Untersuchung und Umsetzung von Graph-Metriken

Studienarbeit

im Studiengang Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart, Campus Horb am Neckar

von

Benedict Weichselbaum

27. April 2021

Bearbeitungszeitraum	28.09.2020 - 31.05.2021
Matrikelnummer, Kurs	6275457, TINF2018
Betreuer & Gutachter	Prof. Dr. ing. Olaf Herden

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema *Untersuchung und Umsetzung von Graph-Metriken* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Nürnberg, 27. April 2021

Benedict Martin Weichselbaum

Abstract

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

CSV Comma Separated Values

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISO International Standard Organisatizon

JSON Java Script Object Notation

JVM Java Virtual Machine

NP nicht-deterministisch polynomiell

POJO Plain Old Java Object

UML Unified Modelling Language

XML Extensible Markup Language

1 Einleitung

1.1 Motivation und Ziel der Studienarbeit

Graphen sind einer der wichtigsten Datenstrukturen der Informatik. Warum kann man das sagen? In seinem Buch „Algorithmische Graphentheorie“ nennt Volker Turau, Professor an der Universität Hamburg-Harburg, den Grund dafür:

> *Graphen sind die in der Informatik am häufigsten verwendete Abstraktion. Jedes System, welches aus diskreten Zuständen oder Objekten und Beziehungen zwischen diesen besteht, kann als Graph modelliert werden.* [Tur04]

Diese netzartigen Strukturen können dabei die verschiedensten Konstrukte repräsentieren. Dazu zählen Straßennetze, Computernetzwerke, elektrische Schaltungen aber auch zum Beispiel chemische Moleküle. [Tit19]

Um Graphen zu beschreiben und zu charakterisieren, haben sich über die Zeit zahlreiche Metriken, bzw. Eigenschaften für diese herausgebildet („graph properties“ [Lov12]). Das heißt, einem Graphen bzw. auch seinen einzelnen Komponenten können gewisse Kennzahlen zugeordnet werden, die ihn auszeichnen. Auch diese Metriken sind, wie die Graphen selbst, oft praktisch anwendbar. Zum Beispiel in der Untersuchung von Netzwerken [EK13].

Diese Studienarbeit soll nun diese Metriken genauer untersuchen. Hierbei ist es zunächst wichtig die verschiedenste Metriken vorzustellen und zu erläutern. Dabei ist es auch relevant herauszufinden, wie verbreitet diese Metriken sind und inwieweit die jeweiligen Kennzahlen im Bezug auf ihre Berechenbarkeit zu bewerten sind. Es soll ein umfassender Überblick über Graphmetriken ermöglicht werden und vor allem klar werden, was eine Metrik ausdrückt und ggf. kurz erläutert werden, welchen Nutzen eine Kennzahl haben kann.

Neben einer theoretischen Betrachtung der Graphmetriken soll auch eine Implementierung bzw. Nutzung dieser Kennzahlen stattfinden. Mithilfe einer selbst erstellten Klassenbibliothek soll aufgezeigt werden, wie die vorgestellten Metriken konkret umgesetzt werden können. Die Entwicklung dieser Klassenbibliothek soll ingenieursmäßig unter Zuhilfenahme von Softwareengineeringwerkzeugen erfolgen. Infolgedessen entsteht das finale Softwareprodukt im Rahmen des klassischen Softwarelebenszyklus und

umfasst die Analyse der Anforderungen, einen Softwareentwurf und die Implementierung und Testung der Software. Bei der Anforderungsanalyse sollen sowohl funktionale und nicht funktionale Aspekte beleuchtet werden. [Bal09; BL11]

Neben der Entwicklung einer Klassenbibliothek zur Berechnung der Graphmetrik ist zudem Ziel der Arbeit zu erarbeiten, wie es möglich wäre die Metriken-Berechnung weiter zu abstrahieren. Hierbei soll ein grobes Konzept erarbeitet werden, wie die Berechnung der Graphkennzahlen potenziell weiter verallgemeinert werden kann, in dem Sinne, dass die eigentliche Berechnungslogik frei wählbar ist. So könnte die Berechnung durch die implementierte Klassenbibliothek erfolgen oder durch ein Graphdatenbanksystem.

1.2 Fragestellungen

Auf Basis dieser Motivation können nun auch die konkreten Fragestellungen formuliert werden, die diese Arbeit betrachten soll. Insgesamt sollen sechs wissenschaftliche Fragen beantwortet werden.

1. Welche Graph-Metriken gibt es und wie sind diese definiert und zu kategorisieren?

Hierzu gehört, wie bereits erwähnt die Vorstellung der einzelnen Metriken, aber auch eine Kategorisierung in Rubriken, um Metriken besser voneinander abzugrenzen, da diverse Metriken höchst unterschiedliche Aussagen über einen Graphen treffen. Es ist dabei auch Ziel bereits aufzuzeigen welche Anwendung die jeweiligen Metriken finden, bzw. welche Motivation hinter ihnen steht, falls eine Aussage darüber getroffen werden kann. Bei der Beantwortung dieser Frage soll außerdem auch darauf eingegangen werden, inwieweit die beschriebene Metrik in bestimmten Mathematikbibliotheken wie „Sage Math“ oder „Wolfram“ vorkommen, um besser bewerten zu können welche Relevanz und Verbreitung diese findet.

2. Wie sind die vorgestellten Metriken zu bewerten?

Bei dieser Frage soll es vor allem darum gehen, die vorgestellten Metriken dahingehend zu bewerten, wie „schwer“ es ist, sie zu ermitteln. Außerdem wird bei der Bewertung auch auf die Verbreitung eingegangen. Speziell soll erwähnt werden, ob die jeweilige Metrik in bekannten Mathematikbibliotheken vertreten ist oder nicht.

3. Welche Anforderungen müssen an eine Klassenbibliothek zur Umsetzung der vorgestellten Metriken gestellt werden?

Im darauffolgenden parktischen Teil der Arbeit sollen die recherchierten Metriken im Rahmen einer Klassenbibliothek implementiert werden. Ziel ist es, das erlangte theoretische Wissen praktisch anzuwenden. Hierbei ist es zunächst wichtig die jeweiligen

Anforderungen an die zu erstellende Software zu ermitteln. Zu diesen Anforderungen gehören sowohl funktionale als auch nicht funktionale Eigenschaften.

4. Wie können die Anforderungen der Klassenbibliothek umgesetzt werden?

Auf Basis der Anforderungen muss anschließend ein Entwurf zur Implementierung der Klassenbibliothek geliefert werden. Anhand des Entwurfs soll klar werden, wie die jeweiligen Anforderungen an die Software umgesetzt wurden. Es sollen sowohl statische als auch dynamische Aspekte des Entwurfes betrachtet werden. Zur Umsetzung ist es zudem notwendig Literatur zu konsultieren, um die richtigen Entwurfs- und Implementierungsentscheidungen zu treffen.

5. Wie wurden die konkreten Algorithmen zur Berechnung der Graphmetriken implementiert?

Die Kernaufgabe der Bibliothek ist es, für einen vorgegebenen Graphen diverse Metriken zu berechnen. Hierfür ist eine Reihe von verschiedenen Algorithmen notwendig. Aufgrund dessen ist es notwenig, dass diese und ihr Zusammenspiel im Rahmen der Entwicklung erläutert werden.

6. Wie wurde sichergestellt, dass die erstellte Klassenbibliothek korrekt funktioniert und die jeweiligen nicht funktionalen Anforderungen eingehalten wurden?

Neben dem Entwurf und der Implementierung ist es unabdingbar, dass geprüft werden muss, ob die definierten Anforderungen auch korrekt von der erstellten Software erfüllt werden. Hierfür sollen vor allem Tests dienen, die jeweiligen Methoden der Bibliothek testen. Des Weiteren wird überprüft, ob die jeweilig definierten nicht funktionalen Anforderungen eingehalten wurden. Dabei ist es wichtig zu erläutern, ob und wie diese erfüllt wurden. Beziehungsweise wird auch erklärt, warum sie nicht erfüllt sind oder es nicht verifiziert werden kann, ob sie erfüllt sind.

6. Wie kann die Berechnung der Graphmetriken weiter abstrahiert werden?

Anstatt die Berechnung der Kennzahlen nur über die entworfene Bibliothek zu ermöglichen, wäre es auch möglich die Berechnung weiter zu abstrahieren und eine Software zu entwerfen, die je nach Wunsch eine andere Metriken-Logik nutzt. Infolgedessen soll untersucht werden, wie eine solche Abstraktion entwickelt werden könnte.

2 Graph-Metriken

Dieser erste Teil der Arbeit wird sich nun ausführlich mit einer weiten Reihe an Graph-Metriken beschäftigen. Hierbei sollen die ersten zwei Fragestellungen der Arbeit genau beantwortet werden. Zur jeweiligen Vorstellung einer Graph-Metrik sollen dabei die folgengen Punkte erläutert werden:

- Was drückt die Metrik aus (Definition)?
- Wie ist die Metrik im Bezug auf den Rechenaufwand zu bewerten?
- Inwieweit ist die Metrik in einschlägigen Mathematikbibliotheken vertreten (Wolfram, SageMath, MatLab)?
- Was ist eine konkrete Motivation bzw. Anwendung für die Metrik, falls diese auszumachen ist?

Es ist zu erwähnen, dass alle im folgenden vorgestellten Metriken über die einzelnen Sektionen der Arbeit in ihre Kategorien eingeteilt sind. Als Synonym für Metrik werden innerhalb dieser Arbeit die Begriffe Kennzahl und Invariante gebraucht. Eine Invariante ist dabei speziell eine Funktion, die zwei isomorphen Graphen den gleichen Wert zuordnet. Dies ist gleichwertig mit dem Begriff Metrik, da eine Metrik auch einen Graphen auf eine Zahl bzw. einen Wert abbildet und dabei zwei isomorphen Graphen den gleichen Wert zuordnet. [Die00] Neben Kennzahlen, die nur auf einen ganzen Graphen angewendet werden, werden auch Metriken vorgestellt, die sich auf einzelne Knoten eines Graphen beziehen.

Darüber hinaus ist noch eine grundsätzliche Notationen während der Arbeit zu klären: Ein **Graph G** ist ein Paar bestehend aus **Knoten V** und **Kanten E**.

$$G = (V, E),\ wobei\ E \subseteq V \times V$$

Für V können wir auch V(G) schreiben, für E auch E(G). [Die00] V ist dabei Englisch und bedeutet „Vertices“, E steht für „Edges“. Wenn es um die Datenstrukturen von Graphen geht, kommen im Rahmen dieser Arbeit hauptsächlich Adjazenzmatrizen und Adjazenzlisten zum Einsatz. Allerdings können bei Bedarf auch Inzidenzen (Beziehung zwischen Knoten und Kanten) im Graphen eine Rolle spielen, wie Inzidenzmatrizen und Inzidenzlisten. [Kne19; Die00] Zudem wird bei der Betrachtung meist auf ungerichtete, simple Graphen Bezug genommen.

Es sei noch zu erwähnen, dass die nachfolgenden Metriken auf Basis einer umfangreichen Recherche zusammengetragen worden sind. Die vorgestellten Kennzahlen

stellen aber bei weitem keine vollständige Liste dar. Beim Zusammentragen der Informationen und der Wahl der Metriken wurde bedacht, wurden vor allem Metriken betrachtet, die entweder grundlegend sind oder im Rahmen der Recherche herausstachen und häufiger auftauchten. Außerdem war ein Kriterium bei der Literaturrecherche die Ergibigkeit der Informationen über die jeweilige Metrik. Infolgedessen sind selten erwähnte oder exotische Graphmetriken in dieser Auflistung nicht zu finden.

2.1 Grundlegende Metriken

Ein einem ersten Teil sollen grundlegende Graph-Kennzahlen vorgestellt werden. Diese beschreiben einen Graphen auf rudimentäre Art und Weise und zeigen die am einfachsten zu verstehenden Eigenschaften des Graphen.

Ordnung und Größe eines Graphen

Die Frage danach, wie viele Knoten ein Graph hat lässt sich mit der **„Ordnung“** eines Graphen beantworten. Die Ördnung”beschreibt dabei einfach die Anzahl der Elemente in der Menge V. Man schreibt: $|V|\ oder\ |V(G)|\ oder\ auch\ |G|$. [Die00] Diese Eigenschaft ist essentiell zur allgemeinen Beschreibung und z.B. graphischen Darstellung eines Graphen. Sie lässt sich dabei in sämtlichen mathematischen Bibliotheken finden, wie SageMath, Matlab und Wolfram [Sag20b; Mat20a; Wol20a]. Die Komplexität zur Erfassung der Metrik gestaltet sich dabei äußerst einfach. Bei einer Adjazenzmatrix lässt sich die Anzahl der Knoten dadurch herausfinden, wie „lang“ eine Dimension des zweidimensionalen Arrays bzw. der zweidimensionalen Liste. Dies kann man, je nach Implementierung der jeweiligen Datenstruktur, in einer Komplexität von $O(n)$ oder $O(1)$ herausfinden.

Eine weitere grundlegende Kennzahl von Graphen ist dessen **„Größe“**. Die Größe beschreibt dabei die Anzahl der Kanten, die im Graphen vorkommen, also die Anzahl der Elemente in der Menge E. Man schreibt analog zur Größe des Graphen: $|E|\ oder\ |E(G)|\ oder\ auch\ ||G||$. [Bal97; Die00] Auch diese Metrik ist weit verbreitet. So lässt sie sich in vielen Büchern zur Graphentheorie finden, aber auch in den genannten Mathematikbibliotheken [Sag20b; Mat20a; Wol20a]. Die Anzahl der Kanten innerhalb eines Graphen herauszufinden, erweist sich nicht ganz so trivial wie das Herausfinden der Ordnung. Ist ein Graph nicht gerichtet, d.h. seine Kanten haben keine feste Richtung [Die00] so ist seine Adjazenzmatrix symmetrisch. Man kann also zählen, wie viele Einträge es innerhalb der Matrix auf der Hauptdiagonalen und einer der Hälften gibt. Das ergäbe immer $\frac{1}{2}n^2$ Schritte, wenn n die Ordnung des Graphen ist. Die Komplexität beträge also $O(n^2)$. Bei der Darstellung durch eine Inzidenzliste wäre das anders. Hier könnte einfach die Größe der Liste gesucht werden und man wüsste die Größe des

Graphen. Die Komplexität wäre hier, wie schon erwähnt, je nach Implementierung $O(n)$ oder $O(1)$.

Der Grad eines Knotens

Während die zwei ersten vorgestellten Metriken vor allem den Graphen als ganzes beschreiben, ist es auch noch wichtig zu wissen, was einen einzelnen Knoten auszeichnet, um einen Graphen besser zu beschreiben. Hierzu gibt es die grundlegende Metrik des **Grades** eines Knotens. Der Grad eines Knotens beschreibt die Anzahl der mit einem Knoten inzidenten Kanten [Die00]. D.h. er drückt aus, wie viele Kanten mit einem Knoten verbunden sind. Man kann dies z.B. durch eine Funktion ausdrücken, die einen Knoten v auf eine natürliche Zahl abbildet: $d(v)$.

Auf Basis dieser Metrik lässt sich auch andere verwandte Metriken ableiten. Hierzu gehört der **„Minimalgrad“** und der **„Maximalgrad“**. Der Minimalgrad ist der kleinste Knoten-Grad eines Graphen G: $\delta(G) := min\{d(v) \mid v \in V(G)\}$. Parallel dazu ist der Maximalgrad der größte Knoten-Grad eines Graphen G: $\Delta(G) := max\{d(v) \mid v \in V(G)\}$. Darüber hinaus kann man noch den **„Durchschnittsgrad“** eines Graphen bestimmen. Dieser bildet den Durchschnitt aller Knotengrade ab: $d(G) := (\sum_{v \in V(G)} d(v))/|V|$. [Die00]

Des Weiteren gibt es bei der Betrachtung eines gerichteten Graphen zusätzliche Abwandlungen der Metrik. Da hier die Kanten immer zu einem Knoten gerichtet sind unterscheidet man speziell zwischen dem **„Eingangsgrad“** und dem **„Ausgangsgrad“**. Der Eingangsgrad eines Knoten beschreibt dabei die Anzahl der Kanten, die auf einen Knoten „zeigen“. Der Ausgangsgrad zeigt wie viele Kanten von einem Knoten „weggehen“. [Bal97]

Auch der Grad eines Knotens und die meisten seiner verwandten Metriken sind weit verbreitet. So sind der allgemeine Grad, der Eingangsgrad, der Ausgangsgrad in allen drei betrachteten Mathematikbibliotheken vorhanden. SageMath unterstützt sogar nativ die Metrik „Durchschnittsgrad“. [Sag20b; Mat20a; Wol20a]

Die Berechnung eines Grades über eine Adjazenzmatrix oder eine Adjazenzliste ist in linearer Zeit lösbar ($O(n)$). Bei der Adjazenzmatrix muss einfach nur die jeweilige Reihe des zugehörigen Knotens durchlaufen werden und gezählt werden, wie häufig ein Eintrag für eine Kante enthalten. Mit Hilfe der Adjazenzliste kann einfach die Größe der Liste als Grad genommen werden, die dem Knoten zugehörig ist.

Anzahl der Zusammenhangskomponenten

Eine weitere Variante einen Graphen grundlegend zu beschreiben, besteht darin seine Zusammenhangskomponenten zu zählen. Hierfür ist es zunächst wichtig zu verstehen, was Zusammenhang bei Graphen bedeutet.

Ein Graph gilt dann als zusammenhängend, wenn gilt: $\forall a, b(a, b \in V \land a \neq b \implies Weg\_existiert(a, b))$. Anschaulich bedeutet das, dass es zwischen zwei beliebigen

Knoten immer einen Weg geben muss, der die beiden Knoten miteinander verbindet. Graphisch erscheint ein zusammenhängender Graph so, dass sich keine verschiedenen, klar voneinander trennbaren Komponenten erkennen lassen. Lassen sich aber einzelne Komponenten erkennen, die für sich stehen und nur als Subgraph als zusammenhängend gelten würden, hat man einen Graphen vor sich liegen, der nicht zusammenhängend ist. Die einzelnen Komponenten oder Partitionen eines nicht-zusammenhängenden Graphen werden „Zusammenhangskomponenten“ genannt. Innerhalb der Zusammenhangskomponenten gilt, wie erwähnt, natürlich wieder die Eigenschaft des Zusammenhangs. [Die00]

Die Anzahl der Zusammenhangskomponenten gibt nun an, wie viele Komponenten innerhalb eines Graphen vorhanden sind. Die Metrik ist sowohl in MatLab als auch in SageMath vertreten [Sag20b; Mat20b]. Zur Ermittlung der Anzahl wird sich eines einfachen Algorithmus bedient, der die Tiefen- oder Breitensuche nutzt, die jeweils jeden Knoten als „besucht“ markiert, den sie traversiert. Folgender Pseudocode beschreibt diesen Algorithmus:

```
1  Integer zaehleKomponenten (graph)
2    int komponentenanzahl = 0
3    for (knoten in graph)
4      if (knoten ist nicht besucht)
5        tiefen_oder_breitensuche(graph, knoten)
6      komponentenanzahl++
7    return komponentenanzahl
```

Da der Algorithmus von einem der Suchalgorithmen abhängig ist, bestimmt dieser die Komplexität zur Ermittlung der Kennzahl. Diese Algorithmen besuchen jeden Knoten genau einmal und gehen jede Kante ab. Es ergibt sich dadurch eine lineare Komplexität von $O(|V| + |E|)$.

2.2 Distanzmetriken

Innerhalb der Graphentheorie gibt es den Begriff des Wegs. Ein Weg beschreibt dabei einen Graphen, der Knoten in einer Reihe hinterander Verbindet. Somit hat der Anfangs- und End-Knoten den Grad 1 und alle „mittleren“ Knoten den Grad 2. Meist sucht man aber einen bestimmten Weg innerhalb eines bestehenden Graphen. Der Weg ist hier dann ein Teilgraph des ursprünglichen Graphen. Anschaulicher lässt sich ein Weg also als eine Folge von Kanten beschreiben, in der kein Knoten zweimal besucht wird. Die Länge eines Weges ist dabei die Anzahl der Kanten, die in einem Weg vorhanden sind. [Die00] Auf Basis des Weges und seiner Längen-Eigenschaft kann nun eine Reihe von Metriken definiert werden.

Abstand/Distanz

Der **„Abstand“** ist eine Metrik, die auf Basis von zwei Knoten innerhalb eines Graphen definiert wird. Sie beschreibt die Länge des kürzesten Weges zwischen den zwei Knoten, von denen man den Abstand wissen will. Aufgeschrieben werden kann die Metrik mittels einer Funktion, die für den Graph G zwei Knoten x und y auf eine natürliche Zahl abbildet: $d_G(x, y)$ [Die00] Diese Metrik ist wichtig als Basis für andere Metriken. Wie die bisherigen Metriken ist auch diese in den jeweiligen Bibliotheken vertreten [Sag20b; Mat20d; Wol20a]. Zur Berechnung der Metrik kann auf verschiedene bekannte, graphtraversierende Algorithmen zurückgegriffen werden. Dazu zählen die Breitensuche, der Djikstra-Algorithmus oder der Bellman-Ford-Algorithmus [Sag20b]. Somit ist auch die Komplexität zum Herausfinden der Metrik gleich mit der der Algorithmen. So wäre bei einer Breitensuche eine Komplexität von $O(|V|+|E|)$ zu erwarten, da jede Kante und jeder Knoten abgegangen wird. Der Djikstra-Algorithmus hingegen hat eine Komplexität von $O(|V|^2)$ [Jun13].

Extrenzität eines Knotens

Mit Hilfe des Abstandes lässt sich nun u.a. die **„Extrenzität“** eines Knotens bestimmen. Die „Extrenzität“ ist der maximale Abstand den ein Knoten von einem anderen Knoten in einem Graphen G haben kann. Eine einfache formale Notierung für den Knoten x wäre: $ecc(x, G) = \max_{y \in V(G)}\{d_G(x, y)\}$, wobei x der gegebene Knoten ist. [Har01] Herauszufinden ist diese Kennzahl beispielsweise über den Djikstra-Algorithmus, der den kürzesten Abstand zu jedem anderen Knoten sucht und dann einfach der größte zu wählen ist. Das bedeutet im Umkehrschluss, dass die Extrenzität die Komplexität einer der vorherigen Algorithmen aufweist, da Algorithmen wie der Djikstra-Algorithmus oder die Breitensuche immer die Abstände zu allen anderen Knoten finden. Hierbei lässt sich auch der größte Abstand zu einem anderen Knoten finden. Die Extrenzität eines Knotens ist anschließend noch für andere Metriken eine wichtige Basis und allgemein weit verbreitet in den genannten Bibliotheken [Sag20b; Mat20d; Wol20a].

Durchmesser

Wie bereits erwähnt, ist es nun möglich auf Basis des Abstands weitere Metriken zu definieren. Hierzu zählt unter anderem auch der **„Durchmesser“** eines Graphen. Der Durchmesser beschreibt den größten Abstand zweier Knoten im Graphen G. [Die00] D.h. es ist der Abstand von allen Knoten zu allen Knoten zu berechnen und davon die größte Zahl auszuwählen. Formal notiert lässt sich die Metrik folgendermaßen beschreiben: $Durchmesser(G) = \max_{x,y}\{d_G(x, y)\}$. Man kann auch anders sagen, dass beim Durchmesser die maximale Extrenzität des Graphen gesucht ist. Nimmt man zur Ermittlung der Abstände dabei den Djikstra-Algorithmus und wendet diesen dann jeweils auf jeden

einzelnen Knoten an, kann eine Komplexität von $O(|V|^3)$ angenommen werden, um die Metrik zu extrahieren. Auch diese Metrik lässt sich z.B. in SageMath oder Wolfram finden. In Matlab kann der Durchmesser über die Distanzmatrix ermittelt werden, die Matlab erstellen kann. [Sag20b; Mat20d; Wol20a] Die Anwendung für diese Metrik kann z.B. sein, rein topologische Eigenschaften des Graphen herausfinden zu wollen. Allerdings kann auch in realen Problemen der Durchmesser als Metrik auftauchen. So ist z.B. der Abstand und damit der Durchmesser auch mit gewichteteten Kanten berechenbar. [Sag20b; GIT14] In einem Navigationssystem wäre der Durchmesser dann die längste fahrbare Strecke ohne einen Knoten doppelt zu besuchen oder Umwege zu fahren.

