Algorithmic analysis of large networks by computing structural indices

Algorithmische Analyse grosser Netzwerke durch die Berechnung von Strukturindizes

Diplomarbeit
von
Christoph Gulden
Fachbereich Mathematik und Statistik
Universität Konstanz
78457 Konstanz, Germany

März 2004

Zusammenfassung

Netzwerke werden verwendet, um Beziehungen zwischen Aktoren darzustellen. Netzwerke können analysiert werden, indem man Indizes, d.h. numerische Werte, für die einzelnen Aktoren und Verbindungen berechnet, welche lediglich von der Netzwerkstruktur abhängen. Es wird ein System von grundlegenden Algorithmen entworfen, um entsprechende Berechnungen kompakt darzustellen und gleichzeitig deren effiziente Umsetzung in einer objektorientierten Umgebung zu erleichtern. Diese Basisalgorithmen werden verfeinert, um Indizes zu berechnen die sowohl lokale als auch globale Eigenschaften von Aktoren widerspiegeln. Um die berechneten Indizes auszuwerten, werden Verfahren aus der diskreten Statistik vorgestellt. Da die Algorithmen für die Berechnung von Abstands-basierten Indizes für grosse Netzwerke zu aufwendig sind, stellen wir Möglichkeiten vor, diese Berechnung zu beschleunigen oder zu approximieren.

Contents

Chapter 1

Introduction

Networks appear in many different situations and express relationships by linking related actors. To illustrate the importance of this subject we start with the presentation of some applications where information can be modeled by networks.

Example 1.1 (Social structures)
Networks are used in social sciences to express relationships between persons on different levels. We can get social networks by using people as actors and link them according to their personal contacts or dependencies. This representation is used to understand the behavior of individuals and groups as well as a whole society.
A well know property for social networks is known as the "Six degrees of separation". This means that all people can be connected by only one short chain of personal contacts of intermediate individuals.

Example 1.2 (Infrastructural networks)
Infrastructural networks for the transportation of goods or the information exchange are essential for industrial societies. We can model them by using good repositories or information devices as actors and link them according to the distributed elements. The dependency on data network technologies and their common use in everyday life rise questions about the reliability of telecommunication, power supply or traffic routes against errors or attacks. This information can be used to coordinate the efficient assignment of resources to get enough bandwidth and redundancy.

Example 1.3 (Data-mining networks)
Data mining delivers a large amount of data that can also be viewed as networks. For example we can get networks from the sale accounting of big stores by using products as actors and link them in case they are sold together. The consumption behavior can be analyzed in order to group similar products together, to keep the product range small and to estimate coming customer trends.

Example 1.4 (Collaboration networks)
Collaboration networks are similar to social networks, because we use persons as actors and link them in case they work together. Now we can rank individuals by aspects like the number of interactions or the interaction with other important actors.
The Erdös number project [31] is a collaboration network that links persons when they write a scientifical publication together. Each scientist gets a number according to the shortest number of intermediate co-authors to Paul Erdös[1].

Example 1.5 (Citation networks)
Citation networks are used to retrieve information by linking documents according to bibliographic citations. The results of searching full-text indices of document contents can be improved by analyzing the citation structure to find works of high relevance for a set of keywords. An important application for analyzing citations are search engines for the World Wide Web.

Example 1.6 (Ecological networks)
We can get an ecological network by using species of the ecological system as actors and link two actors by the relation "species A eats species B". The knowledge of the importance of a species in a complex food chain can change our point of view to the overall context and helps us to understand the dependencies of nature conservation tasks. Such information can be used to enable balanced protective activities.

Example 1.7 (Biological networks)
Metabolic or protein interaction systems can be analyzed to find important substances in an organism and to understand their role. Actors of a metabolic network are the substance of a metabolic system. Two substances are linked, if the metabolic process transforms one substance into the other. Using this information we can develop efficient diagnosis methods for diseases and sufficient therapies.

To understand the properties of networks there are many approaches to analyze different aspects. In the beginning of network analysis scientists dealed with rather small examples and the detection of basic network topologies were the first step undertaken in this respect.
To avoid the necessity of domain specific attributes we can use so-called structural indices which assign the network objects with numerical values that depend only on the raw network structure.
While the first networks were small data sets collected by hand, the introduction of personal computers made it possible for many people to generate data sets more easily. Modern integrated information systems became more

[1]Paul Erdös (1913-1996) was an important mathematician with more than 1,500 published papers.

and more available in the last years and they provide data for networks of large size. Therefore there is a strong interest in algorithmic tools that are efficient as well as flexible.

This paper is structured as follows. In Chapter 2 we give definitions for networks and structural indices to speak about them in a precise way. Additionally we present possibilities to store networks together with index information in-memory as well as persistently. Chapter 3 describes an algorithmic framework to decouple recurring computational tasks. Then we use this framework to describe the computation of popular network indices in Chapter 4. To outline the properties of indices also for large networks we show in Chapter 5 how to describe, visualize and compare them statistically. We demonstrate in Chapter 6 that we can speed-up the computation of distance-based indices, if the network has special properties. To get index values for networks that are too large to get the correct values in an acceptable time Chapter 7 describes an approximation schema for indices that we get by summing up partial results. Finally we conclude with an outlook in Chapter 8.

Chapter 2

Network representation

To represent networks we define them in terms of graph theory. Then we explain the meaning of structural indices and look at a suitable format to store the index information together with the network permanently.

2.1 Network definitions

Graphs are a mathematical model to store relationships by vertices and edges. Remembering the examples from Section 1 we can see that graphs fit perfectly for the representation of networks.

Definition 2.1 (Directed graph)
A *directed graph* $\overrightarrow{G} = (V, \overrightarrow{E})$ is a pair consisting of a vertex set V and a set of directed edges

$$\overrightarrow{E} \subseteq V \times V =: \overrightarrow{E}^*$$

where $\overrightarrow{E}^*$ is the set of all possible directed edges. An edge $e = (v, v) \in \overrightarrow{E}$ is called a loop.

Often it is necessary to have more than one edge between two vertices. We can formalize this by the usage of multi sets.

Definition 2.2 (Multi-set)
A *multi-set* $(X, \#_X)$ is a pair consisting of a set X together with a *multiplicity* of its elements

$$\#_X : X \longrightarrow \mathbb{N}_0.$$

If the context is clear, we simply write X for a multi set $(X, \#_X)$ and define for multi-sets X and Y:

- $x \in_k X :\Leftrightarrow \#(x) := \#_X(x) = k$,
- $x \in X :\Leftrightarrow \#(x) > 0$,
- $x \notin X :\Leftrightarrow \#(x) = 0$,
- $|X| := \sum_{x \in X} \#(x)$ for the size of X and
- $X \subseteq Y$, if for all $x \in X$ the condition

$$x \in_k X \Rightarrow x \in_l Y, \text{ with } k \leq l \text{ holds.}$$

Definition 2.3 (Network)
We define a *(directed) network* to be a graph

$$\overrightarrow{G} = (V, (\overrightarrow{E}^*, \#_{\overrightarrow{E}^*})).$$

We denote

- the number of vertices by $n := |V|$,
- the number of edges by $m := |E|$ and
- the size of the network by $|G| := |V| + |E|$.

Remark 2.1
We write $G = (V, E)$ or simply G for a network $\overrightarrow{G} = (V, \#_{\overrightarrow{E}^*})$ and assume a network to be always directed and finite[1], if not explicitly defined otherwise. To handle the undirected case we can also use directed networks by ignoring the edge direction and omitting loops.

Example 2.1 (Demonstration network)
In Figure 2.1 we have a network with $n = 11$ vertices and $m = 15$ edges. There is a loop $e_8 = (v_4, v_4)$ and an edge with multiplicity 2 from v_6 to v_0. We will use this network to illustrate the structural indices defined later.

Before we can start to analyze networks, we have to define data structures to represent networks in-memory. We get the following representation directly from the definition.

Definition 2.4 (Simple lists representation)
Let $G = (V, E)$ be a network. We define the *simple lists representation* for G by a list for the set of vertices V together with a list for set of edges E. The multiplicity of an edge (v, w) is expressed by multiple edge instances.

[1] A network $G = (V, E)$ is finite, if $|G| < \infty$ holds.

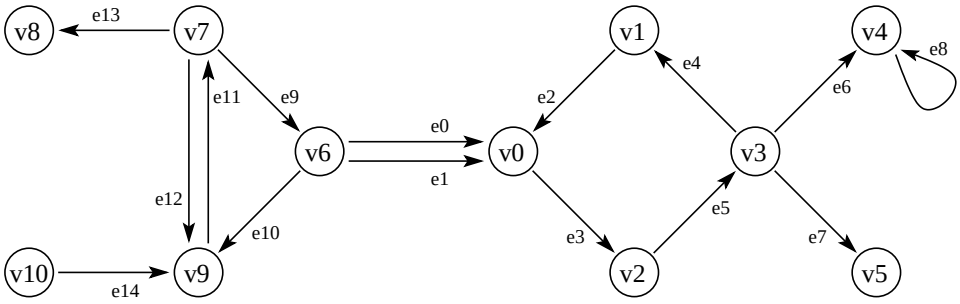

Figure 2.1: Demonstration network.

We can realize the simple lists representation with memory in $\mathcal{O}(|V| + |E|)$ and each serious object-oriented data structure library provides an efficient list data type. On the other hand we have to search in the worst case a whole list to find a given vertex or edge. We avoid this drawback by defining an additional data structure.

Definition 2.5 (Incidence)
Let G be a network. For an edge $e = (v, w) \in E$ we say e

- is incident to v and w,
- has the source v (i.e. $v = source(e)$),
- has the target w (i.e. $w = target(e)$),
- is an outgoing edge of v,
- and an incoming edge of w.

We can improve the simple list representation by storing for each $v \in V$ the sets of outgoing and incoming edges.

Definition 2.6 (Incidence lists representation)
For a network $G = (V, E)$ we define the associated *incidence lists representation* by the simple lists representation with the additional mappings

$$\overrightarrow{I}_G : V \longrightarrow \mathcal{P}(E) \text{ and } \overleftarrow{I}_G : V \longrightarrow \mathcal{P}(E)$$

which return for each vertex $v \in V$ a list with the outgoing resp. incoming edges.

Such mappings can be realized by using a vector of lists indexed by an internal numbering of the vertices with memory also in $\mathcal{O}(|V| + |E|)$. Now we are able to find all incident edges for a vertex without searching the whole set of edges.

We provide some more definitions to make it easier to speak about networks.

Definition 2.7 (Degree)
We define for each vertex v of a network G

- the *outgoing degree* $\overrightarrow{deg}_G(v) := \#\overrightarrow{I}_G(v)$,
- the *incoming degree* $\overleftarrow{deg}_G(v) := \#\overleftarrow{I}_G(v)$ and
- the *degree* $\overline{deg}_G(v) := \overrightarrow{deg}_G(v) + \overleftarrow{deg}_G(v)$.

Definition 2.8 (Path)
Let $s, t \in V$. A *directed path of length* k from s to t is an alternating sequence of vertices and edges

$$\overrightarrow{P}_k := (s, e_1 = (s, v_1), v_1, e_2 = (v_1, v_2), \ldots, v_{k-1}, e_k = (v_{k-1}, t), t)\,.$$

An *undirected path* $\overline{P}_k$ of length k from s to t is an alternating sequence, where we do not have to respect the direction of the edges.

Definition 2.9 (Sub-network)
A network $G = (V, E)$ contains a network $G' = (V', E')$, if

$$V' \subseteq V \text{ and } E' \subseteq (E \cap (V' \times V'))$$

holds. We call G' a *sub-network* of G and write $G' \subseteq G$.

Definition 2.10 (Weak connectivity)
A network G is *(weakly) connected*, if for every pair or vertices $s, t \in V$ an undirected path from s to t exists. Otherwise we call the network *disconnected*.
A (weakly) connected component is an inclusion-maximal weakly connected sub-network.

2.2 Structural indices

Our goal is the computation of real numbers for vertices and edges, i.e. mappings $i : V \longrightarrow \mathbb{R}$ resp. $i : E \longrightarrow \mathbb{R}$ that depend only on the structure of the network. We can formalize this index property by introducing *structural isomorphisms* of networks.

Definition 2.11 (Structural isomorphism)
Two networks $G = (V, E)$ and $G' = (V', E')$ are called *isomorphic*, $G \cong G'$, if a bijection $\alpha : V \longleftrightarrow V'$ exists with

$$(v, w) \in_k E \Longleftrightarrow \alpha(v, w) := (\alpha(v), \alpha(w)) \in_k E'.$$

Then we call α a *structural network isomorphism* and write $\alpha(G) = G'$. We call α an *automorphism on* G, if $V = V'$.

Definition 2.12 (Structural index)
Let $\mathcal{G}$ be a class of non-isomorphic networks closed against the generation of connected components and $\mathbb{R}^*_{\geq 0}$ the set of vectors with nonnegative real numbers. A function $i : \mathcal{G} \longrightarrow \mathbb{R}^*_{\geq 0}$ is called a structural vertex (edge) index on $\mathcal{G}$, if

- i is a vertex (edge) index, i.e. for all networks $G = (V, E) \in \mathcal{G}$

$$i(G) \in \mathbb{R}^V_{\geq 0} \quad \text{resp.} \quad i(G) \in \mathbb{R}^E_{\geq 0}.$$

- i depends only on the network structure, i.e. for all networks $G = (V, E) \in \mathcal{G}$ and automorphism α on G holds

$$\begin{aligned} i(G)(v) &= i(\alpha(G))(\alpha(v)) \text{ for all } v \in V \\ \text{resp.} \quad i(G)(e) &= i(\alpha(G))(\alpha(e)) \text{ for all } e \in E. \end{aligned}$$

- i is consistent, i.e. for all $G = (V, E) \in \mathcal{G}$ and connected components $K = (V_K, E_K) \subset G$ holds

$$\begin{aligned} i(G)(v) \cdot i(K)(w) &= i(K)(v) \cdot i(G)(w) \text{ for all } v, w \in V_K \\ \text{resp.} \quad i(G)(e) \cdot i(K)(f) &= i(K)(e) \cdot i(G)(f) \text{ for all } e, f \in E_K. \end{aligned}$$

We can represent network indices either by storing their values together with the network objects they belong to or we can use an internal numbering to store them separately in an array. The second possibility has the advantage that we do not have to alter the representation for networks. We will refer to the array form by the terms *vertex vector* for $X^{V,G}$ and *edge vector* for $X^{E,G}$, while we mark the direction case by an arrow or line above X.

Example 2.2
From Definition 2.7 we get three structural indices on the class of all networks, for example the outgoing degree index for vertices

$$i_{\overrightarrow{\deg}} : \mathcal{G} \longrightarrow \mathbb{R}^*_{\geq 0}, \quad G \mapsto \overrightarrow{X}^{V,G}_{\text{Deg}}$$

defined by

$$\overrightarrow{X}^{V,G}_{\text{Deg}}(v) := \overrightarrow{deg}_G(v).$$

Figure 2.2 shows the outgoing degree index for the exemplified network. For that purpose we scaled the size of the vertices to illustrate the corresponding index value.

2.3 GraphML

Over time many concurrent formats were defined to store raw network information like vertices and edges as well as additional information like indices persistently. Each of these formats has its own strength and drawbacks.
With the development of XML [32] a general format for data related to graphs named GraphML [29] appeared. Intended for the flexible storage of graph data structures it perfectly meets the requirements of network analysis, because we can store network information and index information together in one file.
Each XML file consists of two parts:

- the preamble, which contains information about the used XML dialect, and
- the actual data, which is marked up by enclosing semantic tags.

XML dialects can be read easily by humans and there are many code libraries (see the Apache project [28] or the Expat XML Parser [30]) to parse and interprete such code. To see the benefits of GraphML we display a part of the GraphML code for the network from Example 2.1 with degree and name information in Figure 2.3

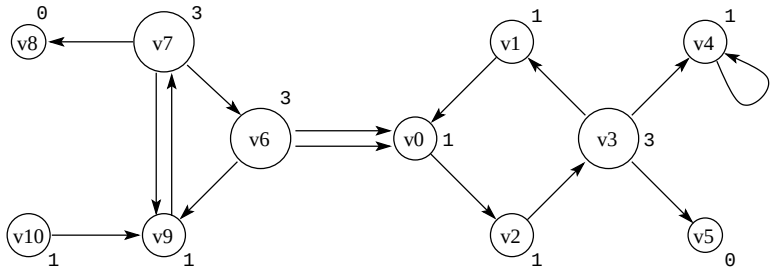

Figure 2.2: Outgoing degree index for the demonstration network.

```
<?xml version="1.0"?> <graphml
xmlns="http://graphml.graphdrawing.org/xmlns/1.0rc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/1.0rc
                    graphml-structure-1.0rc.xsd
                    http://graphml.graphdrawing.org/xmlns/1.0rc
                    graphml-attributes-1.0rc.xsd" >
 <key id="k1" for="node" attr.name="VertexName" attr.type="string">
  <desc>Name of the vertex</desc> <default>Default name</default>
 </key>
 <key id="k2" for="node" attr.name="DegreeIndex" attr.type="double">
  <desc>Vertex degree</desc> <default>-1</default>
 </key>
 <key id="k3" for="edge" attr.name="EdgeName" attr.type="string">
  <desc>Name of the edge</desc> <default>Default name</default>
 </key>
 <graph edgedefault="directed">
  <node id="n0">
   <data key="k1">v0</data> <data key="k2">4</data>
  </node>
  <node id="n1">
   <data key="k1">v1</data> <data key="k2">2</data>
  </node>
  . . .
  <node id="n10">
   <data key="k1">v10</data> <data key="k2">1</data>
  </node>
  <edge source="n6" target="n0"> <data key="k3">e0</data> </edge>
  <edge source="n6" target="n0"> <data key="k3">e1</data> </edge>
  . . .
  <edge source="n10" target="n9"> <data key="k3">e14</data> </edge>
 </graph>
</graphml>
```

Figure 2.3: Extract from the GraphML code for the demonstration network from Figure 2.1 with name and degree information.

Chapter 3

Algorithmic framework

Object-oriented languages like C++ [26], Java [13] and Smalltalk [20] are especially useful for the implementation of network analysis algorithms, because we can express mathematical objects like vertices and edges in a natural way. We will give a short introduction to popular generic programming approaches. Afterwards we describe useful design patterns and apply them for the formulation of basic algorithms.

3.1 Generic programming

Generic programming is an approach to organize structured and reusable software by detecting similar computation steps. This enables us to write shorter code, which helps to find possible errors more easily or to change our solution.
Imperative languages like Fortran or C assist us by methods to do a computation. On the other hand object-oriented languages like C++ or Java help us also to represent the program data, which makes it easier to do the computation.
We can develop data structures and algorithm independently by decoupling both parts with properly defined interfaces. Then it is possible to reuse already existing data structures or combine data structures and algorithms in a flexible way.
The central aspect of generic programming is to formalize the interaction requirements of data structures and algorithms. Handling this aspect appropriately includes the definition of object interfaces, the semantic interpretation of operations and the definition of invariants. We can call such a set of requirements a *concept* and we say that a class is a model of a concept, if it conforms to these requirements.

The requirements of a concept can be classified into

- the valid expressions,
- the associated data types,
- the invariants and
- the complexity guarantees

of a model.

Example 3.1 (Iterators)
A well-known example of a concept is the description of iterators. Iterators are objects to decouple the code for data structures from the code for algorithms, by giving algorithms a uniform access to the elements of different data structures.
The iterator concept gives answers to questions like:

- How can we move the iterator on the data structure and how can we retrieve its elements?
- What are the types of the data structure and its elements?
- Is the iterator valid after changing or deleting an element?
- Which time do we need to find the next element?

Concepts for iterators have hierarchical dependencies, because we need iterators with different facilities. By refining iterator concepts, we can add facilities like random access or circular behavior. To implement such refinements we can use either sub-classing or template parameters.

We look on three libraries with algorithms and data structures for C++ to see how they support the creation of generic code.

3.1.1 STL

The Standard Template Library (STL)[1] offers decoupled data structures, algorithms and iterators. These objects are combined with template parameters and the interfaces are informally described in a detailed concept hierarchy.
Collections of arbitrary objects can be managed in data structures like vectors, lists or trees and the elements in a container class can be manipulated by basic algorithms for tasks like sorting, searching and copying. Additionally there are associative data structures like hash maps or dictionaries to map objects by key-value pairs.
The programmer is restricted to add or remove elements from the container directly, while element access is provided only indirectly by iterators.

[1] A useful starting point for informations about the STL is (Breymann, 2000 [6]).

3.1.2 BGL

The Boost Graph Library (BGL) [25] is part of a collection of source libraries called Boost. While the BGL concentrates on graph computation by offering different graph containers and algorithms, there are also libraries for frequently occurring tasks like input/output handling, memory management, numerical computation or parsing as well as support of generic and higher-order programming. It is developed on top of the STL while extending their functionality in different ways.
Like in the STL there are templated data structures, iterators and algorithms, but the concept approach is augmented by a mechanism for checking the requirements of template parameters at compilation time. Because invariants and complexity guarantees are not computable, it checks whether all defined expressions are satisfiable and the associated data types are valid. This is done by associating information with compile-time entities through traits classes. These traits classes contain generic code for checking the requirements of a concept by trying to generate code for a given model class that uses the associated types and valid expressions. If this generation is successful, it is proven that the given class fulfills the traits concept.
Many graph algorithms can be derived from given basic algorithms by augmenting them by visitors.

