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

Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke

Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Germany
http://parco.iti.kit.edu

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 as well as 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 interactive interface written in Python. The package supports shared-memory parallelism and scales from notebooks to compute servers. 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. The current feature set includes analytics kernels such as degree distribution, connected components, clustering coefficients, community detection, k-core decomposition, degree assortativity and multiple centrality indices. Moreover, NetworKit comes with a collection of graph generators. With the current release, we present and open up the project to a community of both algorithm engineers and domain experts.

Contents

1 Introduction

Complex networks are heterogeneous 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 as networks [19, 13]. Accordingly, network analysis methods are quickly becoming pervasive in science, technology and society: Node ranking by centrality is the basis for modern web search [39], community detection methods find applications on social media sites [25] or in cancer research [30], and tracking social influence through networks is equally interesting to sociologists [24] 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: Recall 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 [4], 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 us to utilize the parallel processing capabilities of multicore systems wherever 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. In this work, we give an introduction to the toolkit and describe it under algorithm and software engineering aspects.

1.1 Design Goals

There is already a variety of software packages which provide graph algorithms in general and network analysis capabilities in particular (see Section 6 for a comparison to related packages). However, NetworKit aims to balance a specific combination of 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 billion edge web graph can be performed on a machine with 16 physical cores and 256 GB of RAM in a matter of minutes [47].

Interface. Networks are as diverse as the series of questions we might ask of them - for example, 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 avoid restricting the user to fixed and predefined tasks. Rather, the aim must be to create convenient and freely combinable functions. 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 allows us to integrate it into a fully fledged computing environment for scientific workflows [40]. It is also straightforward to set up and control a remote server for heavy computations.

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, networkx for additional network analysis tasks, or numpy and scipy for advanced numeric and scientific computation. Furthermore, NetworKit aims to support a variety of input/output formats.

1.2 Architecture

In order to achieve the design goals described above, we implement NetworKit as a 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 use interactively from a Python shell, which is the main usage scenario we envision. However, the code can also be used as a library for application programming either at the Python or C++ level. Throughout the project we use object-oriented and functional concepts. On the C++ level, we make extensive use of closures, using the lambda syntax introduced with C++11. 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 18 000 lines of C++ code include unit tests written in the googletest framework.

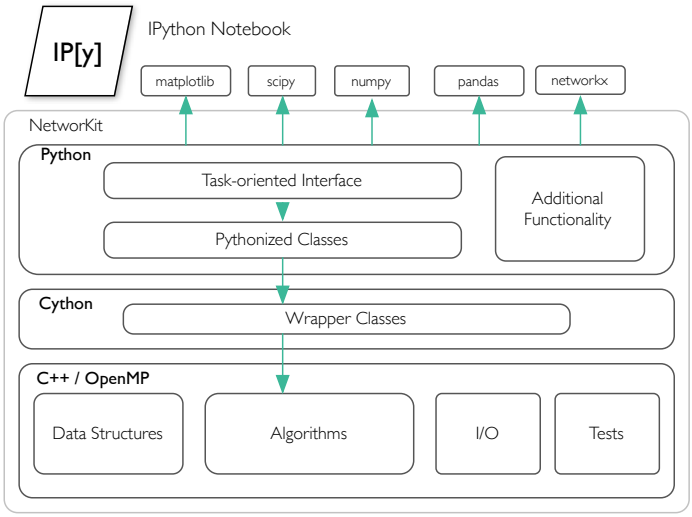

Figure 1: NetworKit architecture overview

Connecting these native implementations to the Python world is enabled by the Cython toolchain [11]. Among other things, Cython can compile pure Python code to C or C++, circumventing the Python interpreter, and also allows for static type annotations – yielding considerable speedup in combination. Currently we use Cython merely to integrate native code by compiling it into a native Python extension module. In order to expose a C++ class to Python with Cython, one needs to a) declare the interface of the class and b) write a Python wrapper class through which method calls are delegated to the native object. The benefits of Python integration are the following: First of all, NetworKit's functionality can be accessed interactively. Thus, analysis kernels can be freely combined. Furthermore, NetworKit can be seamlessly integrated into the rich Python ecosystem for data analysis. We consider this kind of integration crucial for real-world data analysis workflows.

2 Basics

Before discussing the collection of network analysis kernels, we briefly describe basic concepts and data structures used throughout NetworKit. We also introduce some notation used in this text: We denote as $G = (V, E, \omega)$ an undirected graph, consisting of a node set V, an edge set E and an optional weight function

$\omega : E \to \mathbb{R}_{\geq 0}$. An undirected edge $\{u, v\}$ connects nodes u and v, where $u = v$ is possible. We denote the degree of a node, i. e. the number of incident edges, as $\deg(v)$.

2.1 Graph Data Structure

The `NetworKit.Graph` class implements an undirected, optionally weighted graph using an adjacency array data structure with $O(n + m)$ memory requirement. A node is represented by a 64 bit integer index (type `node`), and an edge is identified by an undirected pair of `node`. The basic container is an `std::vector<std::vector<node> >` in which a node index addresses the adjacencies of the respective node. An undirected edge $\{u, v\}$ is stored as two entries, one for the adjacency $u \to v$ and one for $v \to u$. This allows for the retrieval of all neighbors of a node u in $O(\deg(u))$ time. Weights (of type `double` defined as `edgeweight`) are stored in an analog stucture. Edge insertion is therefore possible at the cost of two `std::vector::push_back` operations. Checking for or deleting an edge $\{u, v\}$ requires a scan of the adjacencies in $O(\deg(u))$. A graph can contain nodes from a certain index range $[0, z]$, which can also be extended. Node deletion is implemented by marking the index as deleted in a boolean vector. At the C++ level, the class provides convenient (parallel) iterator methods which hide the details of the data structure and accept arbitrary operations on nodes or edges in the form of a function, e. g., lambda expression. In the following example, we pass a lambda function to the iterator method to initialize a value for each node. Because the lambda expression creates a closure which captures variables from the surrounding scope by reference, the syntax is quite similar to a standard for-loop.

```
std::vector<node> tempMap(G.upperNodeIdBound());
G.parallelForNodes([&](node v){
        tempMap[v] = v; // initialize to identity
});
```

Internally, the `Graph` class enumerates nodes in parallel, checks whether they are present in the graph and calls the given function once per node.

```
template<typename L> inline void NetworKit::Graph::parallelForNodes(L handle) {
#pragma omp parallel for
        for (node v = 0; v < z; ++v) {
                if (exists[v]) {
                        handle(v);
                }
        }
}
```

Identifying nodes by nothing more than an integer index implies that the graph data structure does not need to provide storage for node attributes, which can be handled by any container that can be addressed by the node indices. In practice, this approach has proven beneficial for writing performant algorithm implementations.