Radius

Parallel zum Durchmesser eines Graphen kann man auch dessen **„Radius“** bestimmen. Hierfür wird die Metrik der Extrenzität wichtig und der Begriff der Zentralität. Ein Knoten ist dann *zentral* bzw. im Zentrum eines Graphen, wenn dessen Extrenzität minimal ist. Dies kann nur einen Knoten, aber auch mehrere Knoten betreffen. Die minimale Extrenzität in einem Graphen, die die Knoten des Zentrums haben, nennt man dann auch den **„Radius“** des Graphen. Geschrieben wird $rad(G) = \min_{x \in V(G)} \max_{y \in V(G)} d_G(x, y)$. [Die00] Der Radius lässt sich grundsätzlich auf die gleiche Weise herausfinden, wie der Durchmesser und hat dementsprechend die gleiche Komplexität. Des Weiteren ist diese Metrik auch weit verbreitet und lässt sich in allen untersuchten Bibliotheken finden [Sag20b; Wol20a; Mat20d]

2.3 Kreis-basierte Metriken

Ausgehend von einem Weg innerhalb eines Graphen können wir auch den Begriff des Kreises definieren. Ein Kreis ist dabei einfach ein Weg, der eine zyklische Eckenfolge hat. Sei x_i ein Knoten, so hat ein Kreis folgende typische Eckenfolge: $x_0 ... x_{k-1},\ x_0$. Ist ein solcher Kreis, wie beschrieben gegeben, kann unter anderen an diesem seine Länge abgelesen werden. Die Länge beschreibt dabei die Anzahl der Kanten, die ein Kreis enthält. Ist erst einmal ein Kreis gegeben, lässt sich die Länge leicht berechnen, denn die Länge eines Kreises ist gleich mit der Anzahl der Knoten innerhalb des Kreises. [Die00] Auch auf Basis dieser Kreise und ihrer Länge lassen sich verschiedene Graph-Metriken definieren.

Taillenweite und Umfang

Eine dieser Metriken ist die „Taillenweite“ des Graphen. Die „Taillenweite“ ist so definiert, dass sie den Wert der Länge des kürzesten Kreises innerhalb des Graphen annimmt. Umgekehrt lässt sich auch der „Umfang“ des Graphen bestimmen. Der Umfang ist

dabei so groß wie die Länge des größtmöglichen Kreises innerhalb eines Graphen. Hat ein Graph keinen Kreis, so ist es nicht möglich für beide Kennzahlen einen Wert zu ermitteln. Allerdings haben diese dann einen festen Wert. So beträgt die „Taillenweite“ in diesem Fall ∞ und der Umfang null. [Die00]

Um die Taillenweite und den Umfang eines Graphen herauszufinden, ist es grundsätzlich notwendig die jeweiligen Zyklen innerhalb des Graphen herauszufinden. Hierfür lässt sich unter anderem eine modifizierte Tiefensuche nutzen, die mittels Markierung der Knoten erkennt, ob sie bereits schon einmal bei einem Knoten war und infolgedessen einen Zyklus erkennt. Hierbei wird unterschieden, ob ein Knoten noch nicht bearbeitet wurde, in Bearbeitung ist oder bereits der Algorithmus auf ihm vollständig abgeschlossen wurde. Wird der Algorithmus auf einem Knoten aufgerufen, der sich noch in Bearbeitung befindet, ist ein Zyklus gefunden. Mit einer richtigen Ausgabe kann dieser dann auch benannt werden. Daraus folgt auch, dass die Ermittlung der Taillenweite und des Umfangs komplexitätstechnisch der Tiefensuche entspricht, denn nach der Anwendung dieser, muss einfach der größte bzw. der kleinste Zyklus gewählt werden, um die Metriken zu ermitteln. Alternativ kann man schon während des Algorithmus Variablen halten, die Taillenweite und Umfang enthalten und diese während der Ausführung aktualisieren, falls einer der gefunden Zyklen einen der Werte aktualisiert. Die Komplexität beträgt damit $O(|V| + |E|)$. [Kne19; Vöc+08]

Zur Metrik „Umfang“ lassen sich in den genannten Bibliotheken, außer bei Wolfram, keine direkten Implementierungen finden. Allerdings ist es möglich in SageMath und in Wolfram die „Taillenweite“ direkt zu evaluieren. [Sag20b; Wol20c]

2.4 Zusammenhangsmetriken (Connectivity)

Neben der Definition von Metriken auf Basis von Distanzen ist es auch möglich, Kennzahlen zu ermitteln, die den Zusammenhang eines Graphen betrachten. Hierfür ist es wichtig zu verstehen, wann ein Graph als zusammehängend gilt und was eine Zusammenhangskomponente bzw. Partition eines Graphen ist. Dies wurde bei der Metrik „Anzahl der Zusammenhangskomponenten“ erläutert.

Dichte

Bei der Vorstellung grundlegender Graphmetriken wurden u.a. die Größe und die Ordnung eines Graphen erklärt. Darauf aufbauend kann eine Kennzahl ermittelt werden, die aussagt, wie stark vernetzt ein Graph ist. Die „Dichte“ eines Graphen gibt an, inwiefern der Graph so viele Kanten hat, wie es ihm theoretisch möglich ist. Die „Dichte“ setzt also die tatsächliche Kantenanzahl (Größe) und die mögliche Kantenanzahl in ein Verhältnis. Die Metrik kann darauffolgend einen Wert zischen 0 und 1 annehmen. Ist der Wert 0 hat der Graph keine Kanten. Ist der Wert hingegen 1 so hat man einen

vollständigen Graphen vor sich liegen. Möchte man die Dichte eines Graphen ermitteln, ist es nötig die Größe des Graphen durch die potenzielle Größe zu teilen. Dies ist mit folgender Formel möglich: $\frac{|E|}{\binom{|V|}{2}}$. Hat man statt einem ungerichteten Graphen einen gerichteten, muss die Formel leicht abgewandelt werden, da für einen vollständigen Graphen nun doppelt so viele Kanten nötig sind: $\frac{|E|}{2\binom{|V|}{2}}$. [Die00]

Zur Implementierung der Metrik ist es infolgedessen nur nötig die Größe und die Ordnung des Graphen herauszufinden. Die Komplexität zur Berechnung der „Dichte“ ist deshalb gleich der Komplexität zur Berechnung von Größe und Ordnung addiert. Die Berechnung der „Dichte“ selbst erfolgt in konstanter Zeit. Durch die Ableitung der Metrik aus zwei grundlegenden Kennzahlen ist die Berechnung auch problemlos möglich, ohne dass es eine explizite Implementierung in einer Bibliothek gibt. Allerdings bieten Wolfram und SageMath spezielle Funktionen für die „Dichte“ eines Graphen. [Sag20b; Wol20c; Mat20a]

Stärke

Oft repräsentieren Graphen ein Netzwerk. Im Rahmen von Netzwerken wird unter anderem von deren „Stärke“ gesprochen. Die „Stärke“ gibt dabei das minimale Verhältnis zwischen entfernten Kanten und dadurch erstellter Zusammenhangskomponenten an. Es muss dabei allerdings die Anzahl der Zusammenhangskomponenten insgesamt erhöht werden. Ist die „Stärke“ eines Netzwerks, bzw. eines Graphen, hoch, ist es u.a. schwieriger für einen Angreifer das Netzwerk stark zu beschädigen, greift dieser die Verbindungen, bzw. Kanten, des Netzwerks an. Zur Berechnung der Stärke $\sigma(G)$ seien folgende Annahmen gegeben: Sei Π die Menge aller möglichen Partitionierungen der Knoten V und $\partial\pi$ die Menge an Kanten, die entfernt werden müssten, um die Partitionierung π zu erreichen, gilt folgende Formel zur Errechnung der Stärke:

$$\sigma(G) = \min_{\pi \in \Pi} \frac{|\partial\pi|}{|\pi| - 1}$$

Es wird also jede mögliche Partitionierung der Knoten V betrachtet und ermittelt welche Kanten man entfernen müsste, um diese Partitionierung der Knoten zu erhalten. Die Anzahl der Elemente in der jeweiligen Menge werden dann in ein Verhältnis gesetzt. Dabei wird eine Zusammenhangskomponenten aus π subtrahiert, da ein Graph immer zumindest aus einer Komponente besteht. Aus all diesen erstellten Verhältnissen ist nun das Minimum das Ergebnis. [Tru93; Cun85] Es kann auch anders gesagt werden, dass ein Graph bei dem die Entfernung weniger Kanten zu vergleichsweise vielen Zusammenhangskomponenten führt, ein sehr „schwacher“ Graph ist. Umgekehrt ist es bei einem „starken“ Graphen nicht möglich, selbst durch die Entfernung vieler Kanten (Zähler), eine vergleichsweise hohe Anzahl an Zusammenhangskomponenten (Nenner) zu erreichen.

Die Berechnung der „Stärke“ und die Verbesserung der Komplexität des Algorithmus war Thema mehrerer wissenschaftlicher Arbeiten. Die beste erreichte Komplexität erzielte dabei V. A. Trubin mit einer Komplexität von $O(\min(\sqrt{m}, n^{2/3})mn\log(n^2/m+2))$. m ist hierbei die Anzahl an Kanten im Graphen, n die Anzahl an Knoten. [Tru93]

In SageMath, Wolfram oder MatLab ist die „Stärke“ von Graphen nicht implementiert. Darüber hinaus ist es auch möglich statt über die Entfernung von Kanten die Metrik über die Entfernung von Knoten definieren. In diesem Fall spricht man über die „Härte“ oder „Zähigkeit“ (engl. „Toughness“) des Graphen. [Chv06]

„Vertex Connectivity“/Zusammenhang

Die Stärke eines Graphen ist eine nicht ganzzahlige Metrik zur Beschreibung des allgemeinen Zusammenhangs innerhalb eines Graphen. Wie bei der Stärke schon erwähnt, ist diese Metrik in den einschlägigen Bibliotheken nicht zu finden. Die nächsten zwei Metriken sind sowohl in SageMath als auch in Wolfram zu finden und beschreiben die Stärke des Zusammenhangs des Graphen mittels einer ganzen Zahl [Sag20b; Wol20a]

Die erste dieser Metriken ist die **„Vertex Connectivity“** oder der **„Zusammenhang“**. Hierbei wird ein zusammenhängender Graph betrachtet und ermittelt wie viele Knoten aus dem Graphen mindestens entfernt werden müssen, sodass der Graph nicht mehr zusammehängend ist. *k* entspricht dieser minimalen Anzahl an Knoten. Formal lässt sich sagen, dass ein Graph *k-zusammenhängend* ist, wenn $k < |G|$ und der Graph für jede mögliche Knotenmenge X mit der Mächtigkeit $< k$ zusammenhängend bleibt, sobald man alle Knoten $X \subseteq V$ aus V entfernt ($G - X$). Da ein Graph der *4-zusammenhängend* ist auch *3/2/...-zusammenhängend* ist, ist die finale „Vertex Connectivity“ bzw. der „Zusammenhang“ das größtmögliche *k*, das für den Graphen G möglich ist. Der Zusammenhang ist dann 0, wenn der Graph von Anfang an nicht zusammenhängend ist oder der Graph nur aus einem Knoten besteht. [Die00] Auch bei dieser Metrik gilt wie bei der Stärke, dass der Graph schwerer zu „trennen“ ist, je höher die jeweilige Kennzahl ist. Daraus ist auch zu folgen, dass bei hohem Zusammenhang, beispielweise ein Netzwerk, weniger anfällig für Angriffe ist.

Um den Zusammenhang eines Graphen algorithmisch herauszufinden, können zunächst zwei triviale Fälle abgedeckt werden. Ist ein Graph leer oder trivial (nur ein Knoten) ist der Zusammenhang, wie bereits erwähnt, 0. Ist hingegen der Graph vollständig, beträgt der Zusammenhang $|V|$. Allerdings lässt sich auch ein allgemeiner Algorithmus definieren, der die „Vertex Connectivity“ berechnet. Für den Algorithmus wird eine zusätzliche Funktion benötigt. $N(a, b)$ nimmt zwei Knoten entgegen. Eine Menge an Knoten, die bei Entfernung dafür sorgt, dass zwischen a und b kein Weg mehr existiert, wird „Knoten-Separator“ genannt. N gibt nun die Mächtigkeit des minimalen „Knoten-Separators“ von a und b zurück. Sind a und b direkt mit einer Kante

verbunden, gibt es keinen „Knoten-Separator“. Sich diese Knotenpaare bei der Berechnung der „Vertex Connectivity“ anzusehen, ist unnötig und muss nicht betrachtet werden. Schlussendlich ist die Mächtigkeit des kleinsten minimalen „Knoten-Separators“ die gesuchte Kennzahl. Zur Berechnung gibt Shimon Even in seinem Buch „Graph Algorithms“ folgenden Algorithmus an, der für nicht vollständige Graphen funktioniert [Eve12]:

```
1  Vertex-Connectivity(V, E)
2    Sortiere Knoten v_1, v_2, ..., v_|V| so, dass es von v_1 keine
     direkte Kante zu irgendeinem v gibt
3    gamma = unendlich
4    i = 1
5    while i <= gamma
6      for each v, sodass v_i keine direkte Kante zu v hat gibt
7        gamma = min{gamma, N(v_i, v)}
8    return gamma
```

Der gezeigte Algorithmus terminiert mit γ gleich dem Zusammenhang. In seinen Ausführungen erläutert Even zudem, dass der Algorithmus eine Zeitkomplexität von $O(|V|^{1/2} \cdot |E|^2)$ aufweist.

„Edge Connectivity“

Ähnlich zur „Vertex Connectivity“ ist auch die „Edge Connectivity“ ein ganzzahliges Zusammenhangsmaß für einen Graphen und ist auch ähnlich definiert. Wie die „Vertex Connectivity“ ist diese in SageMath und Wolfram enthalten [Sag20b; Wol20a]. Die „Edge Connectivity“ ist nur definiert, wenn der Graph mindestens 2 Knoten hat. Hat ein Graph nur einen Knoten oder ist von Anfang an nicht zusammenhängend, so ist der auch sogenannte „Kantenzusammenhang“ von G $\lambda(G)$ gleich Null. Ansonsten wird der Kantenzusammenhang so definiert, dass es die minimale Zahl an Kanten ist, die aus einem Graphen entnommen werden kann, sodass dieser nicht mehr zusammenhängend ist. Präziser kann es folgendermaßen definiert werden: Ein Graph hat einen Kantenzusammenhang von $\lambda(G)$, wenn $G - F$ für alle Kantenmengen $F \subseteq E$ der Mächtigkeit $< \lambda(G)$ zusammehängend ist. Speziell ist damit das größtmögliche $\lambda(G)$ gemeint, das für den jeweiligen Graphen möglich ist, da auch hier gilt, das ein *4-kantenzusammenhängender* Graph zudem *3/2/...-kantenzusammenhängend* ist. [Die00] Die „Edge Connectivity“ bildet also das genau Pendant zur „Vertex Connectivity“ und arbeitet komplett analog zu dieser Metrik. Für die Berechnung dieser Metrik gibt es eine Reihe verschiedener Algorithmen, die die „Edge Connectivity“ in polynomialer Laufzeit berechnen. Beispielsweise hat der Algorithmus von David Matula eine Komplexität von $O(|V||E|)$. [Mat87]

2.5 Zentralitätsmetriken

Eine weitere wichtige Eigenschaft eines Graphen ist dessen Zentralität, bzw. dessen Zentralitäten. Bei der Untersuchung dieser Eigenschaften eines Graphen, möchte man herausfinden, welche Knoten oder allgemeiner Regionen eines Graphen besonders wichtig sind. Die Anwendungen für die Verwendung von Zentralitäts-Informationen ist dabei äußerst vielfältig. Vor allem die Analyse von realen sozialen Netzwerken wahr häufig der Ausgangspunkt für etwaige Untersuchungen. Aber auch bei Themengebieten wie Geographie, Stadtentwicklung und Organisationsaufbau wurde Zentralität zur Informationsgewinnung herangezogen. Allgemeiner lässt sich sagen, dass diese Thematik bei allen möglichen Anwendungen interessant sein kann, die Graph-Daten sammeln und die Wichtigkeit von Datenpunkten ermitteln wollen. Infolgedessen soll nun eine Reihe an Zentralitätsmetriken vorgestellt werden, die einen Graphen und speziell dessen Knoten auf diese Eigenschaft auf unterschiedliche Art und Weise untersuchen. [Fre78]

Degree Centrality

Die erste Metrik, die vorgestellt werden soll, ist zugleich die einfachste. Die „Degree Centrality“ wird für einen Knoten eines Graphen definiert und basiert bzw. ist gleich zu einer schon vorgestellten Metrik. Die „Degree Centrality“ oder „Grad-Zentralität“ eines Knotens gleicht dessen Knoten-Grad. D.h. $Degree\_centrality(v) = d(v); v \in V$. Die Berechnung der Zentralität erfolgt meist für jeden Knoten und kann im Fall der „Degree Centrality“ recht simpel berechnet werden. Liegt der Graph als Adjazenzmatrix vor, so muss nur über diese vollständig iteriert werden und für jeden Knoten mitgezählt werden, wie viele Nachbarn er hat. Die Komplexität beläuft sich damit auf $O(|V(G)|^2)$. Die Metrik selbst ist damit auch, wie der Knotengrad in den angeführten Bibliotheken zu finden, bzw. implizit berechenbar. Auch wenn die Metrik auf den ersten Blick recht rudimentär wirkt, so ist sie aber äußerst aussagekräftig z.B. bei der Analyse im Social-Media-Bereich. Geht man davon aus, dass jeder Knoten ein Nutzer oder eine Nutzerin ist und eine Kante eine Beziehung wie „Freund von“ oder „folgt“ gleichkommt, so gibt die „Degree Centrality“ an, wie wichtig der jeweilige User ist. Dies ist vor allem dann interessant, wenn man wissen möchte, ob eine Person besonders einflussreich ist oder nicht. Solche Informationen sind beispielsweise für Werbetreibene von Bedeutung. [Bha19]

Betweenness Centrality

Eine weitere Möglichkeit die Wichtigkeit eines Knoten in einem Graphen zu ermitteln ist über die sogenannte „Betweenness Centrality“. Auch diese Metrik wird häufig für die Analyse von sozialen Netzwerken genutzt. Besonders gibt die Kennzahl für einen Knoten an, wie viel Einfluss dieser hat für den Fluss der Information. Besonders wird

es genutzt, um Knoten zu finden, die als Brücke von einem Graph-Teil zum anderen fungieren. [neo20a]

Zur Berechnung der Metrik ist zunächst wichtig zu verstehen, was unter dem Wert σ_{st} und der Funktion $\sigma_{st}(v)$ zu verstehen ist, wobei gilt $s, t, c \in V(G)$. σ_{st} gibt an, wie viele kürzeste Wege es zwischen den beiden Knoten s und t gibt. Die Funktion $\sigma_{st}(v)$ bildet wiederum einen Knoten v aus G auf die Anzahl der kürzesten Wege zwischen s und t ab, die durch v gehen. Die Funktion ist dabei folgendermaßen definiert, bzw. kann so berechnet werden. [Bra01]

$$\sigma_{st}(v) = \begin{cases} 0, & \text{wenn } d_G(s,t) < d_G(s,v) + d_G(v,t) \\ \sigma_{sv} \cdot \sigma_{vt}, & sonst \end{cases}$$

Die erste Bedingung für $\sigma_{st}(v)$ gilt deshalb, weil v nur dann auf einem der kürzesten Wege zwischen s und t sein kann, wenn die kürzeste Distanz zischen s und t gleich der kürzesten Distanz zischen s und v addiert mit der kürzesten Distanz zwischen v und t ist: $d_G(s,t) = d_G(s,v) + d_G(v,t)$.

Die „Betweenness Centrality“ $C_B(v)$ ist nun so definiert, dass sie die Aufsummierung der Fraktion zwischen $\sigma_{st}(v)$ und σ_{st} für alle möglichen Paare s, t angibt, wobei s und t nie gleich sind und auch nicht gleich v sind:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Die Metrik kann so interpretiert werden, dass Knoten durch die öfter die kürzesten Wege eines Graphen gehen, auch wichtiger sind. Die Betweenness Centrality wird nämlich immer dann für einen Knoten höher, je mehr beliebige kürzeste Wege zwischen zwei anderen Knoten innerhalb des Graphen durch ihn laufen. Dies wird auch durch die Aussage am Anfang des Abschnittes forciert. Dient nämlich ein Knoten als Brücke zwischen bestimmten Graphteilen, ist natürlicherweise die Wahrscheinlichkeit höher, dass kürzeste Wege zwingendermaßen durch diesen laufen müssen. Dies ist vor allem der Fall, wenn ein Weg zwischen zwei Knoten gesucht wird, die jeweils in zwei unterschiedlichen Graphteilen sind.

Die Berechnung dieser Metrik ist in polynomialer Laufzeit möglich und kann mit einem Algorithmus berechnet werden, der eine Komplexität von $O(|V|^3)$ aufweist. Grundsätzlich ist es für die Berechnung zunächst notwendig zwischen allen Knotenpaaren die Anzahl und die Länge der kürzesten Wege zu berechnen. Auf Basis dieser Daten kann dann mit den obigen Funktionen die Kennzahl berechnet werden. In seinem Artikel „A Faster Algorithm for Betweenness Centrality“ zeigt Ulrik Brandes zudem einen Algorithmus, der die Metrik mit einer Zeitkomplexität von $O(|V| \cdot |E|)$ berechnen kann. Es ist zudem erwähnenswert, dass diese Metrik auch explizit für gewichtete Graphen berechenbar ist. [Bra01] Darüber hinaus ist sie in Wolfram, SageMath und auch Matlab nativ vertreten

und verfügbar. [Wol20a; Sag; Mat20c]

Closeness Centrality

Eine weitere Zentralitätsmetrik ist die „Closeness Centrality“. Auch diese Metrik beschreibt die Wichtigkeit eines Knotens. Dabei kann mit der Kennzahl vor allem ausgesagt werden, inwieweit ein Knoten effizient Informationen innerhalb eines Graphen verteilen kann, vorausgesetzt der Graph stellt eine Struktur dar, die diese Interpretation zulässt. [neo20b]

Die „Closeness Centrality“ misst die durchschnittliche invertierte Distanz zu allen anderen Knoten. Erzielt ein Knoten dabei einen hohen Wert, so hat dieser im Schnitt die kleinste Distanz zu allen anderen Knoten. Bei einer Interpretation der Metrik geht man also meist davon aus, dass ein Knoten der eine kurze Distanz zu allen anderen Knoten hat, auch wichtig sein muss. Für die Berechnung der Metrik ist es vor allem wichtig zu wissen, wie hoch die Distanz ist von dem zu untersuchenden Knoten v zu allen anderen. Auf Basis dessen lässt sich die Kennzahl folgendermaßen berechnen: [Coh+14]

$$C_C(v) = \frac{|V| - 1}{\sum_{u \in V} d_G(v, u)}$$

Bei Betrachtung der Metrik wird nun auch klar, warum gesagt werden, kann das die „Closeness Centrality“ angibt, wie gut ein Knoten Informationen weitergibt. Hat nämlich ein Knoten eine möglichst geringe Distanz zu allen anderen Knoten eignet er sich gut zum Verteilen von Informationen, da er im Schnitt hierfür die kürzesten Wege zurücklegen muss.

Die Berechnung dieser Metrik lässt sich in polynomialer Zeit unternehmen. Grundsätzlich sind die Distanzen für den Knoten v auszurechnen und anschließend die obige Funktion anzuwenden. Zur Berechnung der Distanzen kann man z.B., wie schon in 2.2 erwähnt, die Breitensuche verwenden. Als Pseudocode könne die Berechnung der „Closeness Centrality“ dann folgendermaßen aussehen:

```
1  Closeness_Centrality(G(V, E), v)
2    distanz_gesamt = 0
3    for each a in V\v
4      distanz_gesamt += d(v, a)
5    return (G.order - 1)/distanz_gesamt
```

Bedenkt man, dass die Breitensuche eine Zeit-Komplexität von $O(|V| + |E|)$ hat und die $|V|$-mal gemacht wird lässt sich leicht die Gesamtkomplexität der „Closeness Centrality“ für einen Knoten ermitteln: $O(|V|) \cdot O(|V| + |E|) = O(|V| \cdot (|V| + |E|))$. [Sar+13]

Diese Zentralitätsmetrik ist weit verbreitet und lässt sich in den MatLab-, SageMath- und Wolfram-Bibliotheken finden. [Mat20c; Sag; Wol20a]

Eigenvektor Centrality

Die „Eigenvektor Centrality“ ist auch eine Metrik für einen Graph-Knoten. Hierbei soll nicht nur darauf geachtet werden, inwieweit ein Knoten direkten Einfluss auf eine Netzstruktur hat, sondern auch seine transitive Wichtigkeit betrachtet werden. Anwendungen können z.B. Ranking-Systeme sein, die einem Daten-Knoten eine bestimmte Wichtigkeit zuordnen. [neo20c] Die grundlegende Idee der Metrik ist es durch die Transitivität nicht nur zu betrachten, wie wichtig der betrachtete Knoten selbst ist, sondern auch mit einzubeziehen, wie wichtig seine Benachbarten Knoten sind. So ist ein Knoten, der wichtig ist und dazu noch wichtige Nachbarn hat, unter Umständen wichtiger als ein Knoten, der zwar selbst als wichtig eingeschätzt wird, aber keine wichtigen Nachbarn hat.