3.1.3 LEDA

The Library of Efficient Data Types and Algorithms (LEDA) [21] is a library of combinatorial and geometric computing in C++.
LEDA offers a wide range of templated data structures with a STL compliant iterator concept. In contrast to STL and Boost the target of the LEDA approach is not to be a highly reusable framework of decoupled containers and algorithms, but to be efficient and easy to use.
LEDA contains common data types from computer science like graphs realized in the most efficient way. Additionally there are special data structures for node and edge properties. By the use of template parameters of the container classes the user can choose not only the item types but decide also between different internal implementations.
Many common algorithms are implemented on top of the graph data structures and are integrated into the customizable user interface called GraphWin that can also be extended by own algorithms. Despite the existence of iterators LEDA defines convenient and fast loop macros for item access.
LEDA uses its own file format to store attributed graphs permanently, but additionally there is an extension packages to store and retrieve graphs in the GraphML format.

3.2 Design patterns

One of the most difficult tasks for writing reusable and structured code is the definition of suitable interfaces between the code fragments. This is the reason why design patterns came into action. Design patterns are simple structures that occur periodically in the design of software libraries and are useful in different contexts.
Many design patterns discussed so far are extracted from the implementation of user interfaces. We only use behavioral patterns and our goal is to give an insight into the patterns we can apply without going into detail. The standard source for detailed information is the book of Gamma et al [10]. Although there are many formalisms to illustrate design patterns, we only use an informal notation to illustrate our approach.

Example 3.2 (Outgoing vertex degree)
We can compute the outgoing vertex degree index of a network $G = (V, E)$

$$\begin{aligned} \overrightarrow{X}_{Deg}^{V,G}(v) = \#\overrightarrow{I}_G(v) &= \#\{e \in E : e = (v, w)\} \\ &= \sum_{e \in E} \begin{cases} 1 & \text{, if } v \text{ is the source of } e \\ 0 & \text{, else} \end{cases} \end{aligned}$$

simultaneously for all vertices with time in $\mathcal{O}(|E|)$ by performing one iteration over all edges of G like in Procedure outgoingDegreeIndex.

Procedure `outgoingDegreeIndex`

Input : Network $G = (V, E)$
Output: Vertex vector X
begin
  foreach *vertex* $v \in V$ **do**
    assign $\mathsf{X}[v] \leftarrow 0$;
  foreach edge $e = (v, w) \in E$ **do**
    assign $\mathsf{X}[v] \leftarrow \mathsf{X}[v] + 1$;
end

In an object-oriented language Procedure outgoingDegreeIndex can be implemented in a class consisting of information about the corresponding network together with the vertex vector X and a method that contains the code for the loop.
To compute the incoming degree index $\overleftarrow{X}_{Deg}^{V,G}$ and the degree index $\overline{X}_{Deg}^{V,G}$ we can either change the code of the existing class or generate an additional one.

Both options are not contenting:

- By changing the code of the existing class we cannot select which index to compute by executing the computation. Furthermore, the code would lose its clarity.
- If we have to maintain many classes for similar purposes, we have to update all of them each time we improve a common aspect.

Although this drawbacks are acceptable for a simple loop like in Procedure outgoingDegreeIndex, our example has the interesting property that there are two parts of code:

- the code for the loop and
- the code inside the loop.

3.2.1 Template method pattern

The *template method* pattern can be applied, if an algorithm class is implemented where one or more methods of the class can be rewritten in a sub-class to meet special requirements.
We can define a loop over the edges of a network as a class with a method for the edge loop `do_loop()` together with

- a method `template_visit_edge()` that is called inside the loop and
- a method `template_init_loop()` to initialize the data members.

Now we can define the template methods in a derived class. The resulting code for our example could look like in Class 1.

If we want to change the implementation of the loop later into a more efficient version, we have to touch only the base class. Another advantage of this pattern is that we have to test the loop just once and can concentrate later on the variable part assuming that the base algorithm is correct. This helps us to save redundant code and makes it easier to locate errors.
In real world programming tasks we have to compare similar code fragments and put the intersecting parts into a common method. Then we are able to implement the differences in the template methods of sub-classes. A related programming method in purely imperative languages is the usage of call-back functions that are passed by reference.

The template method pattern means that we create one sub-class for each specialization of a base algorithm. Because we merge the variable and constant code in a sub-class, this approach does not reflect the property that parts of Class 1 are decoupled. This implies an own execution of the base

Class 1: Outgoing degree class

```
Data   : Network G = (V, E)
         Vertex vector X

Method template_init_loop() begin
    foreach vertex v ∈ V do
        assign X[v] ← 0;
end

Method template_visit_edge(edge e = (v, w)) begin
    assign X[v] ← X[v] + 1;
end

Method do_loop() begin
    call method template_init_loop();
    foreach edge e = (v, w) ∈ E do
        call method template_visit_edge(e);
end
```

algorithm for each specialization as well as the duplication of its data structures. Although this is not an asymptotical problem, especially for computing many values on large networks we can become more efficient by a constant factor, if we transfer the template methods into an own class.

3.2.2 Visitor pattern

The *visitor* pattern can be applied, if an algorithm class can be decoupled into two parts that maintain their own data structures.
In the case of Class 1 we can get a visitor class by transferring the template methods `template_init_loop()` and `template_visit_edge()` into an own class. Now every augmentions for the edge loop can be done by placing additional code into the template methods of a class inherited from this base visitor class. This leads in our example to the classes in Class 2 and Class 3.
We can see that the visitor pattern helps to decouple parts of algorithms. Another benefit of this approach is the combination of the vertex vector with the elementary computation step. After executing the same edge loop object with different visitor versions we can free its memory, because all results of interest are separated from the edge loop.
By using visitors to augment base algorithms we save memory, but we have to rerun them for each refinement. To save runtime as well we have to be able to invoke the template methods of more than one visitor object at once.

Class 2: Outgoing degree visitor class

Data : Network $G = (V, E)$
Vertex vector X

Method `template_init_visitor()` **begin**
  foreach *vertex* $v \in V$ **do**
    assign $\mathsf{X}[v] \leftarrow 0$;
end

Method `template_visit_edge`(edge $e = (v, w)$) **begin**
  assign $\mathsf{X}[v] \leftarrow \mathsf{X}[v] + 1$;
end

Class 3: Edge loop class

Input : Network $G = (V, E)$
Data : Visitor *vis*

Method `do_loop()` **begin**
  call method *vis*.`template_init_visitor()`;
  foreach $e = (v, w) \in E$ **do**
    call method *vis*.`template_visit_edge`(e);
end

3.2.3 Chain of visitors pattern

A *chain of visitors* can be applied, if we have to invoke the template method of more than one visitor at once.
We can combine visitor objects like in Class 2 in a singly linked list by inserting the interface methods `init_visitor()` and `visit_edge()` together with an information about a possible predecessor. `init_visitor()` and `visit_edge()` try to call the same interface methods of its predecessor first and call then `template_init_visitor()` and `template_visit_edge()` respectively, which become internal methods. A visitor chain class for an edge loop could look like in Class 4.

Now we can compute the three degree indices simultaneously in any combination by chaining appropriate visitor chain objects. But we know even more. When we invoke a template method of a visitor, we know that each predecessor finished its task for the current edge as illustrated in Figure 3.1. This invariant makes it possible to reuse the result of another visitor that is a predecessor in the chain list.
Calling a method of a visitor class is similar to giving over the responsibility during execution. So we can view a visitor as a request handler and it is

Class 4: Visitor chain class for an edge loop

Data : Network $G = (V, E)$
Visitor chain object *pred*

Method `template_init_visitor()`

Method `template_visit_edge`(edge $e = (v, w)$)

Method `init_visitor()` **begin**
  if *pred is defined* **then**
    call method *pred*.`init_visitor()`;
  call method `template_init_visitor()`;
end

Method `visit_edge`(edge $e = (v, w)$) **begin**
  if *pred is defined* **then**
    call method *pred*.`visit_edge`(e);
  call method `template_visit_edge`(e);
end

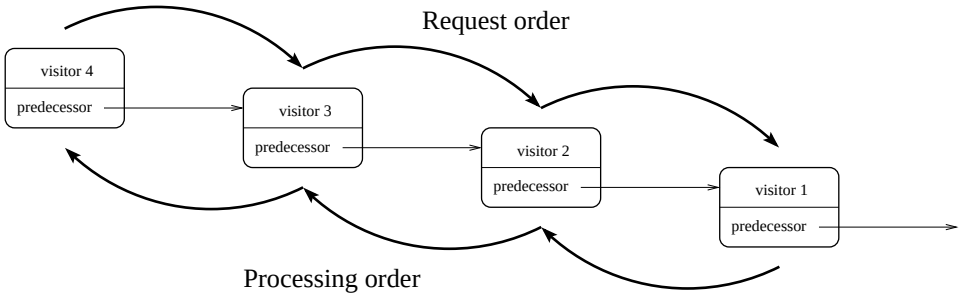

Figure 3.1: Order of requests and processing for a chain of visitors.

related with the *chain of responsibility* pattern. A chain of responsibility is also realized by a singly linked list of handler object, where each object gets the chance to handle requests in its own way. But differently to our situation such a handler object tries first to handle a request by itself and then returns responsibility immediately, if it is possible. Only if all its predecessors in the list are not able to handle a request, an object gets the chance to execute its template method.

Using the chain of visitors pattern we keep our code separated and save additionally memory. In most cases there is no reason to run the edge loop more than once, because we know that each time we invoke a method of the last visitor all visitors in the chain do their job.

Another important effect is that we can compose the set of visitors dynamically at execution time just by deciding whether we need to compute a special result.

3.2.4 Loop kernel pattern

The *loop kernel* pattern can be applied, if we have to control the iteration of a loop object.
By using the chain of visitor pattern, we have to create a sub-class for each refinement we need. Especially during the development of new algorithm implementations we want to concentrate only on our code and not on the class syntax. So it is helpful to change our point of view.
We run an edge loop to invoke the template methods of a visitor chain. But we can also view an edge loop as an object having an initial, a current and a final state. By assuming the ability to determine whether a loop is valid or not[2] together with the possibility to move to the next edge we can control the loop from outside to do according computations. Additionally, we can decide to go on in the loop before (or after) each iteration step. This pattern is called loop kernel[3], because it turns a loop just inside-out. A loop kernel for an edge loop can look like in Class 5.

Looking closer at this concept we recognize the similarity to the well known iterator pattern. The difference is that iterators are so called lightweight pattern, i.e. they are typically "smaller" than useful loop kernel. Additionally, iterators are designed to be representatives of elements in a data structure, while a loop kernel can represent different data types in different situations. To imitate this representative behavior we can give visitors (direct) access to the current element and the loop kernel has to be responsible to set them in the right situation. We can reflect this by invoking the template methods of visitor chain objects not with the current element but with the kernel object as a parameter like in Class 6.
We can augment the edge loop kernel like in Procedure doEdgeLoop, while we have the possibility to insert several conditional breaks without changing the loop kernel.
When we look at this approach, it seems that we produce much more code. Actually, most of this additional code is needed to define the new interface only in the base classes, while the augmentions become more tight and we have the full control over the edge loop.

[2] A loop is valid, if the current state is not equal to the final state. The final state can be signalized by getting an undefined element from a list.

[3] A more detailed discussion of the loop kernel pattern can be found in (Kühl, 1996 [19]).

Class 5: Edge loop kernel class

Data : Network $G = (V, E)$
Edge current

Method `init_loop()` **begin**
assign current $\leftarrow$ first edge in list E;
end

Method `set_next()` **begin**
assign current $\leftarrow$ successor of current in list E;
end

Method `is_valid()` **begin**
return whether current is defined;
end

Class 6: Visitor chain class for edge loop

Data : Network $G = (V, E)$
Visitor chain object *pred*

Method `template_init_visitor()`

Method `template_visit_edge(`edge loop kernel `)`

Method `init_visitor()` **begin**
if *pred is defined* **then**
call method *pred*.`init_visitor()`;
call method `template_init_visitor()`;
end

Method `visit_edge(`edge loop kernel `)` **begin**
if *pred is defined* **then**
call method *pred*.`visit_edge(`kernel `)`;
call method `template_visit_edge(`kernel `)`;
end

Procedure `doEdgeLoop`

Input : Network $G = (V, E)$
Edge loop kernel
Chain of visitor
begin
call method kernel.`init_loop()` and method visitor.`init_visitor()`;
while kernel is valid **do**
call method visitor.`visit_edge(`kernel `)`;
call method kernel.`set_next()`;
end

Remark 3.1
To simplify the following notations we imply that all visitors are implemented as chain visitors. Additionally, we suppress the declaration of the predecessing visitor as well as the methods for the public interface and specify only the internal template methods.

3.3 Basic algorithms

After the presentation of useful design patterns in Chapter 3.2 we adopt them for elementary network analysis tasks that occur frequently. Later we discuss the computation of network indices in Chapter 4 by augmentions for these basic algorithms.

3.3.1 Network loops

The first basic algorithm is the edge loop kernel in Class 5 and we have seen in Procedure doEdgeLoop how to augment loops by visitor objects.
Because we can define analog algorithms for vertex loops, we omit to write them down explicitly. Instead we will describe in detail the iteration for edge-networks, which is important to compute edge indices.

Definition 3.1 (Edge-network)
For a network $G = (V, E)$ we define the associated *edge-network* by the pair $\mathcal{E}(G) := (E, F)$ with

$$((u, v), (v, w)) \in_k F \Leftrightarrow (u, v) \in_r E, (v, w) \in_s E \text{ and } k = r \cdot s.$$

It is clear that we can iterate over the vertices of $\mathcal{E}(G)$ with time in $\mathcal{O}(|E|)$ by performing an edge loop on G. Furthermore, we can iterate over the edges of $\mathcal{E}(G)$ implicitly.

Lemma 3.1 (Implicit edge-network iteration)
We can iterate over the outgoing edges of an implicit edge-network with memory in $\mathcal{O}(|G|)$, while only the running time is linear in the size of $\mathcal{E}(G)$.

Proof: We can do this iteration with the loop kernel in Class 7. This follows directly by definition, because we visit for all $\mathsf{eS} \in E$ all outgoing edges in $\mathcal{E}(G)$. Since we do not have to create an instance of $\mathcal{E}(G)$ in memory, the memory requirements for Class 7 are in $\mathcal{O}(|G|)$ and only the running time is linear in the size of $\mathcal{E}(G)$.

We can connect a loop kernel with a chain of visitors like in Procedure incidentEdgesLoop.

Class 7: Edge pair loop kernel class

Data : Network $G = (V, E)$
Edges eS, eT

Method `init_loop()` **begin**
  assign eS $\leftarrow$ first edge in list E;
  assign eT $\leftarrow$ first outgoing edge in list $\overrightarrow{I}_G(target(\mathsf{eS}))$;
end

Method `set_next()` **begin**
  assign eT $\leftarrow$ successor of eT in list $\overrightarrow{I}_G(target(\mathsf{eS}))$;
  while *eS* is defined and *eT* is not defined **do**
    assign eS $\leftarrow$ successor of eS in list E;
    if *eS* is defined **then**
      assign eT $\leftarrow$ first outgoing edge in list $\overrightarrow{I}_G(target(\mathsf{eS}))$;
end

Method `is_valid()` **begin**
  return whether eS and eT are defined;
end

Procedure `incidentEdgesLoop`

Input : Edge loop kernel for edge-network
Edge loop visitor for edge-network
begin
  call method kernel.`init_loop()` and method visitor.`init_visitor` ;
  while kernel is valid **do**
    call method visitor.`visit_edge_pair(` kernel `)`;
    call method kernel.`set_next()`;
end

Class 8: Edge loop visitor to count the edges of $\mathcal{E}(G)$

Data : $\mathsf{k} \in \mathbb{N}$

Method `template_init_loop()` **begin**
  assign k $\leftarrow 0$;
end

Method `template_visit_edge_pair(`edge pair loop kernel `)` **begin**
  assign k $\leftarrow$ k $+ 1$;
end

Example 3.3 (Edge network size)
To demonstrate the augmention of the loop kernel in Class 7 we define a visitor that counts the edges of an edge-network $\mathcal{E}(G)$.
It turns out that $\mathcal{E}(G)$ can become surprisingly large even if G is of acceptable size. Using the visitor from Class 8 we get the fact that the edge-network of the demonstration network from page 7 has 11 vertices and 15 edges. On the other hand the metabolic network of the bacteria *Helicobacter pylori*[3] with 949 vertices and 2325 edges results in an edge-network with 2325 vertices and 22627 edges.

Remark 3.2
To visit the edges for the incoming case we have to adjust the loop kernel by interchanging the assignments of its source and target properly. To visit the edges for the undirected case we have to combine the outgoing and incoming case and invoke the visitors for both cases.

3.3.2 Network multiplication

The multiplication of a matrix $A \in \mathbb{N}_0^{n \times n}$ and a vector $x \in \mathbb{R}_{\geq 0}^n$

$$\mathrm{m} : \mathbb{N}_0^{n \times n} \times \mathbb{R}_{\geq 0}^n \longrightarrow \mathbb{R}_{\geq 0}^n \tag{3.1}$$

defined by

$$\mathrm{m}(A, x) := A \cdot x = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^n a_{1,j} \cdot x_j \\ \vdots \\ \sum_{j=1}^n a_{n,j} \cdot x_j \end{pmatrix}$$

can be computed with time and memory in $\mathcal{O}(n^2)$ directly from the definition like in Procedure matrixMultiply.

Procedure `matrixMultiply`

Input : Matrix $A \in \mathbb{N}_0^{n \times n}$, vector $x \in \mathbb{R}_{\geq 0}^n$
Output: Vector $y \in \mathbb{R}_{\geq 0}^n$
begin
  for $i \leftarrow 1$ ***to*** n **do**
    assign $y_i \leftarrow 0$;
    for $j \leftarrow 1$ ***to*** n **do**
      assign $\boxed{y_i \leftarrow y_i + a_{i,j} \cdot x_j}$;
end

These requirements hold, because we have to touch each of the n^2 elements of A exactly once. We can avoid the quadratic requirements for sparse matrices by extending the edge loop kernel from Class 5. Therefore we have to

map between networks and matrices.

Definition 3.2 (Adjacency matrix)
For a network $G = (V, E)$ we define the entries of the associated *adjacency matrix* $A(G) = (a_{v,w})_{v,w \in V}$ by

$$a_{v,w} = k \iff (v, w) \in_k E.$$

The definition of the adjacency matrix enables us, to map bijectively between the set of networks with $n \in \mathbb{N}$ vertices and the set of quadratic $n \times n$-matrices with nonnegative natural entries $\mathbb{N}_0^{n \times n}$. This mapping can be realized by an internal vertex numbering of G together with a $n \times n$-matrix with values in $\mathbb{N}_0$ indexed by these node numbers. We express this association by writing

- $A(G)$ for the adjacency matrix of a network G and
- $G(A)$ for the network of a given quadratic matrix $A \in \mathbb{N}_0^{n \times n}$.

This means that we can defined the map from 3.1 equivalently by

$$\mathrm{m} : \mathcal{G}_n \times \mathbb{R}_{\geq 0}^n \longrightarrow \mathbb{R}_{\geq 0}^n$$

where $\mathcal{G}_n$ denotes the set of all networks with n vertices. Because the matrix value $a_{i,j}$ is equal to $\#(v_i, v_j)$, i.e. the number of edges between v_i to v_j, we can compute $y_i = \mathrm{m}(A(G), x)_i$ for a network $G = (V, E)$ by

$$y_i = \sum_{j=1}^{n} a_{i,j} \cdot x_j = \sum_{j=1}^{n} \#(v_i, v_j) \cdot x_j = \sum_{j=1}^{n} (\underbrace{x_j + \cdots + x_j}_{\#(v_i, v_j) \text{ times}}) = \sum_{e=(v_i, v_j) \in E} x_j.$$

We can do this analog to Procedure matrixMultiply for each vertex separately by iterating over all outgoing edges. To do this efficiently the network has to be represented by the incidence list representation. Because each edge updates only the resulting element for its source vertex, we can do the same computation by a simple loop over the edge list like the multiplication loop kernel in Class 9.
Algorithm 9 needs memory in $\mathcal{O}(|G|)$, because the matrix information are now stored as a network and iterating over the set of edges once takes time in $\mathcal{O}(|E|)$. Because this is only an improved version of the our edge loop kernel and we can implement the multiplication kernel as its sub-class with same public interface[4]. Moreover we can augment the multiplication kernel by edge loop visitors like illustrated on page 21, while visitors can additionally use for each edge the information about which vertex is updated from

[4]The public interface does not change, because `template_set_vertices()` is an internal method too.