3 Analytics

This section describes the current set of network analysis algorithms implemented in NetworKit. A basic definition of the concepts is followed by a description of the specific algorithm(s). An index of features pointing to the corresponding function or class is provided by Table 1.

feature group	submodule	feature	class / function
input/output	`graphio`	graph from file	`readGraph(path, [format])`
		write graph to file	`writeGraph(G, path, [format])`
	`nxadapter`	NetworkX compatibility	`nk2nx(G)` `nx2nk(G)`
general properties	`properties`	tabular overview	`overview(G)`
		size	`size(G)`
		density	`density(G)`
		degree distribution	`degreeDistribution(G)`
		degree power law fit	`powerLawFit(G)`
		degree power law exponent	`powerLawExponent(G)`
		degree assortativity	`degreeAssortativity(G)`
		diameter	`Diameter`
		clustering coefficients	`clustering(G)` `ClusteringCoefficient`
		connected components	`components(G)` `ConnectedComponents`
		core decomposition	`CoreDecomposition(G)`
centrality	`centrality`	betweenness	`BetweennessCentrality(G, [normalized])` `ApproxBetweenness(G, [epsilon, delta])`
		PageRank	`PageRank(G, damp, [tol])`
		EigenvectorCentrality	`EigenvectorCentrality(G, [tol])`
community detection	`community`	main function	`detectCommunities(G, [algorithm])`
		PLP	`PLP([theta])`
		PLM	`PLM([refinement?, resolution, parallelism, maxIter])`
		EPP	`EPPFactory`
		CNM	`CNM`
generators	`generators`	Erdös-Renyi	`ErdösRenyiGenerator(n,p)`
		Barabasi-Albert	`BarabasiAlbertGenerator(k, nMax, n0)`
		R-MAT	`RmatGenerator(scale, edgeFactor, a, b, c, d)`
		Chung-Lu	`ChungLuGenerator(degSeq)`
		Havel-Hakimi	`HavelHakimiGenerator(degSeq)`

Table 1: Feature index

3.1 Degree Distribution

The degree distribution 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. Such networks have been categorized as *scale-free* [7], referring to the fact that it is not possible to pick a node of typical degree. Consequently, measures such as the average degree for all nodes convey very little information about the network.

Definition 1. *A scale-free network is a network in which the frequency $p(k)$ of nodes with degree k follows a power law with coefficient γ*

$$p(k) \sim k^{-\gamma} \tag{1}$$

The `powerlaw` Python module, developed by Alstott et al. [3], employs statistical methods from [18, 31] to determine if a probability distribution fits a power law. Since the original code was written for Python 2, we distribute a Python 3 version with NetworKit. We provide a function which returns whether a power law degree distribution is considered a good fit, and also a second parameter R quantifying the goodness of the fit:

> R is the loglikelihood ratio between the two candidate distributions. This number will be positive if the data is more likely in the first distribution, and negative if the data is more likely in the second distribution. The exponential distribution is the absolute minimum alternative candidate for evaluating the heavy- tailedness of the distribution. The reason is definitional: the typical quantitative definition of a "heavy- tail" is that it is not exponentially bounded. Thus if a power law is not a better fit than an exponential distribution (as in the above example) there is scarce ground for considering the distribution to be heavy-tailed at all, let alone a power law. – [3]

In case a power law degree distribution is present, a function returns the exponent γ, which has been found to be a characteristic property of a complex networks. For further analysis the raw degree distribution can also be obtained.

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 the mixing with respect to node degree, which is an important aspect of the network structure, pointing e. g., to a hierarchical composition. Its strength is often expressed as *degree assortativity coefficient* r, which lies in the range $-1 \leq r \leq 1$. High values occur in networks with assortative mixing by degree such as many social networks. There, high-degree nodes are preferentially attached to other high-degree nodes (and low-degree to low-degree). A strong negative value for r indicates disassortative mixing, i. e. high-degree nodes tend to connect to low-degree ones. Note that r is an example of a Pearson correlation coefficient with a covariance in its numerator and a variance in the denominator [36]. Below is Newman's equivalent reformulation, which we implement in NetworKit with $O(m)$ time and constant memory requirements.

Definition 2. *The* degree assortativity coefficient r *is defined as*

$$r = \frac{[\frac{1}{m}\sum_{\{u,v\}\in E}\deg(u)\deg(v)] - [\frac{1}{2m}\sum_{\{u,v\}\in E}\deg(u)+\deg(v)]^2}{[\frac{1}{2m}\sum_{\{u,v\}\in E}\deg^2(u)+\deg^2(v)] - [\frac{1}{2m}\sum_{\{u,v\}\in E}\deg(u)+\deg(v)]^2}.$$

3.3 Diameter

The diameter of a graph is the maximum length of a shortest path between any two nodes [35]. A surprising observation about the diameter of complex networks is often referred to as the *small world phenomenon*,

or specifically in social networks, *six degrees of separation*: The diameter tends to be very small, and often constant or even shrinking with network growth.

Definition 3. *Let $d(u, v)$ denote the distance of nodes u and v in an undirected graph $G = (V, E)$. We define:*

$$\begin{aligned} \mathrm{ecc}(v) &:= \max\{d(v, w) : w \in V\} && \textit{eccentricity of } v \\ \mathrm{diam}(G) &:= \max\{\mathrm{ecc}(v) : v \in V\} && \textit{diameter of } G \end{aligned}$$

While we provide a function to calculate the exact diameter using BFS for unweighted or Dijkstra's algorithm for weighted graphs, this is impractical for the large networks we target. Diameter estimation is therefore a critical feature. The approximation algorithm due to Magnien et al. [34] iteratively yields an upper and lower bound for the diameter, which it iteratively improves by performing two steps until the desired error is reached: Computing the eccentricity of a random node, and computing a BFS tree starting from a random node. The algorithm thereby harnesses the observations that $\mathrm{ecc}(v) \leq \mathrm{diam}(G) \leq 2 \cdot \mathrm{ecc}(v)$ for any node v and $\mathrm{diam}(G) \leq \mathrm{diam}(T)$ for any spanning tree T of G.

3.4 Clustering Coefficients

Clustering coefficients are key figures for the amount of transitivity in complex networks, i. e. the tendency of edges to form between indirect neighbor nodes. For example, the global clustering coefficient is typically quite high in social networks, whose generative processes have a tendency to close triangles: Intuitively speaking, humans tend to connect to friends of friends. In contrast, the clustering coefficient is close to 0 for random graphs.

Definition 4. *Let W be the set of wedges, i. e. of the paths of length 2, in G. Moreover, let Δ be the set of triangles, i. e. of the circles of length 3, in G.*

The global clustering coefficient C_g *measures the fraction of triangles to the total number of wedges [35]:*

$$C_g(G) = 3 \cdot \frac{|\Delta|}{|W|} \tag{2}$$

Let $\Delta(v)$ and $W(v)$ denote the set of triangles containing v and the set of wedges with v as central node, respectively. Moreover, let $V' := V \setminus \{u \in V : \deg(u) \leq 1\}$ and $C(v) := \frac{|\Delta(v)|}{|W(v)|} = |\Delta(v)| \cdot \binom{\deg(v)}{2}^{-1}$.