Die Berechnung der „Eigenvektor Centrality“ basiert auf der Adjazenzmatrix des Graphen. Hierbei ist beim jeweiligen Eintrag $a_{v,t}$ eine Eins eingetragen, falls eine Kante zwischen v und t vorhanden ist, anderen Falls ist eine Null eingetragen. Die „Eigenvektor Centrality“ x von Knoten v ist nun folgendermaßen definiert ($M(v)$ ist die Menge an adjazenten Knoten von v): [BP15]

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

Übersetzt ist die „Eigenvektor Centrality“ also die Aufsummierung der „Eigenvektor Centrality“ aller Nachbarknoten geteilt durch λ. Formuliert man die Formel um und betrachtet sie im Kontext der gesamten Adjazenzmatrix, so kann man auch schreiben: $Ax = \lambda x$, wobei A die quadratische Adjazenzmatrix ist und x der Vektor mit den jeweiligen „Eigenvektor Centrality“-Werten. Hierbei fällt auf, dass dies gleichzeitig auch die Formel für den Eigenvektor und Eigenwert einer Martix ist. Aufgrund dessen erklärt sich auch der Name der Metrik. Stellt man nämlich die Eigenvektor-Formel um, so ergibt sich $(Ax/\lambda) = x$. Betrachtet man in dieser Rechnung nur einen Knoten so ergibt sich durch die Konsequenz der Matrix-Multiplikation die erstgenannte Funktion für x_v. Laut dieser Ausführungen ist λ ein Eigenwert der Adjazenzmatrix. Allerdings wird λ ausdrücklich als Konstante innerhalb dieser Metrik aufgefasst. Warum ist λ konstant? Es ist definiert, dass der Eigenvektor x nicht negativ ist, da die „Eigenvektor Centrality“ für einen Knoten nicht negativ sein kann. Nach dem Perron-Frobenius-Theorem kann mit dieser Einschränkung λ nur der größtmögliche Eigenwert für A und damit auch konstant sein.

Zur Berechnung der Kennzahl für jeden Knoten kann man sich beispielsweise der sogenannten „Power-iteration method“ bedienen. Diese Methode dient dazu, für eine gegebene Matrix möglichst genau einen Eigenvektor und einen Eigenwert zu finden. D.h. über Iterationen wird sich einem Ergebnis angenähert. Für das Verfahren im Falle der „Eigenvektor Centrality“ wird im ersten Schritt die Adjazenzmatrix mit n Zeilen und

n Spalten mit einem n großen Spalten-Vektor multipliziert, der vollständig mit Einsen gefüllt ist. Der entstehende Spalten-Vektor wird normiert und für die nächste Iteration zur Multiplikation mit A verwendet. Dies macht man so lange bis der Spaltenvektor konvergiert, bzw. eine feste Anzahl an Iterationen durchlaufen wurde. Dieser Vektor ist dann idealerweise ein Eigenvektor von A und der berechnete Normalisierungswert ist der korrespondierende Eigenwert λ. Als Beispiel und zum besseren Verständnis sei die Abbildung 2.1 gegeben, bei der ein Beispiel durchgerechnet wurde. [Meg15] Der limitierende Faktor dieser Methode zur Berechnung der „Eigenvektor Centrality“ ist

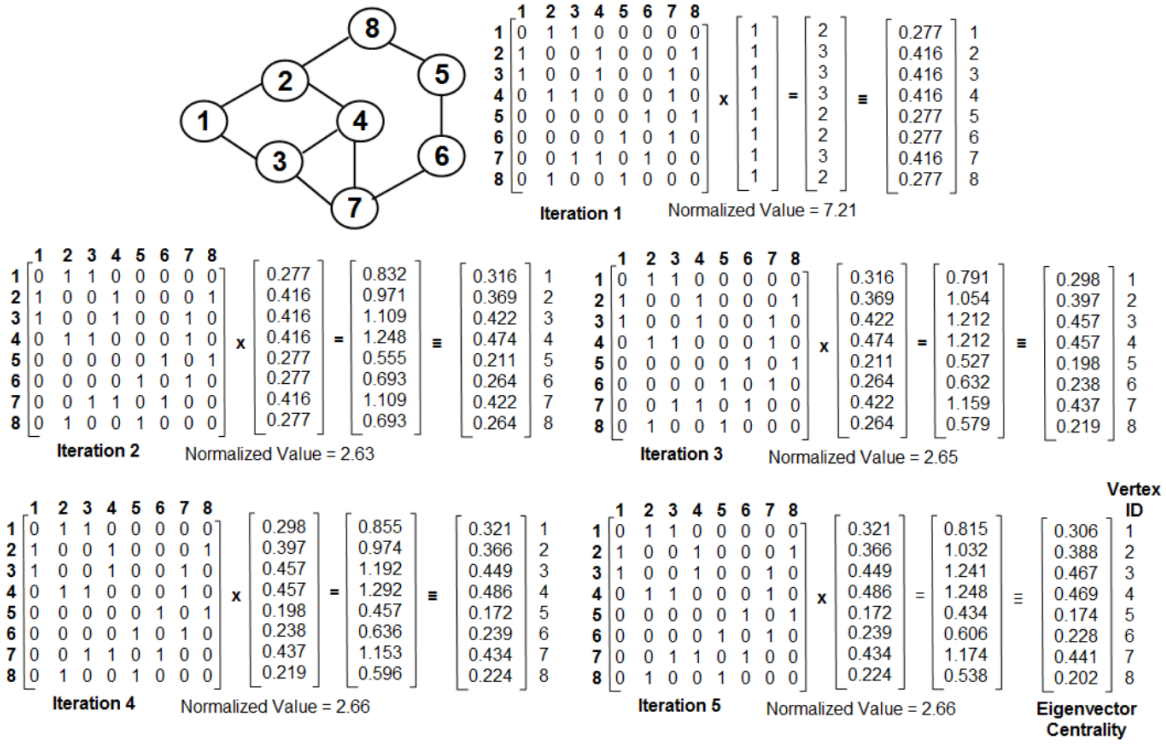

Abbildung 2.1: Eigenvektor Centrality: Power-iteration Method [Meg15]

die Matrix-Multiplikation. Implementiert man diese naiv so ergibt sich eine Komplexität von $O(n^3)$. Die Normierung lässt sich in linearer Laufzeit realisieren. Somit ist die Berechnung mit der „Power-iteration method“ in polynominaler Laufzeit möglich. Die Metrik selbst lässt sich nativ in Wolfram und in Matlab finden. [Wol20a; Mat20c]

Page Rank

Zum Abschluss der Zentralitätsmetriken soll eine Metrik dieser Kategorie vorgestellt werden, die einen hohen praktischen Nutzen findet. Der „Page Rank“ oder die „Page Rank Centrality“ beschreibt die Wichtigkeit eines Knotens auf Basis seines Ausgangsgrads, bzw. seiner Verbindungen zu anderen Knoten und des „Page Ranks“ seiner benachbarten Knoten. Hierbei ähneln sich „Page Rank“ und „Eigenvektor Centrality“. Die Metrik wurde erstmals in einem Google-Paper von Page und Brin veröffentlicht und soll Webseiten im World Wide Web bewerten. [BP98]

Es sei A eine Seite, bzw. ein Knoten und dieser Knoten hat Einwegkanten (Links) zu den Seiten $T_1 \dots T_n$. d sei ein frei wählbarer „Dämpfungs“-Faktor, der meist auf 0,85 gesetzt wird. Zudem ist $C(A)$ als der Ausgangsgrad für eine Seite (Knoten) definiert. Der „Page Rank“ ist dann folgendermaßen definiert.

$$PR(A) = (1-d) + d(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)})$$

Die Berechnung des „Page Ranks“ erfolgt durch einen iterativen Algorithmus, bie dem zunächst jedem Knoten der „Page Rank“ $1/|V|$ gegeben wird. Darauf aufbauend kann dann die obige Funktion zur weiteren Berechnung und die weiteren Iterationen herangenommen werden, bis die Metrik konvergiert. Der Algorithmus weißt dabei eine Zeit-Komplexität von $O(k \cdot |E|)$ auf, wobei k für die Anzahl der Iterationen steht. [Ora17] „Page Rank“ ist in allen angeführten Mathematikbibliotheken vertreten. [Sag; Wol20a; Mat20c]

2.6 Chromatische Zahl und chromatischer Index

Ein bekanntes Problem der Informatik ist das Färbeproblem. Es geht dabei darum einen Graphen so zu färben, sodass zwei benachbarte Knoten nicht die selbe Farbe haben. Anwendungen dafür kann man in vielen Problemen finden. Klassischerweise nimmt man das Beispiel der Karteneinfärbung bei der jedes Land ein Knoten ist und die Karte so eingefärbt werden soll, dass zwei benachbarte Länder nicht die gleiche Farbe haben. Allerdings lässt sich das Färbeproblem auch auf andere relevantere Probleme anwenden. Man kann z.B. so auch einen konfliktfreien Stundenplan erstellen oder andere ähnliche Konflikt-Probleme lösen. Die Knotenfärbung eines Graphen selbst ist eine Abbildung $c : V \mapsto S$, wobei S die Menge an möglichen Färbungen ist. [Die00; Aig15]

Chromatische Zahl

Auf Basis der Färbeproblems können nun eine Reihe an Metriken definiert werden. Betrachtet man das Problem mithilfe der Abbildungsfunktion c, so ist der Graph korrekt gefärbt, wenn gilt: $\forall v, w(v, w \in V \land adjazent(v, w) \implies c(v) \neq c(w))$. Die „chromatische Zahl“ beschreibt nun die Mächtigkeit der minimalen Menge S, für die diese Bedingung für den Graphen G zutrifft. Man bezeichnet die chromatische Zahl auch als $\chi(G)$. Für einen leeren Graphen ist die chromatische Zahl 1. Liegt ein bipartiter Graph vor, so ist die chromatische Zahl offensichtlicherweise 2. [Die00] Die chromatische Zahl kann sowohl von SageMath als auch von Wolfram berechnet werden [Sag20a; Wol20b]

Das Herausfinden der chromatischen Zahl ist ein NP-vollständiges Problem und kann deshalb vermutlich auch nicht effizient berechnet werden. [Weib; Kar96] So ist es

möglich mittels eines Brute-Force-Algorithmus sämtliche Färbemöglichkeiten durchzuarbeiten und dann die Färbung herausnehmen, die gültig ist und die niedrigste Anzahl an Farben hat. Dieser Ansatz hätte eine exponentielle Laufzeit, die schon bei kleinen Graphen zu extrem hohen Rechenzeiten führen würde. Neben dieser naiven Variante werden u.a. auch heuristische Algorithmen genutzt, die versuchen die chromatische Zahl effizient zu approximieren. Ein Greedy-Algorithmus hierfür ist folgendermaßen gegeben:

```
1  Chromatic-Number(G(V, E))
2    vertices <- sortiere_nach_Grad_absteigend(V)
3    highestColor <- 1
4    for (vertex in vertices)
5      color <- lowestPossibleColor(G, vertex)
6      vertex.color = color
7      highestColor <- Max{highestColor, color}
8    return highestColor
```

Dieser Algorithmus hat allerdings den Nachteil, dass es nur eine Annäherung an die chromatische Zahl ist und nicht gesichert ist, dass das Ergebnis korrekt ist. Seine Strategie besteht darin, den Graphen mit möglichst wenig Farben zu färben, indem zuerst der Knoten gefärbt wird, der die meisten Nachbarn hat.

Chromatischer Index

Neben der Markierung bzw. Färbung von Knoten, ist es auch möglich in einem Graphen Kanten zu färben. Hierbei ist ähnlich wie bei der Knotenfärbung gemeint, dass eine Kante aus der Menge $E(G)$ auf eine Zahl, bzw. Farbe abgebildet wird: $k : E \mapsto S$. Im Rahmen der Kantenfärbung ist es nun wichtig, dass zwei inzidente Kanten nicht die selbe Farbe zugewiesen bekommen. Das bedeutet, dass die Farben aller Kanten, die mit dem gleichen Knoten verbunden sind, eine unterschiedliche Farbe haben müssen. Formal lässt sich das mit der Funktion k so ausdrücken: $\forall v, w(v, w \in E \wedge inzident\_zu\_gleichem\_Knoten(v, w) \implies k(v) \neq k(w))$. Findet man eine Abbildung, für die diese Bedingung zutrifft, so nennt man den Graphen k-Kanten-färbbar. k ist dabei die Mächtigkeit der Menge S, der gefundenen Abbildung. Das niedrigst mögliche k für einen Graphen ist dann dessen „chromatischer Index“. Man schreibt dafür auch $\chi'(G)$. Der „chromatische Index“ ist sowohl in Wolfram, als auch in SageMath verfügbar. [And77; Sag20a; Wol20b]

Jeder Graph fällt bei Betrachtung seines chromatisches Indexes in eine von zwei Klassen. Bei der ersten Klasse ist der chromatische Index $\chi'(G) = \Delta(G)$, wobei $\Delta(G)$ der Maximalgrad des Graphen ist. Fällt der Graph nicht in diese Klasse, so ist sein chromatischer Index $\chi'(G) = \Delta(G) + 1$. Obwohl dabei der chromatische Index sich pro Graph in einem extrem engen Korridor aufhält, ist die exakte Ermittlung des Indexes ein NP-vollständiges Problem und somit vermutlich nicht effizient, also in polynomialer

Laufzeit, zu berechnen. Allerdings gibt es Spezialfälle, bei denen die Zuordnung des Graphen in einer der beiden Klassen einfach ist. So ist $\chi'(G) = \Delta(G)$, wenn der Graph bipartit ist. Ist der Graph vollständig und die Anzahl der Knoten ist gerade, ist dies ebenfalls der Fall. Ein kompletter Graph mit ungerader Knotenanzahl hat dann logischerweise den chromatischen Index $\chi'(G) = \Delta(G) + 1$. [Pla83]

Anwendung findet die Kantenfärbung in verschiedenen praktischen Problemen. Beispielsweise kann man es nutzen, um ein sogenanntes „Round-Robin"-Turnier mit möglichst wenig Runden zu planen. Ein „Round-Robin"-Turnier ist ein Turnier, bei dem jeder Teilnehmer auf jeden Teilnehmer einmal trifft. Dieses Problem ist zu lösen, indem jeder Teilnehmer durch einen Knoten repräsentiert wird und jede Begegnung durch eine Kante. Eine Farbe repräsentiert eine Runde. Werden die Kanten nun so gefärbt, dass der Turnier-Graph k-Kanten-gefärbt ist, hat man einen Turnier-Plan gefunden, der keine Konflikte hat. Ist der Graph zudem $\chi'(G)$-Kanten-gefärbt, so hat man den Plan gefunden, der möglichst wenig Runden erfordert. Der chromatische Index gibt in diesem Zusammenhang also an, wie viele Runden man minimal benötigt, sodass sich alle Teilnehmer einmal begegnen. Da der Graph bei dem sich alle einmal begegnen ein vollständiger Graph sein muss, ist auch schnell ersichtlich, wie viele Runden man braucht, betrachtet man die Ergebnisse des letzten Abschnitts. Neben „Round-Robin"-Turnieren lassen sich auch individuelle Turniere so planen. [GY04]

Fraktionierte chromatische Zahl

Neben der klassischen Art und Weise einen Graphen zu färben gibt es auch weitere Methoden. In der fraktionierten Graphentheorie ist es auch möglich, in einem Graphen seinen Knoten mehrere Farben zuzuweisen. Auf Basis dessen lassen wich weitere Metriken definieren, die die Graphenfärbung als Grundlage haben. Hierzu zählt die „fraktionierte chromatische Zahl". Um diese Zahl zu definierten, ist zunächst zu klären, wie fraktionierte Färbung formal definiert werden kann. Eine b-fache Färbung eines Graphen weist jedem Knoten eine Menge an b Farben zu. Auch dies kann durch eine Abbildung dargestellt werden, indem ein Knoten auf eine Menge an Färbungen abgebildet wird. Bei der Färbung muss nun darauf geachtet werden, dass zwei adjazente Knoten mit ihren zugewiesenen Farben keine Schnittmenge bilden. Besser kann dies durch eine Funktion dargestellt werden. Sei A die Menge an verfügbaren Farben, so kann die Färbung mit folgendermaßen dargestellt werden: $c : V \mapsto A^b$. Für die Graphen-Färbung muss nun gelten: $\forall v, w(v, w \in V \wedge adjazent(v, w) \implies c(v) \cap c(w) = \emptyset)$. Für eine b-Färbung wird eine bestimmte Anzahl a an Farben benötigt. Zu einer solchen Färbung wird dann auch gesagt, dass es eine $a : b$-Färbung ist. Das niedrigste a für das ein Graph eine b-Färbung hat, nennt man die b-fache chromatische Zahl ($\chi_b(G)$). Dabei ist logischerweise $\chi_1(G) = \chi(G)$. Die fraktionierte chromatische Zahl ist dabei im

Gegensatz folgendermaßen definiert [SU11]:

$$\chi_f(G) = \lim_{b\to\infty} \frac{\chi_b(G)}{b}$$

Die fraktionierte chromatische Zahl ist also der Grenzwert für die b-fache chromatische Zahl geteilt durch b, wenn b gegen unendlich geht. Im Gegensatz zur „normalen“ chromatischen Zahl ist die fraktionierte chromatische Zahl eine rationale und keine natürliche Zahl. Die Berechnung der Metrik ist ein NP-vollständiges Problem. Allerdings gibt es für einige Graphen vorgefertigte Werte, die genutzt werden können. So ist die fraktionierte chromatische Zahl eines zyklischen Graphen C_{2n+1} z.B. $2 + (\frac{1}{n})$. Der Algorithmus zur Berechnung der Kennzahl bedient sich einem linearem Programm und ist äußerst schwer zu berechnen. Die Komplexität steigt exponentiell zur Ordnung des Graphen. Eine native Implementierung dazu findet sich in SageMath. Wie bei der normalen Färbung kann auch die fraktionierte Färbung für diverse konfliktfreie Planungen und dergleichen verwendet werden. [Weib; Sag20a; SU11]

Fraktionierter chromatischer Index

Wie es auch den chormatischen Index bei einerfacher Graph-Färbung gibt, ist auch ein fraktionierter chromatischer Index für einen Graphen G ermittelbar. Hierbei geht es auch um die Färbung von Kanten. Statt einer Kante nur eine Farbe zuzuordnen, werden ihr b Farben zugeordnet. Auch das lässt sich wieder als Funktion darstellen: $k : E \mapsto A^b$. Ziel der Färbung ist es wieder, dass für zwei Kanten, die inzident zum selben Knoten sind, die Schnittmenge der Farben leer ist: $\forall v, w (v, w \in E \land inzident\_zu\_gleichem\_Knoten(v, w) \implies k(v) \cap k(w) = \emptyset)$. Ein Graph ist nun a-fraktioniert-kantenfärbbar für eine b-Kantenfärbung, wenn es möglich ist mit a Farben eine b-Kantenfärbung für den Graphen G zu finden, für die die obige Bedingung zutrifft. Findet man für eine b-Kantenfärbung das kleinste a hat man die korrespondierende „fraktionierte chromatische Kantenzahl“ $\chi'_b(G)$ gefunden, wobei wieder gilt $\chi'_1(G) = \chi'(G)$. Darauf aufbauend kann der „fraktionierte chromatische Index“ $X'_f(G)$ definiert werden [Weic; SU11]:

$$X'_f(G) = \lim_{b\to\infty} \frac{X'_b(G)}{b}$$

Die Berechnung ist allerdings aber auch anders möglich. Der fraktionierte chromatische Index kann nämlich mithilfe der fraktoinierten chromatischen Zahl berechnet werden. Es gilt: $\chi'_f(G) = \chi_f(L(G))$. $L(G)$ ist dabei der sogenannte korrespondierende Kantengraph von G. Bei einem Kantengraph werden alle Kanten von G genommen und in Knoten umgewandelt. Waren dann in G zwei Kanten inzident zum gleichen Knoten werden sie im Kantengraph mit einer Kante verbunden.

Die Berechnung der Metrik ist in polynomialer Laufzeit möglich. Bei der Betrachtung der Mathematikbibliotheken kam heraus, dass die Metrik ausschließlich in SageMath

vertreten ist. [Sag20b]

2.7 Arborizität

Ein wichtiger Teil der Graphentheorie ist die Betrachtung von Bäumen. Ein Baum ist dabei eine Zusammenhangskomponente eines Waldes. Ein Wald wiederum ist ein Graph, der keine Zyklen aufweist. Mithilfe dieses Wissens kann eine weitere Metrik definiert werden. Betrachtet man einen Graphen, zusammenhängend oder nicht, kann man sich die Frage stellen, wie es möglich ist, diesen Graphen aus einer Menge an Wäldern zu erstellen. Wichtiger noch ist es herauszufinden, aus welcher minimalen Anzahl an Wäldern es möglich ist, den vorliegenden Graphen aufzubauen. Diese minimale Anzahl ist die „Arborizität". Präziser ausgedrückt, ist die „Arborizität" von G ($Y(G)$) die minimale Anzahl an azyklischen Subgraphen (Wäldern), dessen Vereinigung G ergibt. Hierbei teilen sich die Subgraphen keine gemeinsamen Kanten. [Weia]

Zur Berechnung der Arborizität lassen sich sowohl Spezialfälle ausmachen, aber auch eine allgemeine Berechnungsvorschrift finden. So ist es zunächst ersichtlich, dass ein bereits azyklischer Graph eine Arborizität von $Y(G) = 1$ hat. Ein vollständiger Graph K_n hat wiederum eine Arborizität von $Y(K_n) = \lceil n : 2 \rceil$ und ein vollständig bipartiter Graph $K_{m,n}$ weißt den Wert $Y(K_{m,n}) = \lceil (mn) : (m + n - 1) \rceil$ auf. Möchte man die Arborizität allgemein berechnen, ist ein bestimmter Parameter des Graphen zu berechnen. m_p gibt die maximale Anzahl an Kanten eines Subgraphen von G an, der p Knoten hat. Dieser Wert m_p muss nun für alle $|G| > p > 1$ berechnet werden. Mit dieser Berechnung lässt sich die Arborizität mit folgender Formel für jeden G ermitteln. [Weia; Nas61]

$$Y(G) = \max_{p>1} \left\lceil \frac{m_p}{p-1} \right\rceil$$

Die Berechnung der Arborizität ist in polynomialer Laufzeit möglich und kann in verschiedesten Szenarien angewandt werden. Einerseits kann es als Graph-Dichte-Maß herangezogen werden, da ein dichter, mit vielen Kanten durchzogener Graph, logischerweise eine höhere Arborizität aufweisen muss. Wie bereits in vorigen Kapiteln erwähnt, kann die Dichte für die Zuverlässigkeit bzw. Angriffssicherheit eines Netzwerks herangezogen werden. Weiterhin kann die Arborizität aber auch z.B. Aussagen über die Steifigkeit von Strukturen machen oder bei der Analyse von elektrischen Netzwerken helfen. Die Metrik ist in SageMath fest eingebaut. [GW92; Sag20b]

Die Spezialisierung der Arborizität ist die „lineare Arborizität". Um diese Metrik zu definieren, muss zunächst der Begriff des „linearen Waldes" geklärt werden. Wie ein normaler Wald besteht ein linearer Wald ausschließlich aus azyklischen Graphen. Allerdings ist jede Zusammenhangskomponente des linearen Waldes ein Pfad. Man kann sich also einen linearen Wald graphisch als eine Ansammlung von vollständig

unverzweigten Graphen vorstellen. Jeder Knoten hat nur ein oder zwei inzidente Kanten. Die „lineare Arborizität“ $la(G)$ ist nun analog zur normalen Arborizität die minimale Anzahl an linearen Wäldern, dessen Vereinigung G ergibt. [Alo88]

Auch für die Berechnung dieser Metrik können Spezialfälle ausgemacht werden. Beispielsweise hat jeder d-reguläre Graph eine lineare Arborizität von $\lceil (d+1)/2 \rceil$. Außerdem konnte mithilfe des „Linear Arboricity Conjecture“ bewiesen werden, dass für alle planaren Graphen gilt, dass die lineare Arborizität $\lceil \Delta : 2 \rceil$ oder $\lceil (\Delta + 1) : 2 \rceil$ ist. Δ ist dabei der Maximalgrad von G. Im Gegensatz zur normalen Arborizität ist die Ermittlung der linearen Arborizität nicht in polynomialer Laufzeit berechnenbar. Die Ermittlung dieser Metrik ist ein NP-vollständiges Problem. Die Kennzahl ist zudem in keiner er genannten Bibliotheken vorhanden. [Pér84; CKL10]

2.8 Weitere Metriken

Nachdem nun eine weite Reihe an Metriken vorgestellt wurde und diese jeweils in eine Kategorie eingeordnet wurden, sollen in diesem letzten Teil Graph-Kennzahlen vorgestellt werden, die nicht in eine der Kategorien passen aber dennoch Erwähnung finden sollen.