Class 9: Multiplication loop kernel

```
Data   : Network G = (V, E)
         Edge current
         Vertices s, t
         Vertex vectors Y, X

Method template_set_vertices() begin
    assign s ← source(current);
    assign t ← target(current);
end

Method init_loop() begin
    foreach vertex v ∈ V do
        assign X[v] ← Y[v]; assign Y[v] ← 0;
    assign current ← first edge in list E;
    call method template_set_vertices();
    assign Y[s] ← Y[s] + X[t] ;
end

Method set_next() begin
    assign current ← successor of current in list E;
    call method template_set_vertices();
    assign Y[s] ← Y[s] + X[t] ;
end

Method is_valid() begin
    return whether current is defined;
end
```

where.
By setting the vertices s and t in a template method we are able to compute the multiplication with a transposed matrix A^T

$$\mathrm{m}(A^T, x)^T = A^T \cdot x \quad = \quad \begin{pmatrix} a_{1,1} & \dots & a_{n,1} \\ \vdots & \ddots & \vdots \\ a_{1,n} & \dots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} a_{j,1} \cdot x_j \\ \vdots \\ \sum_{j=1}^{n} a_{j,n} \cdot x_j \end{pmatrix}$$

by swapping the assigning of s and t in `template_set_vertices()`.

Remark 3.3
Typical networks have the property $|E| \in \mathcal{O}(|V| \cdot \log(|V|))$ or $|E| \in \mathcal{O}(|V|)$. In this case the network multiplication is significantly faster than the matrix multiplication from Procedure matrixMultiply. Additionally, we can represent the network by simple lists, which is especially important for very large networks.

Another benefit of the loop approach for multiplication is its applicability for edge-networks. Again we omit to write down a suitable kernel class, because we get it from the loop kernel in Class 7 in the same way as above. From the construction of the loop kernel for the edges of an induced edge-network we already know that the time for a multiplication of $\mathcal{E}(G)$ with an edge vector is linear in the time of the edge-network, while the memory requirements stay linear in the size of the underlying network G. But differently to the normal network we now have to represent G by the incident edge lists representation.

3.3.3 Network search

Many network algorithms have to find all vertices and edges reachable from a given root vertex by following paths.

Definition 3.3 (Vertex-induced sub-network)
We define the *vertex-induced sub-network* $G' = (V', E')$ of $V' \subseteq V$ by

$$E' := \left\{ e = (v, w) \in E \,:\, v \in V' \text{ and } w \in V' \right\}.$$

Definition 3.4 (Horizon)
The *horizon* $\overline{\text{hor}}(r)$ of a vertex $r \in V$ is defined by the vertex-induced sub-network of

$$\{ v \in V : \text{ there is an undirected path from } r \text{ to } v \}.$$

The outgoing horizon $\overrightarrow{\text{hor}}(r)$ and the incoming horizon $\overleftarrow{\text{hor}}(r)$ are defined in the same way by considering directed paths from r and to r respectively.

Procedure searchNetwork finds the horizon of a given vertex $r \in V$ by using the loop kernel from Class 10 and shows additionally how to couple the loop kernel with augmenting visitors.

Lemma 3.2
Algorithm searchNetwork finds the horizon of a given vertex $r \in V$ with time and memory in $\mathcal{O}(|G|)$.

Proof:
The algorithm assures that we mark a vertex as discovered at the moment we insert it into search_front (line 1 and 4). Additionally, we call the method `template_set_possible_edges()`. Therefore each vertex is inserted at the most once into search_front.
Assume a vertex $v \in V$ is visited when the algorithm stops. We know that only the root is not discovered by an edge. Therefore we can construct a path from root to v backwards by starting from v and go to its discovering vertex. Applying this argument recursively we get $v \in \overline{\text{hor}}(r)$.
Assume a vertex $v \in \overline{\text{hor}}(r)$ is not visited when the algorithm stops. Because

Procedure `searchNetwork`

```
Input  : Network G = (V, E)
         Vertex r
         Loop kernel for searching a network
         Network search visitor
begin
    call method kernel.init_loop(r);
    call method visitor.init_visitor(r);
    while kernel is valid do
        call method kernel.set_next();
        if kernel.e is defined then
            call method visitor.traverse_edge( kernel );
        else
            call method kernel.template_remove_source_vertex();
end
```

we delete a vertex from search_front only if there are no more unmarked possible edges, all vertices reachable from v have to be undiscovered. But this cannot be true for root.
By definition an edge $e = (v, w) \in E$ is in the sub-network induced by a set of vertices $V' \subseteq V$, if $v \in V'$ and $w \in V'$ holds. The algorithm traverses all possible edges of a discovered vertex and ensures therefore that all edges of the horizon are traversed as well.
To see the running time we can assume that all list operations can be done in constant time. Because each vertex is inserted and deleted from search_front at the most once, we concentrate on the while-loop of the algorithm. Inside the loop we either traverse an edge and discover possibly a vertex or we delete a vertex from search_front (line 3). This can be done in $\mathcal{O}(|G|)$, because each edge is assigned at the most once to kernel.e (line 2).
We can realize the marks with memory in $\mathcal{O}(|G|)$. search_front needs $\mathcal{O}(|V|)$ memory and each edge is represented at the most twice in front_edges. This completes the proof.

Remark 3.4
Lemma 3.2 holds also for the outgoing horizon and for the incoming horizon by setting front_edges in `template_set_possible_edges` only to $\overrightarrow{I}_G(s)$ and $\overleftarrow{I}_G(s)$ respectively.

We demonstrate the augmention of searching a network by a visitor that collects the edges of $\overline{\text{hor}}_G(r)$ in Class 11. Because we call `traverse_edge` for all edges assigned to `e` of the network search kernel, the correctness of Class 11 follows directly from Lemma 3.2. Additionally the visitor preserves the requirements of Class 10.

Class 10: Loop kernel class for network search

```
Data   : Network G = (V, E), vertex root
         Edge e, vertices v, w
         Vertex mark discovered
         Edge mark traversed
         List search_front for vertices
         Vertex vector front_edges of lists of possibly untraversed edges

Method template_set_possible_edges(vertex s) begin
    assign front_edges [ s ] ← →I_G(s) ∪ ←I_G(s);
end

Method template_set_next_source() begin
    assign v ← choose vertex from search_front ;
end

Method template_remove_source_vertex() begin
    remove v from search_front ;
end

Method init_loop(vertex r) begin
    mark all vertices v ∈ V as not discovered;
    mark all edges e ∈ E as not traversed;
    make search_front empty;
    assign root ← r;
1   append root to search_front; mark root as discovered;
    call method template_set_possible_edges( root );
end

Method set_next() begin
    call method template_set_next_source();
2   while front_edges [ v ] is not empty and its first edge is traversed do
        remove first edge in front_edges [ v ];
3   if front_edges [ v ] is not empty then
        assign e ← first edge in front_edges [ v ]; mark e as traversed;
        assign w ← opposite(e,v);
        if w is undiscovered then
4           append w to search_front; mark w as discovered;
            call method template_set_possible_edges(w );
    else
        assign e ← undefined;
        assign w ← undefined;
end

Method is_valid() begin
    return whether search_front is empty or not;
end
```

Class 11: Network search visitor to collect the horizon edges

Data : List edges_in_horizon ;

Method `template_init_visitor`(vertex v) **begin**
  make edges_in_horizon empty;
end

Method `template_traverse_edge`(network search kernel) **begin**
  append kernel.e to edges_in_horizon ;
end

Because `template_set_next_source` assigns `v` arbitrarily at each loop step, there are no more obvious properties for the network search algorithm (we can for example distinct only between traversed and untraversed edges). To utilize network search for other problems we have to restrict the order vertices and edges are assigned in `set_next`. To motivate a classification of the edges during a call of `traverse_edge` we discuss shortly the search of a special class of networks.

Definition 3.5 (Cycle)
A cycle is a path where the first vertex and the last vertex are identical.

Definition 3.6 (Tree)
A *tree* is a connected network $G = (V, E)$ without undirected cycles and we notate the class of all trees by $\mathcal{T}$. By tagging a vertex $r \in V$ we call a pair (G, r) a tree with root r or simply a *root tree*.

Lemma 3.3
Let T be a tree. There is exactly one undirected path between each pair of tree vertices and $|E| = |V| - 1$.

Proof:

- Because a tree is connected, there is at least one path between each pair of tree vertices. Assume a pair of tree vertices is connected by two different paths. Then the union of these paths contains an undirected cycle.
- It is clear that a tree with 1 or 2 vertices has $n - 1$ edges. Assume that each tree T with n vertices has $n - 1$ edges and we want to add a new vertex. Then we have to insert exactly one edge, because by adding another edge we would close an undirected cycle. Therefore we get the identity inductively.

Corollary 3.1
There is at the most one directed path between each pair of vertices of a tree.

Now we want to classify the edges traversed until Procedure searchNetwork by whether it discovers a vertex or not. This means that the vertices in the horizon and the traversed edges that discover a vertex form a root tree. This holds, because it is assured that we have an undirected path between each pair of vertices and closing an undirected cycle would mean that the search discovers a vertex more than once.

Terminology 3.1 (Tree edge)
We call an edge traversed by searching a network a *tree edge*, if it discovers a vertex.

Terminology 3.2
The vertices of a rooted tree (T, r) are always ordered in a natural way. If we have to walk over v to reach w from the root r, we say v is a predecessor of w and w a successor of v. We say that vertices are siblings, if they are direct successors of the same vertex.

Traversing a tree from its root systematically by following a recursive instruction we get two rules with according numbering for vertices and edges:

1. Visit siblings of a vertex before unvisited descendents.
   We assign 0 to the start vertex and give the number of the predecessor incremented by 1 to the successor. Edges get the number of the vertex from which they are traversed. We call this strategy *"Breadth-First Search"*.

2. Visit descendents of a vertex before unvisited siblings.
   We assign 1 to the start vertex and number the other vertices incrementally in the order they are visited. Edges get the number of the vertex from which they are traversed. We call this strategy *"Depth-First Search"*.

Chapter 4

Index computation

In the presentation of the design patterns in Chapter 3.2 we already discussed the computation of the degree index. We start this chapter with the computation of the local clustering index and discuss then the computation of indices based on distances, shortest paths and feedbacks to involve also global information. Therefore we show how to get distances as well as the dominant eigenvector. Because our discussion is restricted on the computation of vertex indices we also give hints how to get them for the edge case.

4.1 Local clustering

Definition 4.1
We define the neighborhood of a vertex $v \in V$ by

$$\mathrm{N}_G(v) := \left\{ w \in V : \overline{\mathrm{d}}_G(v, w) = 1 \right\}.$$

For a network $G = (V, E)$ we define the *number of triples* for a vertex $v \in V$ by

$$t_G(v) := |N_G(v)| \cdot (|N_G(v)| - 1)$$

and the *number of triangles* for a vertex $v \in V$ by

$$\tilde{t}_G(v) := |\{(x, y) \in N_G(v) \times N_G(v) : (x, y) \in E \text{ and } x \neq y\}|.$$

Definition 4.2 (Local clustering index)
We define the *local clustering index* by

$$X_{\mathrm{Clu}}^{V,G}(v) := \begin{cases} \frac{\tilde{t}_G(v)}{t_G(v)} & \text{, if } t_G(v) > 0 \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

The local clustering index is the probability of two vertices adjacent to a common neighbor being neighbors, too. Algorithm localClustering computes

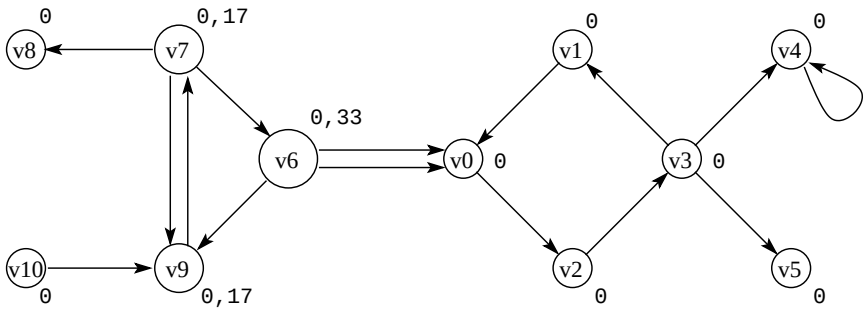

Figure 4.1: Local clustering index (rounded values) for the demonstration network.

the clustering coefficient for the case that each pair of vertices is connected by at the most one edge and the network has no loops. To apply this algorithm for networks in general we have to remove those edges (temporarily). After sorting the list E with a standard sorting algorithm in $\mathcal{O}(|E| \cdot \log(|E|))$ we can find multiple edges and loops in linear time.

Lemma 4.1
Procedure localClustering computes the local clustering index with time in $\mathcal{O}(|V| \cdot |E|)$, if the network has no loops and no multi-edges.

Proof: We can split Procedure localClustering for each vertex into two stages:

- By running the edge loop for the number of triples we count the size of the neighborhood first. This value is correct, because we assume that there are no loops.
- In the second edge loop we find the number of ordered vertex pair inside the neighborhood because we assume that there are no multi-edges.

Remark 4.1
Algorithm localClustering is an example to augment a loop kernel without a special visitor.

4.2 Distance-based indices

Definition 4.3 (Distance)
The outgoing distance $d_G(s,t)$ of a vertex t from a vertex s in a network $G = (V, E)$ is defined by the minimum length of all paths from s to t. If no such path exists, then we set $d_G(s,t) = \infty$.
Additionally, we notate the number of shortest paths from s to t by $\sigma_G(s,t)$ and define the diameter of G by $\mathrm{D}(G) := \max\{d_G(v,w)|v,w \in V\}$.

Procedure `localClustering`

```
Input  : Network G = (V, E)
Data   : Edge loop kernel
         Real numbers triples, triangles
         Vertex flag is_neighbor
         Vertex vector X
begin
    foreach vertex v ∈ V do
        assign triples ← 0 and triangles ← 0;
        assign X [v] ← 0;
        foreach vertex w ∈ V do assign is_neighbor [w] ← false;
        call method kernel.init_loop();
        while kernel is valid do
            if v = source(kernel.e) or v = target(kernel.e) then
                if is_neighbor [opposite(v, kernel.e)]=false then
                    assign is_neighbor [opposite(v, kernel.e)] ← true;
                    assign triples ← triples + 1;
            call method kernel.set_next();
        assign triples ← triples · (triples − 1);
        call method kernel.init_loop();
        while kernel is valid do
            if is_neighbor [source(kernel.e)] = true
            and is_neighbor [target(kernel.e)] = true  then
                assign triangles ← triangles + 1;
        if triples > 0 then  assign X[v] ← triangles/triples;
end
```

Remark 4.2
We can get the incoming and the undirected distances by considering according paths. In the following we discuss only the outgoing case.

4.2.1 Breadth-first search

We can adapt the Breadth-first search (BFS) strategy for trees to search a network[1] by adjusting the methods

`template_set_next_source()` and `template_remove_source_vertex()`

in a sub-class of the loop kernel from Class 10 on Page 29. After assigning a source vertex the first time to kernel.v we have to traverse all unmarked edges before we choose another vertex. We get this behavior by controlling the list kernel.search_front to behave like a queue as in Class 12. By rewriting

[1] More detailed informations about searching networks can be found in (Cormen et al, 1990 [7]).

the method `template_set_possible_edges()` we get the directed Breadth-first search for the outgoing case.

Class 12: Loop kernel class for outgoing BFS (sub-class of class 10)

Method `template_set_possible_edges(`vertex s`)` **begin**
  assign front_edges [s] $\leftarrow \overrightarrow{I}_G(s)$;
end

Method `template_set_next_source()` **begin**
  assign v $\leftarrow$ last vertex from search_front ;
end

Method `template_remove_source_vertex()` **begin**
  remove last vertex from search_front ;
end

Additionally, we define BFS numbers as for trees and use these numbers to distinguish non-tree edges.

Definition 4.4 (BFS numbering and edge types)
All vertices and edges that are not reachable from the root vertex get the BFS number ∞, while we assign the numbers for the discovered vertices recursively by

- the root vertex gets the BFS number 0 and
- if the vertex w is discovered by traversing an edge from the vertex v, we assign the BFS number of v incremented by 1.

Edges get the number of the vertex they are traversed from. At the time a non-tree edge e is traversed we classify it as

- a back edge, if BFS(w) < BFS(v) or
- a cross edge, if BFS(w) = BFS(v) or
- a forward edge, if BFS(w) > BFS(v).

The visitor in Class 13 determines the BFS numbering for vertices and edges as well as the corresponding edge types[2]. The coupling of the BFS kernel and a BFS visitor is the same as in Procedure searchNetwork on Page 28.

We show now the correlation of the BFS number of a vertex and its distance from kernel.root.

[2]Edge types can be realized by an enumerating type.

Class 13: BFS visitor class for traversing information

```
Data   : Network G = (V, E)
         Vertex vector vertex_BFS
         Edge vector edge_BFS ;
         Edge vector bfs_type

Method template_init_visitor(vertex r) begin
    assign for all v ∈ V vertex_BFS [v] ← ∞;
    assign for all e ∈ E edge_BFS [e] ← ∞ and bfs_type [e] ← undefined;
    assign vertex_BFS [r] ← 0;
end

Method template_traverse_edge(BFS kernel ) begin
    assign edge_BFS [e ]=vertex_BFS [v ];
    if vertex_BFS [w ]= ∞ then
        assign vertex_BFS [w ] ← vertex_BFS [v ]+1;
        assign bfs_type [e ] ← tree_edge;
    else
        if vertex_BFS[w] < vertex_BFS[v] then
            assign bfs_type [e ] ← back_edge;
        if vertex_BFS[w] = vertex_BFS[v] then
            assign bfs_type [e ] ← cross_edge;
        if vertex_BFS[w] > vertex_BFS[v] then
            assign bfs_type [e ] ← forward_edge;
end
```

Lemma 4.2
Let $(v_1, \ldots, v_k)$ be the state of kernel.search_front in Class 12. For the vertex numbering from the visitor in Class 13 hold the invariants

$$\mathrm{BFS}(v_i) \leq \mathrm{BFS}(v_{i+1}) \text{ and } \mathrm{BFS}(v_i) \leq \mathrm{BFS}(v_1) + 1$$

hold for $1 \leq i < k$ during the network search in procedure searchNetwork.

Proof: After the call of `template_init_visitor()` only the root vertex is in the list kernel.search_front and the invariants hold.
During the search we change kernel.search_front by either deleting or appending a vertex and we can assume that before such an operation the invariants hold. If we delete a vertex, it is clear that the invariants hold again. Therefore we have to show only the case that we insert a vertex v_{k+1} into the queue. This time the condition holds, because the visitor assigns $\mathrm{BFS}(v_{k+1}) \leftarrow \mathrm{BFS}(v_1) + 1$.

Corollary 4.1
For the visitor in Class 13 the equality

$$\mathsf{vertex\_BFS}(v) = d_G(\mathsf{kernel.root}, v) \text{ holds for all } v \in V.$$

Proof:
The equality is true because of the invariants in Lemma 4.2 and the fact that we follow all possible edges before we choose another vertex to kernel.v.

Corollary 4.2
We can compute the distances of all vertices from a given root with time and memory in $\mathcal{O}(|G|)$, if the network is given in incidence lists representation.

Remark 4.3
We split the tasks for computing the traversing information from the BFS kernel only for presentational reasons. We assume from now on that the BFS kernel also computes this information implicitly.

4.2.2 Closeness

The closeness index was originally defined by Sabidussi [24] for strongly connected networks. We show the computation of closeness for general networks.

Definition 4.5
For each vertex $v \in V$ of a network $G = (V, E)$ we define the number of reachable vertices

$$X_{\mathrm{Rv}}^{V,G}(v) := \#\{t \in hor(v) - \{v\}\}$$

and the sum of distances

$$X_{\mathrm{Ds}}^{V,G}(v) := \sum_{t \in hor(v) - \{v\}} d_G(v, t).$$

Definition 4.6 (Closeness index)
We define the *closeness index* for the vertices of a network $G = (V, E)$ by

$$X_{\mathrm{Clo}}^{V,G}(v) := \begin{cases} \frac{X_{\mathrm{Rv}}^{V,G}(v)^2}{X_{\mathrm{Ds}}^{V,G}(v)} & \text{, if } X_{\mathrm{Ds}}^{V,G}(v) > 0 \\ 0 & \text{, otherwise.} \end{cases}$$

The closeness index is a measurement that gives high values to vertices from which we can reach many other vertices with a short average distance.

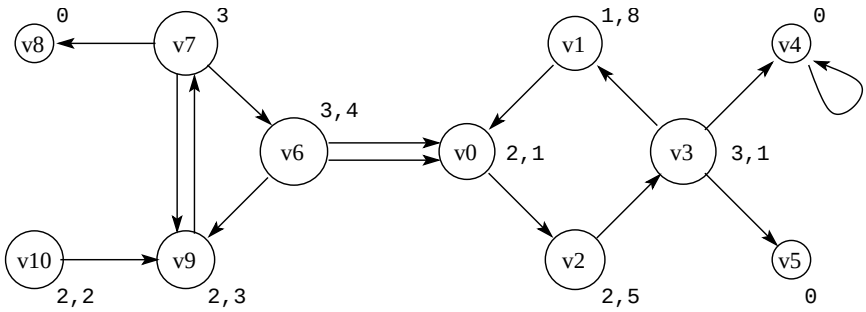

Figure 4.2: Closeness index (rounded values) for the demonstration network.