The average local clustering coefficient C_l *averages the nodes' tendency to form triangles:*

$$C_l(G) := \frac{1}{|V'|} \sum_{v \in V'} C(v) \tag{3}$$

The factor 3 accounts for symmetries (each triangle is part of three wedges). C_g ranges from 0 to 1 and is 1 if the graph is complete (a clique). A straightforward calculation of the clustering coefficient with a node iterator requires $O(nd_{max}^2)$ time. Even with a parallel implementation, this will be very time-consuming for networks in the size range of many million edges. Instead, it is possible to approximate clustering coefficients in essentially linear or even constant time, depending on the respective measure. Details on their wedge sampling approximation algorithm can be found in Schank and Wagner [45]. NetworKit offers both the exact and the approximate calculation of clustering coefficients.

3.5 Components and Cores

Components and cores are related concepts for subdividing 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, containing a large part of all nodes. It is usually accompanied by a large number of very small components.

Definition 5. *For an undirected graph $G = (V, E)$, a connected component is a maximal subgraph $G' = (V', E')$ in which any two nodes from V' are connected by a path.*

We compute connected components in linear time using a label propagation scheme. After initializing each node with a unique integer label, we perform multiple iterations over the node set and let each node adopt the highest label in its neighborhood. It is straightforward that a connected component will eventually adopt a single label.

Core decomposition allows for a more fine-grained subdivision of the node set according to connectedness. k-cores result from successively peeling away nodes of degree k.

Definition 6. *A k-core C_k is a maximal subgraph in which each node is adjacent to at least k other nodes.*

A connected component is thus equivalent to a 1-core. A k-core decomposition also lets us categorize nodes according to the highest-order core in which they are contained, assigning a *core number* to each node.

Definition 7. *The core number k_v of a node $v \in V$ is the highest value of k for which there is a k-core which contains v.*

For each possible $k \in \{0, 1, 2, \dots\}$ in increasing order, we arrive at the k-core by iteratively removing nodes with degrees less than k. The algorithm implemented in NetworKit uses a bucket data structure for managing remaining node degrees, similar to the one used by Fiduccia and Mattheyses for graph partitioning [23]. Our algorithm's total running time is $O(m)$, matching other implementations [10].

3.6 Centrality

Centrality refers to the relative importance of a node within a network. Different ideas of importance are expressed by measures such as degree centrality, betweenness, closeness and eigenvector centrality. We distribute efficient implementations for betweenness, eigenvector centrality and PageRank.

Betweenness. Betweenness centrality expresses the concept that a node is important if it lies on many shortest paths between nodes in the network.

Definition 8. *Given an undirected, optionally weighted graph $G = (V, E, \omega)$, let σ_{st} denote the number of shortest paths between nodes s and t, and $\sigma_{st}(v)$ the number of such paths which contain v as an intermediate node. The betweenness centrality of a node v is defined as*

$$c_b(v) := \sum_{\{s,t\} \in V \times V : s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4}$$

A naive algorithm for calculating all $c_b(v)$ for $v \in V$ would require cubic time. We implement Brandes's algorithm [14], by which betweenness centrality can be computed more efficiently, using $O(nm + n^2 \log n)$ time and $O(n + m)$ space for weighted graphs. For unweighted graphs, running time is reduced to $O(nm)$. For sparse unweighted graphs (e. g., complex networks) this means a running time improvement by the factor $O(n)$. Optionally, our implementation computes the scores for unweighted graphs in parallel, at the expense of a larger memory footprint. Since this is still practically infeasible for the large data sets we target, NetworKit includes also a parallelized implementation of a recent approximation algorithm [43] with probabilistic guarantee that the error is at most an additive constant.

Eigenvector Centrality and PageRank. Eigenvector centrality and its variant PageRank [39] assign relative importance to nodes according to their connections, incorporating the idea that edges to high-scoring nodes contribute more. This recursive definition has the following analogon in algebraic graph theory:

Definition 9. *Given a graph G and its adjacency matrix A, we require a centrality score x_i of node v_i to be proportional to the scores of its neighbors:*

$$x_i = c \sum_{j=1}^{n} A(i,j)\ x_j \qquad c \neq 0$$

By the Perron-Frobenius theorem, there exists a nonnegative eigenvector x of A which corresponds to the largest eigenvalue λ. An entry x_i constitutes the centrality score for node v_i.

PageRank is a version of eigenvector centrality which introduces a damping factor, modeling a random web surfer which at some point stops following links and jumps to a random page. In PageRank theory, centrality is understood as the probability of such a web surfer to arrive on a certain page. Implementations of both variants based on parallel power iteration are distributed with NetworKit.

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. NetworKit started as a testbed for the novel parallel community detection algorithms PLP, PLM and EPP [48, 47], which remain a prominent feature.

Community detection 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 optimization problem by using community quality measures, first and foremost *modularity* [27]. We approach community detection from the perspective of modularity maximization: Faced with an NP-hard optimization problem [15], we engineered parallel heuristics which deliver a good tradeoff between modularity and running time. Other objective functions can be integrated into these algorithms fairly easily as long as the effect on the objectives by a local community change can be computed locally as well. In the following, we briefly introduce modularity and then describe a number of algorithms.

Modularity. Modularity measures the quality of a partition ζ of V by comparing its *coverage*, i. e. the fraction of edges contained within a community, with the coverage it would achieve in a randomized null-model graph. Thus, a high modularity partition covers significantly more edges than expected by chance

Definition 10. *Let $\omega(u, C) := \sum_{\{u,v\}:v\in C} \omega(u,v)$ be the weight of all edges from u to nodes in community C, and define the* volume *of a node and a community as $vol(u) := \sum_{\{u,v\}:v\in N(u)} \omega(u,v) + 2\cdot\omega(u,u)$ and $vol(C) := \sum_{u\in C} vol(u)$, respectively. The modularity of a partition is defined as*

$$mod(\zeta, G) := \sum_{C\in\zeta} \left(\frac{\omega(C)}{\omega(E)} - \frac{vol(C)^2}{4\omega(E)^2} \right) \tag{5}$$

We provide parallel code for the fast computation of the modularity of a partition of the node set with respect to a graph.

PLP. Community detection by label propagation, as originally introduced by Raghavan et al. [42], extracts communities from a labelling $V \to \mathbb{N}$ of the node set. Initially, each node is assigned a unique label, and then multiple iterations over the node set are performed: In each iteration, every node adopts the most

frequent label in its neighborhood. Densely connected groups of nodes thus agree on a common label, and eventually a globally stable consensus is reached, which usually corresponds to a good solution for the network. Label propagation therefore finds communities in near linear time: Each iteration takes $O(m)$ time, and the algorithm has been empirically shown to reach a stable solution in only a few iterations. The local update rule and the absence of global variables make label propagation well suited for a parallel implementation.