Unabhängigkeitszahl

Bei der Betrachtung von Graphen ist es möglich sogenannte „unabhängige“ Knotenmengen zu finden. Eine „unabhängige“ Knotenmenge ist eine Untermenge S der Knoten $V(G)$ ($S \subseteq V(G)$), sodass keiner der Knoten in S adjazent zu einem anderen adjazent ist. Betrachtet man beispielsweise einen bipartiten Graphen, so ist ja dessen Definiton so gestaltet, dass der Graph in zwei „unabhängige“ Knotenmengen gegliedert werden kann. Um nun die sogenannte „Unabhängigkeitszahl“ $\alpha(G)$ für einen Graphen G zu ermitteln, muss man die größtmögliche „unabhängige“ Knotenmenge für G finden. Die Mächtigkeit dieser Menge ist dann die „Unabhängigkeitszahl“ von G [Die00; Weie; Weie] Unterstützung für diese Metrik lässt sich SageMath und Wolfram direkt oder indirekt (Ausgabe der größten „unabhängigen“ Knotenmenge) finden. [Sag20b; Weid]

Die Berechnung der „Unabhängigkeitszahl“ erfolgt durch das finden der größten „unabhängigen“ Knotenmenge. Dieses Problem ist NP-schwer und lässt sich durch einen navien Brute-Force-Algorithmus lösen. Dieser iteriert über jedes mögliche Subset von G und prüft, ob das Subset unabhängig ist. Ist über alle Untermengen iteriert worden, kann die Mächtigkeit der größten, unabhängigen Untermenge ermittelten werden. Die Zahl der zu untersuchenden Untermengen steigt exponentiell mit jedem weiteren Knoten. Durch Optimierungen und intelligente Algrotihmen kann das Problem mittlerweile in einer Zeitkomplexität von $O(1,1996^n)$ gelöst werden. Darüber hinaus gibt

es neben exakten Algorithmen zur Ermittlung auch approximierende Algorithmen, die schneller sind. [XN17]

Cliquenzahl

Verwandt mit der Unabhängigkeitszahl ist die sogenannte „Cliquenzahl“. Wenn von Cliquen bei Graphen die Rede ist, ist eine Untermenge in einem Graphen G gemeint, in der jeder Knoten adjazent zu jedem anderen Knoten innerhalb der Untermenge ist. Deutlicher kann man es so ausdrücken, dass eine Clique in einem Graphen eine Untermenge an Knoten ist, die gemeinsam einen vollständigen Graphen bilden. Die Cliquenzahl $\omega(G)$ ist nun die Mächtigkeit der größten Knotenmenge, die in G eine Clique formt. [Die00] Auch die Cliquenzahl lässt sich in Wolfram und in SageMath finden [Res15; Sag20b] Das auffinden von Cliquen („Cliquenproblem“) und damit auch das ermitteln der Cliquenzahl ist ein NP-vollständiges Problem und kann deshalb vermutlich auch nicht effizient gelöst werden.

Das Prinzip der Clique kann, wie der Name schon nahelegt, beispielsweise auf sozialen Netze übertragen werden. Stellen Knoten Personen dar und Kanten Freundschaften, können durch das Auffinden von Graph-Cliquen echte Personen-Cliquen gefunden werden. Die Cliquenzahl eines Graphen gäbe in diesem Kontext dann Größe der größten Clique innerhalb eines sozialen Netzes an.

Buchdicke/Seitenzahl

Als letztes soll noch eine geometrische Invariante untersucht und beschrieben werden: Die „Buchdicke“ bzw. „Seitenzahl“ $bt(G)$ eines Graphen G. Hierfür muss zunächst geklärt werden, was ein „Buch“ im Rahmen der Graphentheorie ist. Ein n-Buch, bzw. ein „Buch“ mit n Seiten, besteht aus einer Linie („spine“ oder „Rücken“) L, die sich in einem dreidimensionalen Raum aufhält und n Halb-Ebenen („Seiten“), die L als ihre gemeinsame Grenze haben, an denen sie sich treffen. In ein solches Buch kann man nun einen Graphen hineinlegen. Hierbei liegt jeder Knoten auf dem „Buchrücken“ L und jede Kante liegt auf einer der „Seiten“. Hierbei ist es wichtig, dass sich die Kanten auf einer Seite nicht kreuzen drüfen. Um dies besser zu verdeutlichen, wie sich dies graphisch wiederspiegelt sei Abbildung 2.2 gegeben. Gezeigt wird, wie ein vollständiger Graph mit fünf Knoten in ein Buch eingebettet wird. Die „Buchdicke“ bzw. „Seitenzahl“ $bt(G)$ ist nun das kleinsmögliche n das gefunden werden kann, damit der Graph in das korrespondierende n-Buch hineingelegt werden kann. [BK79] Die ermittlung von $bt(G)$, bzw. das Entscheidungsproblem, ob ein Graph G in ein n-Buch eingebettet werden kann, ist ein NP-vollständiges Problem. [CLR87]

Die „Buchdicke“ und das Einbetten von Graphen in ein „Buch“ hat mehrere Anwendungsszenarien. Eines davon ist die Planung einer Ampelschaltung einer Kreuzung.

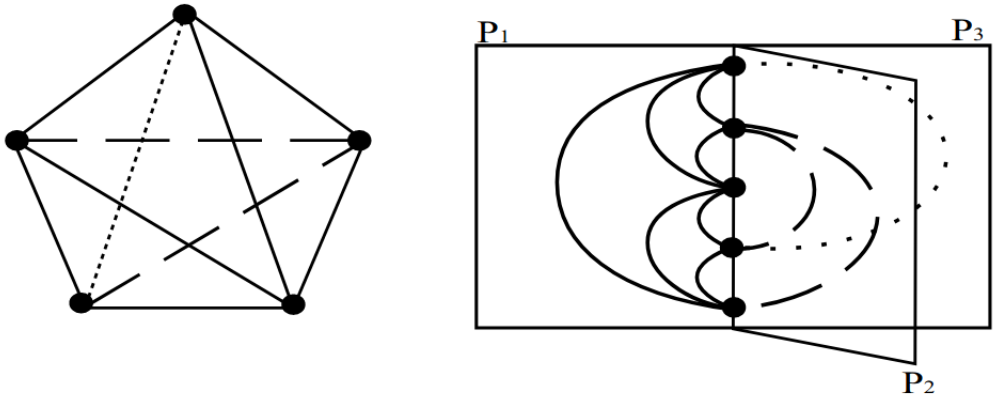

Abbildung 2.2: Einbettung K_5 in Buch [Bla03]

Hierbei betrachten wir die Knoten eines Graphen als die ein- und ausgehenden Straßenspuren (inkl. Fuß- und Radwege). Die Kanten sind die möglichen Wege, die ich von einem Knoten zum anderen begehen/befahren kann. Die Knoten sind nun so auf dem „Buchrücken“ zu arrangieren, dass ihre Reihenfolge auf dem „Rücken“ der Reihenfolge der Spuren im Uhrzeigersinn auf der Kreuzung gleicht. Die Kanten werden anschließend in die „Seiten“ eingebettet. Jede „Seiten“ repräsentiert dann eine Ampelphase, die so geschaltet ist, dass die Verbindungen der jeweiligen Seite befahrbar sind, ohne das es zu Konflikten kommt. Die „Buchdicke“ ist infolgedessen auch die minimal mögliche Anzahl an Ampelphasen an einer gegeben Kreuzung, die durch den Graphen G repräsentiert wird. [Kai90]

2.9 Übersicht der vorgestellten Graphmetriken

Nachdem nun eine weite Reihe an Graph-Metriken vorgestellt wurden, sollen diese mit Hilfe einer Aufzählung zusammengefasst werden. Aufgelistet werden jeweils der Name, bzw die Bezeichnung, der Metrik und ihre Definition. Zudem soll klar werden in welche der Kategorien die Metrik gehört Bei der Definition wird dabei eine mathematische oder eine wörtliche Beschreibung angegeben. Gegebenenfalls auch beides, falls es für das Verständnis förderlich ist.

Ordnung $|V(G)|$

Anzahl der Knoten eines Graphen

Größe $|V(E)|$

Anzahl der Kanten eines Graphen

Knotengrad $d(v)$

Anzahl der inzidenten Kanten des Knotens v

Minimal- ($\delta(G)$) und Maximalgrad ($\Delta(G)$)

Der kleinste bzw. größte Knotengrads des Graphen G; $\delta(G) := min\{d(v) \mid v \in V(G)\}$, $\Delta(G) := max\{d(v) \mid v \in V(G)\}$

Anzahl der Zusammenhangskomponenten

Anzahl der Komponenten/Subgraphen eines Graphen, die selbst zusammenhängend sind. Ein Graph (eine Komponente) ist dann zusammehängend, wenn sich zwischen zwei beliebigen seiner Knoten immer ein Weg finden lässt.

Abstand/Distanz $d_G(x, y)$

Die Länge des kürzesten Weges zwischen den beiden Knoten x und y.

Extrenzität eines Knotens $ecc(x, G)$

Der Maximale Abstand, den ein Knoten zu einem anderen Knoten im Graphen G haben kann. $ecc(x, G) = \max_{y \in V(G)}\{d_G(x, y)\}$

Durchmesser $Duchmesser(G)$

Der größte Abstand zweier Knoten innerhalb des Graphen G. $Durchmesser(G) = \max_{x,y}\{d_G(x, y)\}$.

Radius $rad(G)$

Kleinste Extrenzität innerhalb des Graphen. Extrenzität der Knoten des Zentrums. $rad(G) = \min_{x \in V(G)} \max_{y \in V(G)} d_G(x, y)$.

Taillenweite

Länge des kürzesten Kreises innerhalb eines Graphen

Umfang

Länge des größten Kreises innerhalb eines Graphen

Dichte

Anzahl der Kanten des Graphen geteilt durch die Anzahl der möglichen Kanten. Ungerichteter Graph: $\frac{|E|}{\binom{|V|}{2}}$. Gerichteter Graph: $\frac{|E|}{2\binom{|V|}{2}}$

Stärke $\sigma(G)$

Das Minimale Verhältnis zwischen entfernten Kanten und dadurch entstandenen Zusammenhangskomponenten. Sei Π die Menge aller möglichen Partitionierungen der Knoten V und $\partial\pi$ die Menge an Kanten, die entfernt werden müssten, um die Partitionierung π zu erreichen, gilt folgende Formel zur Errechnung der Stärke:

$$\sigma(G) = \min_{\pi \in \Pi} \frac{|\partial\pi|}{|\pi| - 1}$$

„Vertex Connectivity“

Minimale Anzahl k Knoten, die aus einem zusammenhängenden Graphen entfernt werden müssen, sodass er nicht mehr zusammenhängend ist.

„Edge Connectivity“

Minimale Anzahl k Kanten, die aus einem zusammenhängenden Graphen entfernt werden müssen, sodass er nicht mehr zusammenhängend ist.

Degree Centrality

Wichtigkeitsmaß für einen Knoten. Bewertet Knoten nach seinem Grad: $d(v); v \in V$

Betweenness Centrality $C_B(v)$

Wichtigkeitsmaß für einen Knoten, das angibt inwieweit beliebige kürzeste Wege zwischen zwei anderen Knoten durch ihn verlaufen. Sei σ_{st} Anzahl der kürzesten Wege zwischen Knoten s und t und $\sigma_{st}(v)$ die Anzahl dieser Wege, die durch v laufen.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Closeness Centrality $C_C(v)$

Zentralitätsmetrik, die Knoten danach bewertet wie weit sie von jedem Anderen Knoten entfernt sind.

$$C_C(v) = \frac{|V| - 1}{\sum_{u \in V} d_G(v, u)}$$

Eigenvektor Centrality x_v

Zentralitätsmetrik, die einen Knoten dahingehend bewertet wie vernetzt er ist und wie vernetzt seine Nachbarn sind. Wendet man die Metrik auf die gesamte Adjazenzmatrix A an, so sind die Eigenvektor Centralities aller Knoten im Eigenvektor x zu

finden, der nur positive Werte enthält.

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

Page Rank $PR(A)$

Praktisch angewandte Zentralitätsmetrik, die ähnlich wie die Eigenvektor Centrality einen Knoten nach seiner v+Vernetztheit und der Vernetztheit seiner Nachbarn bewertet. Sei A der zu untersuchende Knoten (Seite) und $T_1 \dots T_n$ die Knoten auf die von A aus gezeigt wird (Links), dass ist der Page Rank PR(A) so definiert:

$$PR(A) = (1-d) + d(\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{C(T_n)})$$

Chromatische Zahl $\chi(G)$

Die minimale Anzahl an Farben mit denen es möglich ist die Knoten eines Graphen so zu färben, das keine zwei adjazenten Knoten die gleiche Farbe haben.

Chromatischer Index $\chi'(G)$

Die minimale Anzahl an Farben mit denen es möglich ist die Kanten eines Graphen so zu färben, das keine zwei inzidenten Kanten die gleiche Farbe haben.

Fraktionierte chromatische Zahl $\chi_f(X)$

Statt einem Knoten nur eine Farbe zuzuweisen, können ihm auch b Farben zugewiesen werden. $\chi_b(G)$ gibt die minimale Zahl an unterschiedlichen Farben an, für die G so färbbar ist, dass zwei adjazente Knoten mit b Farben in ihren Farben eine leere Schnittmenge bilden. Die fraktionierte chromatische Zahl ist auf Basis dessen so formuliert:

$$\chi_f(G) = \lim_{b \to \infty} \frac{\chi_b(G)}{b}$$

Fraktionierter chromatischer Index $\chi'_f(X)$

Statt einer Kante nur eine Farbe zu geben, können ihr auch b Farben gegeben werden. $\chi'_b(G)$ gibt die minimale Zahl an unterschieldichen Farben an, für die G so färbbar ist, dass zwei zum gleichen Knoten inzidenten Kanten mit b Farben in ihren Farben eine leere Schnittmenge bilden. Der fraktionierte chromatische Index lässt sich daraufhin so formulieren:

$$X'_f(G) = \lim_{b \to \infty} \frac{X'_b(G)}{b}$$

Arborizität $Y(G)$

Minimale Anzahl an azyklischen Subgraphen (Wäldern), die zusammen wieder G ergeben. Sei m_p die maximale Anzahl an Kanten eines Subgraphen von G mit p Knoten, dann kann die Arborizität folgendermaßen berechnet werden:

$$Y(G) = \max_{p>1} \left\lceil \frac{m_p}{p-1} \right\rceil$$

Lineare Arborizität $la(G)$

Minimale Anzahl an Wäldern, die zusammen wieder G ergeben. Dabei sind die Zusammenhangskomponenten jedes Waldes ausschließlich Pfade.

Unabhängigkeitszahl $\alpha(G)$

Eine Knotenmenge ist dann unabhängig, wenn alle Knoten in ihr nicht adjazent zueinander sind. Die Mächtigkeit der größtmöglichen unabhängigen Knotenmenge ist die Unabhängigkeitszahl $\alpha(G)$

Cliquenzahl $\omega(G)$

Eine Clique ist eine Knoten-Untermenge von G, die einen vollständigen Graphen ergebibt. Die Mächtigkeit der größtmöglichen Knoten-Untermenge von G, die eine Clique ist, ist die Cliquenzahl $\omega(G)$

Buchdicke/Seitenzahl $bt(G)$

Ein Graph kann in ein Buch eingebettet werden, wobei die Knoten auf dem Buchrücken L angebracht sind und die Kanten über anliegende Seiten (Halbebenen) laufen und sich dabei nicht kreuzen dürfen. Die Buchdicke $bt(G)$ gibt nun an, wie viele Seiten man für einen Graphen G braucht, um ihn in ein Buch einbetten zu können.

---

3 Implementierung und Umsetzung der Graph-Metriken

Nachdem nun eine umfassende Anzahl an Graphmetriken besprochen wurde, ist es Ziel der Studienarbeit einen Teil dieser im Rahmen einer Klassenbibliothek umzusetzen. Die Entwicklung erfolgt dabei im Rahmen des klassischen Softwarelebenszyklus und besteht aus Anforderungsanalyse, Entwurf/Design, Implementierung und Testung. [Bal09; BL11] Darüber hinaus soll auch die Algorithmik einzelner Metriken betrachtet werden und infolgedessen auf die Betrachtungen aus Kapitel 2 angeschlossen werden.

3.1 Anforderungsanalyse

Im ersten Teil der Entwicklung sollen Anforderungen an die verschiedenen Softwareteile definiert werden. Unter Anforderungen (Requirements) werden dabei die Eigenschaften verstanden, die vom jeweiligen Softwaresystem erwartet werden. Zunächst wird definiert welche globalen Ziele die Implementierung der Graphmetriken verfolgt. Anschließend wird beschrieben, in welchen Rahmenbedingungen die Umsetzung stattfinden. Die Definition konkreter Anforderungen erfolgt im Anschluss und trennt sich in Anforderungen an die Graphdatenstruktur, auf der die Berechnungen stattfinden und in Anfordreungen an die eigentliche Berechung der Metriken. Es werden hierbei sowohl funktionale als auch nicht funktionale Anforderungen formuliert. Die hier durchgeführte Anforderungsanalyse orientiert sich bei den meisten Schritten an den aufgeführten Punkten und Schritten aus Helmut Balzerts „Lehrbuch der Softwaretechnik“ und folgt dabei in vielen Punkten der Anforderungsschablone der IEE 830-1998. [Bal09; IEE98]

3.1.1 Ziele der Graphmetriken-Implementierung

Finales Ziel der Graphmetrik-Implementierung soll es sein, eine Klassenbibliothek zur Verfügug zu stellen, die es einem Anwender oder einer Anwenderin ermöglicht, mithilfe einer bereitgestellten Graphen-Datenstruktur diverse Graphmetriken zu berechnen. Besonders wichtig und praxisrelevant sind dabei Kennzahlen zur „Centrality“. Dieses Hauptziel soll noch einmal in diese Unterziele untergliedert werden:

1. Die Klassenbibliothek soll eine Graph-Datenstruktur bereitstellten, die einerseits die grundlegenden Operationen eines Graphen beherrscht und andererseits es ermöglicht die Berechungen der Metriken auf ihr auszuführen. Dies ist nötig, da nur durch die Definition einer eigenen Datenstruktur eine Implementierung der Metriken möglich ist, da hierfür zumindest der Zugriff auf den Graph standardtisiert sein muss.

2. Die Klassenbibliothek soll eine Reihe von Klassen bereitsstellen, die es ermöglichen diverse Metriken für Graphen zu berechen. Die Berechnungen sollen dabei auf der vordefinierten Graph-Datenstruktur stattfinden, die zuvor definiert wurde. Dieses Ziel beschreibt die Hauptmotivation hinter dem Softwareprojekt.

3. Da es sich bei der Software um eine Klassenbibliothek handelt, ist es zudem zwingend notwendigt, dass die Bibliothek in andere Programme einbindbar ist. D.h. für einen anderern Entwicker oder eine Entwicklin muss es möglich sein, die Bibliothek in deren eigenen Programmen einzubinden und zu nutzen. Der letztendliche Sinn der Bibliothek soll es sein, das sie ggf. wiederauftretende Probleme im Zusammenhang mit Graphen lösen kann, ohne das ein Entwickler selbst die Datenstruktur oder die Metrik-Berechnung selbst implementieren muss. [MW17]

3.1.2 Rahmenbedingungen

Durch die Zielformulieren wurde auf einer abstrakten Ebene klar, was die zu erstellende Software können muss. Im folgenden soll geklärt werden, mit welchen Rahmenbedingungen die Klassenbibliothek erstellt werden soll. Es ist wichtig zu betrachten, welche Anwendungsbereiche das finale Produkt hat, wer die Zielgruppe ist und welche Betriebsbedingungen für die Bibliothek herrschen. Zudem müssen die technische Produktumgebung betrachtet und definiert werden. [Bal09]

Bei der Ausmachung des Anwendungsbereiches der Klassenbibliothek kann keine konkrete Räumlichkeit oder professionelle Arbeitsumgebung (writschaftlich oder akademisch) angegeben werden. Grundsätzlich ist der Anwendungsbereich dort, wo in einem Programm Graphen und Graphmetriken genutzt werden müssen. Wie bereits in Kapitel 1 erwähnt, ist der Graph und damit die zugehören Metriken vielverwendete Abstraktionen für viele Systeme. [Tur04] Damit ist auch der finale Anwendungsbereich einer solchen zugehörigen Bibliothek beliebig.

Da Anwendungsbereich der Klassenbibliothek weit gefasst ist, besteht auch die potenzielle Zielgruppe aus vielen verschiedenen Personengruppen. Was diese Personen aber eint, ist, dass es sich immer um Softwareentwickler oder -entwicklerinnen handelt, da eine Klassenbibliothek immer im Rahmen einer anderen Software verwendet wird. In diesem Zusammenhang sei aber zu erwähnen, dass die zu entwickelnde Bibliothek vor

allem im akademischen Kontext dieser Studienarbeit entsteht. Infolgedessen werden auch die realen Benutzer der Software sich in diesem Kontext bewegen.

Die Betriebsbedingungen der Bibliothek sind grundsätzlich beliebig. Die Ausführung der Programme kann auf jedem Computer geschehen, der die jeweiligen technischen Kriterien erfüllt, die anschließend in der technischen Produktumgebung beschreiben sind.

Die technische Produktumgebung ist folgendermaßen gegeben. Die Klassenbibliothek wird mittels der Programmiersprache **„Java“** umgesetzt. Das Softwareinkrement selbst wird „gebaut“ mittels des Abhängigkeitsmanagementtools **„Maven“**. Durch Maven wird es auch möglich sein die erstellte Software in andere Projekte einzubinden. Mit der Wahl von Java und Maven wird vorausgesetzt, dass der Endnutzer der Bibliothek die „Java Virtual Machine“ installiert und einsatzbereit hat. Weitere Voraussetzungen sind nicht vorhanden. Die Auswahl von „Java“ ist dadruch begründet, dass die Sprache sehr stark verbreitete und universell ausführbar ist, was u.a. dazu führte, dass es bereits einen großen Satz an Bibliotheken für die Sprache gibt. Das Verwenden von „Maven“ als Abhängigkeitsmanagement- und Build-Tool ist dahingehend sinnvoll, da es einerseits speziell für Java entwickelt ist, andererseits wichtige Eigenschaften in Sachen Testing und Artefaktbereitstellung aufweist. So kann, wie bereits erläutert, die erstellte Bibliothek in anderen Proejkten mittels Maven eingebunden werden. [Ull16; Sri11]

3.1.3 Anforderungen an die Graphdatenstruktur

Nachdem Ziel und Rahmenbedingungen des Softwareproduktes geklärt sind, müssen nun konkrete Anforderungen formuliert werden, die die Klassenbibliothek erfüllen soll. Im ersten Teil der Anforderungen wird bestimmt, welche Eigenschaften die Graph-Datenstruktur haben muss, die anschließend für die Berechnung der Metriken verwendet wird. Bei der Formulierung der Anforderungen wird natürliche Sprache verwendet. Zunächst werden funktionale Anforderungen beschrieben, anschließend nicht funktionale Anforderungen. [Bal09]

Zunächst ist es wichtig, dass einem potenziellen Verwender der Klassenbibliothek möglich gemacht wird, ein Objekt zur Verfügung zu haben, das einen Graphen repräsentiert. Genauer gesagt, muss die Bibliothek einem Programmierer oder einer Programmiererin, die diese verwendet, eine Möglichkeit geben, eine Datenstruktur zu erstellen, die einen Graphen repräsentiert. D.h. es wird eine Struktur geschaffen, die Knoten und Kanten kennt, wobei die Kanten eine Verbindung bzw. Beziehung zwischen Knoten darstellen. Besonders wichtig ist die Implementierung eines simplen und ungerichteten Graphen. Diese Anforderung korrespondiert dierekt mit dem ersten Produktziel. Die Graph-Datenstruktur stellt den zentralen Einsprungspunkt dar, um mit der Bibliothek zu arbeiten und muss daher über eine leicht zu verstehende Schnittstelle oder Abstraktion zu erreichen sein. Dies bedeutet, dass der Zugriff auf das jeweilige

Graphen-Objekt über eine einheitliche Schnittstelle erfolgt.

Darüber hinaus ist es wichtig neben der reinen Bereitstellung einer Datenstruktur auch zu definieren, welche Aktionen auf diese möglich sind. D.h. die Bibliothek muss es dem Nutzer oder der Nutzerin ermöglichen die Graphdatenstruktur zu benutzen. Zur Benutzung zählen die folgenden Aktionen bzw. Operationen:

- Das Hinzufügen eines Knotens.
- Das Löschen eines Knotens. Hierbei werden auch alle inzidenten Kanten gelöscht.
- Das Hinzufügen einer Kante.
- Das Löschen einer Kante.

Auch diese Anforderung ist essentiell für die Erreichung des ersten Projektzieles. Die jeweiligen Operationen sind Teil der Graph-Schnittstelle, wie sie in der ersten Anforderung beschreiben wurde. Jedes Graph-Objekt kann über die Schnittstelle dementsprechend manipuliert werden.

Ein Graph dient meist zur Repräsentation und Abstraktion eines bestimmten Sachverhaltes [Tur04]. Viele Beispiele dazu konnten z.B. in Kapitel 2 gesehen werden, wobei unter anderem Anwendungen für Metriken besprochen und vorgestellt wurden. Auch die Graphdatenstruktur muss es ermöglichen, dass eine gewünschte Abstraktion dargestellt werden. Die Bibliothek muss es also ermöglichen Knoten und Kanten im Graphen zu „markieren“. Durch die Markierung muss es zudem möglich werden, die Knoten eindeutig zu identifizieren. Diese eindeutige Identifikation dient anschließend auch zur Durchführung der diversen Aktionen, die auf den Graphen möglich sind. Beispielsweise ist das Einfügen einer Kante über die Angabe der jeweiligen Knotenmarkierungen möglich. Die verwendete Markierung soll dabei beliebig sein. Allerdings ist es notwendig, dass Markierungen untereinander vergleichbar sind, damit die Funktionalität des Graphen gewährleistet werden kann.