Remark 4.4
For a strongly connected network G all vertices have the same number of reachable vertices $(n-1)$. Thus we have the case that we can rewrite the above definition to

$$X_{\text{Clo}}^{V,G}(v) = (n-1)^2 \cdot \frac{1}{\sum_{t \in V} d_G(v,t)}.$$

This is the classical definition except for a constant factor (compare [24]).

Because the closeness index is defined by a fraction we show the computation separately for the enumerator $X_{\text{Rv}}^{V,G}(v)$ and the denominator $X_{\text{Ds}}^{V,G}(v)$. We can compute the number of reachable vertices cumulatively by a visitor for the incoming BFS kernel.

Class 14: BFS visitor for cumulative closeness computation

Data : Vertex vector X_{Rv}
Vertex vector X_{Ds}

Method `template_init_visitor`(vertex r) **begin**
  foreach vertex $v \in V$ **do** assign $\mathsf{X}_{\text{Rv}}[v] \leftarrow 0$ and $\mathsf{X}_{\text{Ds}}[v] \leftarrow 0$;
end

Method `template_traverse_edge`(Network search kernel) **begin**
  if kernel.e is tree edge **then**
    assign X_{Rv}[kernel.w] $\leftarrow \mathsf{X}_{\text{Rv}}$[kernel.w] $+ 1$;
    assign X_{Ds}[kernel.w] $\leftarrow \mathsf{X}_{\text{Ds}}$[kernel.w] $+$ vertex_BFS[kernel.w];
end

Lemma 4.3
Procedure closenessIndex computes the closeness index of a network $G = (V, E)$ time in $\mathcal{O}(|V| \cdot |E|)$.

Procedure `closenessIndex`

Input : Network $G = (V, E)$
Vertex vector $\mathsf{X}_{\mathrm{Clo}}$
Data : Incoming BFS kernel
Cumulative closeness visitor
begin
  call method visitor.`init_visitor()` ;
  foreach vertex $t \in V$ **do**
    call method kernel.`init_loop(`t`)` ;
    while kernel is valid **do**
      call method visitor.`traverse_edge(`kernel `)`;
      call method kernel.`set_next()`;
  foreach vertex $v \in V$ **do**
    if visitor.$\mathsf{X}_{\mathrm{Ds}}[v] > 0$ **then**
      assign $\mathsf{X}_{\mathrm{Clo}}[v] \leftarrow \frac{\text{visitor.}\mathsf{X}_{\mathrm{Rv}}[v]^2}{\text{visitor.}\mathsf{X}_{\mathrm{Ds}}[v]}$;
    else
      assign $\mathsf{X}_{\mathrm{Clo}}[v] \leftarrow 0$;
end

Proof: Because of Lemma 3.2 the BFS kernel discovers all vertices in the incoming horizon of the root. All those vertices are discovered by traversing a tree edge despite of the root. Therefore $X_{\mathrm{Rv}}^{V,G}(v)$ is computed correctly. Because of Lemma 4.1 the kernel assigns the distance value of a non-root vertex by traversing a tree edge. This is completed by the construction of a shortest path to the root vertex.

Remark 4.5
We can compute the closeness value of a single vertex directly by one outgoing BFS in linear time. On the other hand the cumulative strategy will be important for the approximation in Chapter 7.

4.2.3 Betweenness

The betweenness index is defined by Anthonisse [2] and Freeman[9] independently.

Definition 4.7 (Dependency)
We define the *dependency* of a vertex $s \in V$ on a single vertex $v \in V$ by

$$\delta_G(s|v) := \sum_{t \in V} \begin{cases} \frac{\sigma_G(s,t|v)}{\sigma_G(s,t)} & \text{, if } \sigma_G(s,t) > 0 \text{ and} \\ 0 & \text{, otherwise.} \end{cases}$$

$\sigma_G(s,t|v)$ denotes the number of shortest paths from s to t using v as an inner vertex (i.e. v is neither the start nor the end vertex).

Definition 4.8 (Betweenness index)
For a network $G = (V, E)$ we define the *betweenness index* of all $v \in V$ by

$$X_{\text{Bet}}^{V,G}(v) := \sum_{s \in V} \delta_G(s|v).$$

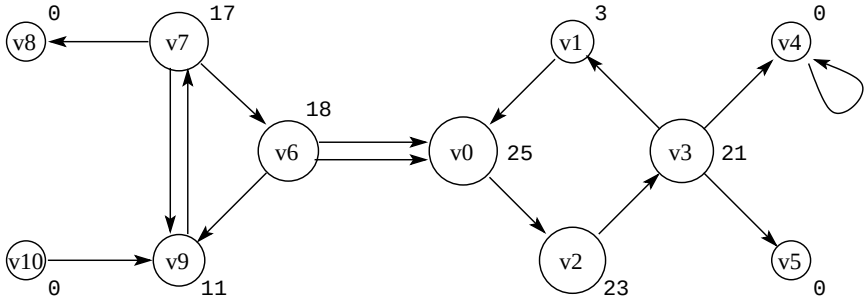

Figure 4.3: Betweenness index for the demonstration network.

To compute the betweenness index with an augmention of the BFS kernel we have to use the following lemma of Brandes [5].

Lemma 4.4 (Recursive dependencies)
Let

$$P_G(s, v) := \{(u, v) \in E : d_G(s, v) = d_G(s, u) + 1\}$$

be the multi-set of predecessing edges on shortest path from s to v.
For $s \neq v \in V$ the recursive equality

$$\delta_G(s, v) = \sum_{\substack{(v,w) \in P_G(s,w) \\ \text{for a } w \in V}} \frac{\sigma_G(s, v)}{\sigma_G(s, w)} \cdot (1 + \delta_G(s, w)) \text{ holds.}$$

A proof of this lemma can be found in the original text (see [5]).

To apply the identity from Lemma 4.4 for a given start vertex s we have to know

- the predecessor set $P_G(s, v)$ and
- the number of shortest path $\sigma_G(s, v)$

for all vertices $s \neq v$, together with

- the reachable vertices ordered by distance from s.

Because of the invariant from the proof of Lemma 4.2, we conclude that for each tree or forward edge e exists a shortest path using e as an intermediate edge. Therefore we can compute all predecessing edges of a given vertex s by BFS visitor in Class 15.

Class 15: BFS visitor for recursive dependencies information

Data : Network $G = (V, E)$
Vertex array P of edge lists
Vertex array σ
List S of vertices

Method `template_init_visitor(`vertex r`)` **begin**
  foreach vertex $v \in V$ **do** make $P[v]$ empty;
  foreach vertex $v \in V$ **do** assign $\sigma[v] \leftarrow 0$;
  assign $\sigma[r] \leftarrow 1$;
  make S empty;
end

Method `template_traverse_edge(` BFS kernel `)` **begin**
  if kernel.e is tree edge or forward edge **then**
    append kernel.e to P[kernel.w];
  if kernel.e is tree edge or forward edge **then**
    assign σ[kernel.w] $\leftarrow \sigma$[kernel.w] + σ[kernel.v];
  if kernel.e is tree edge **then**
    append kernel.w to S;
end

Procedure `betweennessIndex`

Input : Network $G = (V, E)$
Data : BFS visitor for recursive dependencies information
Vertex vector δ
Output: Vertex vector X

begin
  foreach vertex $v \in V$ **do** assign X[v] $\leftarrow 0$;
  foreach vertex $s \in V$ **do**
    call method kernel.`init_loop(`s`)`;
    call method visitor.`init_visitor(`s`)`;
    call Procedure searchNetwork(s,kernel,visitor);
    while visitor.S not empty **do**
      assign vertex $w \leftarrow$ last element of visitor.S;
      foreach edge $e \in$ visitor.$P[w]$ **do**
        assign vertex $v \leftarrow opposite$(kernel.e, w);
        assign $\delta[v] \leftarrow \delta[v] + \frac{\text{visitor.}\sigma[v]}{\text{visitor.}\sigma[w]} \cdot (1 + \delta[w])$;
    foreach vertex $v \in V$ **do**
      if $v \neq s$ **then**
        assign X[v] $\leftarrow$ X[v] + $\delta[v]$;
end

Corollary 4.3
Procedure betweennessIndex computes the betweenness index with time in $\mathcal{O}(|V| \cdot |E|)$.

4.3 Matrix-based indices

While the indices discussed in Chapter 4.2 consider only shortest paths, we now discuss the computation of indices that involve all paths in the network. We show how the number of paths can be computed by using the associated adjacency matrix from Definition 3.2 on Page 25. Because solving a linear equation system is too expansive, we will only discuss methods to approximate matrix-based indices.

The idea of matrix-based indices becomes clear by considering the following lemma that show how to compute the number of paths from a given root.

Lemma 4.5
Let $G = (V, E)$ be a network with a vertex $r \in V$, $A(G) = (a_{i,j})_{i,j \in V}$ the adjacency matrix of G, $A^k = (a^k_{i,j})_{i,j \in V}$ its k-th power and X_r the vertex vector defined by

$$\mathsf{X}_r[v] := \begin{cases} 1 & \text{, if } v = r \\ 0 & \text{, otherwise.} \end{cases}$$

Then the value $\mathsf{Y}^k_r[v]$ of the vertex vector

$$\mathsf{Y}^k_r := (A^T)^k \cdot \mathsf{X}_r$$

is equal to the number of edge sequences from r to v for all $v \in V$.

Proof: We proof the lemma by an induction over $k \in \mathbb{N}$.
For $k = 0$ we get

$$\mathsf{Y}^0_r = \left(A^T\right)^0 \cdot \mathsf{X}_r = E \cdot \mathsf{X}_r = \mathsf{X}_r$$

where E denotes the identity matrix. The lemma holds, because only the empty path from r to r is of length 0.
Assume now that for $k' < k$ the number of edge sequences of length k' from r to v is equal to $Y^{k'}_r[v]$ for all $v \in V$. We complete the proof by

$$\mathsf{Y}^k_r = \left(A^T\right)^k \cdot \mathsf{X}_r = A^T \cdot \left(A^T\right)^{k-1} \cdot \mathsf{X}_r = A^T \cdot \mathsf{Y}^{k-1}_r = \begin{pmatrix} \sum_{v \in V} \mathsf{Y}^{k-1}_r[v] \cdot a_{v,1} \\ \vdots \\ \sum_{v \in V} \mathsf{Y}^{k-1}_r[v] \cdot a_{v,n} \end{pmatrix}.$$

Remark 4.6
The computation of the number of paths can be done with the network multiplication from Chapter 3.3.2.

4.3.1 Influence index and status index

The influence index defined by Katz [17] involves all paths in the network by considering longer paths with decreasing values.

Definition 4.9 (Influence index)
Let A be the adjacency matrix of the network $G = (V, E)$ and

$$\alpha = \frac{1}{\min\{\max_{v \in V} \overrightarrow{\deg}_G(v), \max_{v \in V} \overleftarrow{\deg}_G(v)\} + 1}.$$

The *influence index* of G is defined by

$$X_{\text{Inf}}^{V,G} := \left(\sum_{k=1}^{\infty} (\alpha A)^k \right) \cdot \mathbf{1}.$$

Remark 4.7
The influence index is well-defined, because the choice of the factor α assures that the influence matrix converges.

The following lemma gives us the possibility to approximate the influence index. Because we only use the multiplication of the network with a vector, we can do this computation with the basic algorithm from Chapter 3.3.2.

Lemma 4.6
We can compute the influence index recursively by

$$X_{\text{Inf}}^{V,G} = A \cdot \alpha \left(\mathbf{1} + X_{\text{Inf}}^{V,G} \right).$$

Proof:

$$\begin{aligned} X_{\text{Inf}}^{V,G} &= \left(\sum_{k=1}^{\infty} (\alpha A)^k \right) \cdot \mathbf{1} = \left((\alpha A) \cdot \sum_{k=0}^{\infty} (\alpha A)^k \right) \cdot \mathbf{1} \\ &= A \cdot \alpha \left(E_n + \sum_{k=1}^{\infty} (\alpha A)^k \right) \cdot \mathbf{1} = A \cdot \alpha \left(\mathbf{1} + X_{\text{Inf}}^{V,G} \right) \end{aligned}$$

Because of Lemma 4.6 we can approximate $X_{\text{Inf}}^{V,G}$ by multiplying an initial vector – for example $\mathbf{0}$ – with the network. To decide whether we can stop the approximation we can look at the maximal difference of all vertices after a multiplication. If this value is smaller than a given ε, we can break the iteration. This leads directly to Procedure approximateInfluence.

Procedure `approximateInfluence`

Input : Network $G = (V, E)$
Real value $\varepsilon > 0$
Data : Multiplication kernel
Output: Vertex vector X
begin
  foreach vertex $v \in V$ **do** assign $\mathsf{kernel.Y}[v] \leftarrow 0$;
  repeat
    foreach vertex $v \in V$ **do**
      assign $\mathsf{kernel.Y}[v] \leftarrow \alpha\,(1 + \mathsf{kernel.Y}[v])$;
    call Procedure doEdgeLoop(kernel);
  until $\max_{v \in V} |\mathsf{kernel.Y}[v] - \mathsf{kernel.X}[v]| < \varepsilon$;
  copy $\mathsf{X} \leftarrow \mathsf{kernel.Y}$;
end

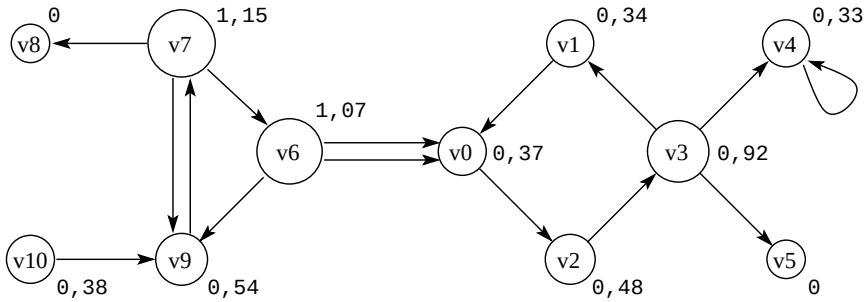

Figure 4.4: Approximated influence index (rounded values) for the demonstration network. The approximation with $\alpha = 0.25$ and $\varepsilon = 10^{-6}$ stops after 15 iterations.

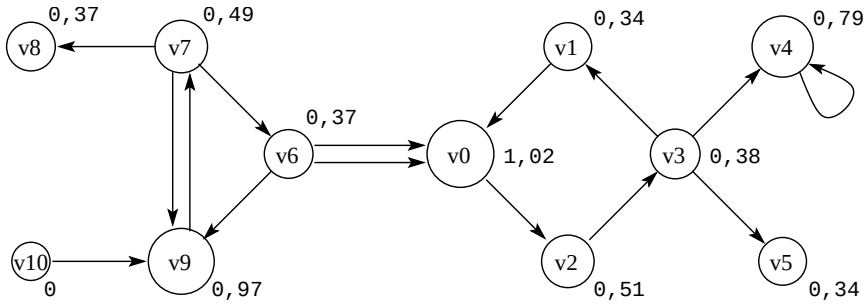

Figure 4.5: Approximated status index (rounded values) for the demonstration network. The approximation with $\alpha = 0.25$ and $\varepsilon = 10^{-6}$ stops after 14 iterations.

By using incoming paths instead of outgoing paths in Definition 4.9 we get the so-called status index

$$X_{\text{Sta}}^{V,G} := \left(\sum_{k=1}^{\infty} (\alpha A^T)^k \right) \cdot \mathbf{1}.$$

that can be computed like in Procedure approximateInfluence by using a network multiplication kernel for transposed multiplication.

4.3.2 Vector iteration

In the following we discuss the computation of indices that are defined on an equilibrium condition for a matrix. To illustrate the approximation of these values we introduce the idea in general and apply it then for the concrete indices.

Definition 4.10 (Eigenvalues and eigenvectors)
A scalar $\lambda \in \mathbb{R}$ is called an *eigenvalue* of a matrix $A \in \mathbb{R}^{n \times n}$, if there exists a vector $x \in \mathbb{R}^n - \{0\}$ such that the equation

$$A \cdot x = \lambda \cdot x$$

is satisfied. The vector x is called an *eigenvector*.

While there are many methods to solve the eigenvalue problem under several circumstances[3], we discuss a simple method that integrates well with the network multiplication from Chapter 3.3.2.

Definition 4.11 (Vector iteration)
Let $A \in \mathbb{R}^{n \times n}$ and $y \in \mathbb{R}^n$. The vector iteration (power method, von-Mises iteration) is defined by the sequence

$$y^0 := y \text{ and } y^k := \frac{Ay^{k-1}}{\|Ay^{k-1}\|}.$$

To illustrate the idea of the vector iteration, we can investigate its behavior for a matrix $A \in \mathbb{R}^{n \times n}$ with n eigenvalues

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|.$$

Lemma 4.7
If we apply the iteration from Definition 4.11 to a vector that can be represented by $y^0 = \sum_{i=1}^{n} \alpha_i v_i$ with $\alpha_1 \neq 0$, where $\{v_1, \ldots, v_n\}$ are the corresponding eigenvectors. Then we get for $k \to \infty$

$$y^k \longrightarrow \frac{v_1}{\|v_1\|}.$$

[3]For an introduction to numerical matrix computation see for example (Golub and van Loan, 1996 [12]).

Proof: After k iterations we get

$$\begin{aligned} A^k \cdot y^0 &= A^k \cdot (\sum_{i=1}^{n} \alpha_i v_i) = \sum_{i=1}^{n} \alpha_i A^k v_i = \sum_{i=1}^{n} \alpha_i \lambda_i^k v_i \\ &= \alpha_1 \lambda_i^k v_1 + \sum_{i=2}^{n} \alpha_i \lambda_i^k v_i = \lambda_i^k \cdot \left(\alpha_1 v_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1} \right)^k v_i \right). \end{aligned}$$

Because of the assumption $|\lambda_1| > |\lambda_i|$ for $1 < i \leq n$ this goes for large k to $A^k \cdot y^0 \longrightarrow \lambda_i^k \alpha_1 v_1$. This completes the proof, because the resulting vectors y^k are scaled after each iteration step.

We can use the vector iteration to approximate the associated eigenvector of the so-called *dominant eigenvalue*, if the vector iteration converges. The assumption $\alpha_1 \neq 0$ is not so important for numeric computation, because we can expect a sufficient rounding error in order to get a proper vector during the iteration. More problematic can be the convergence rate, because it depends on the fraction $\frac{|\lambda_2|}{|\lambda_1|}$. Each iteration step takes time in $\mathcal{O}(|E|)$.
To apply this method for approximation we need a proper abort criterion. To demonstrate the vector iteration we will use the easy condition $|\|Ay^{k-1}\|_2 - \|Ay^{k-2}\|_2| < \varepsilon$ and remark that this can be replaced by better possibilities.

Definition 4.12
A network $G = (V, E)$ is called *strongly connected*, if for each pair of vertices $s, t \in V$ a path from s to t exists. An inclusion-maximal strongly connected sub-network is said to be a *strongly connected component*.
A network is *bipartite*, if we can decompose V into two disjoint sets so that no two vertices within the same set are adjacent.

The following version of the Perron-Frobenius theorem for graphs (see [11]) gives a condition, so that the vector iteration converges actually.

Theorem 4.1 (Perron-Frobenius)
If A is the adjacency matrix of a strongly connected graph G, then there is an ordering

$$\lambda_1 \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$$

of its eigenvalues such that λ_1 is real and simple and $-\lambda_1$ is an eigenvalue of A if and only if G is bipartite. Moreover, the entries of a non-zero eigenvector associated with λ_1 are either all negative or all positive real numbers.

An immediately application of the vector iteration method is the approximation of the Eigenvector index originally defined by Bonacich [4].

Procedure `approximateEigenvectorIndex`

Input : Strongly connected network $G = (V, E)$
Real value $\varepsilon > 0$
Data : Multiplication kernel
Real value ρ, ρ_{-1}
Output: Vertex vector X
begin
  foreach vertex $v \in V$ **do** assign kernel.Y$[v] \leftarrow \frac{1}{|V|}$;
  assign $\rho \leftarrow 1$;
  repeat
    call method kernel.`init_loop()`;
    call Procedure doEdgeLoop(kernel);
    assign $\rho_{-1} \leftarrow \rho$;
    assign $\rho \leftarrow \|$kernel.Y$\|$;
    foreach vertex $v \in V$ **do** assign kernel.Y$[v] \leftarrow \frac{\text{kernel.Y}[v]}{\rho}$;
  until $|\rho - \rho_{-1}| < \varepsilon$;
  copy X $\leftarrow$ kernel.Y;
end

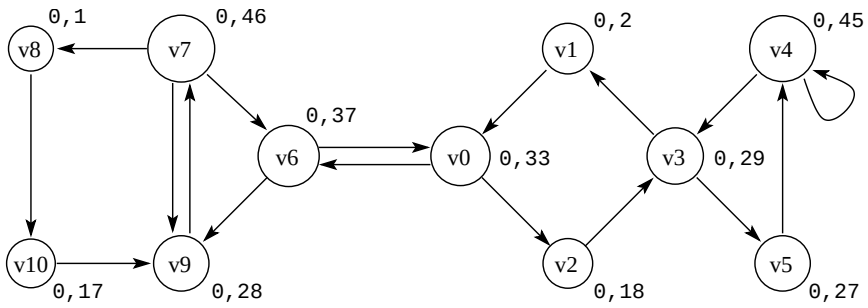

Figure 4.6: Approximated eigenvector index (rounded values) for a strongly connected variation of the demonstration network. The approximation with $\varepsilon = 10^{-6}$ stops after 77 iterations.