PLP is our parallel implementation of community detection by label propagation. We adapt the original algorithm to make it applicable to weighted graphs. Iteration continues until the number of nodes which changed their labels falls below a threshold θ, which often yields considerable speedup at the cost of a negligible loss of modularity. PLP is by far the fastest community detection method in NetworKit, but generally stays significantly behind other methods in terms of the achieved modularity.

PLM. The *Louvain method* for community detection was first presented by Blondel et al. [12]. It can be classified as a locally greedy, bottom-up multilevel algorithm and uses modularity as the objective function. In each pass, nodes are repeatedly moved to neighboring communities so that the locally maximal increase in modularity is achieved, until the communities are stable. Then, the graph is coarsened according to the solution and the procedure continues recursively, forming communities of communities. Finally, the communities in the coarsest graph determine those in the input graph by direct prolongation.

We provide a shared-memory parallelization of the Louvain method (PLM) in which node moves are evaluated and performed in parallel instead of sequentially. We also extend the method by an optional refinement phase, i. e. when a partition is returned from prolongation, nodes are again moved for modularity gain. This approach was described as the PLMR algorithm and can be switched on by a parameter of the implementation.

EPP. In machine learning, *ensemble learning* is a strategy in which multiple *base classifiers* or *weak classifiers* are combined to form a strong classifier. Classification in this context can be understood as deciding whether a pair of nodes should belong to the same community.

The EPP scheme works as follows: In a preprocessing step, assign G to an ensemble of base algorithms. The graph is then coarsened according to the *core communities* $\bar{\zeta}$, which represents the consensus of the base algorithms. Coarsening reduces the problem size considerably, and implicitly identifies the contested and the unambiguous parts of the graph. After the preprocessing phase, the coarsened graph G' is assigned to the final algorithm, whose result is applied to the input graph by prolongation. We instantiate this scheme with PLP as a base algorithm and PLM or PLMR as the final algorithm.

CNM. For comparison, we also include a sequential implementation of the well-known CNM method for modularity maximization [17]. However, algorithm and implementation are not nearly as scalable as the ones described above.

Choice of Algorithm. In extensive experiments, we compared our community detection methods to other current efforts in the field [47]. A Pareto evaluation (see Figure 2), relating modularity and running time, showed that our algorithms deliver a favorable tradeoff. We recommend the PLM and PLMR algorithms as the default choice for modularity-driven community detection in large networks. A small quality boost can be achieved by enabling refinement (i. e. the PLMR algorithm), at the cost of some running time. For very large networks in the range of billions of edges, PLM running time can be undesirably high. In this case, PLP delivers a better time to solution, albeit with a quality loss. A middle ground can be achieved by combining PLP and PLM in the EPP ensemble scheme.

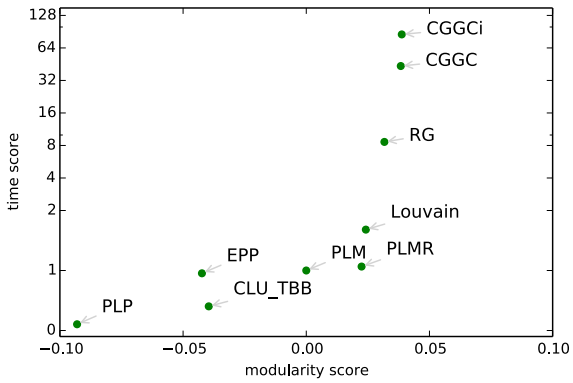

Figure 2: Pareto evaluation of current community detection algorithms

3.8 Additional Graph Algorithms

In addition to the aforementioned network analysis methods, NetworKit also includes a collection of basic graph algorithms, breadth-first and depth- first search, Dijkstra's algorithm for shortest paths and code for computing approximate maximum weight matchings.

4 Generators

Generative 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, software engineers want generators for synthetic datasets which can be arbitrarily scaled and parametrized and produce graphs which resemble the real application data. On the other hand, network scientists need models to increase their understanding of network phenomena: What are the rules that guide the formation and evolution of the complex networks we observe? If we observe common structures in very different domains (e.g. as different as society and biochemistry), is it because common abstract principles are at work? What are "typical" and "untypical" structures that emerge? Can we make predictions based on models?

So far, NetworKit provides efficient generators for basic Erdős-Rényi random graphs, the Barabasi-Albert model (which produces a power law degree distribution) and the Chung-Lu and Havel-Hakimi model (which replicate any realizable degree distribution) as well as the R-MAT model.

Erdős-Rényi Model. In this simple probabilistic model edges are created among n nodes with a uniform probability of p for each of the $\{u, v\}$ pairs. It is the earliest attempt to create a formal method for generating graphs, developed by Paul Erdős and Alfred Rényi in the 1950s [38]. Not intended to generate realistic graphs, it was viewed as a source of mathematical examples, and an extensive analytical examination followed. We include an efficient implementation as described in [9].

Clustered Random Graphs. A simple variation of the Erdös-Renyi model is useful for generating graphs which have distinctive dense areas with sparse connections between them (i. e. communities). Nodes are equally distributed over k subsets, while nodes from the same subset are connected with probability p_{in} and nodes from different subsets with a smaller probability p_{out}.

Barabasi-Albert Model. The Barabasi-Albert model [2] implements a preferential attachment process ("rich become richer") which results in a power-law degree distribution. The model was introduced in order to produce scale-free networks. Both Erdős–Rényi and Watts-Strogatz models (chronological predecessors) are not able to generate networks with such properties. According to the model, the probability that a new node will be attached to an existing node v is defined as:

$$p(v) = \frac{deg(v)}{\sum_{u \in V} deg(u)}$$

User-defined parameters are the number of nodes to be generated and the number of edges each new node introduces into the graph. Focused on generating a power law degree distribution, the model lacks realism in other aspects (e.g. community structure [44]), and the scalability of the generator is limited.

R-MAT Generator The Recursive Matrix (R-MAT) model [16] 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 R-MAT generator operates on the initially empty adjacency matrix A of the result graph, an $n \times n$ matrix where n is a power of two. The matrix is recursively subdivided into four equal-sized quadrants. Each is assigned one of the probabilities a, b, c, d which add up to 1. Edges are "dropped" into the matrix and land in one of the quadrants according to their probabilities. The process continues recursively by further subdividing the quadrant into four parts with probabilities a, b, c, d, until a 1×1 partition is reached, which determines the location of the edge. NetworKit includes an efficient sequential implementation of R-MAT.