Neben dem Graphen selbst ist die Ermittlung diverser Metriken einer der Hauptziele der Klassenbibliothek. Um dies zu erreichen, ist es notwendig, dass ein Graph-Objekt es ermöglicht, Informationen über sich preis zu geben. Die Datenstruktur muss dementsprechend dem Nutzer oder der Nutzerin eine Möglichkeit zur Verfügung stellen, auf Informationen des Graphen zuzugreifen. Die Operationen, die hierfür nötig sind, sind die folgenden:

- Information, ob Graph einen bestimmten Knoten enthält.
- Inforamtion, ob Graph eine bestimmte Kante enthält.
- Herausgabe aller Knoten des Graphen.
- Herausgabe aller Kanten des Graphen.
- Auffinden eines Knotens über ein gleichwertiges Knoten-Objekt (nicht die gleiche Referenz)
- Auffinden aller Nachbarn eines Knotens

- Rückgabe einer Repräsentation des Graphen als Adjazenzmatrix.

Die Eigenschaft, die diese Anforderung dem Softwareprodukt verleiht, ist schlussendlich die Bereitstellung sämtlicher nötigen Informationen, die für die Berechung der Metriken von Bedeutung sind.

Die nächste funktionale Anforderung an die Graphdatenstruktur hat mit der bereits beschriebenen Eigenschaft zu tun, dass die jeweiligen Graph-Objekte über eine einheitliche Schnittstelle erreichbar sind. Die Klassenbibliothek muss es nämlich explizit ermöglichen, dass der Programmierer oder die Programmiererin verschiedenartige Graphen mit der Bibliothek verwenden kann. D.h. die jeweilige Implementierung der Datenstruktur muss frei sein, während der Zugriff genormt ist. Infolgedessen ist es möglich verschiedene Implementierungsarten für den Graphen umzusetzten und zu nutzen. Die Bibliothek muss dabei selbst mindestens eine Implementierung bereitstellten. Sie sollte, wenn das möglich ist, mehrere Graphimplementierungen zur Verfügung stellen.

Die bisherigen Anforderungen beschrieben vor allem welche Funktionen der Graph selbst haben muss. Bei der Arbeit mit Graphen und den korrespondierenden Problemen ist es allerdings meist erforderlich oder erwünscht, dass es möglich ist einen Graphen zu persistieren und umgekehrt in das Programm und den Hauptspeicher zu laden. Deshalb muss die Klassenbibliothek auch ermöglichen einen beliebigen definierten Graphen in eine Datei eines bestimmten Formates zu speichern. Gleichermaßen müssen solche Dateien auch eingelesen werden können.

3.1.4 Anforderungen an die Metriken-Berechnung

Durch die Definition der Eigenschaften der Graphdatenstruktur ist es nun auf Basis dessen möglich, Anforderungen zu definieren, die beschreiben, wie und welche Metriken berechnet werden. Bezug wird dabei auf die vorgestellten Metriken genommen, die in Kapitel 2 vorkamen. Da die Anzahl der Metriken allerdings verhältnismäßig hoch ist, werden im Zuge der Anforderungsformulierung Schwerpunkte gesetzt. Besonders wenig praktisch anwendbare Metriken sollen aus der implementierungstechnischen Betrachtung ausgeschlossen werden. Hohes Augenmerk soll auf die Zentralitätsmetriken gelegt werden (siehe Abschnitt 2.5).

Die erste funktionale Anforderung besteht darin, dass die Klassenbibliothek einen Mechanismus oder eine Struktur zur Verfügug stellen muss, die es dem Nutzer oder der Nutzerin ermöglichen Metrik-Kennzahlen aus einem erstellten Graphen zu extrahieren. Der Graph, der hierfür verwendet werden kann, ist durch die Schnittstelle vorgegeben, die im vorigen Kapitel erwähnt wurde. Mit dieser Anforderung ist nicht die Implementierung einer oder mehrerer konkreter Metriken gemeint, sondern der Entwurf eines Mechanismus, der bestimmt, wie Metriken-Berechnung adressiert werden kann.

Mit Hilfe eines Mechanismus zur Berechnung der Graphmetriken muss definiert werden, welche Metriken die Klassenbibliothek berechnen können soll. Hierbei sind

zunächst die grundlegenden Metriken wichtig (siehe Abschnitt 2.1). Die Metriken sind fundamentale Kennzahlen zur Beschreibung eines Graphen. Daher muss es die Bibliothek ermöglichen für einen Graphen diese grundlegenden Metriken zu berechnen. Dazu zählen die folgenden Kennzahlen:

- Ordnung (Order) eines Graphen
- Größe (Size) eines Graphen
- Knotengrad und zugehörig Minimal-, Maximal- und Durchschnittsgrad.
- Anzahl der Zusammenhangskomponenten

Neben diesen Basis-Metriken sind auch die Distanz-Metriken eine wichtige Menge an Metriken, die für die Implementierung relevant sind. Besonders die kürzeste Distanz zwischen zwei Knoten ist äußerst wichtig, da die Kennzahl Basis für weitere Metriken ist, die auch Teil der Anforderungen sind. Aufgrunddessen muss die Bibliothek eine Berechungslogik bereitstellen, die es ermöglicht folgende Distanzmetriken zu berechnen:

- Minimale Distanz zwischen zwei Knoten
- Extrenzität eines Knotens
- Durchmesser
- Radius

Wie anfangs beschrieben, sollen im Rahmen dieser Betrachtung die Zentralitätsmetriken besondere Betrachtung erfahren. In Kapitel 2 konnte gezeigt werden, dass diese Metriken besondere praktische Relevanz haben. So dienen sie z.B. zur Analyse in sozialen Netzen oder der Bewertung von Internetseiten. Infolgedessen muss die Klassenbibliothek dem Nutzer oder der Nutzerin es ermöglichen für einen Graphen die folgenden Zentralitätsmetriken zu berechnen:

- Degree Centrality
- Betweenness Centrality
- Closeness Centrality
- Eigenvector Centrality
- Page Rank

Praktische Anwendung findet auch die Kennzahl der chromatischen Zahl. Darüber hinaus genießt das Färbeproblem Prominenz in der Informatik. Deshalb muss die Bibliothek die Berechung der chromatischen Zahl zur Verfügug stellen. Zudem soll es auch möglich sein den chromatischen Index zu ermitteln. In Kapitel 2.6 wurde auch gezeigt, dass es für die chromatische Zahl einen approximierenden Algrotihmus gibt, der effizienter arbeitet als der klassiche Algorithmus. Auch dieser Algrotihmus muss im Rahmen der Klassenbibliothek umgesetzt werden, damit der dem Entnutzer der der Endnutzerin zur Verfügug gestellt werden kann.

Um auch eine Metrik zur Verfügung zu haben, die Aussagen dazu trifft, wie stark der Graph zusammenhängt, muss die Klassenbibliothek auch eine Kennzahl dieser Kategorie berechnen können. Infolgedessen soll hierfür die „Dichte“ eines Graphen berechenbar sein.

3.1.5 Nicht funktionale Anforderungen

In den letzten zwei Abschnitten der Studienarbeit wurde erläutert und definiert, welche funktionalen Eigenschaften die Klassenbibliothek haben soll. Im Folgenden wird nun dargelegt, welche nicht funktionalen Eigenschaften die Software haben soll. Hierbei wurde sich an den Merkmalen des Standards ISO/IEC 9126-1 orientiert. [Bal09] Dieser Standard heißt in seiner aktuellsten Fassung ISO/IEC 25010:2011.

Die ersten nicht funktionalen Anforderungen beschäftigen sich mit dem Aspekt der „Usability“. Da es sich um kein Softwareprodukt mit einer Benutzeroberfläche handelt, wird der Aspekt der Usability im Bezug auf die Verwendung der Bibliothek durch einen Entwickler oder eine Entwicklering betrachtet. Hierbei sind besonders zwei Eigenschaften wichtig und relevant. Im Rahmen der Usability muss die Bibliothek beim Verwenden der Graphdatenstruktur und der Metriken darauf achten, dass der Nutzer oder die Nutzerin das System leicht versteht und gut bedienen kann [Bal09]. Hierfür ist es besonders wichtig, dass darauf geachtet wird, wie Parameter, Felder und Methoden benannt sind. Aber auch die statische Struktur der Bibliothek muss einem logischen Aufbau folgen und konsistent sein. Sowohl die Verständlichkeit als auch die Bedienbarkeit eines Systems können auch auf die klassischen Usability-Eigenschaften übertragen werden, wie sie Jakob Nielsen in „Usability Engineering“ beschreibt [Nie10]. So fördert die Verständlichkeit eines Systems die Erlernbarkeit (*Learnability*) und die Befriedigung (*Satisfaction*). Die gute Bedienbarkeit geht wiederum mit einer erhöhten Benutzungs-Effizienz (*Efficiency*) daher. Neben diesen Usability-Zielen ist es für eine Klassenbibliothek zudem unabdingbar, dass sie möglichst fehlerfrei ist (*Errors*). Ohne diese Eigenschaft ist keine gute Usability zu erreichen, da ein Nutzer oder eine Nutzerin einer Klassenbibliothek davon ausgeht, dass die Programmlogik dieser möglichst fehlerfrei, bzw. vollständig fehlerfrei ist.

Auch die Eigenschaft, dass das Produkt die gewünschten Funktionalitäten bereitstellt, ist eine nicht funktionale Eigenschaften und soll selbstverständlich von der Klassenbibliothek erfüllt werden. Besonders die Eigenschaften der Agemessenheit und Genauigkeit sollen beachtet werden. Angemessenheit beschreibt die Fähigkeit einer Software geeignete Funktionen bereitzustellen, um die spezifizierte Aufgabe zu erledigen. Genauigkeit wiederum sagt aus, wie genau und akkurat die Software die Anforderungen erfüllt. Für beide Eigenschaften kann definiert werden, dass die Klassenbibliothek sowohl volle Angemessenheit als auch Genauigkeit für den Nutzer oder die Nutzerin liefern muss. Angemessenheit wird erreicht, wenn die formulierten Anforderungen wie be-

schrieben erfüllt werden. Die Genauigkeit der Software ist gegeben, wenn die jeweiligen Funktionen vollständig korrekt arbeiten.

Nicht nur müssen Funktionalität genau und angemessen bereitgestellt werden, sie müssen auch effizient ausgeführt werden. Besonders zwei Punkte sind dabei bei Computerprogrammen interessant. Das Zeitverhalten und das Verbrauchsverhalten. Beim Verbrauchsverhalten ist u.a. die Auslastung der Ressource Arbeitsspeicher gemeint. Im Rahmen der Studienarbeit soll die Klassenbibliothek möglichst zeit- und ressourcenschonend sein. Allerdings geniesen beide Eigenschaften nicht höchste Priorität. Vor allem bei der Berechnung einiger Metriken kann es bei großen Graphen zu längeren Antwortzeiten kommen. Vor allem bei NP-vollständigen Problemen ist die Berechnung unvermeidbar langsam.

Darüber hinaus ist auch die Wartbarkeit einen Systems äußerst wichtig, damit Änderungen und Fehlersuche- und Behebung erleichtert werden. Im Zuge der Wartbarkeit soll besonders die Eigenschaft der Testbarkeit hervorgehoben werden. Die Bibliothek muss eine hohe Testbarkeit aufweisen. D.h. speziell, dass besonders Methoden so konzipiert werden müssen, dass sie testbar sind. Nur so kann die Korrektheit der Algorithmen, die in der Klassenbibliothek Verwendung finden sichergestellt werden, der relevanten Programmzeilen.

Eine weitere nicht funktionale Eigenschaft, die die Bibliothek mitbringen soll, ist die der Anpassbarkeit und Installierbarkeit. Beide Eigenschaften fallen unter den Oberbegriff der Portabilität. Darunter ist zu verstehen, dass die Software anpassbar und variabel in dem Sinne sein muss, als das sich das System gut an verschiedene Umgebungen und Anwendungsszenarien anpassen kann, damit die verwendende Person die Funktionen des Softwareprodukt vielfältig einsetzen kann. Eine gute Installierbarkeit wiederum beschreibt, dass im Rahmen der dokumentierten technischen Voraussetzungen die Software leicht installiert und verwendet werden können muss. Ist die Installierbarkeit nämlich nicht ausreichend gut, ist die Hürde zur Verwendung der Bibliothek zu hoch.

3.1.6 Nutzbarkeit als Klassenbibliothek

Neben programmlogischen und nicht funkionalen Anforderungen ist noch eine Anforderung festzuhalten, die mit der Beschaffenheit als Klassenbibliothek der Software einhergeht. Nach dem dritten formulierten Software-Ziel ist festgeschrieben, dass im Sinne einer Bibliothek es möglich sein muss, diese innerhalb anderer Programme einzubinden. Auf Basis dessen lässt sich die Anforderung herausarbeiten, dass die Bibliothek für einen nutzenden Entwickler oder Entwicklerin in ein anderes Softwareprodukt einbinbar sein muss. In den Rahmenbedingungen wurde bereits festgelegt, dass diese Einbindung durch das Abhängigkeitsmanagementtool „Maven“ geschehen soll.

3.1.7 Use Cases der Klassenbibliothek

Durch die Ermittlung und Formulierung der Anforderungen an die Klassenbibliothek zur Berechung von Graphmetriken, können diese auch in Use Cases überführt werden. Ein Use Case beschreibt und repräsentiert eine Sequenz von Aktionen, die ein Akteur in Interaktion mit dem Softwaresystem ausführen kann, um ein bestimmtes Ziel zu erreichen. Die formulierten Anforderungen werden nun also in Use Cases überführt. So ist es besser möglich zu sehen mit welchen Aktionen die Aktoren auf das System Metriken-Klassenbibliothek zugreifen. Auch wird klar, welche Funktionen die Klassenbibliothek gegenüber dem Aktor bietet. [Bal09]

Bei der Bestimmung der Aktoren kann sich diesem Fall nur eine Entität ausmachen lassen: Der Entwickler bzw. die Entwicklerin, die die Bibliothek potenziell nutzt, indem diese beispielweise in andere Softwareprodukte eingegliedert wird. Weitere Akteure gibt es nicht, da eine Bibliothek kein System ist, dass von anderen Endbenutzern direkt verwendet wird.

Bei Bestimmung der Use Cases muss definiert werden welche Aktionen der Aktor mit den System durchführen kann. Die erste Anforderungsbetrachtung erfolgte über die Graphdatenstruktur und was diese leisten können muss. Aus diesen Anforderungen können drei Use Cases ausgemacht werden, die noch weiter konkretisiert oder erweitert werden können.

- Graph erzeugen
- Graph manipulieren
- Graph analysieren

Der Graph wird laut den Anforderungen über eine einheitliche Schnittstelle erreicht. Das bedeutet wiederum, dass der Use Case darum erweitert werden kann, dass eine bestimmte Graphimplementierung gewählt werden muss. Die Bibliothek erlaubt es auch auf Basis der Schnittstelle eine eigene Implementierung zu stellen, falls das vom Nutzer gewünscht ist. Der Use Case „Graph manipulieren“ ist abstrakt und wird durch Use Cases konkretisiert, die die Manipulationen beschreiben. Dazu zählt die Löschung von Knoten und Kanten und das Einfügen von Knoten und Kanten. Da Markierungen für Kanten und Knoten ist jeweils als Use Case beim Einfügen inkludiert. Auch die Analyse des Graphen ist ein abstrakter Use Case. Abgeleitet aus den Funktionen, die die Bibliothek diesbezüglich leisten muss, können zwei Use Cases abgeleitet werden. Zum einen kann der Akteur die Existenz eines Graphelements erfragen, andererseits sich bestimmte Graphelemente zurückgeben lassen. Dazu zählen alle Knoten des Graphen, alle adjazenten Knoten eines Knoten und so weiter.

Neben den Use Cases zur Graphdatenstruktur können noch zwei weiterte Use Cases definiert werden. Ein zentraler und wichtiger Use Case ist die Berechung einer Metrik für einen Graphen. Die jeweiligen Metriken, die berechnet werden sollen, wurden bei den

Anforderungen erwähnt. Ein weiterer Use Case ist das Einbinden der Klassenbibliothek in ein anderes Softwareprodukt. Die vorgestellten und definierten Use Cases werden im folgenden Use Case-Diagramm zusammengefasst (Abbildung 3.1).

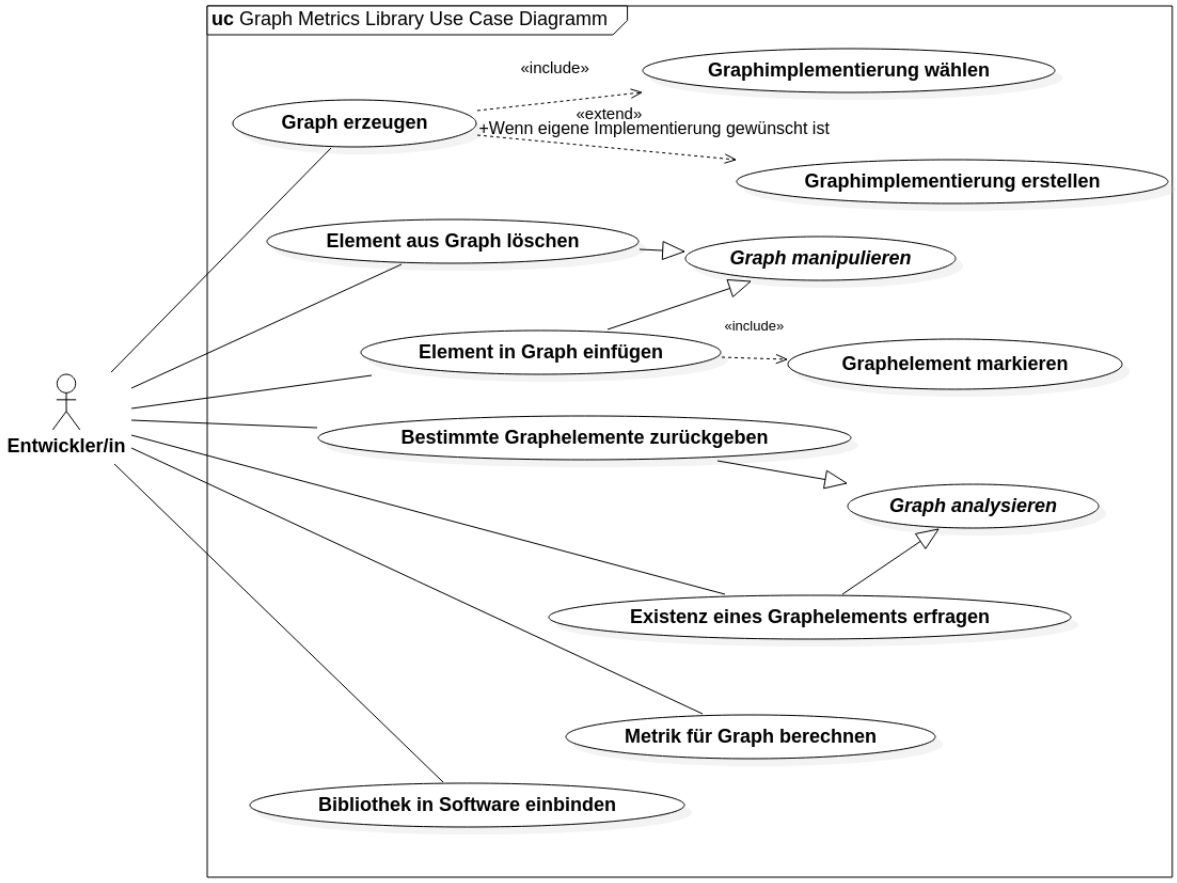

Abbildung 3.1: Use Case-Diagramm: Graphmetriken-Klassenbibliothek

Für die mit dem Entwickler assoziierten Use Cases sollen noch Use Case-Schablonen formuliert werden [Bal09]:

Use Case: Graph erzeugen

Ziel:	Ein Graphobjekt wurde innerhalb einer Variable oder eines Klassenfelds erzeugt.
Vorbedingung:	-
Nachbedingung Erfolg:	Neues Graphobjekt in gewünschtem Wertebehälter
Nachbedingung Fehler:	Fehlermeldung wird dem Nutzer oder der Nutzerin gemeldet.
Akteure:	Entwickler/in
Auslösendes Ereignis:	Entwicker oder Entwicklerin muss einen Graphen erstellen, der zur Repräsentation eines Sachverhalts dient.
Beschreibung:	Über einen Konstruktoraufruf wird ein neues Graphobjekt innerhalb eines Programms erstellt. Dabei muss eine bestimmte Graphimplementierung gewählt werden.
Erweiterung:	Ist eine andere Implementierung des Graphen gewünscht, kann auf Basis der vorgegebenen Schnittstelle eine eigene Implementierung erstellt werden.

Use Case: Element aus Graph löschen

Ziel:	Ein Element wurde aus einem Graphobjekt gelöscht.
Vorbedingung:	Das zu löschende Element ist in Graph vorhanden.
Nachbedingung Erfolg:	Element nicht mehr in Graph vorhanden.
Nachbedingung Fehler:	Fehlermeldung wird dem Nutzer oder der Nutzerin gemeldet.
Akteure:	Entwickler/in
Auslösendes Ereignis:	Entwickler/in will einen Knoten aus Graph löschen.
Beschreibung:	Über einen Zugriff auf das Graphobjekt wird ein Element identifiziert und gelöscht.
Erweiterung:	-

Use Case: Element in Graph einfügen

Ziel:	Ein Element wurde in ein Graphobjekt eingefügt.
Vorbedingung:	Das zu löschende Element ist noch nicht im Graphen vorhanden.
Nachbedingung Erfolg:	Element ist im Graphobjekt vorhanden.
Nachbedingung Fehler:	Fehlermeldung wird dem Nutzer oder der Nutzerin gemeldet.
Akteure:	Entwickler/in
Auslösendes Ereignis:	Entwickler/in will einen Knoten in Graph einfügen.
Beschreibung:	Über einen Zugriff auf das Graphobjekt wird ein neues Knotenobjekt eingefügt.
Erweiterung:	-

Use Case: Bestimmte Graphelemente zurückgeben

Ziel:	Der Nutzer oder die Nutzerin konnte bestimmte Knoten und Kanten aus dem Graphen extrahieren.
Vorbedingung:	-
Nachbedingung Erfolg:	-
Nachbedingung Fehler:	-
Akteure:	Entwickler/in
Auslösendes Ereignis:	Entwickler/in möchte Graph analysieren und Knoten und Kanten extrahieren.
Beschreibung:	Um den Graph zu analysieren muss man auf seine Elemente über eine Schnittstelle zugreifen können. Hierzu zählt das Ausgeben aller Knoten, aller Kanten und eines bestimmten Knotens.
Erweiterung:	-

Use Case: Existenz eines Graphelements erfragen

Ziel:	Der Nuter oder die Nutzerin weiß, ob ein bestimmter Knoten oder eine Kante im Graph existiert.
Vorbedingung:	-
Nachbedingung Erfolg:	-
Nachbedingung Fehler:	-
Akteure:	Entwickler/in
Auslösendes Ereignis:	Die Entwickler/in möchte erfahren, ob ein Knoten oder eine Kante im Graph existiert.
Beschreibung:	Über die Graphschnittstelle ist es möglich zu erfragen, ob der Graph einen bestimmten Knoten oder eine bestimmte Kante enthält.
Erweiterung:	-

Use Case: Metrik für Graph berechnen

Ziel:	Der Nutzer oder die Nutzerin konnte sich vom Programm eine Metrik für einen bestimmten Graphen berechnen lassen.
Vorbedingung:	-
Nachbedingung Erfolg:	Die korrekte Kennzahl wurde zurückgegeben.
Nachbedingung Fehler:	Fehlermeldung wird dem Nutzer oder der Nutzerin gemeldet.
Akteure:	Entwicker/in
Auslösendes Ereignis:	Die Entwickler/in ist an einer Kennzahl für einen vorliegenden Graph interessiert.
Beschreibung:	Über die Berechnungslogik wird für einen Graphen eine vorgegebene Metrik berechnet. Das Ergebnis wird dem/der Entwickler/in zurückgemeldet.
Erweiterung:	-

Use Case: Bibliothek in Software einbinden

Ziel:	Der Entwickler bzw. die Entwicklerin, die die Bibliothek in eine Software einbinden will, kann dies erfolgreich durchführen.
Vorbedingung:	-
Nachbedingung Erfolg:	Bibliothek ist in ein anderes Softwareprodukt eingebunden.
Nachbedingung Fehler:	-
Akteure:	Entwickler/in
Auslösendes Ereignis:	Der/Die Entwickler/in möchte die Bibliothek in ein anders Softwareprojekt einbinden.
Beschreibung:	Durch den Einsatz von Maven ist es möglich die Klassenbibliothek in anderen Mavenprojekten einzubinden und zu verwenden. Dafür muss das Artefakt als Abhängigkeit eingebunden werden.
Erweiterung:	-

3.2 Analyse zur Implementierung der Graphdatenstruktur

Neber der Definition der Anforderungen und der Formulierung der Use Cases ist eine kurze Analyse über die Implementierung der Graphdatenstruktur notwendig. Weil es eine Reihe von möglichen Graphimplementierungen gibt, muss recherchiert werden, welche Implementierung für die Klassenbibliothek gewählt werden soll. Hierbei sei zu erwähnen, dass laut den Anforderungen die Klassenbibliothek grundsätzlich unabhängig von der konkreten Graphimplementierung arbeiten soll. Der Graph wird durch eine einheitliche Schnittstelle angesprochen. Allerdings ist aus den Anforderungen auch zu entnehmen, dass die Klassenbibliothek mindestens eine Implementierung liefern muss. Ziel ist es also herauszufinden, welche Implementierungen besonders relevant sind und umgesetzt werden sollen.