Definition 4.13
Let $G = (V, E)$ be a strongly connected network with the adjacency matrix A and $\rho(G)$ its dominant eigenvalue. The Eigenvector index $X_{\text{Eig}}^{V,G}$ of G is defined by the unique solution of

$$X_{\text{Eig}}^{V,G} = \frac{1}{\rho(G)} \cdot A \cdot X_{\text{Eig}}^{V,G}.$$

Additionally we require $X_{\text{Eig}}^{V,G}$ to be standardized according to $\|X_{\text{Eig}}^{V,G}\|_2 = 1$.

We can approximate the eigenvector index of a network with the vector iteration method like in Procedure approximateEigenvectorIndex.

4.3.3 Hubs & Authorities index

Another application for the vector iteration method is the approximation of the Hubs & Authorities index defined by Kleinberg [18].

Definition 4.14 (Kleinberg 1999)
Let $G = (V, E)$ be a network with the adjacency matrix A.
Let B be the network with the symmetric adjacency matrix AA^T (the so-called *bibliographic coupling* of G). The *Hub index* is defined by

$$X_{\text{Hub}}^{V,G} := X_{\text{Eig}}^{V,B}.$$

Let C be the network with the symmetric adjacency matrix A^TA (the so-called *Co-citation* of G). The *Authority index* $X_{\text{Aut}}^{V,G}$ is defined by

$$X_{\text{Aut}}^{V,G} := X_{\text{Eig}}^{V,C}.$$

The actual approximation of both indices is simple because of the recursive dependencies

$$X_{\text{Hub}}^{V,G} = A \cdot X_{\text{Aut}}^{V,G} \quad \text{and} \quad X_{\text{Aut}}^{V,G} = A^T \cdot X_{\text{Hub}}^{V,G}$$

and can be done like in procedure approxHubAuthority.
If G contains at least on edge, then neither B nor C are the adjacency matrix of a bipartite network. Because of the symmetry of B and C, it follows that all their components are strongly connected.

Vertices with a high hub value point to vertices with high authority values, while good authorities are pointed to by good hubs (see Figure 4.7).

Procedure `approxHubAuthority`

Input : Network $G = (V, E)$
Real $\varepsilon > 0$
Data : Multiplication kernel and transposed multiplication kernel^T
Real values $\rho_{\mathrm{H}}, \rho_{\mathrm{H}-1}, \rho_{\mathrm{A}}, \rho_{\mathrm{A}-1}$
Output: Vertex vectors $\mathsf{X}_{\mathrm{H}}, \mathsf{X}_{\mathrm{A}}$
begin
  foreach vertex $v \in V$ **do**
    assign $\mathsf{kernel.Y} \leftarrow \frac{1}{|V|}$ and $\mathsf{kernel}^T\mathsf{.Y} \leftarrow \frac{1}{|V|}$;
  assign $\rho_{\mathrm{H}} \leftarrow 1$ and $\rho_{\mathrm{A}} \leftarrow 1$;
  repeat
    copy $\mathsf{kernel.Y} \leftarrow \mathsf{kernel}^T\mathsf{.Y}$; call method kernel.`init_loop()`;
    call procedure doEdgeLoop(kernel);
    assign $\rho_{\mathrm{H}-1} \leftarrow \rho_{\mathrm{H}}$; assign $\rho_{\mathrm{H}} \leftarrow \|\mathsf{kernel.Y}\|$;
    copy $\mathsf{kernel}^T\mathsf{.Y} \leftarrow \mathsf{kernel.Y}$; call method kernel^T.`init_loop()`;
    call procedure doEdgeLoop(kernel^T);
    assign $\rho_{\mathrm{A}-1} \leftarrow \rho_{\mathrm{A}}$; assign $\rho_{\mathrm{A}} \leftarrow \|\mathsf{kernel}^T\mathsf{.Y}\|$;
    foreach vertex $v \in V$ **do**
      assign $\mathsf{kernel.Y}[v] \leftarrow \frac{\mathsf{kernel.Y}[v]}{\rho_{\mathrm{H}}}$; assign $\mathsf{kernel}^T\mathsf{.Y}[v] \leftarrow \frac{\mathsf{kernel}^T\mathsf{.Y}[v]}{\rho_{\mathrm{A}}}$;
  until $|\rho_{\mathrm{H}} - \rho_{\mathrm{H}-1}| < \varepsilon$ and $|\rho_{\mathrm{A}} - \rho_{\mathrm{A}-1}| < \varepsilon$;
  copy $\mathsf{X}_{\mathrm{H}} \leftarrow \mathsf{kernel.Y}$ and $\mathsf{X}_{\mathrm{A}} \leftarrow \mathsf{kernel}^T\mathsf{.Y}$;
end

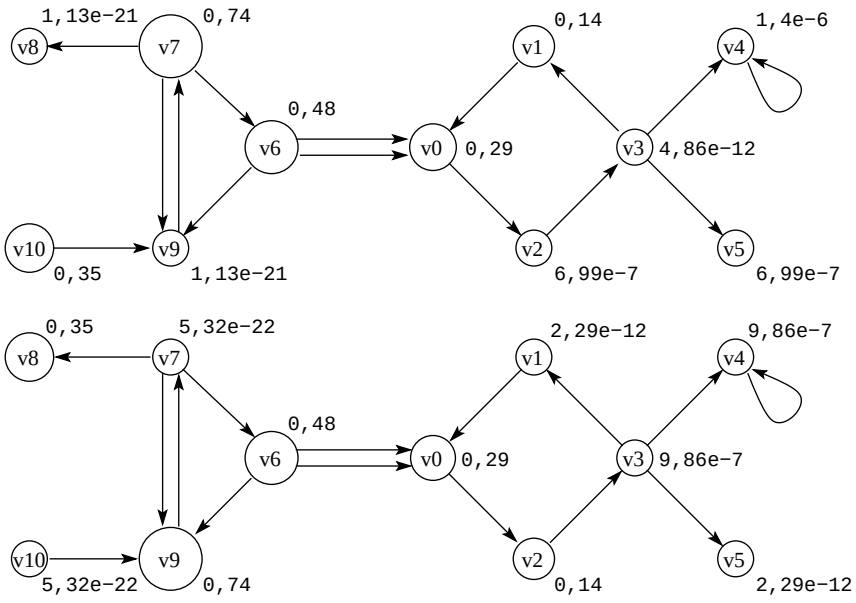

Figure 4.7: Approximated hubs and authority index (rounded values) for the demonstration network. The approximation with $\varepsilon = 10^{-6}$ stops after 32 iterations.

4.3.4 PageRank index

The PageRank index defined by Page and Brin[22] is used by the search engine *Google*. To get a relevance measure for the importance of each web site the PageRank simulates the presence behavior of an internet user with a random walk by introducing a transition probability $\omega \in [0,1]$.

Definition 4.15 (PageRank index)
For a network $G = (V, E)$ with adjacency matrix A we define the *inverse outdegree matrix*

$$D_G = (d_{v,w})_{v,w \in V} := \begin{cases} \frac{1}{\overrightarrow{\deg}_G(v)} & \text{if } v = w \text{ and } \overrightarrow{\deg}_G(v) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and the *(outdegree) normalized adjacency matrix* $M_G := D_G \cdot A$.
For $\omega \in [0,1]$ is the *PageRank* index defined as the unique solution of

$$\overrightarrow{X}^{V,G}_{\text{Pag}} = (1-\omega) \cdot M_G^T \cdot \overrightarrow{X}^{V,G}_{\text{Pag}} + \frac{\omega}{n} \cdot \mathbf{1}.$$

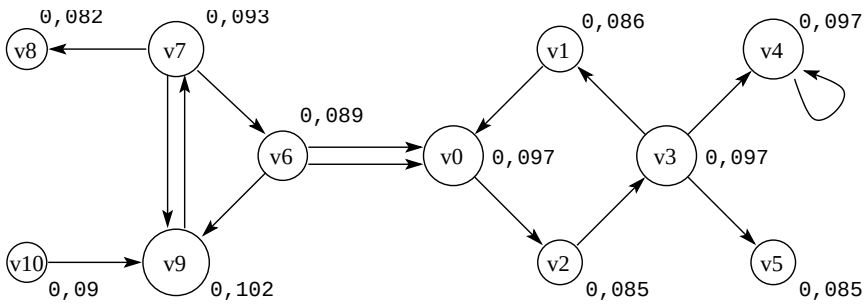

Figure 4.8: Approximated page rank index (rounded values) for the demonstration network. The approximation with $\varepsilon = 10^{-6}$ stops after 9 iterations.

By applying a vector iteration we get the stationary probabilities of presence for the user on all sites. To find the solution with the vector iteration method we have to transform the equilibrium condition from the definition to

$$\begin{aligned} \overrightarrow{X}^{V,G}_{\text{Pag}} &= (1-\omega) \cdot M_G^T \cdot \overrightarrow{X}^{V,G}_{\text{Pag}} + \frac{\omega}{n} \cdot \mathbf{1} \\ &= (1-\omega) \cdot (D_G \cdot A)^T \cdot \overrightarrow{X}^{V,G}_{\text{Pag}} + \frac{\omega}{n} \cdot \mathbf{1} \\ &= A^T \cdot \underbrace{(1-\omega) \cdot D_G}_{=:S} \cdot \overrightarrow{X}^{V,G}_{\text{Pag}} + \frac{\omega}{n} \cdot \mathbf{1}. \end{aligned}$$

The transition probability ω assures that the vector iteration has a good convergence rate and makes the underlying matrix strongly connected and non-bipartite.

Procedure `approxPageRank`

Input : Network $G = (V, E)$
Real values $\varepsilon > 0$ and $\omega \in [0, 1]$
Data : Transposed multiplication kernel^T
Vertex vector S and real value ρ
Output: Vertex vector X
begin
  foreach vertex $v \in V$ **do**
    assign $\mathsf{X}[v] \leftarrow \frac{1}{|V|}$;
    if $\overrightarrow{d}_G(v) > 0$ **then**
      assign $S[v] \leftarrow \frac{1-\omega}{\overrightarrow{d}_G(v)}$;
    else assign $S[v] \leftarrow 0$;
  repeat
    foreach vertex $v \in V$ **do** assign $\mathsf{kernel}^T.\mathsf{Y}[v] \leftarrow S[v] \cdot \mathsf{X}[v]$;
    call method kernel^T`.init_loop();`
    call procedure doEdgeLoop(kernel^T);
    foreach vertex $v \in V$ **do** assign $\mathsf{kernel}^T.\mathsf{Y}[v] \leftarrow \mathsf{kernel}^T.\mathsf{Y}[v] + \frac{\omega}{|V|}$;
    assign $\rho \leftarrow \|\mathsf{X} - \mathsf{kernel}^T.\mathsf{Y}\|$;
    copy $\mathsf{X} \leftarrow \mathsf{kernel}^T.\mathsf{Y}$;
  until $\rho < \varepsilon$;
end

4.4 Edge indices

To get corresponding index values for the edges of a network $G = (V, E)$ we can define them by the vertex indices for the edge-network $\mathcal{E}(G)$ given in Definition 3.1. Because $\mathcal{E}(G)$ can become very large also for networks of acceptable size, we need methods to do the index computation by using the implicit representation like in Chapter 3.3.1.
We have already seen that we can perform loops on the vertices and edges of $\mathcal{E}(G)$ with memory requirements linear in the size of G and we therefore get the multiplication of an edge vector with the network $\mathcal{E}(G)$. This means that we can compute matrix-based edge indices also for large networks, while only the running time is linear in the size of $\mathcal{E}(G)$.
To compute distance-based edge indices on $\mathcal{E}(G)$ we cannot adapt the BFS kernel, because we have to store edge information. Because of Lemma 4.5 we can compute the distances of all vertices from a given vertex r by multiplying the network with a vector. The visitor in class 16 computes these distances, if we increment the counter $\mathsf{visitor}.k$ before each run of the edge loop. We have to repeat the edge loop as long as we have the possibility to find a new vertex. After one edge loop that did not find a new vertex we are sure that we found all vertices.

Thus we are able to compute distances for $\mathcal{E}(G)$ even for large networks by transferring this method to the loops on incident edge pairs. Visitors for such a kernel are formulated similar to the BFS variant.

Class 16: Multiplication visitor for distance from root

Data : Counter $k \in \mathbb{N}$
Vertex vector dist for distance from root
Flag found_new_vertex

Method `template_init_visitor(` vertex r`)` **begin**
  foreach $v \in V$ **do** assign dist$[v] \leftarrow \infty$;
  assign $k \leftarrow 0$;
  assign dist$[r] \leftarrow k$;
end

Method increment_counter() **begin**
  assign $k \leftarrow k + 1$;
  assign found_new_vertex $\leftarrow false$;
end

Method `template_visit_edge(` loop kernel `)` **begin**
  if dist$[\mathsf{s}_{kernel}] = \infty$ **then**
    assign dist$[\mathsf{s}_{kernel}] \leftarrow k$ and found_new_vertex $\leftarrow true$;
end
Method `is_valid()` **begin**
  return whether found_new_vertex is true or not;
end

Remark 4.8
We split the distance visitor from the kernel here only to show the underlying algorithm more clearly. Because of the importance for the distance computation the task of this visitor is typically implemented inside the kernel.

Chapter 5

Statistical analysis

The evaluation of indices for large networks is not possible by looking at each single value. Thus we have to summarize these data sets with methods from discrete statistics. In this chapter we discuss the statistical

- visualization,
- description and
- comparison

of vertex indices[1] $X \in \mathbb{R}^V$.

Example 5.1 (Escherichia coli)
We will demonstrate the statistical methods from this chapter for the metabolic network [3] of the bacteria *Escherichia coli* with 2275 vertices and 5763 (directed) links. For that purpose we analyze the closeness index $\overrightarrow{X}^{V,G}_{\mathrm{Clo}}$ and compare it then with the betweenness index $\overrightarrow{X}^{V,G}_{\mathrm{Bet}}$.

For a given finite network $G = (V, E)$ we get the (discrete) probability space (V, P) by the uniform distribution $P(v) = \frac{1}{|V|}$.
Because each vertex index $X \in \mathbb{R}^V$ maps the set of vertices to the real numbers, we can understand a network index as a discrete random variable

$$X : V \longrightarrow \mathbb{R}$$

and notate it by $X = (x_v)_{v \in V} \in \mathbb{R}^V$.

5.1 Visualization

In order to get an impression about the whole network index we can display it graphically in different ways. First of all we can display the raw discrete

[1] It is easy to see that the statistical analysis of edge indices is exactly the same by using an EdgeLoop.

random variable $X : V \longrightarrow \mathbb{R}$ by just plotting the points $(v, X(v))$ for all vertices into the two-dimensional plane. The disadvantage of this approach is that it depends heavily on the order of the samples. We can easily eliminate this order by plotting the increasingly sorted values like in Figure 5.1.

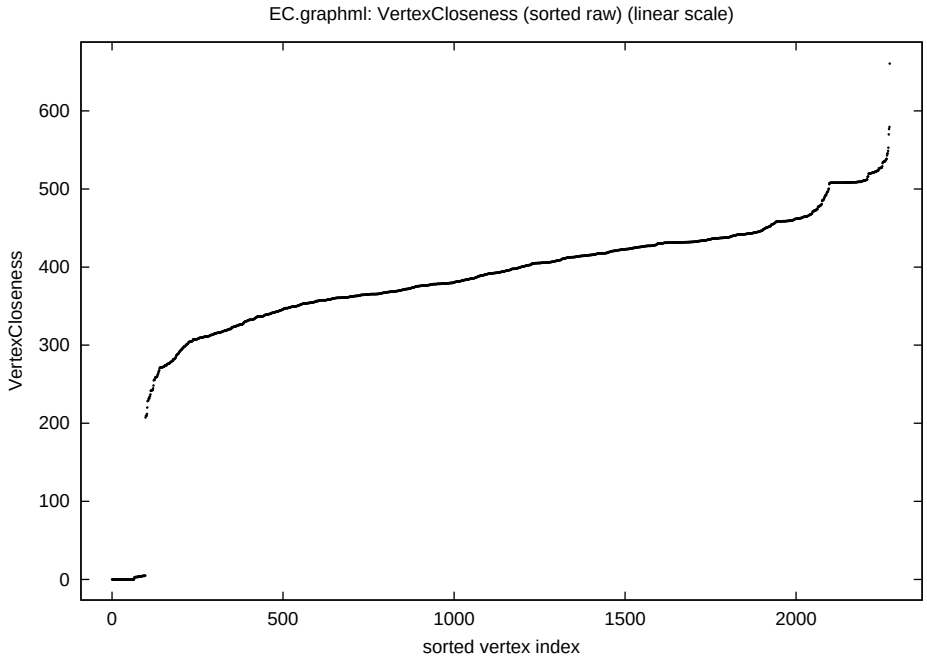

Figure 5.1: The closeness index for the metabolic network of Escherichia coli with increasingly sorted values.

Another approach is to compute the induced *probability mass function*

$$P^X : \mathbb{R} \supsetneq X(V) \longrightarrow [0, 1]$$

defined by

$$P^X(x) := \frac{|\{v \in V : x_v = x\}|}{|V|}.$$

Because of

- $P^X(x) > 0$ holds by definition for all $x \in X(V)$ and
- $\sum_{x \in X(V)} P^X(x) = \sum_{x \in X(V)} \frac{|\{v \in V : x_v = x\}|}{|V|} = \frac{1}{|V|} \sum_{v \in V} 1 = 1$

P^X is actually a probability mass function and reflects the "frequency" of the resulting values from the image of X. We can compute the probability mass function P^X of a network index X with time in $\mathcal{O}(|V|)$ by using the the values computed by the vertex loop visitor from Class 17.

Class 17: Vertex loop visitor for probability masses

Data : Network $G = (V, E)$
Vertex vector X
Real k
Map $f : \mathbb{R} \to \mathbb{R}$

Method `template_init_visitor` () **begin**
  assign $k \leftarrow 0$;
  foreach $x \in \mathbb{R}$ **do** assign $f(x) \leftarrow 0$;
end

Method `template_visit_vertex` (Vertex loop kernel) **begin**
  assign $k \leftarrow k + 1$;
  assign $f(\mathsf{X}[\mathsf{kernel.current}]) \leftarrow f(\mathsf{X}[\mathsf{kernel.current}]) + 1$;
end

The mapping $\mathbb{R} \longrightarrow \mathbb{R}$ can be realized efficiently by the container type *map* with keys and values in the real numbers. The frequency plot for Example 5.1 looks like in Figure 5.2.

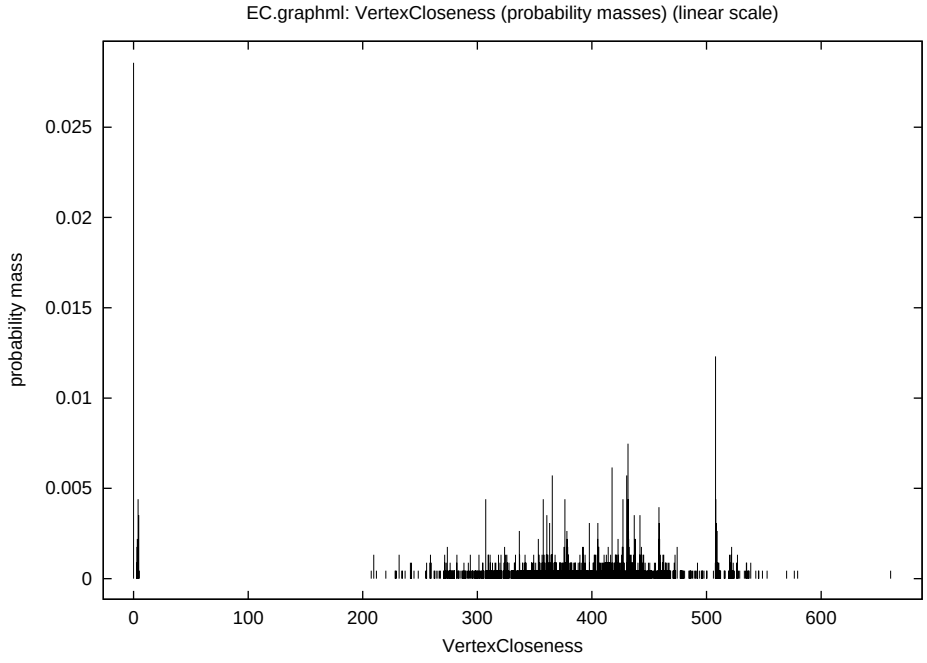

Figure 5.2: The probability mass function of the closeness index for the metabolic network of Escherichia coli.

To visualize the probability mass function of a given network index in another way we can use the *cumulative distribution function*

$$F : X(V) \longrightarrow [0, 1]$$

defined by

$$F(x) := \sum_{y \in X(V): y \leq x} P^X(y) = \frac{|\{v \in V : x_v \leq x\}|}{|V|}.$$

We can find the significant value of a discrete random variable X, i.e. the values with $P^X(x) \neq 0$ for $x \in \mathbb{R}$, by iterating over $|V|$. Sorting a set and eliminating multiple elements can be realized in $\mathcal{O}(k \log(k))$ time using standard sorting algorithms like merge sort. Now we can compute the values of the cumulative probability function F by summing up the probability masses sequentially from $\min(X)$ to $\max(X)$ in linear time.