Chung-Lu Model. The Chung-Lu model [1] is a random graph model which aims to replicate a given degree distribution. Given a degree sequence, the method creates edges between nodes with a probability of

$$p(u, v) = \frac{deg(u)deg(v)}{\sum_k deg(k)},$$

which recreates the degree sequence in expectation. The model can be conceived as a weighted version of the Erdös-Renyi model, and has been shown to have similar capabilities as the SKG or R-MAT model [41].

Havel-Hakimi Generator. For a given realizable degree sequence, the algorithm of Havel and Hakimi [29] generates a graph with exactly this degree sequence. While this is similar to the Chung-Lu model, the generative process promotes the formation of closed triangles, leading to a higher (and possibly more realistic) clustering coefficient.

PubWeb Generator. This network model is motivated by the P2P computing library *PubWeb* and has been presented previously [26]. For the generative process nodes are embedded into the 2D Euclidean unit torus (square with wrap-around boundaries). To create edges, a variation of the disc graph model is employed with a uniform communication radius r for all nodes. A node is connected to up to k nearest neighbors within its communication radius.

5 Example Workflows

In the following, we present a few examples for possible workflows, highlighting the interactive capabilities of 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.

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. The following example gives an overview of the `PGPgiantcompo` graph, a social network and web of trust resulting from signatures on PGP public keys [46]. As we see, the network consists of a single connected component, has a power law degree distribution, a diameter in the range $[22, 24]$, shows assortativity with respect to node degrees, moderate clustering and a distinctive modular structure.

```
In [1]: from NetworKit import *

In [2]: G = readGraph("input/PGPgiantcompo.graph")

In [3]: properties.overview(G)

Network Properties: PGPgiantcompo
==================
Basic Properties
-------------------------  --------------
nodes (n)                  10680
edges (m)                  24316
density                    0.000426
isolated nodes             0
self-loops                 0
min. degree                1
max. degree                205
avg. degree                4.553558
degree power law fit?      True, 2.101144
degree power law exponent  1.6997
degree assortativity       0.2382
-------------------------  --------------
Path Structure
-------------------------  --------
connected components       1
size of largest component  10680
estimated diameter range   (22, 24)
-------------------------  --------
Community Structure
----------------------------------------  -----------  --------
approx. avg. local clustering coefficient               0.437505
PLP community detection
                                          communities  845
                                          modularity   0.780827
PLM community detection
                                          communities  98
                                          modularity   0.883877
----------------------------------------  -----------  --------
```

Comparison of networks is now straightforward. Consider for example `soc- LiveJournal`, a network extracted from the LiveJournal webpage, which combines online social networking and blogging – at 43 million edges a data set that we consider medium-sized [5]. The network consists of many components, but clearly one connected component is dominant. Clustering and community structure seem somewhat less pronounced, but still present. There is a clear power-law degree distribution, but practically no degree assortativity. Creating this overview takes less than two minutes.

```
Network Properties: soc-LiveJournal
==================
Basic Properties
-------------------------  ---------------
nodes (n)                  4847571
edges (m)                  43369619
density                    0.000004
isolated nodes             0
self-loops                 518382
min. degree                1
max. degree                20334
avg. degree                17.786404
degree power law fit?      True, 11.284630
degree power law exponent  1.3853
degree assortativity       0.0217
-------------------------  ---------------
Path Structure
-------------------------  -------
connected components       1876
size of largest component  4843953
estimated diameter range   None
-------------------------  -------
Community Structure
---------------------------------------  -----------  --------
approx. avg. local clustering coefficient              0.366108
PLP community detection
                                         communities  29931
                                         modularity   0.547508
PLM community detection
                                         communities  4704
                                         modularity   0.768956
---------------------------------------  -----------  --------
```

Using Generative Models. Much early work in network science has focused on degree distributions of complex networks as well as generative models to reproduce them. In this workflow we demonstrate one of the available graph generators, and also show that the degree distribution yields only a very partial characterization of a network. Given the degree sequence of any network, the Havel-Hakimi generator produces a synthetic graph which realizes this degree sequence exactly. In this case, we input the degree sequence of `soc-LiveJournal`.

As expected, properties of the degree distribution are replicated closely. However, the generator also introduces degree assortativity, manifolds the number of connected components and communities, and increases the clustering coefficient. In general, the replica is a very different network compared to the original.

```
In [10]: %time H = generators.HavelHakimiGenerator([G.degree(v) for v in G.nodes()]).generate()
Wall time: 24.8 s

In [11]: properties.overview(H)
Network Properties: G#10
=================
Basic Properties
-------------------------  ---------------
nodes (n)                  4847571
edges (m)                  43110428
density                    0.000004
isolated nodes             0
self-loops                 0
min. degree                1
max. degree                20334
avg. degree                17.786404
degree power law fit?      True, 11.284630
degree power law exponent  1.3853
degree assortativity       0.2014
-------------------------  ---------------
Path Structure
-------------------------  -------
connected components       699256
size of largest component  3259587
estimated diameter range   None
-------------------------  -------
Community Structure
---------------------------------------  -----------  --------
approx. avg. local clustering coefficient               0.650352
PLP community detection
                                         communities  312047
                                         modularity   0.823445
PLM community detection
                                         communities  699988
                                         modularity   0.983613
---------------------------------------  -----------  --------
```

A Web Scale Network. So far, these analyses did not pose a challenge in terms of processing time or memory. In fact, an analysis of the `soc-LiveJournal` network is practical with NetworkX (see Table 2 for a running time comparison). In the following, we test the scalability of our software with an analysis of the large web graph `uk-2007-05`, a hyperlink graph of the `.uk` domain consisting of more than 100 million nodes and 3.3 billion edges.

	NetworkX	NetworKit	Speedup factor
connected components	42s	5s	8.4
avg. local clustering coefficient	39min 30s	15 min 47 s	2.5
degree assortativity	3min 36s	650 ms	332.3
core decomposition	3min 55s	26.8 s	8.8

Table 2: Running time comparison on `soc-LiveJournal`.