Bei der Wahl der Implementierung muss zunächst zwischen Graphart und der repräsentierenden Datenstruktur unterschieden werden. Ersteres beschreibt die Art und Weise wie der Graph agiert und was in ihm möglich ist, wenn man ihn manipuliert. Hier kann eine Unterscheidung zwischen einem simplen und nicht simplen Graph vorgenommen werden. Auch wird festgelegt, ob der Graph gerichtet ist oder nicht. Außerdem können sich Graphen dahingehend unterscheiden, dass sie Markierungen zulassen oder nicht. Bei der in Kapitel 2 vorgestellten Metriken wurde hauptsächlich

auf simple, ungerichtete Grphen eingegangen. Aufgrunddessen soll auch die Standardimplementierung in der Klassenbibliothek einen solchen Graphen umsetzten. Dies bedeutet, dass beim Einfügen von Kanten darauf geachtet werden muss, dass eine Kante nicht bereits eingefügt worden ist und eine Kante immer in beide Richtungen eingefügt werden muss. Bei der Löschung einer Kante ist im Gegenzug darauf zu achten, das jeder Eintrag gelöscht wird, der die jeweilige Kante rerpäsentiert. Da allerdings einige Metriken und auch Graph-Anwendungsfälle auf gerichtete Graphen setzten, soll auch eine Implementierung für einen solchen Graphen erfolgen. Die Implementierung beim Einfügen und Entfernen von Kanten muss dementsprechend anders sein. Aus den Anforderungen geht weiterhin heraus, dass ein Graph meist einen bestimmten Sachverhalt wiederspiegelt und deshalb markierbar sein muss. Dies begründet auch, dass die Graphimplementierung in der Klassenbibliothek Markierungen zulässt.

Zur Wahl der Datenstruktur muss zunächst betrachtet werden, welche Möglichkeiten es gibt, einen Graphen zu repräsentieren, wenn man ihn programmatisch im Arbeitsspeicher darstellt. Grundsätzlich lassen sich vier Datenstrukturen ausmachen [Kne19]:

- Adjazenzmatrix
- Adjazenzlisten
- Inzidenzmatrix
- Inzidenzliste

Alle diese Datenstrukturen bringen ihre eigenen Vor- und Nachteile mit sich. Besonders bei den verschiedenen Einfüge-, Lösch- und Abfrageoperationen weisen die jeweiligen Implementierungen Unterschiede auf. Auch beim Speicherverhalten sind starke Unterschiede auszumachen.

Die Adjazenzmatrix ist eine sehr einfache Art und Weise einen Graph umzusetzten. Eine Markierung der Knoten und der Kanten ist mit ihr möglich. Besonders das Einfügen einer Kante ist in der Matrix sehr schnell und in konstanter Zeit möglich, da nur die Knotenindices addressiert werden müssen. Gleich verhält es sich für das Löschen einer Kante. Im Gegenzug ist allerdings das Einfügen und Löschen eines Knotens äußerst aufwendig. Da eine Adjazenzmatrix meist über Arrays realisiert wird, muss beim Einfügen und Löschen eines Knotens eine neue größere oder kleinere Martix initialisiert werden, in die dann alle vorigen Werte der alten Martix kopiert werden müssen. Ein weiterer schwerwiegender Nachteil der Matrix ist, das diese oft dünn besetzt ist, da Graphen selten vollständig sind. Dadurch wird äußerst viel Speicherplatz verschwendet. Dies ist besonders kritisch, da der Speicherbedarf der Matrix quadratisch mit der Anzahl der Knoten steigt.

Anders verhält sich das bei den Adjazenzlisten. Das einfügen eines Knotens ist hier sehr einfach möglich, da nur eine neue Liste angelegt werden muss, die an den neuen Knoten gebunden ist. Auch beim Einfügen einer Kante ist die Adjazenzliste leicht zu manipulieren, da nur in eine oder mehrere der Listen ein Eintrag hinzugefügt

werden muss. Ähnlich verhält es sich beim Löschen von Knoten und Kanten. Bei der Kantenlöschung müssen nur die entsprechenden Einträge aus den Adjazenzlisten gelöscht werden. Bei der Knotenlöschung kann die entsprechende Liste vollständig gelöscht werden und die übrigen Kanteneinträge aus den anderen Listen werden auch entfernt. Beim Speicherverhalten verbraucht die Adjazenzliste nur so viel, wie der Graph Informationen trägt. Der Verbraucht steigt also linear mit Knoten- und Kantenanzahl.

Inzidenzliste- und Matrix repräsentieren den Graphen durch das aneinanderreihen von Kanten und ihren Informationen (inzidente Knoten). Dadurch sind Kanteneinfüge- und löschoperationen leicht zu realisieren. Allerdings ist man bei der Matrix durch ihre festgelegte Größte stärker beschränkt und muss beim Löschen und Einfügen die Matrix neu initialisieren. Der Speicherbedarf steigt linear bei beiden Lösungen mit der Anzahl der Kanten. Bei der Inzidenzmatrix wird die Matrix zudem linear größer, desto mehr Knoten im Graphen sind.

Neben den manipulierenden Operationen auf die Datenstrukturen sind auch analysierende Zugriffe wichtig zu betrachten. Hier stellt sich heraus, dass die adjazenzbasierten Datenstrukturen sich besser zur Informationsbeschaffung eignen als die inzidenz-basierten. Besonders die Adjazenzliste ermöglicht es einfach die Nachbarn eines Knotens auszugeben. Soll die Nachbarschaft zweier Knoten nachgewiesen werden, ist die Matrix zwar am schnellsten, die Liste schafft es aber dennoch in linearer Zeit. Bei den inzidenz-basieren Strukturen sind vor allem lesende Operationen günstig, die die Inzidenz aufzeigen wollen. [Bet19]

Aufgrund der guten Speicherverbrauchseigenschaften und der relativen Einfachheit der Manipulation soll für die Klassenbibliothek der Graph mittels Adjazenzlisten implementiert werden. Zudem sind die Adjazenzlisten günstig, da sie sich leicht auswerten lassen. Besonders bei Nachbarschaftsbeziehungen ist dies der Fall. Bedenkt man, dass für die Berechnung der Metriken besonders die Auswertung des Graphen wichtig ist, wird die Wahl noch einmal untermauert. So werden beispielsweise bei der chromatischen Zahl oder beim Page Rank die adjazenten Knoten eines Knotens immer wieder benötigt.

3.3 Analyse zur Persistierung des Graphs

Um einen Graphen innerhalb einer Datei zu persistieren, gibt es eine weite Reihe an Möglichkeiten und Formaten. Es soll deshalb diskutiert werden, welche Persistierungsform für die zu umsetzende Klassenbibliothek gewählt werden soll.

Die Darstellung von Graphen in Textform wurde bereits von vielen Datenformaten unternommen. Dabei werden eigene Textformate verwendet oder auf bewährte Formate wie XML, CSV und JSON zurückgegriffen. Vertreter dieser bereits existenten Formate sind z.B. das *„Graph Exchange XML Format“*, *„GDF“* oder *„CSV“*. [Gep17]

Weiterhin ist das Feld der Graph-Textformate allerdings immer noch im Wandel und so gibt es weiterhin auch neue Ansätze. So wurde 2021 von Adreas Kollegger, einem Mitarbeiter von Neo4j Inc., ein weiteres Format namens „gram“ vorgeschlagen. Dieses Format kombiniert Elemente des JSON-Formats und der Abfragesprache Cypher, die für Anfragen an die Graphdatenbank Neo4j genutzt wird. So können Knoten-Objekte über die JSON-Syntax definiert werden und Kanten zwischen diesen Knoten über eine Cypher-artige Syntax erstellt werden (z.B. *(a: A)-[:Mark]->(b: B)*). [Kol21]

Für die Persistierung innerhalb der Klassenbibliothek soll sich nun auch an diesem „gram“-Format orientiert werden. Hierfür wird zunächst in der Datei angegeben, ob es sich um einen gerichteten oder um einen ungerichteten Graphen handelt. Anschließend werden wie in „gram“ die Graphobjekte (Knoten **und** Kanten) über JSON definiert und bekommen einen Identifier. Zum Schluss des Dokuments werden die einzelnen Beziehungen, wie bereits gezeigt, einzeln aufgeführt. Mithilfe der erweiterten Backus–Naur Form, können die Ableitungsregeln der definierten Grammatik der Format-Sprache folgendermaßen dargestellt werden. Hierbei werden bestimmte Ableitungen abgekürzt. So beschreibt das Nicht-Terminal „JSON“ die normale JSON-Sytax. Das Startsymbol ist „graph“:

```
graph = ('directed' | 'undirected'), '\n', {object}, {connection};
object = '(', IDENTIFIER, JSON, ')', '\n';
connection = '(', IDENTIFIER, ')', '-[', IDENTIFIER, ']->',
'(', IDENTIFIER, ')', '\n';
```

Das definierte Format bietet nicht nur den Vorteil, dass beliebige Graphen intuitiv gespeichert werden können, sondern kann auch einfach Implementiert werden, da die Erstellung von JSON-Strings durch Bibliotheken wie „Jackson“ übernommen werden kann.

3.4 Entwurf der Klassenbibliothek

Auf Basis der der Anforderungen und der angefertigten Analysen wird nun ein Entwurf erarbeitet, der die geforderten Anforderungen umsetzt. Dabei wird vor allem auf die Struktur und Dynamik der Klassenbibliothek eingegangen. Es soll hervorgehen, wie die Bibliothek konzipiert ist und auch wie sie effektiv genutzt werden kann. Anschließend wird als Teil des detailierten Entwurfs auch auf die konkrete Algorithmik der Graphmetriken eingegangen. [BL11]

Im ersten Teil der jeweiligen Entwurfsbeschreibung (Statik und Dynamik) geht es um die Umsetzung und Implementierung zu den Anforderungen der Graphdatenstruktur. Im Zuge dessen werden auch die Use Cases „Graph erzeugen“, „Graph manipulieren“ und „Graph analysieren“ umsetzt. Im zweiten Teil, wird jeweils darauf eingegangen, wie die Berechnung der Metirken umgesetzt wird.

3.4.1 Statischer Entwurf

Zunächst soll ein statischer Entwurf der Klassenbibliothek angegeben werden. Der statische Entwurf beschreibt die Struktur der Software und damit den Teil, der sich zur Laufzeit einer Programms nicht ändert. Durch den statischen Entwurf soll vor allem klar werden, wie die Klassenbibliothek verwendet werden kann, indem die relevanten Schnittstellenklassen identifiziert und beschrieben werden.

Im Rahmen dieses Entwurfes wird vor allem auf die Beschreibung durch Klassen zurückgegriffen. Das finale Ergebnis stellt auch stehts ein Klassendiagramm dar.

Graphdatenstruktur

Um die Graphdatenstrukur zur Verfügung zu stellen, sind zwei Dinge notwendig. Zunächst muss eine Schnittstelle für ein Graphobjekt definiert werden. Danach ist es nötig eine Implementierung für diese Schnittstelle zu definieren. Teil der Schnittstelle sind alle nötigen Methoden, die in den Anforderungen definiert wurden. Dazu zählen die manipulierenden und analysierenden Operationen. Dadurch, dass der Graph durch eine Schnittstelle definiert wird, ist auch direkt gewährleistet, dass ein anderer Programmierer oder eine andere Programmiererin eine eigene Graph-Datenstruktur entwickeln kann. Er oder sie muss dafür nur die definierte Schnittstelle implementieren.

Weiterhin muss es ermöglicht werden, dass die zu erstellenden Graphen markiert werden müssen. Hierbei muss eine Markierung am Knoten und an den Kanten möglich sein. Damit die Markierung vollkommen frei wählbar ist, wird für den Graphen parametrisierte Polymorphie angewandt. D.h. im speziellen, dass das definierte Interface für den Graphen generifiziert wird über zwei Typvariablen. Die erste Typvariable gibt dabei den Typen der Knotenmarkierung an, die zweite den Typen der Kantenmarkierung. Da es laut den Anforderungen verpflichtend ist, dass ein Knoten im Graphen über seine Markierung identifiziert wird, ist es zudem notwendig, den Typparameter für die Knotenmarkierung einzugrenzen. Speziell ist es notwendig, dass Knotenmarkierungen vergleichbar sind. Um dies zu gewährleisten wird festgelegt, dass alle Knotenmarkierungstypen das Interface *Comparable* implementieren. Dieses Interface ist dabei schon Teil der Java Standard-Bibliothek [Ora21]. Durch diese Einschränkgung wird es möglich, dass Knoten untereinander vergleichbar werden, da über die Methode *compareTo* ein Abgleich stattfinden kann.

Da Kanten nicht wie Knoten ausschließlich über ihre Markierung beschreiben werden können, ist es notwendig eine eiegne Kantenklasse zu definieren. Diese ist ein einfaches Java-Objekt (POJO) bestehend aus dem Ausgangsknoten, dem Eingangsknoten und der Markierung.

Auf Basis von Kapitel 3.2 wurde klar, welche Grapharten implementiert werden sollen. Umgesetzt werden zwei simple Adjazenzlistengraphen. Einer davon ist gerichtet, der andere nicht. Da sich beide Implementierungen bis auf die Handhabung mit Kanten

gleichen, ist es sinnvoll eine Abstraktionsebene in die Umsetzung einzubauen. So implementiert eine abstrakte Klasse *AdjacencyListGraph* das Graph-Interface und definiert alle Methoden bis auf das Einfügen, Löschen und Existenzabfragen von Kanten. Durch die abstrakte Klasse ist es möglich Codewiederholungen zu vermeiden, da Code für die jeweiligen Grapharten wiederverwendet werden kann. Zur eigentlichen umsetzung der Adjazenzlisten nutzt *AdjacencyListGraph* eine Java-Map, wobei als Instantzobjekt eine Hash-Map verwendet wird. Die Map hat den Typen *Map<N, List<Edge<N, E>>>*, wobei *N & E* die generischen Typparameter für den Knoten und die Kantenmarkierung sind. Wie zu erkennen ist, wird ein Knoten auf seine jeweilige Adjazenzliste abgebildet. Die Map ist einer geschachtelten Liste überlegen, da die Suche nach der jeweiligen Adjazenzliste in konstanter Zeit möglich ist, anstatt eine ganze Liste zu durchsuchen. Zudem ist die Zuordung des Knoten zu seiner Liste bereits in der Datenstruktur integriert.

Für manche Graphmetriken ist es relevant welche Graphart vorliegt. So ist z.B., wie in Kapitel 2 gezeigt, die Berechnung der Dichte abghängig davon, ob ein gerichteter oder ein ungerichteter Graph vorliegt. Um bei der Metrikenberechnung eine Unterscheidung zwischen den Arten treffen zu können, ist es nötig diese zu markieren. Hierfür wird sich sogenannter „Marker-Interfaces“ bedient. Diese beinhalten keine Methoden und können so beliebig von Klassen „implementiert“ werden. Im Falle der Klassenbibliothek gibt es die vier Marker-Interfaces *SimpleGraph*, *NonSimpleGraph*, *DirectedGraph* und *UndirectedGraph*. Die beiden Graphimplementierungen „implementieren“ dann ihre jeweiligen Schnittstellen. Innerhalb von Java kann dann mit einer *instanceof*-Abfrage erfragt werden, um welche Art von Graph es sich handelt.

Auch wenn die Bibliothek keine direkte Implementierung des Graphen über eine Adjazenzmatrix bereitstellt, ist die Repräsentation durch diese für eine Metrik relevant. Die Eigenvector Centrality benötigt zur Berechnung über die „Power Iteration“ die Adjazenzmatrix. Infolgedessen muss noch eine weitere Klasse hinzugefügt werden, die die Klassenrepräsentation übernimmt. Da keine Manipulation dieser Matrix vorgesehen ist, bietet sie nur einen Konstrktor zum Aufbau der Datenstruktur. Hierbei hält die Klasse *AdjacencyMatrix* eine Map, die einen Knoten auf seinen Matrix-Index abbildet und die Matrix selbst, die die Kantenmarkierungen enthält.

Um die Anforderung zur Persistierung des Graphen zu erfüllen, muss eine weitere Klasse zur Verfügung gestellt werden. Diese hält zwei statische Methoden, die einen Graphen persitieren bzw. laden können. Hierbei kann ein freier Dateipfad gewählt werden.

Außerhalb der Klassen, die zur Erfüllung der Anforderungen dienen, wurde zudem eine Klasse erstellt, die nur statische Methoden enthält, die dazu bestimmt sind, Standard-Graphen zu erstellen. Dies ist vor allem für das spätere Testen relevant, da so die Grapherstellung bei den Tests erleichert wird und einige Metriken bekannte Werte für diese Standardgraphen haben. Zur den verfügbaren Graphen gehören der vollständige Graph, der vollständig bipartite Graph, der Kreisgraph und der Liniengraph.

Schlussendlich ist es im Rahmen der Implementierung noch wichtig sich, um mögliche Fehler und Ausnahmen zu kümmern. Hierfür stellt die Klassenbibliothek im Rahmen der Graphdatenstruktur drei eigene Exceptions zur Verfügung. Die *GraphManipulationException* wird geworfen, wenn bei der Graphmanipulation entwas illegales passiert. Dies ist z.B. der Fall, wenn ein gleicher Knoten zum zweiten Mal eingefügt wird. Die *GraphCreationException* wird genutzt, wenn bei der Standardgrapherstellung etwas schief läuft. Die *GraphAnalyticException* wiederum wird geworfen, wenn bei der Graph-Analyse ein Fehler auftritt. Ein Beispiel hierfür ist die Abfrage nach den adjazenten Knoten eines Knotens, der nicht im Graph existiert.

Durch die oben angegebene Menge an Klassen und Methoden können nun alle funktionalen Anforderungen aus 3.1.3 umgesetzt werden. Es wird eine markierbare Graphdatenstruktur als Schnittstelle vorgegeben und zwei Implementierungen für diese bereitgestellt. Durch die Schnittstelle wird aber einem fremden Entwickler bzw. einer fremden Entwicklerin die Freiheit gelassen, eine eigene Implementierung zu nutzen. Weiterhin ist die Persistierung und das Laden von Graphen mittels Textdateien möglich.

Die beschriebenen Klassen und Methoden zur Graphdatenstruktur können übersichtlich mittels des Klassendiagramms in Abbildung 3.2 zusammengefasst werden.

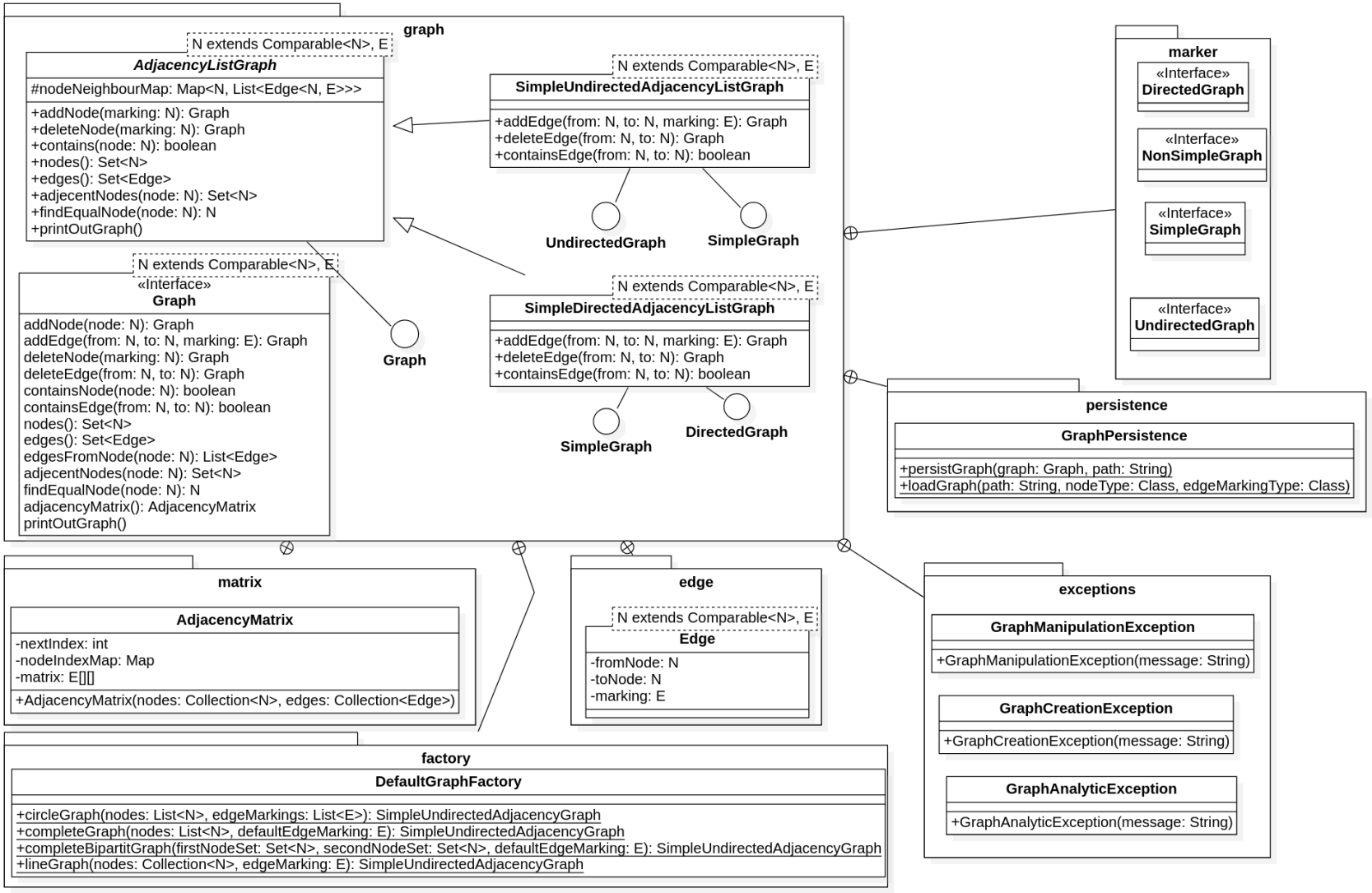

Abbildung 3.2: Klassendiagramm: Graphdatenstruktur

Metriken-Berechnung

Durch die Bereitstellung einer Graph-Datenstruktur ist es nun möglich, eine Struktur bereitzustellen, die es ermögllicht alle gewünschten Metriken zu berechnen. Um die Berechung zu nutzen soll ein Mechanismus zur Verfügung gestellt werden, der den Zugriff auf die Kalkulation homogenisiert. D.h. es wird eine Schnittstellen-Klasse erzeugt, die sämtliche Berechnungsanfragen annimmt und an die Business-Logik weitergibt.

Zur Umsetzung dieses Mechanismus wird auf ein vereinfachtes Schichten-Muster zurückgegriffen [BL11]. Die Klasse *MetricsCalculation* stellt als eine Art Schnittstelle („bondary“) statische Methoden zur Berechung aller Metriken zur Verfügung. Es wird dabei zwischen drei Verschiedenen Metrik-Arten unterschieden.

- Graph-umfassende Metriken
- Knoten-beschreibende Metriken
- Metriken zur Beschreibung der Beziehung zwischen zwei Knoten

Die jeweiligen Methoden nehmen den zu analysierenden Graphen und ggf. die zu untersuchenden Knoten entgegen. Zudem wird eine Instanz einer *GraphMetric*, *NodeMetric* oder *NodeToNodeMetric* übergeben, die die zu berechnende Metrik identifiziert. Die aufgeführten „Klassen“ sind Enumerationen, die jeweils die unterstützten Kennzahlen enthalten. Die Schnittstelltenfunktionen liefern dem Aufrufer schlussendlich eine Java-*Number*, die sowohl Ganzzahlen als auch Gleitkommazahlen darstellen kann.

Nachdem die „boundary“ die initiale Anfrage entgegennimmt, muss diese weiter verteilt werden. Genauer heißt das, dass die Anfrage an die jeweiligen Klassen weitergegeben wird, die die Berechung der eigentlichen Metrik übernehmen. Für diese Verteilung wird in der zweiten Schnicht („control“) die Klasse *GraphMetricCalculationDistribution* genutzt, die je nach der gewählten Metrik, die korrekte Berechnungsklasse aufruft.