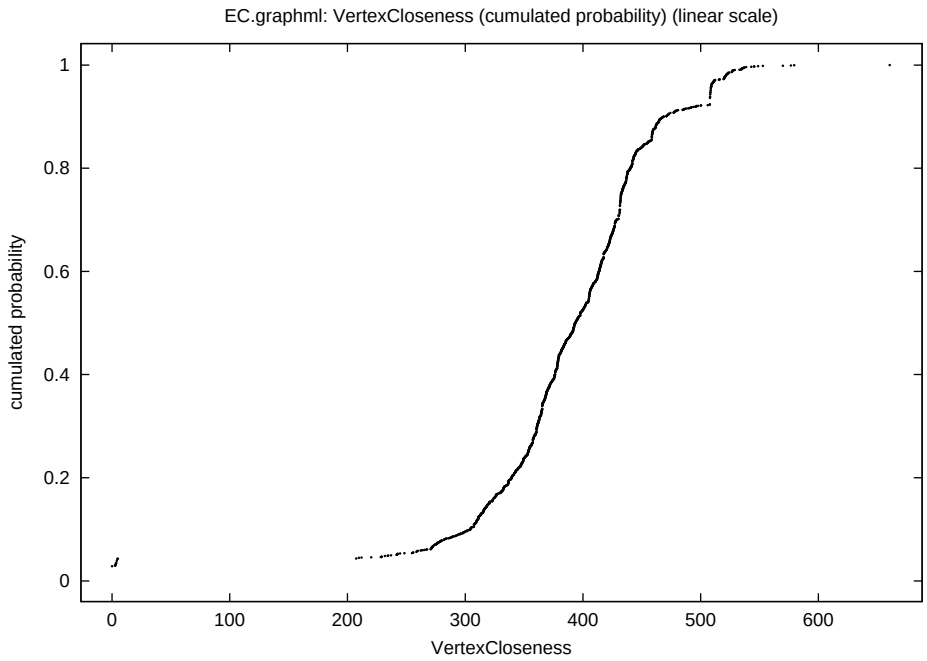

Figure 5.3: The cumulative distribution function of the closeness index for the metabolic network of Escherichia coli.

Therefore we can generate a graphical representation of the cumulative probability function by plotting the points $(x, F(x))$ into the two-dimensional plane like in Figure 5.3 with time in $\mathcal{O}(k \log(k))$.

The graphical representation of the probability mass function does not reflect the characteristic properties of many network indices, especially if they have pairwise different values. We can extend P^X to a probability distribution on the set of all real intervals of the form $[a, b) \subset \mathbb{R}$ with $a < b$ by

$$P^X\big([a,b)\big) := \sum_{a \le x < b} P^X(x) = \frac{|\{v \in V : a \le x_v < b\}|}{|V|}.$$

The probability of a given interval $[a, b)$ can be computed by using the values from the vertex loop visitor in Class 18 with time in $\mathcal{O}(|V|)$.

Class 18: intervalPropabilityVisitor for vertex or edge loop

Data : Vertex vector X
Interval bounds $a, b \in \mathbb{R}$
Map $f : \mathbb{R} \longrightarrow \mathbb{R}$

Method `template_init_visitor` () **begin**
  assign $f(x) \leftarrow 0$ for all $x \in \mathbb{R}$;
end

Method `template_visit_vertex` (Vertex loop kernel) **begin**
  if $a \le$ X[kernel.current] $< b$ **then**
    assign $f \leftarrow f + 1$;
end

Now we can visualize the density of X as a *histogram* of P^X by covering range(X) with disjoint intervals of the form $[a, b)$ and plotting for each interval $P^X\big([a,b)\big)$ at the center point $\frac{a+b}{2}$.

Again there are two disadvantages. First we cannot guarantee that we can generate the histogram for P^X in linear time, because we have to compute P^X for each interval in linear time. On the other hand it is important to realize that in general the resulting histogram depends heavily on choosing the intervals.
Now we discuss how to compute the histogram values for a given network index in linear time under the restriction that we use an equidistant partition of $X(V)$.
We can cover range(X) by m disjoint intervals[2] of length

$$l := \frac{\text{range}(X)}{m-1}$$

and center points

$$p_j := \min(X) + j \cdot l \text{ for } j \in \{0, \ldots, m-1\}.$$

[2]It is a good heuristic rule to choose $m := \ulcorner \sqrt{k} \urcorner$.

This leads to the set of disjoint intervals

$$\{[\min(X) - \frac{l}{2}, \min(X) + \frac{l}{2}), \ldots, [\max(X) - \frac{l}{2}, \max(X) + \frac{l}{2})\}$$

that covers range(X) by construction and we can compute for $x \in \text{range}(X)$ the index j of the corresponding interval by

$$j(x) = \llcorner \frac{x - \min(X)}{l} + \frac{1}{2} \lrcorner.$$

Class 19: histogramVisitor for vertex or edge loop

Data : Vertex vector X
Map $h : \mathbb{N} \to \mathbb{R}$
$j \in \mathbb{N}$

Method `template_init_visitor` () **begin**
  assign $h(x) \leftarrow 0$ for all$x \in \mathbb{R}$;
end

Method `template_visit_vertex` (Vertex loop kernel) **begin**
  assign $j \leftarrow \llcorner \frac{\text{X[kernel.current]} - \min(X)}{l} + \frac{1}{2} \lrcorner$;
  assign $h(j) \leftarrow h(j) + 1$;
end

Now we are able to generate an equidistant histogram in linear time by using the vertex loop visitor from Class 19 and plotting the resulting points $(p_i, h(i))_{0 \leq i \leq m}$ on the two-dimensional plane like in Figure 5.4.

5.2 Description

Beside the graphical representations above we can describe a given network index shortly by using statistical values to unhide some characteristic properties by real values. The advantage of such values is that they can easily be checked automatically. We can compute the maximum and the minimum of a given network index X by a visitor for a vertex loop. Because only real values have to be stored and the operations inside the template method can be realized in constant time, we can see that the computation of

- the maximum $\max(X) := \max(x_i)_{1 \leq i \leq k}$,
- the minimum $\min(X) := \min(x_i)_{1 \leq i \leq k}$ and
- the range[3] $\text{range}(X) := \max(X) - \min(X)$

[3]We will use range(X) also to denote the interval $[\min(X), \max(X)]$.

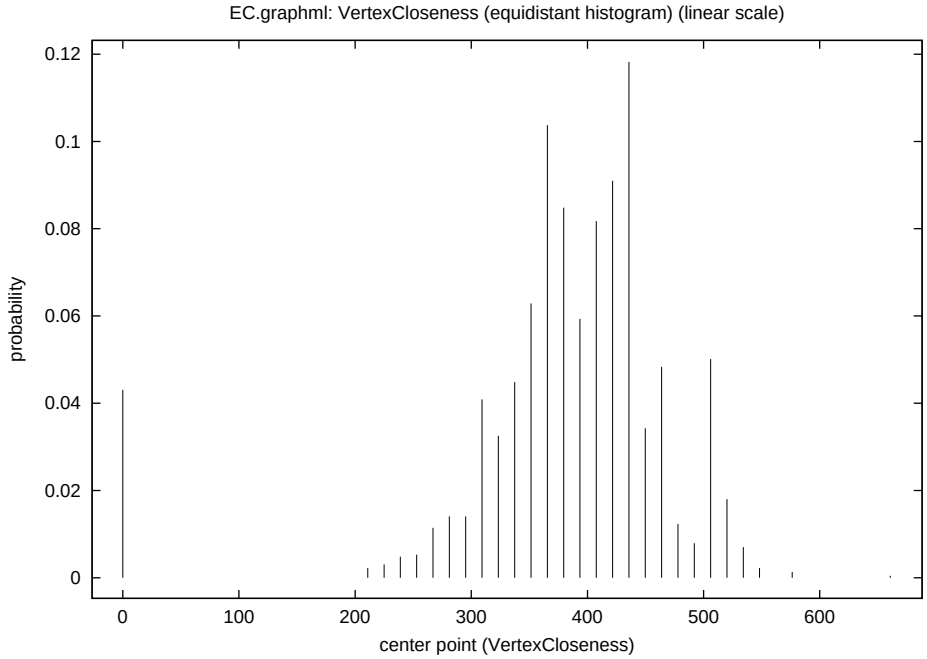

Figure 5.4: Histogram of the closeness index for the metabolic network of Escherichia coli with 48 equidistant intervals.

of a network index X can be done with time in $\mathcal{O}(|V|)$.

We can compute the expected value of a vertex index

$$\mathbb{E}(X) := \sum_{x \in X(V)} x \cdot P^X(x) = \sum_{x \in X(V)} x \cdot \frac{|\{v \in V : X(v) = x\}|}{|V|} = \frac{1}{|V|} \sum_{v \in V} X(v)$$

with time in $\mathcal{O}(|V|)$ by the average of all values x_v using the vertex loop visitor from Class 20.

The expected value can also be utilized to get an idea "how much" the index values $X(v)$ are spread around the expected value $\mathbb{E}(X)$. A first approach is to compute the *mean absolute deviation*

$$\mathbb{E}\big(|X - \mathbb{E}(X)|\big) = \sum_{x \in X(V)} |x - \mathbb{E}(X)| \cdot P^X(x)$$

to get the average difference of $X(v)$ from $\mathbb{E}(X)$. We can do this again by using the vertex loop visitor from class 20 with the vertex vector of the absolute differences from $\mathbb{E}(X)$.

Class 20: expectationVisitor for vertex loop

Data : Vertex vector X
Expectation $\bar{x} \in \mathbb{R}$
Real values sum and k

Method `template_init_visitor` () **begin**
  assign $\bar{x} \leftarrow 0$, sum $\leftarrow 0$ and k $\leftarrow 0$;
end

Method `template_visit_vertex` (Vertex loop kernel) **begin**
  assign sum $\leftarrow$ sum $+$X[kernel.current];
  assign $k \leftarrow k + 1$;
  assign $\bar{x} \leftarrow \frac{\text{sum}}{k}$;
end

Although the average distance of the values from the expected value is a useful parameter, it is more popular to investigate the standard deviation of a data set

$$\sigma(X) := \sqrt{\mathbb{V}\text{ar}(X)}$$

as the square root of the variance of the data set

$$\mathbb{V}\text{ar}(X) := \mathbb{E}\big((X - \mathbb{E}(X))^2\big) = \mathbb{E}\big(X^2 - \mathbb{E}(X)^2\big) = \mathbb{E}(X^2) - \mathbb{E}(X)^2.$$

This means that we only have to compute the value $\mathbb{E}(X^2)$ in the same vertex loop as the expected value $\mathbb{E}(X)$ to get the variance and the standard deviation of a vertex index.

Now we can describe the statistical properties of the closeness index for the metabolic network of Escherichia coli by:

- $\min(\overrightarrow{X}^{V,G}_{\text{Clo}}) = 0.0$,
- $\text{range}(\overrightarrow{X}^{V,G}_{\text{Clo}}) = \max(\overrightarrow{X}^{V,G}_{\text{Clo}}) = 660.654$,
- $\mathbb{E}(\overrightarrow{X}^{V,G}_{\text{Clo}}) = 380.284$,
- $\mathbb{V}\text{ar}(\overrightarrow{X}^{V,G}_{\text{Clo}}) = 10122.4$ and
- $\sigma(\overrightarrow{X}^{V,G}_{\text{Clo}}) = 100.61$.

5.3 Comparison

We can display the correlation of two different indices $X, Y \in \mathbb{R}^G$ of the same network by plotting the pair $(X(v), Y(v)) \in \mathbb{R}^2$ for each vertex $v \in V$ into the two-dimensional plane like in Figure 5.5.

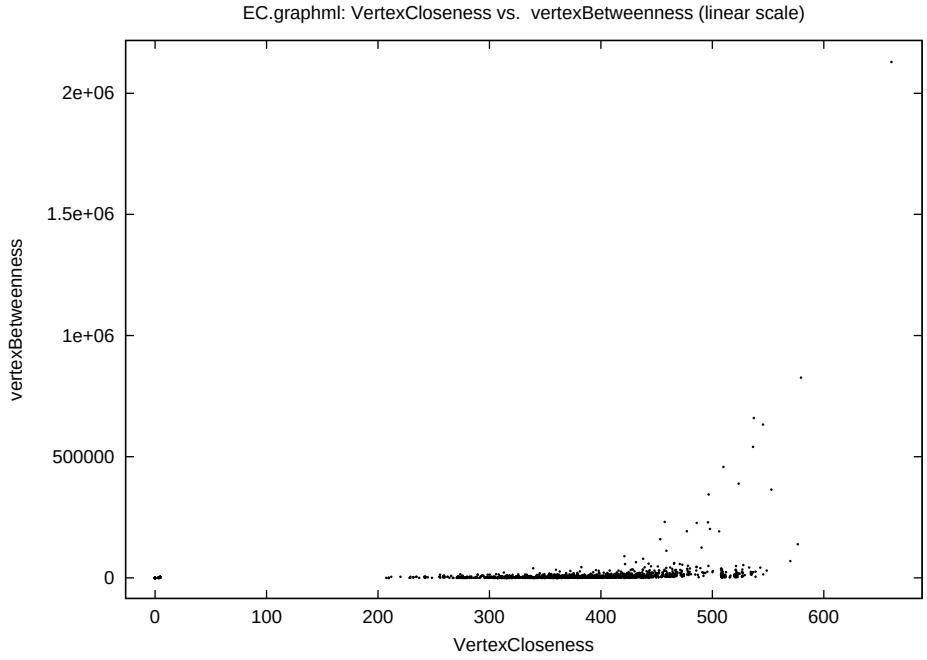

Figure 5.5: Comparison plot with the closeness index against the betweenness index for the metabolic network of Escherichia coli.

Now we can measure the correlation of the two data sets statistically by the *correlation coefficient*

$$\mathbb{C}\mathrm{or}(X, Y) := \frac{\mathbb{C}\mathrm{ov}(X, Y)}{\sigma(X) \cdot \sigma(Y)}$$

where the numerator is just the covariance

$$\begin{aligned} \mathbb{C}\mathrm{ov}(X, Y) &:= \mathbb{E}\big((X - \mathbb{E}X)(Y - \mathbb{E}Y)\big) \\ &= \mathbb{E}\big(XY - X\mathbb{E}(Y) - \mathbb{E}(X)Y + \mathbb{E}(X)\mathbb{E}(Y)\big) \\ &= \mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y) - \mathbb{E}(X)\mathbb{E}(Y) + \mathbb{E}(X)\mathbb{E}(Y) \\ &= \mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y). \end{aligned}$$

This means that we have to compute only $\mathbb{E}(XY)$ and we can do this together with $\mathbb{E}(X)$ and $\mathbb{E}(Y)$ in the same vertex loop.

We can express the correlation of the closeness index and the betweenness index numerically by

$$\mathbb{C}\mathrm{or}(X_{\mathrm{Clo}}, X_{\mathrm{Bet}}) = 0.163853.$$

Remark 5.1 (Properties of the correlation coefficient)
The following elementary properties hold for the correlation coefficient:

- $|\,\mathbb{C}\mathrm{or}(X,Y)| \leq 1$ and
- $|\,\mathbb{C}\mathrm{or}(X,Y)| = 1 \Leftrightarrow \exists a, b \in \mathbb{R} : P(Y = aX + b) = 1$,
  i.e. X, Y are linearly correlated.

Proofs for this properties can be found in the standard literature (see for example (Henze, 2000 [15])).

Chapter 6

Heuristics for connectivity components

We know from Chapter 4 that the computation of distance-based indices like the closeness index and the betweenness index can be done with time in $\mathcal{O}(|V| \cdot |E|)$. Because these time requirements can become unacceptable for large networks, we present a method to speed up these computations for networks with a special structure.

Definition 6.1 (Bi-edge connectivity)
A connected network is called *bi-edge connected*, if there exists no single edge, whose removal disconnects the network. For a network an inclusion-maximal bi-edge connected sub-network is said to be a *bi-edge component*. An edge that violates the bi-edge connectivity is called a *bridge edge*.

For algorithms to detect bi-edge components with Depth-First search (DFS) in linear time see (Tarjan, 1972 [27]). The algorithms presented there can be implemented by visitors for a DFS kernel like in Chapter 3.3.3.

6.0.1 Closeness

We have seen that we can compute the closeness index $\overrightarrow{X}^{V,G}_{\mathrm{Clo}}(v)$ by a fraction of

$$\overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v) = \sum_{t \in \overrightarrow{rv}_G(v)} 1 \quad \text{and} \quad \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v) = \sum_{t \in \overrightarrow{rv}_G(v)} d_G(v,t)$$

with

$$\overrightarrow{rv}_G(v) := \{t \in V : t \neq v \text{ and } \exists \text{ path from } v \text{ to } t\}.$$

We know from Chapter 4.2.2, that we can do this computation for one vertex $v \in V$ with time in $\mathcal{O}(|E|)$ by using BFS visitors. Now we show how to speed up the computation of the closeness index heuristically, if bridge edges exist.

We assume that there is a bridge edge e. Without loss of generality we can look at a network $G = (V, E)$ consisting of two connected components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ together with the bridge edge $e = (v_1, v_2)$ from $v_1 \in V_1$ to $v_2 \in V_2$ as illustrated in Figure 6.1.

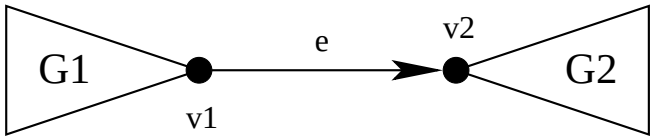

Figure 6.1: Two components connected by a bridge edge.

Lemma 6.1
Assume $v \in V_2$. We can compute

$$\overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v) \text{ and } \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v)$$

without leaving the component G_2 with time in $\mathcal{O}(|E_2|)$.

Proof:
The direction of e guarantees that there is no path to a $t \in V_1$.

Now we can concentrate on the vertices of G_1 and assume $v \in V_1$.

Lemma 6.2
Assume $v \in V_1$ and $t \in V_2$. If there is a path from v to t, it has to use the bridge edge $e = (v_1, v_2)$ and

$$d_G(v, t) = d_{G_1}(v, v_1) + 1 + d_{G_2}(v_2, t)$$

holds.

Proof:

- Assume there is a path from $v \in V_1$ to $t \in V_2$ that does not use e. Then we can construct an undirected cycle and therefore e cannot be a bridge edge between G_1 and G_2.
- Each shortest path from $v \in V_1$ to $t \in V_2$ has to use the bridge edge $e = (v_1, v_2)$. Therefore we have to use a shortest path from v to v_1 in G_1 first. Then we have to expand this to a shortest path to v_2 by using the edge e. Finally we can connect this with a shortest path from v_2 to t in G_2.

Notation 6.1
Because we can disjointly split the vertices of G into $V = V_1 \cup V_2$, we set

$$\overrightarrow{rv}_{G_1}(v) := (\overrightarrow{rv}_G(v) \cap V_1) \quad \text{and} \quad \overrightarrow{rv}_{G_2}(v) := (\overrightarrow{rv}_G(v) \cap V_2).$$

Using the fact that $(\mathbb{P}(V), \cup, \cap, \emptyset, V, ^c)$ is a boolean algebra we are able to express the set of vertices reachable from $v \in V_1$ by

$$\begin{aligned} \overrightarrow{rv}_G(v) &= \overrightarrow{rv}_G(v) \cap V = \overrightarrow{rv}_G(v) \cap (V_1 \cup V_2) \\ &= (\overrightarrow{rv}_G(v) \cap V_1) \cup (\overrightarrow{rv}_G(v) \cap V_2) \\ &= \overrightarrow{rv}_{G_1}(v) \cup \overrightarrow{rv}_{G_2}(v). \end{aligned}$$

Because this is also a disjoint partition we get

$$\overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v) = |\overrightarrow{rv}_{G_1}(v)| + |\overrightarrow{rv}_{G_2}(v)|$$

and

$$\begin{aligned} \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v) &= \sum_{t \in \overrightarrow{rv}_{G_1}(v)} d_G(v,t) + \sum_{t \in \overrightarrow{rv}_{G_2}(v)} d_G(v,t) \\ &= \sum_{t \in \overrightarrow{rv}_{G_1}(v)} d_{G_1}(v,t) + \sum_{t \in \overrightarrow{rv}_{G_2}(v)} (d_{G_1}(v,v_1) + 1 + d_{G_2}(v_2,t)). \end{aligned}$$

Theorem 6.1
Assume $v \in V_1$. If we already know the values

$$|\overrightarrow{rv}_{G_2}(v_2)| = \overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v_2) \quad \text{and} \sum_{t \in \overrightarrow{rv}_{G_2}(v_2)} d_{G_2}(v_2,t) = \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v_2),$$

we can compute

$$\overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v) \quad \text{and} \quad \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v)$$

without leaving the component G_1 with time in $\mathcal{O}(|E_1|)$.