One of the most time-consuming tasks is reading the graph from file. After this is done, we can quickly determine the size of the graph and minimum, maximum and average node degree. For a graph of this size, we were so far not able to analyze the degree distribution using the third-party `powerlaw` module because the implementation exhausts our available main memory of 256 GB.

```
In [1]: %time G = readGraph("uk-2007-05.metis.graph")
Wall time: 14min 4s

In [2]: properties.size(G)
Out[2]: (105896555, 3301876564)

In [3]: %time properties.degrees(G)
CPU times: user 1.35 s, sys: 2 ms, total: 1.35 s
Wall time: 397 ms
Out[3]: (0, 975419, 62.3604151050995)
```

When running NetworKit from the IPython Notebook, matplotlib integration makes it very easy to produce publication-quality figures within the same session, and without a detour through file I/O. Here, plotting the degree distribution of the network with logarithmic axes shows some similarity to the characteristic power law pattern, but with some anomaly for low-degree nodes.

```
In[4]:  dd = properties.degreeDistribution(G)
        xscale("log")
        xlabel("degree")
        yscale("log")
        ylabel("number of nodes")
        plot(dd)
```

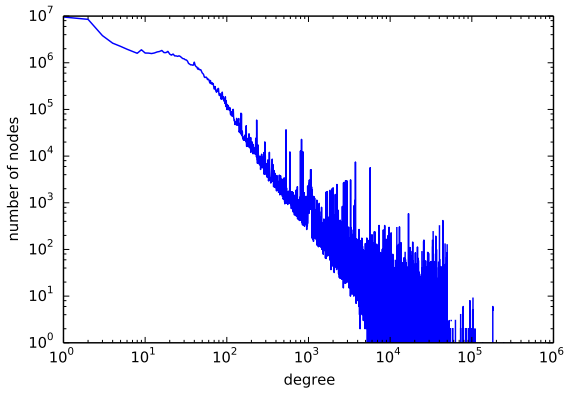

Figure 3: Degree distribution of uk-2007-05

The following function determines the number and size distribution of connected components. The network is not connected, and there are over 700 000 components.

```
In [5]: %time cc = properties.components(G)
Wall time: 2min 45s

In [6]: cc[0]
Out[6]: 756936
```

We now want to find out if this web graph is strongly clustered. While exact calculation of clustering coefficients would be very tedious, we can get a good estimate in a fraction of a second:

```
In [7]: %time properties.clustering(G)
INFO:root:taking 23026 samples
Wall time: 300 ms
Out[7]: 0.7375575436463129
```

Next, we perform community detection using the PLP and PLM algorithm. Modularity and number of communities indicate whether the network has a distinctive community structure. There is a qualitative difference in the results of both algorithms – which one is more appropriate depends very much on the research question asked and requires domain-specific evaluation.

```
In [6]: %time community.detectCommunities(G, community.PLP())
PLP(updateThreshold=1058) detected communities in 50.325546741485596 [s]
solution properties:
-------------------  -------------
# communities        906041
min community size         1
max community size    354494
avg. community size      116.878
modularity                 0.970867
-------------------  -------------
Wall time: 1min 26s
Out[6]: <_NetworKit.Partition at 0x7f2620709030>

In [8]: %time community.detectCommunities(G, community.PLM())
PLM(balanced,refinement) detected communities in 399.4341833591461 [s]
solution properties:
-------------------  -------------
# communities        762739
min community size         1
max community size    822890
avg. community size      138.837
modularity                 0.996288
-------------------  -------------
Wall time: 7min 13s
Out[8]: <_NetworKit.Partition at 0x7f2620709a80>
```

For later analysis, the communities can also be saved to disk as a partition file in which line i contains the partition index of node i.

```
In [17]: community.writeCommunities(Out[8], "uk2007-plm.ptn")
```

Calculating the degree assortativity coefficient can also be done in a short amount of time, showing that degrees are correlated slightly negatively.

```
In [10]: %time properties.degreeAssortativity(G)
Wall time: 26.4 s
Out[10]: -0.10445556727737432
```

6 Comparison to Related Software

In the following, we are going to compare NetworKit to software packages with 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 4 shows a comparison of the feature sets. Table 3 compares non-functional aspects.

6.1 Description of Related Software

	NetworKit	NetworkX	KDT	JUNG	GraphCT	SNAP	STINGER	Pajek	Gephi	igraph	graph-tool
language	C++, Python	Python	C++, Python	Java	C, Java, Python, Ruby, R	C	C	Closed source	Java	C, Python, Ruby, R	C++, Python
interface	object-oriented, functional	object-oriented, functional	object-oriented	object-oriented	procedural	procedural	procedural	GUI	GUI, object-oriented	procedural	object-oriented
platform	cross-platform	cross-platform	cross-platform	cross-platform (JRE)	x86 / CrayXMT	cross-platform	x86 / CrayXMT	x86 - Windows	cross-platform (JRE)	cross-platform	cross-platform
parallelism	shared memory (OpenMP, TBB)	-	distributed memory (MPI)	-	shared memory (Cray XMT)	shared memory (OpenMP)	shared memory (OpenMP / Cray XMT)	-	shared memory (JVM)	-	shared memory (OpenMP)
license	MIT	BSD	BSD	BSD	BSD	GPL	BSD	proprietary	CDDL 1.0, GPL v3	GPL	GPL v3
first release	1.0 (Mar 2013)	0.22 (Jun 2005)	0.1 (Mar 2011)	Prefuse Alpha (May 2005)	0.3.3 (Jul 2009)	0.1 (Feb 2008)	2008	0.01 (Jan 1997)	0.6 alpha 1 (Jul 2008)	0.1 (Jan 2006)	1.1.0 (May 2007)
latest release	2.1 (Feb 2014)	1.8.1 (Aug 2013)	0.3 (Apr 2013)	2.0.1 (Jan 2010)	0.8.0 (Sep 2012)	0.4 (Aug 2010)	r633 (Aug 2013)	3.15 (Mar 2014)	0.8.2 beta (Jan 2013)	0.6.5 (Mar 2013)	2.2..31 (Mar 2014)
web	http://parco.iti.kit.edu/software/networkit.shtml	http://networkx.github.io/	http://kdt.sourceforge.net/	http://jung.sourceforge.net/	https://trac.research.cc.gatech.edu/graphs/wiki/GraphCT	http://snap.stanford.edu/snap/index.html	http://www.stingergraph.com/	http://pajek.imfm.si	https://gephi.org/	http://igraph.sourceforge.net/	http://graph-tool.skewed.de/

Table 3: Comparison of network analysis software

NetworkX. NetworkX [28] is a feature-rich Python package for network analysis whose development started in 2002. NetworkX 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, and it is also currently the basis for our graph drawing capabilities. NetworkX has a large feature set and provides a highly flexible graph data structure, but its applicability to large graphs is limited.

igraph. igraph [20] 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 (all use the same C code base).

graph-tool. graph-tool is a Python library for manipulation and analysis of networks. In order to achieve high performance, the core of graph-tool is written in C++, utilizing the Boost Graph Library, and uses OpenMP for parallelization. The module has a rich collection of features for analysis and visualization of networks.

Gephi. Gephi [8] is a vizualisation and analysis package. It has a rich graphical user interface that allows to use the package without coding, as well as advanced visualization capabilities, analytics and plotting functionality, as well as "data laboratory" view. This makes Gephi a popular popular choice for data exploration. Gephi Toolkit makes the functionality accessible as a Java library.

Pajek. Pajek is a program for analyzing and visualizing large networks. A closed sourced stand-alone tool, it is available for free for noncommercial use. Currently, Pajek only runs on Microsoft Windows operating system. It was first released in 1997 and is still occasionally updated. A special edition – Pajek XXL aims to allow analysis of larger networks. The XXL version stores only a part of the network in memory. Some of the Pajek functionality is not available in the XXL version.