Die jeweiligen Berechnungsklassen enthalten ausschließlich statische Methoden, die eine Metrik für den Graphen berechnen und dabei logischerweise eine Zahl zurückgeben (Integer, Long oder Double). Diese Klassen sind dabei einmal nach den drei bereits bekannten Kategorien in Pakete sortiert und enthalten dann noch mal nur Metriken jeweils passend zu einer der in Kapitel 2 formulierten Kategorien. Da Metriken teilweise aufeinander aufbauen interagieren diese Klassen auch untereinander. Die jeweiligen Klassen heißen und berechnen:

- **BasicGraphMetricCalculation**: Berechnung grundlegender Graph-Metriken (Ordnung, Größe etc.)
- **DistanceGraphMetricCalculation**: Berechnung distanzbasierter Metriken (Radius und Durchmesser)
- **ChromaticNumberMetricCalculation**: Berechnung der chromatischen Zahl (exakt und mit Greedy-Algorithmus)

- **ChromaticIndexMetricCalculation**: Berechnung des chromatischen Indexes (exakt und mit Greedy-Algorithmus)
- **DensityMetricCalculation**: Berechnung der Graph-Dichte
- **BasicNodeMetricCalculation**: Berechnung grundlegender Metriken für einen Knoten (z.B. Grad)
- **DistanceNodeMetricCalculation**: Berechnung der Extrenzität eines Knoten
- **CentralityMetricCalculation**: Berechnung der aufgeführten Zentralitätsmetriken
- **NodeToNodeDistanceMetricCalculation**: Berechnung der Distanz zwischen zwei Knoten innerhalb eines Graphen

Weiterhin benötigen einige Metriken bestimmte Hilfsobjekte und -algorithmen, damit sie berechnet werden können. Hierzu zählen ein generisches Tupel-Objekt und Algorithmen zur Berechnung des Binomialkoeffizienten und der Breiten- und Tiefensuche. Zudem werden weitere Hilfsmethoden zur Verfügung gestellt zum Füllen und klonen von Java-Maps und zum invertierten von Graphen. Beim invertieren werden werden alle Kanten zu Knoten und die neuen Knoten sind adjazent zueinander, wenn die ursprünglichen Kanten inzident zueinander waren. Dies ist wichtig zur Berechnung des chromatischen Index.

Aus statischer Sicht sind keine weiteren Klassen notwendig, um die Berechung der Graphmetriken zu gewährleisten. Einzig gibt es noch zwei weitere Exceptions, die hinzugefügt werden. Die *MetricChoiceException* wird geworfen, falls die Berechnung einer Metrik stattfindet, die nicht implementiert ist. Die *MetricCalculationException* wird dann verwendet, falls bei der Berechnung einer Metrik ein Fehler auftritt. Beispielsweise ist das der Fall, falls ein für einen Graph die Größte berechnet werden soll, der aber nicht als gerichtet oder ungerichtet markiert wurde.

Wie an der Beschreibung gesehen werden konnte, sind die Methoden für die Metrikenberechnung alle ausschließlich statisch. Das entspricht zwar nicht dem Paradigma der Objektorientierung, ist allerdings vorteilhaft, da so nicht unnötigerweise attributlose Berechnungsobjekte erstellt werden müssen. Außerdem drückt dies auch den funktionalen Charakter der Bibliothek aus. Die jeweiligen Methoden zur Berechnungen der Kennzahlen lösen keine Seiteneffekte aus und liefern so bei gleichem Input den gleichen Output. Dies ähnelt dem funkionalen Programmierparadigma, was die statischen Methoden imitieren. [CB15]

Die beschriebenen Klassen können nun auch wieder in einem Klassendiagramm aufgezeigt werden, dass in Abbildung 3.3 zu sehen ist.

3.4.2 Dynamische Aspekte

Dadurch, dass die Statik der Klassenbibliothek nun feststeht und durch diese die formulierten funktionalen Anforderungen gedeckt sind, soll noch auf den dynamischen

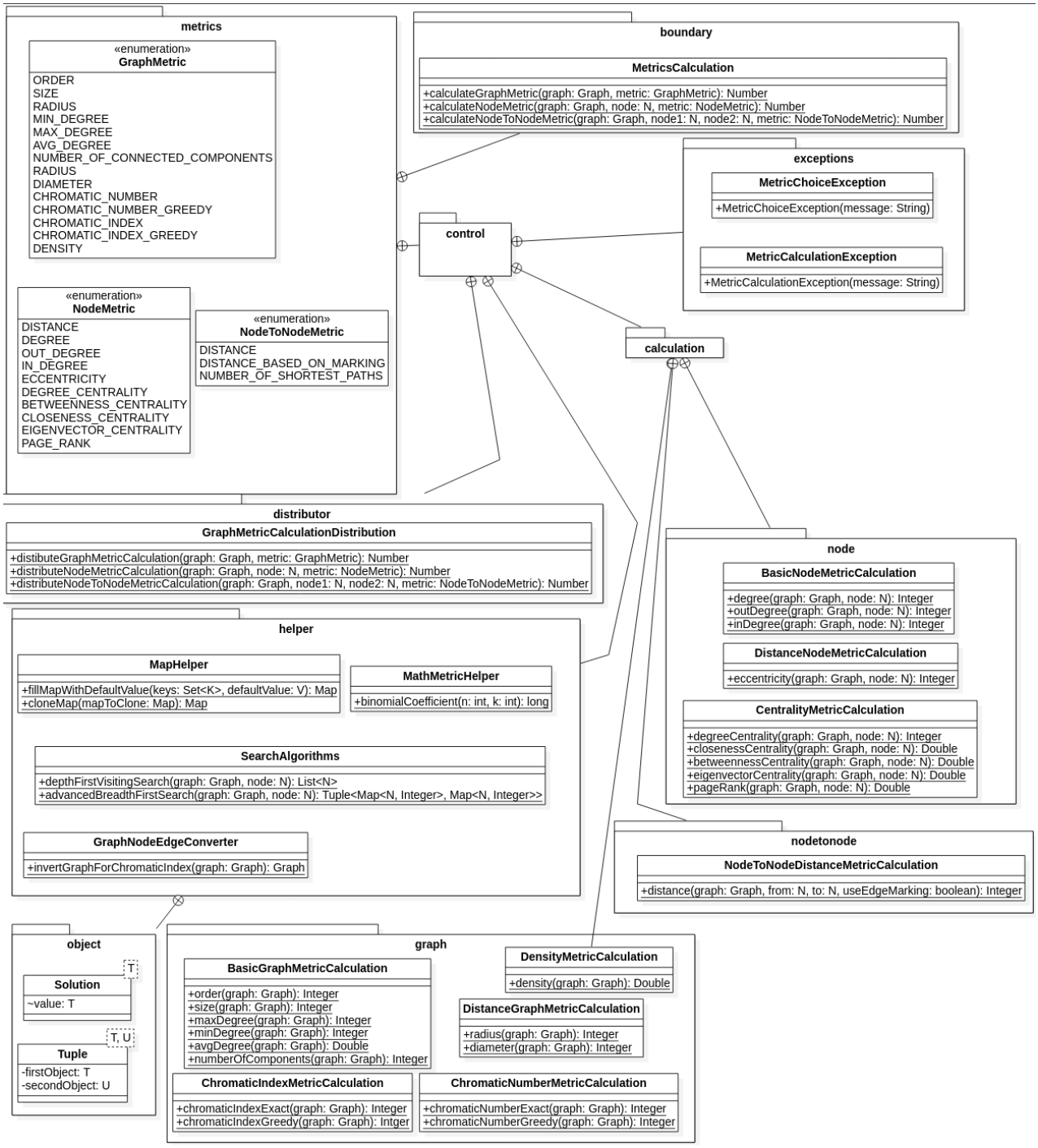

Abbildung 3.3: Klassendiagramm: Metriken-Berechnung

Entwurfsaspekt der Bibliothek eingegangen werden. Konkret soll erläutert werden, wie eine Metriken-Berechnung von Statten geht.

Zur Beschreibung soll als Beispiel die Berechnung der Closeness Centrality herhalten. Die Berechnung der anderen Metriken erfolgt analog, bzw. ähnlich. Bevor aber eine Metrik berechnet werden kann, ist es nötig einen Graphen zu erstellen und zu manipulierten bzw. zu füllen. Im Rahmen des Beispiels soll ein simpler, ungerichteter Adjazenzlistengraph erstellt werden, der als Knoten und Kantenmarkierung Ganzzahlen entgegennimmt. Hierfür lässt sich einer der bereits implementierten Graphen der Bibliothek verwenden. Der Graph selbst wird in einer Variable gespeichert, die den Typ *Graph* hat. Damit ist der Graph nur über die definierte Schnittstelle zu manipulieren und kann für die Metrikenberechnung genutzt werden. Anschließend an die Erstellung können Knoten und Kanten eingefügt werden. Im Beispiel sollen die zwei Knoten 1 und 2 hinzugefügt werden. Zudem wird eine Kante zwischen beiden erstellt, die mit 1 markiert ist. Wie das im Rahmen eines Sequenzdiagramms aussieht, zeigt Abbildung 3.4. Nachdem der Graph erstellt wurde, können auf ihm alle Metriken berechnet

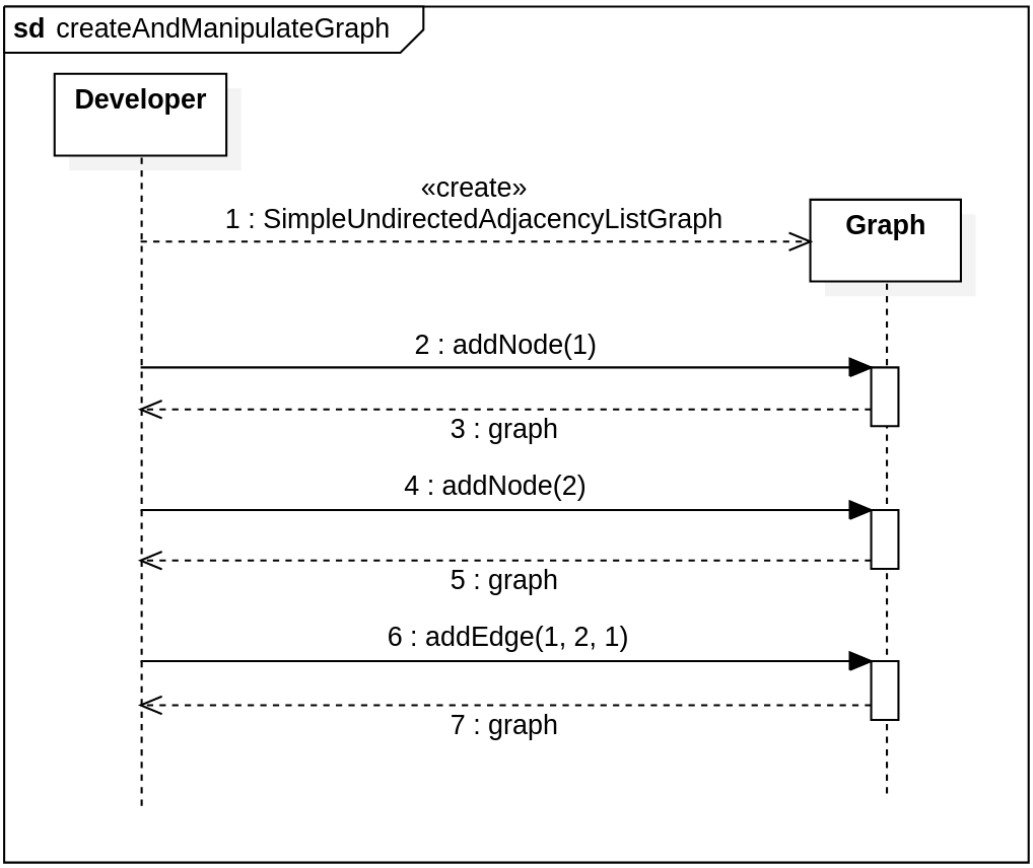

Abbildung 3.4: Sequenzdiagramm: Grapherstellung und- manipulation

werden. In diesem Fall übergibt der Entwickler den Graph mit den sonstigen Argumenten der *MetricCalculation*, die wiederum die Anfrage an die weiter unten liegende Schicht weitergibt. Hier verteilt die *GraphMetricCalculationDistribution* die Anfrage an die korrekte Berechnungsklasse. Im Fall der Closeness Centrality ist das die Klasse

CentralityMetricCalculation. Diese kann die Metrik allerdings nicht alleine berechnen und benötigt die Hilfe eines Suchalgorithmus (Breitensuche). Infolgedessen ruft die Klasse die *SearchAlgorithms*-Klasse auf und kann mit deren Ergebnis die Metrik berechnen und schlussendlich den höheren Schichten zurückgeben. Abbildung 3.5 zeigt die dynamische Folge der Metriken-Berechnung für die Closeness Centrality in einem Sequenzdiagramm.

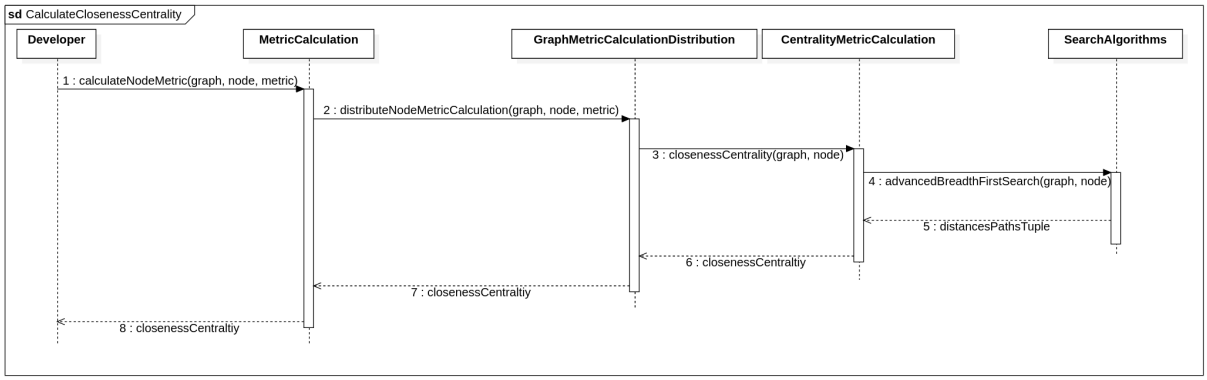

Abbildung 3.5: Sequenzdiagramm: Metriken-Berechnung (Closeness Centrality)

3.5 Algorithmik und Implementierung der Graph-Metriken

Nachdem die Beschaffenheit und grundsätzliche Funktionsweise der Klassenbibliothek erläutert wurde, soll nun noch darauf eingegangen werden, wie die jeweiligen Metriken berechnet werden. Vor allem ist es interessant die Algorithmik einzelner Metirken zu beleuchten. Es werden dabei alle implementierten Metriken betrachtet. Zuerst werden die Graph-übergreifenden, dann die Knoten-beschreibenden und abschließend die Distanz zwischen zwei Knoten beschrieben.

Graph-umfassende Metriken

Zu den Graph-umfassenden Metriken gehören zunächst die vorgestellten grundlegenden Metriken (siehe. 2.1), die für einen ganzen Graph gehören. Diese sind meist auch recht einfach zu berechnen. Für die Ordnung eines Graphen wird abgefragt, wie groß die Menge der Knoten ist. Die Knotenmenge kann dabei mit dem Aufruf *graph.nodes()* erfragt werden, der im Graph-Interface definiert ist. Ähnlich verhält es sich auch bei der Größe des Graphen, wobei hier die Kantenmenge abgerufen wird (*graph.edges()*). Eine Besonderheit besteht hierbei noch in der Unterscheidung zwischen gerichteten und ungerichteten Graphen. Ist der Graph nämlich ungerichtet so wird seine Größe durch Zwei geteilt. Dies wird durch den *instanceof*-Aufruf von Java und die Marker-Interfaces erreicht.

Zur Berechnung von Minimal-, Maximal- und Durchschnittsgrad bedient sich die Bibliothek der Stream-API von Java. Diese API ermöglicht es u.a. mithilfe von Lambda-Ausdrücken Datenmengen in Java zu filtern, abzubilden, zu reduzieren und zu sammeln. [Ull14] Zur Berechnung werden die Knoten des Graphen in einem Stream verarbeitet und auf ihren jeweiligen Grad abgebildet. Hierfür wird die Grad-Metrik der Bibliothek genutzt. Auf den anschließenden Ganzzahl-Stream kann eine *min()-, max()-* oder *average()*-Anfrage aufgeführt werden. Da die API hierbei einen Java *Optional* zurückgibt, muss dieser noch aufgelöst werden, indem bei Nichtvorhandensein eines Ergebnisses eine *MetricCalculationException* geworfen wird. Dies ist beispielsweise der Fall, wenn der Graph leer ist. Der gesamte Aufruf zur Berechnung der Metrik sieht dann folgendermaßen aus (Beispiel für Maximalgrad):

```
1 return graph.nodes().stream().mapToInt(node ->
2       BasicNodeMetricCalculation.degree(graph, node)).max()
3       .orElseThrow(() -> new MetricCalculationException(MESSAGE));
```

Für die Berechnung der Anzahl der Zusammenhangskomponenten wird sich dem Algorithmus bedient, der bereits in 2.1 beschreiben wurde. D.h. wird eine Tiefensuche für alle Knoten initiiert, falls diese nicht bereits von einer anderen Tiefensuche besucht wurden. Hierfür gibt der Tiefensuche-Algorithmus immer die Menge an Besuchten Knoten zurück. Die Tiefensuche selbst ist Teil der Helfermethoden und bedient sich eines Stapels zur Koordinierung der Suche. Zu Beginn der Algrotihmus wird eine List für die besuchten Knoten und ein Stapel initialisiert. Der Start-Knoten wird in beide Kollektionen eingefügt. Anschließend wird solange ein Knoten vom Stapel genommen und seine nicht besuchten Nachbarn in die Kollektionen gelegt, bis der Stapel final leer ist. Danach wird Liste der besuchten Knoten zurückgegeben.

Nach grundlegenden, graph-übergreifenden Metriken soll erklärt werden, wie das einzige Dichtemaß des Bibliothek berechnet wird. Die Dichte eines Graphen wurde in 2.4 beschrieben und kann genauso berechet werden, wie die Formel im Kapitel es vorschreibt. So wird einfach die Größe des Graphen durch den Binomialkoeffizienten $|V|$ über 2 geteilt und das Ergebnis zurückgegeben. Hervorzuheben ist, dass beim Berechnen sowohl geprüft wird, ob ein simpler Graph vorliegt und ob der Graph gerichtet ist oder nicht. Der Graph muss simpel sein, da die Berechnung nur möglich ist, wenn keine Schlingen und Doppelkanten im Graph möglich sind. Weiterhin ist es relevant ob der Graph gerichtet ist oder nicht, da bei einem nicht gerichteten Graph der Binomialkoeffizienten verdoppelt werden muss, da der Graph ja potenziell doppelt so viele Kanten tragen kann. Zur Berechnung des Binomialkoeffizienten wird sich eines effizienten iterativen Algorithmus bedient, der die Produktformel für den Binomialkoeffizienten implementiert, die folgendermaßen lautet:

$$\binom{n}{k} = \prod_{i=1}^{k} \frac{n+1-i}{i}$$

Die Implementierung dieser Produktformel, kann innerhalb von Java folgendermaßen aussehen. Hierbei wird zuerst abgefangen, falls k Null ist. Ist dies der Fall, ist das Ergebnis immer 0. Ist wiederum $2 * k$ größer als n, so wird k zu $n - k$. Dies ist möglich, da das Pascal'sche Dreieck, in dem die Werte des Binomialkoeffizienten stehen, symmetrisch ist und k die Spalten in diesem Dreieck rerpäsentiert. Zudem verkürzt das verkleinerte k die Zählschleife und somit dem Algrotihmus.

```
1  if (k == 0) return 1L;
2  if (2 * k > n) k = n - k;
3  long solution = 1L;
4  for (int i = 1; i <= k; i++) {
5    solution = solution * (n + 1 - i) / i;
6  }
7 return solution;
```

Für die distanzbasierten, graph-übergreifenden Metriken wird sich wieder der Java Stream-API bedient. Sowohl für den Radius als auch für den Durchmesser wird die Extrenzität aller Knoten benötigt. Ähnlich wie beim Maximal- oder Minimalgrad werden alle Knotens des Graphen auf eine Granzzahl abgebildet. In diesem Fall auf die Extrenzität. Dann kann wieder über die *max()* (Durchmesser) und *min()* (Radius) die jeweilige Metrik extrahiert werden. Auch hier gilt, dass bei einem leeren Graphen über die *Optional*-Logik eine *MetricCalculationException* geworfen wird.

Die letzten graph-übergreifenden Metriken, die umgesetzt wurden, sind die chromatische Zahl und der chromatische Index. Zur Berechnung der chromatischen Zahl wurden zwei Algorithmen implementiert. Eine exakte Berechnung des NP-vollständigen Problems und die bereits vorgestellte Greedy-Variante. Die Greedy-Variante konnte so umgesetzt werden, wie sie in 2.6 beschreiben wurde. Um für den Algorithmus die jeweilige niedrigst mögliche Farbe zu wählen, werden jeweils die zum Knoten adjazenten Knotenfarben in eine Liste geladen und anschließend, beginnend mit der niedrigsten Farbe, durch die Farben iteriert und geprüft, ob die Farbe in der Liste ist. Ist sie das nicht, kann sie als Farbe für den Knoten verwendet werden. Die Ermittlung der exakten chromatischen Zahl erwies sich als schwieriger. Um über alle möglichen Färbungen zu iterieren, wie das bei der exakten Berechnung nötig ist, müssen jeweils n Zählschleifen ineinander geschachtelt werden, die jeweils bis zur maximal höchsten Färbung zählen und dies ihrem korrespondierenden Knoten zuweisen. Sowohl n als auch die höchst mögliche Farbe entspricht der Ordnung des Graphen. Um diese beliebige Schachtelung der Zählschleifen zu erreichen, muss ein rekursiver Algorithmus entworfen werden. Um den rekursiven Abstieg zu koordinieren, wird eine Level-Variable eingefügt, die anfangs so groß ist, wie die Ordnung des Graphen. Bei jedem Abstieg wird diese um eins reduziert und sie markiert, welcher Knoten bei welchem Aufruf zu Färben ist. Ist das Level gleich 0, tritt der Basis-Fall der Rekursion ein und es wird geprüft, ob die jeweilige aktuelle Färbung legal ist. Fall diese das ist, wird geprüft, wie viele verschiedene

Farben genutzt wurden und in ein einheitliches Lösungsobjekt die aktuell kleiste legale Färbungszahl geschrieben. Das Lösungsobjekt, das dem Algorithmus übergeben wird, kann anschließend ausgelesen werden. Um zu prüfen, ob eine Färbung legal, wird über alle Knoten iteriert und geprüft, ob einer der Nachbarn jeweils die gleiche Farbe hat wie der aktuelle Knoten. Ist dies nur einmal der Fall, so ist die Färung nicht legal. Im Pseudocode ausformuliert, sieht der Algorithmus folgendermaßen aus:

```
1  chromaticNumberExact(graph, nodeColorings: List<[Node, Integer]>,
    level, maxColor, solution) {
2    if (level == 0) { // Basisfall
3      if (isLegalColoring(graph, nodeColorings)) {
4        solution = Math.min(solution, colorNumber(nodeColorings));
5      }
6    } else { // Rekursionsfall
7      for (int i = 1; i <= maxColor; i++) {
8        nodeColorings.get(level).setColor(i);
9        chromaticNumberExact(graph, nodeColorings, level - 1, maxColor,
10       solution);
11     }
12   }
13 }
```

Bereits beim statischen Entwurf der Bibliothek wurde gezeigt, wie die Berechnung des chromatischen Index erfolgt. Hierfür wird der Code der chromatischen Zahl einfach wiederverwandt. Auch beim chromatischen Index wird eine exakte und eine Greedy-Variante bereitgestellt. Grundsätzlich wird der Graph zur Berechnung also invertiert und anschließend die schon beschriebenen Algorithmen angewandt. Bei der Invertierung wird jede Kante in einen Knoten umgewandelt und anschließend dort Kanten eingefügt, falls beim ursprünglichen Graph zwei Kanten inzident zueinander waren.

Knoten-beschreibende Metriken

Auch bei den Knoten-beschreibenden Metriken gibt es bestimmte grundlegende Metriken. Der Ausgangsgrad eines Knotens wird dabei berechnet, indem die Größe der Menge seiner adjazenten Knoten (*graph.adjacentNodes(node)*) zurückgegeben wird. Zur Berechnung des Eingangsgrads wird die Liste aller Kanten (*edges()*) gefiltert und dabei geschaut, ob der Knoten, zu der die Kante zeigt, gleich dem zu untersuchenden Knotens ist. Für den Eingangsgrad kann dann diese gefilterte Liste gezählt werden. Bei der Berechnung des allgemeinen Knotengrades muss wieder unterschieden werden, ob ein gerichteter Graph vorhanden ist. Ist er ungerichtet so kann als Graph einfach der Ausgangsgrad genutzt werden. Ist er gerichtet, müssen sowohl ein- als auch ausgehende Kanten einzeln gezählt werden. Dies wird erreicht durch die Addition des Ausgangs- und Eingangsgrads.