Proof: We have to distinguish two cases:

1st case $\underline{v_1 \notin \overrightarrow{rv}_{G_1}(v)} \Longrightarrow \overrightarrow{rv}_{G_2}(v) = \emptyset$ **:** We can conclude that

$$|\overrightarrow{rv}_{G_2}(v)| = \sum_{t \in \overrightarrow{rv}_{G_2}(v)} (d_{G_1}(v,v_1) + 1 + d_{G_2}(v_2,t)) = 0.$$

Therefore we can compute both values without leaving the component G_1.

2nd case $\underline{v_1 \in \overrightarrow{rv}_{G_1}(v)} \Longrightarrow \overrightarrow{rv}_{G_2}(v) = \overrightarrow{rv}_{G_2}(v_2)$ **:** We can conclude that

$$|\overrightarrow{rv}_{G_2}(v)| = |\overrightarrow{rv}_{G_2}(v_2)|\,.$$

This is true, because a vertex $t \in V_2$ is reachable from $v_2 \in V_2$, if it is reachable from $v \in V_1$ by using the bridge edge e.

To compute the distances sum of all reachable vertices in V_2 we can transform the appropriate expression to

$$\begin{aligned} & \sum_{t \in \overrightarrow{rv}_{G_2}(v_2)} (d_{G_1}(v, v_1) + 1 + d_{G_2}(v_2, t)) \\ = \; & |\overrightarrow{rv}_{G_2}(v_2)| \cdot (d_{G_1}(v, v_1) + 1) + \sum_{t \in \overrightarrow{rv}_{G_2}(v_2)} d_{G_2}(v_2, t). \end{aligned}$$

Therefore we have to compute

$$\overrightarrow{X}^{V,G}_{\mathrm{Rv}}(v) = |\overrightarrow{rv}_{G_1}(v)| + |\overrightarrow{rv}_{G_2}(v_2)|$$

and

$$\begin{aligned} \overrightarrow{X}^{V,G}_{\mathrm{Ds}}(v) \; = \; & \sum_{t \in \overrightarrow{rv}_{G_1}(v)} d_{G_1}(v, t) + \sum_{t \in \overrightarrow{rv}_{G_2}(v_2)} d_{G_2}(v_2, t) \\ & + |\overrightarrow{rv}_{G_2}(v_2)| \cdot (d_{G_1}(v, v_1) + 1)\,. \end{aligned}$$

Corollary 6.1
Theorem 6.1 can be applied recursively to the components G_1 and G_2. Therefore we can compute the closeness index $\overrightarrow{X}^{V,G}_{\mathrm{Clo}}(v)$ for G with time in

$$\mathcal{O}\left(|G| + |V| \cdot |E\left(C_{\max}\right)|\right)$$

where $C_{\max}$ denotes a bi-edge component with the biggest number of edges.

Proof: We can compute $|\overrightarrow{rv}_G(v)|$ and $\sum_{t \in \overrightarrow{rv}_G(v)} d_G(v, t)$ directly for a vertex v with visitors of an outgoing BFS. If we discover a bridge edge target while searching the network, we can omit to insert it into the vertex repository of the BFS kernel. To apply instead Theorem 6.1 we have to start an independent BFS, only if we do not already know its values. Because we can reuse this value later, we have to start for all vertices exactly one BFS without leaving its component.
Together with the time for detecting the bi-edge components we get the running time of the corollary.

6.0.2 Betweenness

We already know from Chapter 4.2.3 that we need time in $\mathcal{O}(|V| \cdot |E|)$ for computing the betweenness index for all vertices $v \in V$ conventionally by using BFS visitors.
Now we want to speed up the computation of the betweenness index

$$\overrightarrow{X}^{V,G}_{\mathrm{Bet}}(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)} = \sum_{s \in V} \sum_{t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)} \tag{6.1}$$

for a connected network $G = (V, E)$, if G has bridge edges. This approach is similar to the method developed for closeness above.

We only have to investigate the case that there are two bridge edges and notice that the case with a single bridge edge is also covered. Without loss of generality we can look at a connected network $G = (V, E)$ with the bi-edge components

$$G_1 := (V_1, E_1), G_2 := (V_2, E_2) \text{ and } G_3 := (V_3, E_3)$$

connected by the bridge edges

$$e_1 := (\underline{v}_1, \overline{v}_2) \text{ from } \underline{v}_1 \in V_1 \text{ to } \overline{v}_2 \in V_2$$

and

$$e_2 := (\underline{v}_2, \overline{v}_3) \text{ from } \underline{v}_2 \in V_2 \text{ to } \overline{v}_3 \in V_3$$

as illustrated in Figure 6.2.

Figure 6.2: Three components connected by two bridge edges.

We assume additionally that we look at a vertex $v \in V_2$.

Initially we can rewrite Equation 6.1 to

$$\overrightarrow{X}^{V,G}_{\text{Bet}}(v) = \sum_{s \in V_1} \sum_{t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} + \sum_{s \in V_2} \sum_{t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)} + \underbrace{\sum_{s \in V_3} \sum_{t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}}_{=0} \tag{6.2}$$

and see that the last addend is always zero, because there is no shortest path that starts in V_3 and uses the vertex $v \in V_2$. This is guaranteed by the direction of the bridge edges.
We can simplify the expression from Equation 6.2 again to

$$\begin{aligned} \overrightarrow{X}^{V,G}_{\text{Bet}}(v) \quad = \quad & \sum_{s \in V_1} \sum_{t \in V_2} \frac{\sigma_G(s,t|v)}{\sigma_G(s,t)} + \sum_{s \in V_2} \sum_{t \in V_3} \frac{\sigma_G(s,t|v)}{\sigma_G(s,t)} \\ & + \sum_{s \in V_1} \sum_{t \in V_3} \frac{\sigma_G(s,t|v)}{\sigma_G(s,t)} + \sum_{s \in V_2} \sum_{t \in V_2} \frac{\sigma_G(s,t|v)}{\sigma_G(s,t)} \end{aligned} \tag{6.3}$$

by applying the same argument for $s, t \in V_1$ and $s \in V_2, t \in V_1$.

Lemma 6.3
Let G be a connected network with two bi-edge components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ together with a bridge edge $e = (v_1, v_2)$ from $v_1 \in V_1$ to $v_2 \in V_2$ as illustrated in Figure 6.1. For the number of shortest paths from $s \in V_1$ to $t \in V_2$ holds the equality

$$\sigma_G(s,t) = \sigma_{G_1}(s, \underline{v}_1) \cdot \sigma_{G_2}(\overline{v}_2, t).$$

Proof: Each shortest path from a vertex $s \in V_1$ to a vertex $t \in V_2$ has to use the bridge edge $e = (v_1, v_2)$. Therefore e extends each shortest path from s to v_1 in G_1 to a shortest path from s to v_2. All those shortest paths can be extended again by all shortest paths from v_2 to t in G_2.

Corollary 6.2
Under the assumptions from Lemma 6.3 we also get the equalities

$$\sigma_G(s,t|v) = \sigma_{G_1}(s, \underline{v}_1|v) \cdot \sigma_{G_2}(\overline{v}_2, t)$$

for the number of shortest paths, which use $v \in V_1$ as an inner vertex and

$$\sigma_G(s,t|v) = \sigma_{G_1}(s, \underline{v}_1) \cdot \sigma_{G_2}(\overline{v}_2, t|v)$$

for the number of shortest paths, which use $v \in V_2$ as an inner vertex.

After applying the identities from Lemma 6.3 and Corollary 6.2 for both bridge edges to Equation 6.3, we get

$$\begin{aligned}
\overrightarrow{X}^{V,G}_{\mathrm{Bet}}(v) &= \sum_{s\in V_1}\sum_{t\in V_2} \frac{\sigma_{G_1}(s,\underline{v}_1)\cdot\sigma_{G_2}(\overline{v}_2,t|v)}{\sigma_{G_1}(s,\underline{v}_1)\cdot\sigma_{G_2}(\overline{v}_2,t)} \\
&\quad + \sum_{s\in V_2}\sum_{t\in V_3} \frac{\sigma_{G_2}(s,\underline{v}_2|v)\cdot\sigma_{G_3}(\overline{v}_3,t)}{\sigma_{G_2}(s,\underline{v}_2)\cdot\sigma_{G_3}(\overline{v}_3,t)} \\
&\quad + \sum_{s\in V_1}\sum_{t\in V_3} \frac{\sigma_{G_1}(s,\underline{v}_1)\cdot\sigma_{G_2}(\overline{v}_2,\underline{v}_2|v)\cdot\sigma_{G_3}(\overline{v}_3,t)}{\sigma_{G_1}(s,\underline{v}_1)\cdot\sigma_{G_2}(\overline{v}_2,\underline{v}_2)\cdot\sigma_{G_3}(\overline{v}_3,t)} \\
&\quad + \sum_{s\in V_2}\sum_{t\in V_2} \frac{\sigma_{G_2}(s,t|v)}{\sigma_{G_2}(s,t)} \\
&= \left(\sum_{s\in V_1}\frac{\sigma_{G_1}(s,\underline{v}_1)}{\sigma_{G_1}(s,\underline{v}_1)}\right)\cdot\sum_{t\in V_2}\frac{\sigma_{G_2}(\overline{v}_2,t|v)}{\sigma_{G_2}(\overline{v}_2,t)} \\
&\quad + \sum_{s\in V_2}\left(\frac{\sigma_{G_2}(s,\underline{v}_2|v)}{\sigma_{G_2}(s,\underline{v}_2)}\cdot\sum_{t\in V_3}\frac{\sigma_{G_3}(\overline{v}_3,t)}{\sigma_{G_3}(\overline{v}_3,t)}\right) \\
&\quad + \left(\sum_{s\in V_1}\frac{\sigma_{G_1}(s,\underline{v}_1)}{\sigma_{G_1}(s,\underline{v}_1)}\right)\cdot\left(\frac{\sigma_{G_2}(\overline{v}_2,\underline{v}_2|v)}{\sigma_{G_2}(\overline{v}_2,\underline{v}_2)}\cdot\sum_{t\in V_3}\frac{\sigma_{G_3}(\overline{v}_3,t)}{\sigma_{G_3}(\overline{v}_3,t)}\right) \\
&\quad + \sum_{s\in V_2}\sum_{t\in V_2}\frac{\sigma_{G_2}(s,t|v)}{\sigma_{G_2}(s,t)}. \qquad (6.4)
\end{aligned}$$

Lemma 6.4
Using abbreviations like in Notation 6.1, the identities

$$\sum_{t \in V_3} \frac{\sigma_{G_3}(\overline{v}_3, t)}{\sigma_{G_3}(\overline{v}_3, t)} = |\overrightarrow{rv}_{G_3}(\overline{v}_3)| \quad \text{and} \quad \sum_{s \in V_1} \frac{\sigma_{G_1}(s, \underline{v}_1)}{\sigma_{G_1}(s, \underline{v}_1)} = |\overleftarrow{rv}_{G_1}(\underline{v}_1)|$$

hold.

Proof: In Definition 4.8 we required $\frac{0}{0} = 0$ for the fractions of shortest paths numbers. Therefore we get the correctness of the first identity from

$$\frac{\sigma_{G_3}(\overline{v}_3, t)}{\sigma_{G_3}(\overline{v}_3, t)} = \begin{cases} 1 & \text{, if there are shortest paths from } \overline{v}_3 \text{ to } t \\ 0 & \text{, else.} \end{cases}$$

This is just the number of vertices reachable from $\overline{v}_3$. We complete the proof by applying the same argument symmetrically for the second identity.

Theorem 6.2
We can compute $\overrightarrow{X}^{V,G}_{\text{Bet}}(v)$ for all vertices $v \in V_2$ without leaving component G_2 with time in $\mathcal{O}(|V_2| \cdot |E_2|)$, if we already know the values

$$|\overrightarrow{rv}_{G_3}(\overline{v}_3)| = \overrightarrow{X}^{V,G}_{\text{Rv}}(\overline{v}_3) \quad \text{and} \quad |\overleftarrow{rv}_{G_1}(\underline{v}_1)| = \overleftarrow{X}^{V,G}_{\text{Rv}}(\underline{v}_1).$$

Proof: Substituting the identities from Lemma 6.4 in Equation 6.4, we get

$$\begin{aligned} \overrightarrow{X}^{V,G}_{\text{Bet}}(v) \;=\; & |\overleftarrow{rv}_{G_1}(\underline{v}_1)| \cdot \sum_{t \in V_2} \frac{\sigma_{G_2}(\overline{v}_2, t|v)}{\sigma_{G_2}(\overline{v}_2, t)} \\ & + \sum_{s \in V_2} \left(\frac{\sigma_{G_2}(s, \underline{v}_2|v)}{\sigma_{G_2}(s, \underline{v}_2)} \cdot |\overrightarrow{rv}_{G_3}(\overline{v}_3)| \right) \\ & + |\overleftarrow{rv}_{G_1}(\underline{v}_1)| \cdot \left(\frac{\sigma_{G_2}(\overline{v}_2, \underline{v}_2|v)}{\sigma_{G_2}(\overline{v}_2, \underline{v}_2)} \cdot |\overrightarrow{rv}_{G_3}(\overline{v}_3)| \right) \\ & + \sum_{s \in V_2} \sum_{t \in V_2} \frac{\sigma_{G_2}(s, t|v)}{\sigma_{G_2}(s, t)}. \end{aligned} \tag{6.5}$$

Because of Equation 6.5 we have to update the betweenness computation in the following way. If we start a BFS for a bridge edge target, we have to involve the dependencies for all incoming paths from the other component and if we reach a bridge edge source, we have to involve the dependencies for all outgoing paths into the other component.

Corollary 6.3
We can apply Theorem 6.2 recursively on the components G_1, G_2 and G_3 and we can compute the betweenness index for G with time in

$$\mathcal{O}(|G| + |V| \cdot |E(C_{\max})|),$$

where $C_{\max}$ denotes a bi-edge component with the biggest number of edges.

Proof: We have to show that Theorem 6.2 is also applicable, if G has only one bridge edge between two components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as illustrated in Figure 6.1 on page 64. We have to distinguish two cases:

1st case : $\underline{v \in V_1}$

Assuming $|\overrightarrow{rv}_{G_3}(\overline{v}_3)| = 0$ we get from Theorem 6.2

$$\overrightarrow{X}^{V,G}_{\text{Bet}}(v) = \sum_{s \in V_2} \sum_{t \in V_2} \frac{\sigma_{G_2}(s,t|v)}{\sigma_{G_2}(s,t)} + |\overleftarrow{rv}_{G_1}(\underline{v}_1)| \cdot \sum_{t \in V_2} \frac{\sigma_{G_2}(\overline{v}_2,t|v)}{\sigma_{G_2}(\overline{v}_2,t)}.$$

2nd case : $\underline{v \in V_2}$

Assuming $|\overleftarrow{rv}_{G_1}(\underline{v}_1)| = 0$, we get

$$\overrightarrow{X}^{V,G}_{\text{Bet}}(v) = \sum_{s \in V_2} \sum_{t \in V_2} \frac{\sigma_{G_2}(s,t|v)}{\sigma_{G_2}(s,t)} + \sum_{s \in V_2} \left(\frac{\sigma_{G_2}(s,\underline{v}_2|v)}{\sigma_{G_2}(s,\underline{v}_2)} \cdot |\overrightarrow{rv}_{G_3}(\overline{v}_3)| \right)$$

also from Theorem 6.2.

The running time for detecting the bi-edge components and the computation of the number of reachable vertices for vertices incident to bridge edges is in

$$\mathcal{O}(|G| + |V| \cdot |E(C_{\max})|)$$

because of corollary 6.1. We can then apply Theorem 6.2 to all bi-edge components with time in

$$\mathcal{O}(|V| \cdot |E(C_{\max})|).$$

Chapter 7

Sampling approximation

In Chapter 4.3 we have already seen network indices that can be computed approximately by iterating the matrix multiplication. Now we discuss an approximation schema [8] for network indices

$$X(v) = \sum_{t \in V} X_t(v)$$

that can be computed by a finite sum of values, which depend on the parameter $t \in V$. This situation is given for distance-based indices like those in Chapter 4. We assume that $X(v)$ can be extrapolated from the partial values for an independent random sample $(t_1, \ldots, t_k)$ with elements in V

$$\frac{\sum_{t \in V} X_t(v)}{n} \approx \frac{\sum_{i=1}^{k} X_{t_i}(v)}{k} \Longrightarrow \sum_{t \in V} X_t(v) \approx \frac{\sum_{i=1}^{k} n \cdot X_{t_i}(v)}{k},$$

if k is large enough. To get an estimation for the probability of a given error we present an appropriate bound first. After the formulation of the general schema we demonstrate its application by constructing suitable estimators for betweenness and closeness of large networks.

7.1 Error probability bound

The following theorem of Hoeffding gives a bound for the probability that the average of a random sample with k elements $\overline{X}^k := \frac{\sum_{i=1}^{k} X_i}{k}$ exceeds its expected value $\mathbb{E}\left[\overline{X}^k\right]$ more than a given $\xi > 0$. A proof for this theorem can be found in the original article [16].

Theorem 7.1 (2nd Hoeffding Theorem)
Let $X_1, X_2, \ldots, X_k$ be independent random variables bounded by $a_i \leq X_i \leq b_i$ $(i = 1, \ldots, k)$. For $\xi > 0$ holds

$$\mathbb{P}\left\{\overline{X}^k - \mathbb{E}\left[\overline{X}^k\right] \geq \xi\right\} \leq \exp\left(-2 \cdot \frac{k^2 \xi^2}{\sum_{i=1}^{k} (b_i - a_i)^2}\right). \tag{7.1}$$

We use Theorem 7.1 to get a bound for the probability that the absolute error of $\overline{X}^k$ from $\mathbb{E}\left[\overline{X}^k\right]$ is greater than $\xi > 0$. We can assume for finite networks additionally that there is a common bound $0 \leq X_t(v) \leq M$ for all vertices $v, t \in V$.

Corollary 7.1 (Hoeffding bound)
Let $X_1, X_2, \ldots, X_k$ be independent random variables bounded by

$$0 \leq X_i \leq M \text{ for } i = 1, \ldots, k.$$

For $\xi > 0$ holds

$$\mathbb{P}\left\{\left|\overline{X}^k - \mathbb{E}\left[\overline{X}^k\right]\right| \geq \xi\right\} \leq 2 \cdot \exp\left(-2k \cdot \left(\frac{\xi}{M}\right)^2\right).$$

Proof: We get directly from Inequality 7.1

$$\begin{aligned}\mathbb{P}\left\{\overline{X}^k - \mathbb{E}\left[\overline{X}^k\right] \geq \xi\right\} &\leq \exp\left(-2 \cdot \frac{k^2\xi^2}{\sum_{i=1}^k (b_i - a_i)^2}\right) \\ &= \exp\left(-2 \cdot \frac{k^2\xi^2}{k \cdot M^2}\right) = \exp\left(-2k \cdot \left(\frac{\xi}{M}\right)^2\right)\end{aligned}$$

by setting $a_i = 0$ and $b_i = M$.
We can also estimate the probability

$$\begin{aligned}\mathbb{P}\left\{\overline{X}^k - \mathbb{E}\left[\overline{X}^k\right] \leq -\xi\right\} &= \mathbb{P}\left\{\left(-\overline{X}^k\right) - \mathbb{E}\left[-\overline{X}^k\right] \geq \xi\right\} \\ &\leq \exp\left(-2k \cdot \left(\frac{\xi}{M}\right)^2\right)\end{aligned}$$

for $-M \leq -X_i \leq 0$ $(i = 1, \ldots, k)$, to complete the proof.

Remark 7.1
It can be shown that the inequality from Corollary 7.1 also holds for a random sample without replacement from a finite population.

We can apply Corollary 7.1 for indices of finite networks by considering the following lemma.

Lemma 7.1
The equality $\mathbb{E}\left(\frac{\sum_{i=1}^k X_i(v)}{k}\right) = X(v)$ holds for $X_i(v) := n \cdot X_{t_i}(v)$.

Proof: Because we get a random sample by choosing k times from the finite population V, we can show the above equality by averaging over all samples with k vertices combinatorially and get

$$\begin{aligned}
& \mathbb{E}\left(\frac{\sum_{i=1}^{k} X_i(v)}{k}\right) = \mathbb{E}\left(\frac{\sum_{i=1}^{k} n \cdot X_{t_i}(v)}{k}\right) \\
= \; & \frac{1}{n^k} \sum_{t_1,\ldots,t_k \in V} \frac{n \cdot \sum_{i=1}^{k} X_{t_i}(v)}{k} = \frac{n}{n^k \cdot k} \sum_{t_1,\ldots,t_k \in V} (X_{t_1}(v) + \ldots + X_{t_k}(v)) \\
= \; & \frac{1}{n^{k-1} \cdot k} \left(\sum_{t_1,\ldots,t_k \in V} X_{t_1}(v) + \ldots + \sum_{t_1,\ldots,t_k \in V} X_{t_k}(v) \right) \\
= \; & \frac{1}{n^{k-1} \cdot k} \left(\sum_{t_1 \in V} \left(\sum_{t_2,\ldots,t_k \in V} X_{t_1}(v) \right) + \ldots + \sum_{t_k \in V} \left(\sum_{t_1,\ldots,t_{(k-1)} \in V} X_{t_k}(v) \right) \right) \\
= \; & \frac{1}{n^{k-1} \cdot k} \left(\sum_{t_1 \in V} n^{k-1} \cdot X_{t_1}(v) + \ldots + \sum_{t_k \in V} n^{k-1} \cdot X_{t_k}(v) \right) \\
= \; & \frac{k}{n^{k-1} \cdot k} \sum_{t \in V} n^{k-1} \cdot X_t(v) = \sum_{t \in V} X_t(v) \\
= \; & X(v).
\end{aligned}$$

7.1.1 Approximation schema

To construct a suitable estimator for a network index $X(v) = \sum_{t \in V} X_t(v)$ we can create an estimator for a properly scaled index $X_{\mathrm{s}}(v)$ first. By scaling the values of $X_{\mathrm{s}}(v)$ into the fixed interval $[0, 1]$ we can analyze the error probability independently from the network size.
Now we have to determine the required size k of a random sample to get a bounded error $\varepsilon > 0$ for $X_{\mathrm{s}}(v)$ with high probability. Afterwards we get the error bound for the original index $X(v)$ with the same probability by a simple back transformation.
Then we formulate an approximation algorithm that estimates the index for a given error with high probability and can finally analyze the possible error to improve the approximated values by refining the algorithm.