JUNG. JUNG [37] (Java Universal Network/Graph Framework) is a Java-based library for modeling, analysis, and visualizing data that can be represented as a network. In addition to static networks, JUNG has support for dynamic (evolving) networks, which are unsupported by the majority of popular libraries. The latest version of the library – JUNG 2.0.1 was released in 2010. Thus, JUNG lacks some of the modern algorithms and features.

GraphLab. The GraphLab project [32] 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.

KDT. The Knowledge Discovery Toolbox (KDT) [33] 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. Version 0.3 of the package appeared in May 2013. Due to its early stage of development, it currently has a limited feature set.

GraphCT. GraphCT (Graph Characterization Toolkit) [21] 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.

SNAP. SNAP (Small-world Network Analysis and Partitioning) [6] is a C library using OpenMP primitives for parallelization. SNAP's feature set includes several fundamental graph algorithms as well as network analysis queries, in particular betweenness centrality and community detection. Interactive workflows are not supported, but SNAP's design is meant to be extensible. The last release dates from August 2010.

STINGER. STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation) [22] has started as a dynamic graph structure and has been extended over time by various 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 focusses on fast execution. Recently, productivity-oriented interfaces to Python and Java for loading data into the graph, querying the graph, and analytics methods have been added as well.

6.2 NetworKit in Comparison

Clearly, network analysis tools vary widely in terms of target platform, user interface, scalability and feature set. We therefore locate NetworKit relative to these related efforts. As frequent users of NetworkX, we consider the package a model of feature completeness and usability. However, processing billion-edge networks in NetworkX is out of reach. Due to the similar interface, users of NetworkX are likely to move easily to NetworKit for larger networks. Related efforts such as KDT, SNAP, GraphCT and STINGER offer high performance through native implementations. However, to characterize a complex network in practice we need a substantial set of analysis kernels which not all of these frameworks currently provide. The hybrid approach of native kernels and a high-level language interface is also followed by igraph and graph-tool, which are closest to NetworKit in scope, performance and usability. While a systematic experimental comparison is not within

Feature Group	Feature	Sub-Feature	NetworKit	NetworkX	Gephi	JUNG	Pajek	KDT	STINGER	GraphCT	SNAP	igraph	GraphLab	graph-tool
General	API		●	●	●	●	●	●	●	●	●	●	●	●
	interactive		●	●	●	○	●	●	○	○	○	●	○	●
Data Structures	graph (undirected/directed/ weighted)		●○●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●
	hypergraph		○	○	○	●	○	●	○	○	○	○	○	○
Algorithms	connected components		●	●	●	●	●	●	●	●	●	●	●	●
	search (BFS/DFS)		●●	●●	●●	●○	●●	●○	●○	○○	●●	●●	○○	●●
	weighted shortest path		●	●	●	●	●	●	○	○	●	●	○○	●
	degree distribution		●	●	●	●	●	●	●	●	●	●	○○	●
		power law estimation	●	○	●	○	○	○	○	○	○	●	○	○
	degree assortativity		●	●	○	○	○	○	○	○	○	●	○	●
	core decomposition		●	●	●	○	●	○	●	●	●	●	●	●
	diameter (ex./ap.)		●●	●○	●○	●○	●○	○○	●○	○●	●●	●○	○●	○●
	centrality	betweenness (ex./ap.)	●●	●○	●○	●○	●○	●●	●○	●○	●○	●●	○○	●○
		closeness (ex./ap.)	○○	●○	●○	●○	●○	○○	○○	○○	●○	●●	○○	●○
		eigenvector/page rank	●	●	●	●	●	●	●	○	●	●	●	●
	clustering coefficients (ex./ap.)		●●	●	●	●	●○	○	●	●	●○	●○	●○	●○
	community detection		●	○	●	●	●	○	●	○	●	●	○	●
		modularity	●	○	●	○	○	○	●	●	●	●	○	●
	matching		●	●	○	○	●	○	○	○	○	●	○	○
	independent set		○	●	○	○	○	●	○	○	○	●	○	●
	flows		○	●	●	●	●	○	○	○	○	●	○	●
Generators	Erdös-Renyi		●	●	●	●	●	○	○	○	●	●	○	●
	Barabasi-Albert		●	●	●	●	●	○	○	○	●	●	○	●
	R-MAT		●	○	○	○	○	●	●	●	●	○	○	○
	Watts-Strogatz		○	●	●	○	○	○	○	○	●	●	○	○
	Chung-Lu		●	●	○	○	○	○	○	○	●	○	○	○
	Havel-Hakimi		●	●	○	○	○	○	○	○	○	○	○	○
Drawing			●	●	●	●	●	○	○	○	●	○	○	●

Table 4: Feature matrix for NetworKit and related network analysis software

network (m)	soc-LiveJournal ($43M$)			uk-2002 ($261M$)			uk-2007-05 ($3.3G$)		
	NX	*GT*	*NK*	*NX*	*GT*	*NK*	*NX*	*GT*	*NK*
connected components	42s	4.4s	2.2s	⋈	14s	7.9s	⋈	†	45 s
avg. local clustering coeff.	39min 30s	1min 16s	1min 27s	⋈	13min 9s	7min 44s	⋈	†	⋈
" approx.	○	○	0.1s	○	○	0.1s	○	○	0.2s
degree assortativity	3min 36s	1.88s	0.7s	⋈	3.5s	2.1s	⋈	†	25 s
k-core decomposition	3min 55s	6.9s	29s	⋈	23s	2min 38s	⋈	†	⋈
betweenness approx. ± 0.1	○	○	5min 24s	○	○	11min 49s	○	○	†

Table 5: Running times for some analysis kernels for NewtworkX, graph-tool and NetworKit. (Legend: ○: feature missing, ⋈: takes > 2h, †: runs out of memory)

the scope of this paper, Table 5 shows some example running times in comparison to NetworkX and graph-tool. graph-tool delivers similar high performance for standard kernels, but NetworKit's memory efficiency and approximation features make even a graph like `uk-2007-05` tractable. Through further development, NetworKit will continue to set itself apart with a feature set geared towards large-scale network analysis and including novel algorithmic approaches (e. g., to community detection). 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.

7 Miscellaneous

Input/Output. As the primary file format for graphs, we support the format introduced by the graph partitioning software Chaco and further popularized by METIS, which stores graphs with optional weights in a simple text-based adjacency file. The collection of supported formats will be extended as needed.

