benchmark results broken down by network instance
A.2 Code Example. The following block of C++ code illustrates NetworKit’s programming model for parallel graph algorithms. It is a parallelized implementation of Brandes’ algorithm for betweenness centrality which is used for the Betweenness kernel. The core function runs a search from a source node to compute the so-called dependencies of other nodes on the source node, with the centrality score of a node being the sum over all dependencies. This is formulated as a lambda expression which is then passed to the node iterator Graph::balancedParallelForNodes, which executes the function in parallel on the set of nodes while also performing load balancing. To enable efficient parallelization and avoid data races, the algorithm first computes thread-local centrality scores which are then added for the final result. We strive for an API on the C++ layer that facilitates the concise formulation of graph algorithms, and thereby the extensibility of the library.

```cpp
void Betweenness::run() {
    count z = G.upperNodeIdBound();
    scoreData.clear();
    scoreData.resize(z);
    // thread-local scores for efficient parallelism
    count maxThreads = omp_get_max_threads();
    std::vector<std::vector<double>> scorePerThread(maxThreads, std::vector<double>(G.upperNodeIdBound()));
    auto computeDependencies = [&]() {  
        std::vector<double> dependency(z, 0.0);
        // run SSSP algorithm and keep track of everything
        std::unique_ptr<SSSP> sssp;
        if (G.isWeighted()) {
            sssp.reset(new Dijkstra(G, s, true, true));
        } else {
            sssp.reset(new BFS(G, s, true, true));
        }
        sssp->run();
    
    // compute dependencies for nodes in order of decreasing distance from s
    std::stack<node> stack = sssp->getStack();
    while (!stack.empty()) {
        node t = stack.top();
        stack.pop();
        for (node p : sssp->getPredecessors(t)) {
            double weight;
            tmp.ToDouble(weight);
            dependency[p] += weight * (1 + dependency[t]);
        }
        if (t != s) {
            scorePerThread[omp_get_thread_num()][t] += dependency[t];
        }
    }
    G.balancedParallelForNodes(computeDependencies);
    INFO("adding thread-local scores");
    // add up all thread-local values
    for (auto local : scorePerThread) {
        G.parallelForNodes([&](node v){
            scoreData[v] += local[v];
        });
    }
    if (normalized) {
        // divide by the number of possible pairs
        count n = G.numberOfNodes();
        count pairs = (n-2) * (n-1);
        G.forNodes([&](node u){
            scoreData[u] = scoreData[u] / pairs;
        });
    }
}
```