Remark 7.2
For a network consisting of more than one component we can estimate the values separately for each component and we can decide to compute either exactly for small components or approximately for large components. For very large components we can also try to do a combination with the heuristics for bi-edge or bi-vertex components like in Chapter 6.

7.2 Betweenness

To approximate the betweenness index

$$X_{\text{Bet}}^{G,V}(v) = \sum_{t \in V} X_t(v) \quad \text{with} \quad X_t(v) := \sum_{s \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

for a large network $G = (V, E)$ we look first at the scaled betweenness index

$$X_{\text{sBet}}^{V,G}(v) := \frac{X_{\text{Bet}}^{V,G}(v)}{(n-1)(n-2)} = \sum_{t \in V} \frac{X_t(v)}{(n-1)(n-2)} \tag{7.2}$$

with values in the fixed interval $[0, 1]$. The scaling becomes clear by considering the betweenness value of the center vertex in Figure 7.1.

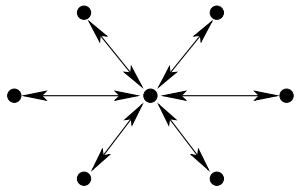

Figure 7.1: Star-shaped network.

We approximate the scaled betweenness index with the estimator

$$X_{\text{sBet}}^{V,G}(v) \approx \frac{\sum_{i=1}^{k} \left(n \cdot \frac{X_{t_i}(v)}{(n-1)(n-2)} \right)}{k} \tag{7.3}$$

by defining the random variables

$$X_i(v) := n \cdot \frac{X_{t_i}(v)}{(n-1)(n-2)}$$

for independently chosen target values $t_i \in V$. We can compute these values simultaneously for all vertices $v \in V$ by using the visitors from Chapter 4.2.3 with an incoming breadth first search.
Because the sample values $X_i(v)$ are bounded by

$$X_i(v) = n \cdot \frac{X_{t_i}(v)}{(n-1)(n-2)} \leq \frac{n \cdot (n-2)}{(n-1)(n-2)} = \frac{n}{n-1} =: M$$

we can estimate the probability that the error for a sample with

$$k := \ulcorner \frac{2 \log(n)}{\varepsilon^2} \urcorner$$

elements is greater than

$$\xi := \varepsilon$$

by applying Corollary 7.1 and get[1]

$$\begin{aligned}
& \mathbb{P}\left(\left|\frac{\sum_{i=1}^{k} X_i(v)}{k} - X_{\text{sBet}}^{V,G}(v)\right| \geq \varepsilon\right) \\
\leq\ & 2 \cdot \exp\left(-2 \cdot \ulcorner \frac{2\log(n)}{\varepsilon^2} \urcorner \left(\frac{\varepsilon}{\frac{n}{n-1}}\right)^2\right) \\
\leq\ & 2 \cdot \exp\left(-2 \cdot \frac{2\log(n)}{\varepsilon^2} \cdot \varepsilon^2 \cdot \left(\frac{n-1}{n}\right)^2\right) \\
=\ & 2 \cdot \exp\left(-2\log(n) \cdot 2 \cdot \left(\frac{n-1}{n}\right)^2\right) \\
\leq\ & 2 \cdot \exp\left(-2\log(n)\right) = \frac{2}{n^2}. \qquad (7.4)
\end{aligned}$$

Therefore we know that the probability to get an error greater than ξ for the standardized betweenness index $X_{\text{sBet}}^{V,G}(v)$ at any vertex $v \in V$ is smaller than $\frac{2}{n}$.
We will see what this means for the values of the original betweenness index $X_{\text{Bet}}^{V,G}(v)$ by looking at the inequality

$$\begin{aligned}
\varepsilon \leq\ & \left|\frac{\sum_{i=1}^{k} X_i(v)}{k} - X_{\text{sBet}}^{V,G}(v)\right| \\
=\ & \left|\sum_{i=1}^{k} \frac{n \cdot \frac{X_{t_i}(v)}{(n-1)(n-2)}}{k} - \frac{X_{\text{Bet}}^{V,G}(v)}{(n-1)(n-2)}\right| \\
=\ & \left|\frac{1}{(n-1)(n-2)}\left(\sum_{i=1}^{k} \frac{n \cdot X_{t_i}(v)}{k} - X_{\text{Bet}}^{V,G}(v)\right)\right|. \qquad (7.5)
\end{aligned}$$

We conclude that the approximation error for a sample with $k = \ulcorner \frac{2\log(n)}{\varepsilon^2} \urcorner$ elements is bounded by

$$\left|\sum_{i=1}^{k} \frac{n \cdot X_{t_i}(v)}{k} - X_{\text{Bet}}^{V,G}(v)\right| < \varepsilon(n-1)(n-2) \qquad (7.6)$$

for any vertex with probability $\left(1 - \frac{2}{n}\right) = \frac{n-2}{n}$. So we can utilize this trade-off between running time and exactness for large networks like in Procedure approxBetweenness.

[1]The inequality holds, because we get $1 \leq 2 \cdot \left(\frac{n-1}{n}\right)^2$ for $n \geq 4$.

Procedure `approxBetweenness`

Input : Network $G \in \mathcal{G}$
Relative accuracy $\varepsilon \in \mathbb{R}^+$
Output: Vertex vector X
begin
  foreach $v \in V$ **do** assign $\mathsf{X}[v] \leftarrow 0$;
  $k \leftarrow \lceil \frac{2\log(n)}{\varepsilon^2} \rceil$;
  for $i \leftarrow 1$ ***to*** k **do**
    choose vertex $t_i \in V$;
    compute dependencies vector δ_{t_i} with incoming BFS;
    foreach $v \in V \backslash \{t_i\}$ **do**
      $\mathsf{X}(v) \leftarrow \mathsf{X}(v) + \delta_{t_i}(v)$;
  foreach $v \in V$ **do**
    $\mathsf{X}(v) \leftarrow \frac{n \cdot \mathsf{X}[v]}{k}$;
end

Corollary 7.2 (Betweenness approximation)
Let G be a network. If G is large, we can compute a network index $X \in \mathbb{R}^V$ that satisfies the error bound

$$|X(v) - X_{\mathrm{Bet}}^{V,G}(v)| < \varepsilon(n-1)(n-2)$$

for nearly all $v \in V$, with time in $\mathcal{O}(\frac{\log(n)}{\varepsilon^2} \cdot m)$.

Proof:
We can compute X by using Procedure approxBetweenness. The correctness of this algorithm follows directly from Inequality 7.6, while the running time is clear because each BFS run needs time in $\mathcal{O}(|E|)$.

If we choose the vertices t_i independently, we get a fixed maximal error bound $\varepsilon(n-1)(n-2)$ as well as a fixed probability $\frac{n-2}{n}$, while we would expect that the approximated values become better for $k \longrightarrow n$. This behavior can be achieved by refining Procedure approxBetweenness to choose each vertex $t \in V$ at the most once. Now the refined algorithm computes the correct betweenness values for $k = n$. Because of remark 7.1 Corollary 7.2 holds also for this situation, while the approximated values become even better.

7.3 Closeness

An approximation for the closeness index

$$X_{Clo}^{V,G}(v) := \frac{X_{\mathrm{Rv}}^{V,G}(v)^2}{X_{\mathrm{Ds}}^{V,G}(v)}$$

for a large network $G = (V, E)$ can be computed by a fraction of approximated values for the number of reachable vertices

$$H(v) \approx X_{\mathrm{Rv}}^{V,G}(v)$$

and the sum of distances

$$D(v) \approx X_{\mathrm{Ds}}^{V,G}(v).$$

We will discuss the approximation of these values later and assume now that we already know estimations for them. By defining the errors at a vertex v to

$$e_H(v) := \left(H(v) - X_{\mathrm{Rv}}^{V,G}(v)\right) \quad \text{and} \quad e_D(v) := \left(D(v) - X_{\mathrm{Ds}}^{V,G}(v)\right),$$

we get the resulting error for the approximated closeness value by

$$\begin{aligned}
&\left|\frac{H(v)^2}{D(v)} - X_{Clo}^{V,G}(v)\right| = \left|\frac{H(v)^2}{D(v)} - \frac{X_{\mathrm{Rv}}^{V,G}(v)^2}{X_{\mathrm{Ds}}^{V,G}(v)}\right| = \left|\frac{H(v)^2}{D(v)} - \frac{(H(v) - e_H(v))^2}{D(v) - e_D(v)}\right| \\
= &\left|\frac{H(v)^2\,(D(v) - e_D(v)) - D(v)\left(H(v)^2 - 2e_H(v)H(v) + e_H(v)^2\right)}{D(v)\,(D(v) - e_D(v))}\right| \\
= &\left|\frac{-e_D(v)H(v)^2 + 2e_H(v)H(v)D(v) - e_H(v)^2D(v)}{D(v)\,(D(v) - e_D(v))}\right| \\
= &\left|\frac{-e_D(v)H(v)^2}{D(v)\,(D(v) - e_D(v))} + \frac{2e_H(v)H(v) - e_H(v)^2}{D(v) - e_D(v)}\right| \\
= &\left|\frac{-e_D(v) \cdot \frac{H(v)^2}{D(v)}}{D(v) - e_D(v)} + \frac{e_H(v) \cdot (2H(v) - e_H(v))}{D(v) - e_D(v)}\right|. \qquad (7.7)
\end{aligned}$$

From the last term of Equation 7.7 we can see:

- The resulting error is the worst, if e_H and e_D have alternating leading signs.
- The error scales with the approximated closeness values.
- e_H and e_D are more significant, if $D(v)$ is small, while e_H is also more significant, if $H(v)$ is large.

Therefore we can improve the approximated closeness values $X_{Clo}^{V,G}(v)$ by increasing $X_{\mathrm{Rv}}^{V,G}(v)$ and $X_{\mathrm{Ds}}^{V,G}(v)$ always together. We can do this by using the same random sample $(t_1, \ldots, t_k)$.

7.3.1 Number of reachable vertices

To approximate the number of reachable vertices

$$X_{Rv}^{V,G}(v) := \sum_{t \in V} X_t(v) \quad \text{with} \quad X_t(v) := \begin{cases} 1 & \text{if } d_G(v,t) < \infty \text{ and } t \neq v, \\ 0 & \text{else} \end{cases}$$

for a large network $G = (V, E)$ we begin with an approximation for the scaled number of reachable vertices

$$X_{sRv}^{V,G}(v) := \frac{X_{Rv}^{V,G}(v)}{n-1} = \sum_{t \in V} \frac{X_t(v)}{n-1}$$

with values in $[0, 1]$. We can use the estimator

$$X_{sRv}^{V,G}(v) \approx \frac{\sum_{i=1}^{k} \left(n \cdot \frac{X_{t_i}(v)}{n-1} \right)}{k}$$

and define the random variables

$$X_i(v) := n \cdot \frac{X_{t_i}(v)}{n-1}.$$

By applying Corollary 7.1 with

$$M := \frac{n}{n-1} \quad , \quad \xi := \varepsilon \quad \text{and} \quad k := \lceil \frac{2\log(n)}{\varepsilon^2} \rceil$$

we get the probability

$$\begin{aligned} \mathbb{P}\left(\left| \frac{\sum_{i=1}^{k} X_i(v)}{k} - X_{sRv}^{V,G}(v) \right| \geq \varepsilon \right) &\leq 2 \cdot \exp\left(-2 \lceil \frac{2\log(n)}{\varepsilon^2} \rceil \left(\frac{\varepsilon}{\frac{n}{n-1}} \right)^2 \right) \\ &\leq 2 \cdot \exp\left(-2\log(n) \cdot 2 \left(\frac{n-1}{n} \right)^2 \right) \\ &\leq \frac{2}{n^2} \end{aligned} \tag{7.8}$$

that the approximation error for the scaled number of reachable vertices for a given $v \in V$ is bounded by ε. Therefore we get the error estimation

$$\left| \frac{n}{k} \sum_{i=1}^{k} X_{t_i} - X_{Rv}^{V,G}(v) \right| < \varepsilon(n-1) \quad \text{with probability} \quad \frac{n-2}{n}$$

for the original number of reachable vertices $X_{Rv}^{V,G}(v)$ at any vertex $v \in V$.

Corollary 7.3
Let G be a network. If G is large, we can compute a network index $X \in \mathbb{R}^V$ that satisfies the error bound

$$\left| X(v) - X_{Rv}^{V,G}(v) \right| < \varepsilon(n-1)$$

for nearly all $v \in V$, with time in $\mathcal{O}(\frac{\log(n)}{\varepsilon^2} \cdot m)$.

7.3.2 Sum of distances

To approximate the sum of distances

$$X_{\mathrm{Ds}}^{V,G}(v) := \sum_{t \in V} X_t(v) \text{ with } X_t(v) := \begin{cases} d_G(v,t) & \text{if } d_G(v,t) < \infty \text{ and } t \neq v, \\ 0 & \text{else} \end{cases}$$

for a large network $G \in \mathcal{G}$ with diameter

$$D := \max_{s,t \in V} d_G(s,t) \leq (n-1)$$

we start with the approximation of the scaled sum of distances

$$X_{\mathrm{sDs}}^{V,G}(v) := \frac{X_{\mathrm{Ds}}^{V,G}(v)}{n-1} = \sum_{t \in V} \frac{X_t(v)}{n-1}$$

with values in $[0, D]$. We can approximate $X_{\mathrm{sDs}}^{V,G}(v)$ with the estimator

$$X_{\mathrm{sDs}}^{V,G}(v) \approx \frac{\sum_{i=1}^{k} \left(n \cdot \frac{X_{t_i}(v)}{n-1} \right)}{k}$$

by defining the random variables

$$X_i(v) := n \cdot \frac{X_{t_i}(v)}{n-1}.$$

We apply Corollary 7.1 with

$$M := \frac{n \cdot D}{n-1} \quad , \quad \xi := \varepsilon \cdot D \quad \text{and} \quad k := \ulcorner \frac{2\log(n)}{\varepsilon^2} \urcorner$$

and get the probability

$$\begin{aligned} \mathbb{P}\left(\left| \frac{\sum_{i=1}^{k} X_i(v)}{k} - X_{sDs}^{V,G}(v) \right| \geq \varepsilon \cdot D \right) &\leq 2 \cdot \exp\left(-2 \ulcorner \frac{2\log(n)}{\varepsilon^2} \urcorner \left(\frac{\varepsilon \cdot D}{\frac{n}{n-1} \cdot D} \right)^2 \right) \\ &\leq 2 \cdot \exp\left(-2\log(n) \cdot 2 \left(\frac{n-1}{n} \right)^2 \right) \\ &\leq \frac{2}{n^2}. \end{aligned} \tag{7.9}$$

We conclude that the approximation error for the scaled sum of distances for any vertex $v \in V$ is bigger than $\varepsilon \cdot D$ with probability smaller than $\frac{2}{n}$. Together we get the error estimation

$$\left| \frac{n}{k} \sum_{i=1}^{k} X_{t_i} - X_{Ds}^{V,G}(v) \right| < \varepsilon \cdot D \cdot (n-1) \quad \text{with probability} \quad \frac{n-2}{n}$$

for the original sum of distances $X_{Ds}^{V,G}(v)$ at any vertex $v \in V$.

Corollary 7.4
Let G be a network. If G is large, we can compute a network index $X \in \mathbb{R}^V$ that satisfies the error bound

$$\left| X(v) - X_{Ds}^{V,G}(v) \right| < \varepsilon \cdot D \cdot (n-1)$$

for nearly all $v \in V$, with time in $\mathcal{O}(\frac{\log(n)}{\varepsilon^2} \cdot m)$.

Remark 7.3
The error bound shrinks with the diameter D and the small world behavior

$$D \ll (n-1)$$

is typical for natural networks.

Chapter 8

Summary and outlook

After defining networks and structural indices we demonstrated the flexible and efficient computation and statistical evaluation of structural indices. Especially for large networks we developed methods to speed up or approximate their values.
In this paper we provided specific examples of network analysis by index computation. These results can be applied to similar analysis tasks in further projects.
While we focused on the computation possibilities, it would be interesting to apply our results to real situations as described in the introduction.

Bibliography

[1] A. Aho, and J. E. Hopcroft and J. D. Ullman
Data structures and algorithms.
Addison-Wesley, 1983

[2] J.M. Anthonisse
The rush in a directed graph.
Technical Report BN 9/71, Stichting Mathematisch Centrum, Amsterdam, 1971

[3] H. Jeong, B. Tombor, R. Albert, Z. N. Oltavi and A.-L. Barábasi
The large-scale organization of metabolic networks.
Nature 407, 651-654, 2000

[4] P. Bonacich
Power and centrality: A family of measures.
American Journal of Sociology, 92:1170-1182, 1987

[5] U. Brandes
A Faster Algorithm for Betweenness Centrality.
Journal of Mathematical Sociology 25(2):163-177, 2001

[6] U. Breymann
Designing Components with the C++ STL: A new approach to programming.
Addison-Wesley, Boston, 2000

[7] T.H. Cormen, C.E. Leisserson and R.L. Rivest
Introduction to Algorithms.
The MIT Press, Cambridge, 1990

[8] D. Eppstein and J. Wang
Fast approximation of centrality.
12th ACM-SIAM Symp. Discrete Algorithms, Washington, 2001, pp. 228-229

[9] L.C. Freeman
A set of measures of centrality based on betweenness.
Sociometry 40:35-41, 1977

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides
Design patterns: elements of reusable object-oriented software.
Addison-Wesley, 1994

[11] C. Godsil and G. Royle
Algebraic Graph Theory, volume 207 of Graduate Texts in Mathematics. Springer, 2001

[12] G.H. Golub and C.F. van Loan
Matrix Computations.
Johns Hopkins University Press, 3rd edition, 1996

[13] J. Goslin, B. Joy, G. Steele and G. Bracha
The Java Language Specification.
Addison-Wesley, 2nd edition, 2000

[14] P. Hage and F. Harary
Eccentricity and centrality in networks.
Social Networks 17:57-63, 1995

[15] N. Henze
Stochastik für Einsteiger: Eine Einführung in die faszinierende Welt des Zufalls.
Vieweg, 2000

[16] W. Hoeffding
Probability inequalities for sums of bounded random variables.
J. Amer. Statistical Assoc. 58(301):713-721, March 1963

[17] L. Katz
A new status index derived from sociometric analysis.
Psychometrika, 18:39-43, 1953

[18] J.M. Kleinberg
Authoritative Sources in a Hyperlinked Environment.
Journal of the Association for Computing Machinery, 46(5):604-632, 1999

[19] D. Kühl
Design patterns for the implementation of graph algorithms.
Master's thesis, Technische Universität Berlin, 1996

[20] W. LaLonde
Discovering Smalltalk.
Addison-Wesley, 1994

[21] K. Mehldorn and S. Näher
Leda: a platform for combinatorial and geometric computing.
Cambridge University Press, Cambridge, 1999

[22] L. Page, S. Brin, R. Motwani and T. Winograd
The PageRank Citation Ranking: Bringing Order to the Web.
Stanford Digital Library Technologies Project, 1998

[23] D. Pollard
Convergence of Stochastic processes.
Springer, 1984

[24] G. Sabidussi
The centrality index of a graph.
Psychometrica 31:581-603, 1966

[25] J. Siek, L.-Q. Lee and A. Lumsdaine
The Boost Graph Library: User Guide and Reference Manual.
Addison Wesley, Boston, 2001

[26] B. Stroustrup
The C++ Programming Language.
Addison-Wesley, Boston, 1991

[27] R.E. Tarjan
Depth-First search and Linear Graph Algorithms.
SIAM Journal of Computation 1,146-160, 1972

[28] The Apache Software Foundation
The Apache XML Project.
http://xml.apache.org/

[29] graphdrawing.org
The GraphML File Format.
http://graphml.graphdrawing.org/

[30] sourceforge.net
The Expat XML Parser.
http://expat.sourceforge.net/

[31] Jerry Grossman
The Erdös Number Project.
http://personalwebs.oakland.edu/∼grossman/erdoshp.html

[32] World Wide Web Consortium
Extensible Markup Language (XML).
http://www.w3.org/XML/