Visualization. NetworKit includes basic graph drawing capabilities, implemented on the basis of matplotlib and NetworkX. A major overhaul is due in a future release. A small example is shown below, a force-directed drawing of the "Les Miserables" character coappearance network, in which the node sizes are proportional to betweenness centrality.

```
In [1]: L = readGraph("input/lesmis.graph")
In [2]: bc = centrality.Betweenness(L, True)
        bc.run()
In [3]: viztasks.drawGraph(L, nodeSizes=[b * 1000 for b in bc.scores()])
```

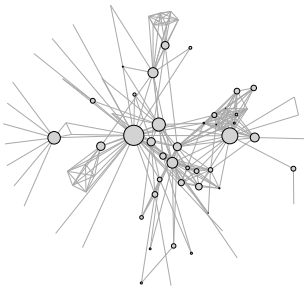

Figure 4: "Les Miserables" character coappearance network

7.1 Infrastructure

NetworKit is free software licensed under the permissive MIT License. As argued by Stodden [49], we chose this 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 source code is distributed via a public Mercurial repository at http://algohub.iti.kit.edu/parco/NetworKit/NetworKit. Write access for contributions is granted on request. The included documentation provides support for setting up and getting started with NetworKit. General discussion (updates, support, feature requests etc.) takes place on the open e-mail list `networkit@ira.uni-karlsruhe.de`.

7.2 History and Roadmap

NetworKit started as a collection of static community detection algorithms written in C++, first released in March 2013. With version 2.0 the Python interface was introduced in November 2013. Version 3.0, released in March 2014, experienced several bug fixes and changes under the hood. New features include fundamental graph and network analysis algorithms as well as network generators. Version 3.1 of April 2014 extends the feature set further. The package is under constant development while we aim for several releases per year. Features of upcoming releases are likely to focus on multiple areas: Algorithms for layouting and visualizing large complex networks; tools for the analysis of dynamic graphs; algorithms for additional community detection scenarios such as selective, dynamic, and overlapping; better support for case studies by improved handling of attributes as well as algorithms that incorporate graph semantics. In another future release we want to support directed graphs as well.

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 Miriam Beddig, Stefan Bertsch, Andreas Bilke, Guido Brückner, Patrick Flick, Daniel Hoske, Yassine Marrakchi, Florian Weber, Michael Wegner and Jörg Weisbarth for contributing code to NetworKit.

References

[1] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 171–180. Acm, 2000.

[2] R. Albert and A.-L. Barabasi. Statistical mechanics of complex networks. June 2001.

[3] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: a python package for analysis of heavy-tailed distributions. *PLoS ONE*, 9(1):e85777, 2014.

[4] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, R. Lent, S. Herculano-Houzel, et al. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513(5):532–541, 2009.

[5] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[6] D. A. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. *Parallel and Distributed Processing Symposium, International*, 0:1–12, 2008.

[7] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. 286:509–512, 1999.

[8] M. Bastian, S. Heymann, and M. Jacomy. Gephi: an open source software for exploring and manipulating networks. In *International Conference on Weblogs and Social Media*, pages 361–362, 2009.

[9] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[10] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[12] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[13] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006.

[14] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

[15] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE Trans. Knowledge and Data Engineering*, 20(2):172–188, 2008.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. *Computer Science Department*, page 541, 2004.

[17] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[18] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[19] L. d. F. Costa, O. N. Oliveira Jr, G. Travieso, F. A. Rodrigues, P. R. Villas Boas, L. Antiqueira, M. P. Viana, and L. E. Correa Rocha. Analyzing and modeling real-world phenomena with complex networks: a survey of applications. *Advances in Physics*, 60(3):329–412, 2011.

[20] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5), 2006.

[21] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader. Graphct: Multithreaded algorithms for massive graph analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11):2220–2229, 2013.

[22] D. Ediger, R. McColl, J. Riedy, and D. Bader. Stinger: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5, Sept 2012.

[23] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th ACM/IEEE Design Automation Conf.*, pages 175–181, Las Vegas, NV, June 1982.

[24] J. H. Fowler, N. A. Christakis, Steptoe, and D. Roux. Dynamic spread of happiness in a large social network: longitudinal analysis of the framingham heart study social network. *BMJ: British medical journal*, pages 23–27, 2009.

[25] U. Gargi, W. Lu, V. S. Mirrokni, and S. Yoon. Large-scale community detection on youtube for topic discovery and exploration. In *International Conference on Weblogs and Social Media*, 2011.

[26] J. Gehweiler and H. Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proc. 7th High-Performance Grid Computing Workshop (HGCW'10) in conjunction with 24th Intl. Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society, 2010.

[27] M. Girvan and M. Newman. Community structure in social and biological networks. *Proc. of the National Academy of Sciences*, 99(12):7821, 2002.

[28] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2008.

[29] S. L. Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph. i. *Journal of the Society for Industrial & Applied Mathematics*, 10(3):496–506, 1962.

[30] P. F. Jonsson, T. Cavanna, D. Zicha, and P. A. Bates. Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis. *BMC Bioinformatics*, 7:2, 2006.

[31] A. Klaus, S. Yu, and D. Plenz. Statistical analyses support power law distributions found in neuronal avalanches. *PloS one*, 6(5):e19779, 2011.

[32] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[33] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proceedings of the Twelfth SIAM International Conference on Data Mining (SDM12)*, pages 930–941, April 2012.

[34] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics (JEA)*, 13:10, 2009.

[35] M. Newman. *Networks: an introduction*. Oxford University Press, 2010.

[36] M. E. J. Newman. Assortative mixing in networks. *Phys. Rev. Lett.*, 89:208701, Oct 2002.

[37] J. O'Madadhain, D. Fisher, S. White, and Y. Boey. The JUNG (java universal network/graph) framework. *University of California, Irvine, California*, 2003.

[38] A. R. P. Erdős. On the Evolution of Random Graphs. *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[40] F. Perez, B. E. Granger, and C. Obispo. An open source framework for interactive, collaborative and reproducible scientific computing and education, 2013.

[41] A. Pinar, C. Seshadhri, and T. G. Kolda. The Similarity between Stochastic Kronecker and Chung-Lu Graph Models. Oct. 2011.

[42] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[43] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proceedings of the 7th ACM Conference on Web Search and Data Mining, WSDM*, volume 14, 2013.

[44] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proceedings of the 19th international conference on World wide web - WWW '10*, page 861, New York, New York, USA, Apr. 2010. ACM Press.

[45] T. Schank and D. Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[46] M. A. Serrano, M. Boguñá, R. Pastor-Satorras, and A. Vespignani. Correlations in complex networks. *Large scale structure and dynamics of complex networks: From information technology to finance and natural sciences*, pages 35–66, 2007.

[47] C. L. Staudt and H. Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. *arXiv preprint arXiv:1304.4453*, 2013.

[48] C. L. Staudt and H. Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. In *proceedings of the 2013 International Conference on Parallel Processing*. Conference Publishing Services (CPS), 2013.

[49] V. Stodden. Enabling reproducible research: Licensing for scientific innovation. *Int'l J. Comm. L. & Pol'y*, 13:1, 2009.