Um die Extrenzität, die auch für den Durchmesser und den Radius gebraucht wird,

eines Knoten zu berechnen, werden wieder alle Knoten des Graphen auf eine Granzzahl abgebildet. Die entstehenden Ganzzahlen sind die Distanzen zwischen dem zu Untersuchenden Knoten und allen anderen Knoten. Aus der so entstanden Distanzliste kann dann die höchste Zahl entnommen werden, die wiederum die Extrenzität des Knotens darstellt.

Die Metrikenklasse mit den meisten Implementierungen bei den Knoten-beschreibenden Metriken sind die Zentralitätsmetriken. Die erste Zentralitätsmetrik ist die Degree Centrality. Diese ist äußerst einfach zu berechnen, da sie deckungsgleich zum Grad des Knotens ist. D.h. die Aufgabe wird einfach an die Grad-Berechnung deligiert, damit kein doppelter Code geschrieben werden muss.

Die erste komplexere Centrality-Berechnung ist die der Closeness Centrality. Für diese Metrik muss man wissen, wie weiter der zu untersuchende Knoten von allen anderen Knoten entfernt ist. Um diese Distanzen herauszufinden, wird sich einer Breitensuche bedient, die eine Map zurückliefert, die die anderen Knoten auf ihre Distanz vom Ausgangsknoten abbildet. Die Breitensuche wird genauer in einem weiteren Abschnitt beschrieben. Sind zunächt die Distanzen vorhanden, können diese einfach aufsummiert werden, um anschließend, die Formel für die Closeness Centrality anzuwenden. Ist die aufsummierte Distanz, wird als Ergebnis die höchstmögliche Ganzzahl zurückgegeben, um ∞ zu repräsentieren.

Um die Betweenness Centrality zu berechnen, ist es nötig die Funktionen $\sigma_{st}(v)$ und σ_{st} zu implementieren. Da die Funktion $\sigma_{st}(v)$ auf σ_{st} basiert, muss voralledem die zweitere algorithmisch umgesetzt werden. Auch hierbei hilft die Breitensuche. Da σ_{st} die Anzahl der kürzesten Wege zwischen zwei Knoten darstellt und damit eine Metrik für sich ist, wird diese später noch genauer beschrieben. Ist σ_{st} umgesetzt, kann $\sigma_{st}(v)$ einfach so im Programm umgesetzt werden, wie in Kapitel 2.5 beschrieben. Hierbei wird für die Berechnung der Distanzen zwischen zwei Knoten auf die korrespondierende Implementierte Metrik zugegriffen, die noch genauer erläutert wird. Mit den beiden implementierten Funktionen kann die Betweenness Centrality einfach über zwei Zählschleifen umgesetzt werden. Die jeweiligen Schleifen iterieren über alle Knoten des Graphen. Falls die beiden betrachteten aktuellen Knoten s und t nicht dem zu untersuchenden Knoten entsprechen und s und t nicht gleich sind, kann zur Betweenness Centrality des Knotens ein Summand addiert werden. Hierbei ist der Summand einfach die Anzahl $\sigma_{st}(v)$ durch σ_{st} geteilt, wie es die Formel auch vorgibt. Ist die Iteration vorbei kann das Ergebnis der Betweenness Centrality zurückgegeben werden.

Eine weitere Zentralitätsmetrik ist die Eigenvector Centrality. Diese wird in der Klassenbibliothek mittels des vorgestellten Power Iteration-Algorithmus berechnet. Da dieser Algoritmus mittels Matrixmultiplikation verläuft benötigt es zwei Dinge. Zuerst muss für den Graph eine Repräsentation mittels einer Adjazenzmatrix bereitgestellt werden. Zweitens muss eine Funktion zur Matrixmultiplikation vorhanden sein. Für ersteres dient die Methode *adjacencyMatrix()* des *Graph*-Interfaces. Die im *AdjacencyMatrix*-Objekt

enthaltene Matrix wird anschließend so umgewandelt, dass statt der Kantenmarkierung eine 1 oder 0 in den Matrixpositionen steht. Eine 1 steht im Eintrag, falls dort vorher auch eine Kante eingetragen war. Ansonsten wird eine 0 eingetragen. Für die Matrixmultiplikation wird mittels Maven die „Apache Commons“-Bibliothek eingebunden, da diese eine effiziente Implementierung für die Matrixmultiplikation bietet. Neben der Adjazenzmatrix muss für die Poweriteration auch der Startvektor erstellt werden, der so viele Zeilen wie die Adjazenzmatrix hat und eine Spalte hat, wobei jeder Eintrag 1 ist. Die Power Iteration selbst ist ein rekursiver Algorithmus. In diesem wird zunächst ein neuer Ergebnisvektor berechnet, indem die Adjazenzmatrix mit dem aktuellen Eigenvector Centrality-Vector multipliziert wird. Zu Anfang ist das der erstellte Startvektor. Der so entstehende neue Vektor wird normalisiert und der Normalisierungswert wird zusätzlich gespeichert. Der Basisfall der Rekursion tritt nun auf, wenn die Berechnung konvergiert. Dies ist der Fall, wenn der aktuelle Normalisierungswert fast gleich dem alten Normalisierungswert ist (maximale Differenz 0,001). Dann kann der Ergebnisvektor mit den Eigenvector Centralities zurückgegeben werden und der Wert für den zu untersuchenden Knoten herausgenommen werden. Konvergiert die Berechung nicht, tritt der Rekursionsfall ein. Hierbei wird einfach die Power Iteration mit dem neuen Eigenvector Centrality-Vektor erneut aufgerufen. Zudem wird der vorher berechnete Normalisierungswert mitgegeben, um zu prüfen, ob die Berechnung konvergiert. Als Pseudocode kann der Algorithmus folgendermaßen zusammengefasst werden:

```
1  double[][] powerIteration(adjacencyMatrix, vector, normalizedValue) {
2    newVector = adjacencyMatrix * vector;
3    newNormalizedValue = calcNormalizedValue(newVector);
4    normalizeVector(newVector, newNormalizedValue);
5    if (powerIterationConverged(normalizedValue, newNormalizedValue)) {
6      return newVector;
7    } else {
8      return powerIteration(adjacencyMatrix, newVector,
9        newNormalizedValue);
10   }
11 }
```

Die letzte Zentralitätsmetrik, die umgesetzt wurde, ist der Page Rank. Auch hier wird ein Rekursiver Algoritmus zur Berechnung genutzt. Zu Beginn bekommt jeder Knoten innerhalb einer Java-Map den Page Rank $1/|V|$ zugewiesen. Anschließend kann der eigentliche Algorithmus gestartet werden. Da dieser auch terminiert, wenn die Werte in der Map gegen einen bestimmten Wert konvergieren, wird im ersten Schritt die Map kopiert. Schlussendlich kann die in 2.5 vorgestellte Formel für den Page Rank angewandt werden. Hierfür wird über alle Knoten des Graphen iteriert und die Formel angewandt und anschließend der neue Wert in der Map für den Knoten gespeichert. Sind alle Werte sehr ähnlich zur Map aus vorigen Funktionsaufruf (Differenz ¡ 0.001), so tritt der Basisfall ein und der Algorithmus terminiert und aus der Map kann der Page

Rank aller Knoten abgelesen werden. Ansonsten tritt der Rekursionsfall ein und der Page Rank-Algrotihmus wird erneut mit der neuen Map aufgerufen. Im Pseudocode sieht das folgendermaßen aus. Der der Dämpfungsfaktor, der für den Algorithmus gewählt werden muss, steht als Konstante in der Klasse zur Verfügung und beträgt 0,85.

```
1  Map<Node, Double> pageRank(graph, previousMap, pageRankMap) {
2    newPageRankMap = clone(pageRankMap);
3    for (node : graph.nodes()) {
4      newPageRank = 1 - MODERATION_FACTOR;
5      pageRankAddend = 0;
6      for (adjacentNode: graph.adjacentNodes(node)) {
7        pageRankAddend += previousMap.get(adjacentNode) /
8          outDegree(graph, adjacentNode);
9      }
10     pageRankAddend *= MODERATION_FACTOR;
11     newPageRankMap.put(node, newPageRank + pageRankAddend);
12   }
13   if (pageRankConverges(previousMap, newPageRankMap)) {
14     return newPageRankMap;
15   } else {
16     return pageRank(graph, pageRankMap, newPageRankMap);
17   }
18 }
```

Metriken zwischen zwei Knoten

Die erste Metrik, die die Klassenbibliothek zwischen zwei Knoten definiert, ist die kürzeste Distanz zwischen diesen. Zur Berechung dieser haben sich einige Algorithmen verausgebildet. So können z.B. eine Breitensuche, der Djikstra-Algorithmus oder der A*-Algortihmus das Problem korrekt lösen. Bei der Breitensuche muss allerdings die Distanz zwischen zwei Knoten immer 1 sein und der A*-Algorithmus benötigt eine Heuristik, um effizient zu arbeiten. Im Rahmen dieser Bibliothek wurde zur Berechnung der Distanz die „Uniform Cost Search“ gewählt, wie sie im Buch „Artificial Intelligence - A modern approach“ von Stuart Russell und Peter Norvig beschrieben ist. Zu Beginn des Algrithmus wird zunächst eine Vorrangwarteschlange initialisiert, die immer den Knoten an vorderste Stelle stellt, der die niedrigste Distanz zum Ausgangsknoten hat. Zudem wird eine Liste der besuchten Knoten angelegt. Anfangs wird in diese Warteschlange der Ausgangsknoten mit Distanz 0 gelegt. Anschließend wird das erste Element aus der Warteschlange genommen und geprüft, ob dies der Zielknoten ist. Wenn ja, kann seine hinterlegte Distanz zurückgegeben werden. Falls nicht, wird der Knoten in die Liste der besuchten Knoten hinzugefügt und in die Vorrangwarteschlange werden alle seine noch nicht besuchten Nachbarknoten gelegt. Diese bekommen die Distanzmarkierung *Markierung aktueller Knoten + Distanz zu jeweiligen Nachbar*. Dieser Vorgang wird solange wiederholt, bis die Warteschlange leer ist. Falls dies eintrifft und somit kein Weg

zwischen den Knoten gefunden wurde, gibt der Algorithmus den maximalen Ganzahlwert des Typs *Integer* zurück, um die Distanz ∞ zu repräsentieren. Der implementierte Algorithmus kann sowohl mit Kantenmarkierungen für die Distanz arbeiten oder für jede Kante die Distanz 1 annehmen. Falls die Kantenmarkierung berücksichtig werden soll, muss dies bei Aufruf angegeben werden und die Kantenmarkierung muss zu einer Zahl castbar sein. [RN16]

Die letzte Metrik deren Implementierung beschreiben wird, ist eine spezielle Metrik, die deswegen implementiert wurde, da sie für die Betweenness Centrality gebraucht wird. Damit die Anzahl der kürzesten Wege zwischen zwei Knoten berechnet werden kann, wird sich wie beschrieben einer speziellen Breitensuche bedient. Die gleiche Breitensuche wird auch für die Distanzen bei der Closeness Centrality genutzt. Für beide Nutzungsszenarien liefert die implementierte Breitensuche ein Ergebnis vom Typ *Tuple<Map<N, Integer>, Map<N, Integer>>*. In der ersten Map werden alle erreichbaren Knoten gelistet und auf ihre Distanz vom Ausgangsknoten abgebildet. In der zweiten Map sind die gleichen Knoten verzeichet und werden auf die Anzahl der kürzesten Wege vom Ausgangsknoten aus abgebildet. Somit kann das Ergebnis für die Berechnung der Anzahl der kürzesten Wege zwischen zwei Knoten einfach aus der zweiten Map gelesen werden. Die verwendete spezialisierte Breitensuche wurde von Said Sryheni in seinem Artikel „Number of Shortest Paths in a Graph“ beschrieben [Sry20]. Die Suche verfolgt dabei grundsätzlich das gleiche Prinzip wie die beschriebene Tiefensuche, nur dass anstatt eines Stapels eine Warteschlange als Hilfsstruktur gewählt wird. Aus der Warteschlange wird zunächst der erste Knoten entnommen und dann über all seine Nachbarn iteriert. Ist ein Nachbar noch nicht besucht, wird er in die Warteschlange eingereit und als besucht markiert. Zusätzlich wird nun geprüft, ob die akutelle Distanz zum jeweiligen adjazenten Knoten kleiner ist, als die eingetragene. Ist dies der Fall wird die nun kleinere gefundene Distanz eingetragen. Zudem kann die Anzahl der gefunden kürzesten Wege zum Knoten auf die Anzahl der kürzesten Wege gesetzt werden, die der aktuell betrachtete Knoten hat. Ist dies nicht der Fall wird geprüft, ob die neu gefundene Distanz gleich zu der bereits eingetragenen ist. Fall dies zutrifft, kann gesagt werden, dass ein neuer kürzester Weg zum jeweiligen Knoten gefunden wurde. In diesem Fall wird die Anzahl der kürzesten Wege für diesen Knoten um so viel erhöht, wie es kürzeste Wege zum gerade betrachteten Knoten gibt. Der beschriebene Vorgang wird solange wiederholt, bis die Warteschlange leer ist. Anschließend werden beide Maps zurückgeben. Um den Algorithmus verständlicher zu machen, ist folgender Pseudocode gegeben.

```
1  Tuple<Map<N, Integer>, Map<N, Integer>> bfs(graph, node) {
2    distanceMap = mapWithDefaultValues(graph.nodes(), Int.MAX_VALUE);
3    pathsMap = mapWithDefaultValues(graph.nodes(), 0);
4    distanceMap.replace(node, 0);
5    pathsMap.replace(node, 1);
```

```
6       queue = new List();
7       queue.add(node);
8       vistedNodes = new List();
9
10      while (!queue.isEmpty()) {
11        actNode = queue.poll();
12        for (child : graph.adjacentNodes(actNode)) {
13          if (!visitedNodes.contains(child)) {
14            queue.add(child);
15            visitedNodes.add(child);
16          }
17          if (distanceMap.get(child) > distanceMap.get(actNode) + 1) {
18            distanceMap.replace(child, distanceMap.get(actNode) + 1);
19            pathsMap.replace(child, pathsMap.get(actNode));
20          } else if (distanceMap.get(child) == distanceMap.get(actNode)
21            + 1)) {
22            pathsMap.replace(child, pathsMap.get(child) + pathsMap
23              .get(actNode));
24          }
25        }
26      }
27      return new Tuple(distanceMap, pathsMap);
28    }
```

3.6 Testung der Klassenbibliothek

Nachdem nun beschreiben wurde, wie die Klassenbibliothek statisch und dynamisch beschaffen ist und ausführlich erläutert wurde, wie die Algrithmen zur Berechung der Metriken funktionieren, ist es noch nötig kurz aufzuzeigen, wie die fertige Bibliothek getestet wurde.

Das Testen von Software ist aus vielen Gesichtspunkten heraus wichtig. So stellen Tests die korrekte Ausführung des Codes sicher und machen infolgedessen auf fehlerhafte Änderungen während des Entwicklungsprozesses aufmerksam, bzw. zeigen sehr schnell auf, ob der geschriebene Code korrekt ist. Weiterhin können Tests auch als Documentationskomponente dienen, da sie zeigen, welche Ausgabe eine Softwarekomponente für eine bestimmte Eingabe erbringen soll. Schlussendlich sind Test aber auch deshalb wichtig, da sie das Design der Software verbessern, da sie einen Entwickler oder eine Entwicklerin zwingen Methoden und Komponenten so zu schreiben, dass sie testbar sind. Beispielsweise fällt schneller auf, dass Code aus einer Klasse ausgelagert werden sollte, der in privaten Methoden steht. Diese können nämlich nicht getestet werden, obwohl es vielleicht erwünscht ist. [GS17]

Das primäre Ziel der Klassenbibliothek ist es die Metriken für beliebige Graphen zu berechnen. Aufgrund dessen ist es auch vorrangiges Ziel der Tests, zu prüfen, ob

die jeweiligen Metriken korrekt berechnet werden. Um die Tests zu schreiben, werden jeweils Graphen erstellt, für die das Ergebnis der zu testenden Metrik bekannte ist. Beispielsweise ist die chromatische Zahl eines vollständigen Graphens gleich seiner Ordnung. Für jede Metrik innerhalb der Bibliothek wurde folgendermaßen ein Test mit dieser Struktur geschrieben. Bei Bedarf wurden mehrere solche Testgeschrieben, um Randfälle der Metriken abzudecken. So wurde z.B. getestet, ob die Distanzmetrik für zwei Knoten zwischen denen es keinen Weg gibt, auch den Wert *Integer.MAX_VALUE* zurückgibt. Weitere wichtige Randfälle sind wiederum der leere Graph oder ein Graph mit nur einem Knoten, bzw. keinen Kanten.

```
1 testMetric() {
2   graph = bekannterGraph();
3   assertEquals(bekanntesErgebnis, MetricCalculation.metric(graph));
4 }
```

Neben den Metriken selbst wurden auch die anderen Komponenten der Bibliothek getestet. Beispielsweise wurde geprüft, ob die Hilfsmethoden wie der Binomialkoeffizient richtig arbeiten oder ob die Erstellung der Standardgraphen korrekt verläuft.

Eine weiterer wichtiger Teil der Bibliothek ist die Graphdatenstruktur selbst. Auch die Methoden des implementierten Graphen selbst wurden getestet. Hierbei wird vor allem darauf geachtet, dass die jeweiligen Modifikationen effektiv sind und die analysierenden Methoden die richtigen Antworten liefern.

3.7 Überprüfung der nicht funktionalen Anforderungen

Durch die Testung der Klassenbibliothek und den Programmentwurf selbst konnte gezeigt werden, wie und ob die erstellte Klassenbibliothek ihre funktionalen Anforderungen erfüllt. Neben den funktionalen Anforderungen wurden allerdings zusätzlich nicht funktionale Anforderungen definiert. Diese sollen auch überprüft werden.

Die erste nicht funktionale Anforderung, die an die Bibliothek gestellt wurde ist die Erfüllung gewisser Usability-Merkmale. Besonders wichtig war dabei eine einfache Bedienung und die Fehlerfreiheit der Bibliothek. Letzteres konnte erreicht werden, indem die Software ausführlich getestet wurde. Durch die erstellten Unit-Tests und die zugehörige Überprüfung von Standard- und Randfällen konnte programmatisch festgestellt werden, dass die Berechung der Graphmetriken und die zugehörigen Helfer-Klassen korrekt funktionieren. Aus den Rahmenbedingungen der Bibliothek (3.1.2) kann darüber hinaus entnommen werden, dass das Abhängigkeitsmanagementtools „Maven“ genutzt wird. Wird mit „Maven“ die Bibliothek gebaut, werden automatisch auch alle geschriebenen Tests mit ausgeführt. Dieser Mechanismus sorgt dafür, dass kein Softwareartefakt ausgeliefert werden kann, dass fehlerbehaftet ist. Dies ist wichtig, da ansonsten die Usability nicht gewährleistet werden kann. Weiterhin wurde definiert,

dass die Software leicht zu bedienen sein soll. Da solche nicht funktionalen Anfroderungen schwer zu überprüfen sind, wurde bereits während der Anforderungsformulierung eingeschränkt, dass sich dieser Aspekt auf die Bennennung von Methoden und Klassen weitesgehend beschränkt. Anderenfalls müsste mittels Anwenderlaboren und anderen Überprüfungsmethoden aufwendig die Usability der Bibliothek evaluiert werden. Dies ist allerdings nicht der Fokus dieser Studienarbeit. So kann nur ausgesagt werden, dass beim Entwurf der Klassenbibliothek darauf geachtet wurde, dass sich diese möglichst intuitiv bedienen lässt. So gibt es bei der Metrikenberechnung einen einheitlichen Einstiegspunkt. Hierbei wird über die Mitgabe eines Enumerationsparameters bestimmt, welche Metrik berechnet wird. Der Rest der Berechnung bleibt dem Anwender vollständig verborgen und ist auch unwichtig für ihn. Bei der Graphdatenstruktur wurde beachtet, dass ein Graph verschiedenste Sachverhalte repräsentieren kann. Hierfür ist die gesamte Klassenbibliothek generisch desiged und es wird infolgedessen häufig von parametrisierter Polymorphie Gebrauch gemacht. Dadruch wird der Graph beliebig einsetzbar. Der spätere Einsatz des Graphen ist durch die Schnittstelle klar vorgegeben und alle Methodennamen sagen klar aus, welche Funktion ausgeführt wird. Dadurch dass die Bibliothek zwei Implementierungen zur Verfügung stellt, wird der Einstieg in die Nutzung der Bibliothek einfacher.

Eine weitere geforderte nicht funkionale Anforderung ist die der Angemessenheit und der Genauigkeit. Beide stehen im direkten Zusammenhang mit den normalen funktionalen Eigenschaften der Software. Da die funktionalen Anforderungen der Software erfüllt sind und diese durch die Testung auch genau erfüllt wurden, kann gesagt werden, dass die Bibliothek sowohl angemessen als auch genau ist.

Weiterhin war die Wartbarkeit des Systems eine wichtige Eigenschaft die erreicht werden sollte. Besonders wichtig war hierbei die Testbarkeit. Diese konnte dadurch nachgewiesen werden, dass beinahe alle Methoden der Bibliothek direkt oder indirekt getestet wurden. Allein bei der Berechnung der Graphmetirken konnte eine Testabdeckung von über 95 % erreicht werden.

Neben der Wartbarkeit war zudem die Portabilität eine nicht funkionale Eigenschaft, die von der Bibliothek erfüllt werden musste. Im Rahmen der Portabilität sollte speziell eine hohe Anpassbarkeit und Installierbarkeit erreicht werden. Beide Punkte können vollständig durch die gesetzten Rahmenbedingungen der Software abgedeckt werden. Da die Software mit Java geschrieben wurde und mithilfe von Maven gebaut wird, ist sie auch automatisch in sämtliche Maven/Java-Projekte inbindbar. Wie bereits in 3.1.2 erwähnt ist Java äußerst weit verbreitet und kann auf vielen Maschinen ausgeführt werden, bzw. teils in andere JVM-Sprachen wie Scala integiert werden. [Ull16] Auch dies erhöht die Portabilität noch einmal. Die beschriebene Portabilität des Software schließt zudem die in 3.1.6 gewünschte Nutzbarkeit als Klassenbibliothek mit ein, da durch Maven die Software überall als Bibliothek mit eingebunden werden kann.

Die letzte nicht funkionale Anforderung die überprüft werden soll, ist die Effizienz.

Diese Anforderung hatte keine hohe Priorität und wurde deshalb auch nicht konsequent umgesetzt. Zwar wurde darauf geachtet, dass die genutzen Algroithmen keine unnötig lange Laufzeit haben, allerdings waren manche Berechnungen gar nicht effizient zu berechnen. So sind die chomarische Zahl und der chormatische Index NP-vollständige Probleme, für die es vermutlich keine effiziente Lösung gibt. Bei der Implementierung der Graphdatenstruktur wurde wiederum darauf geachtet eine Implementierung zu wählen, die möglichst günstig ist (siehe. 3.2). So wurde beispielsweise zur Darstellung der Adjazenzlisten eine Java-Hashmap genutzt, da hier z.B. die Suche nach der richtigen Liste zu einem Knoten in konstanter Zeit verläuft. Im Falle einer klassischen Implementierung mit einer Liste von Listen wäre dies nur in linearer Zeit möglich. Vor allem bei einem großen Graphen kann dies einen Zeitvorteil verschaffen. Zusammenfassend kann also gesagt werden, dass bei der Erstellung der Software darauf geachtet wurde nicht unperformant zu programmieren, allerdings kein Fokus darauf lag, eine besonders hohe und quantitaiv messbare Effizienz zu erreichen.

4 Abstraktion der Graphmetrik-Berechnung

Neben der Umsetzung der Graphmetriken mithilfe einer Klassenbibliothek, ist es noch Ziel der Studienarbeit zu beleuchten, wie es möglich ist die Metriken-Berechnung noch weiter zu abstrahieren. Damit kann auch die letzte Fragestellung der Arbeit beantowrtet werden. Um dieses Ziel zu erreichen soll, zunächst noch einmal die Problemstellung der Abstraktion genau beschrieben werden und anschließend auf Lösungsansätze eingegangen werden.

4.1 Problemstellung der Abstraktion

Im letzten Kapitel wurde äußerst ausführlich gezeigt, wie eine Klassenbibliothek zur Umsetzung von einer Menge an Graphmetriken aussehen kann und wie die jeweiligen Metriken zur Berechnung der Kennzahlen und Invarianten algorithmisch umgesetzt sind. Die Umsetzung über eine eigene Klassenbibliothek ist allerdings nicht die einzig denkbare Variante, um die Metriken zu berechnen. So gibt es anderer Bibliotheken, die selbst Graphdatenstrukturen und eine Kennzahlberechnung besitzen. Dazu zählen z.B. die Java-Bibliotheken „JgraphT“ und „Java Universal Network/Graph Framework“. Weiterhin gibt es aber auch Bibliotheken, die in vollkommen anderen Sprachen umgesetzt wurden, aber das gleiche Ziel verfolgen. So können Graphen auch mit den Bibliotheken „The Boost Graph Library“ und „LEMON“ in C++ erstellt und analysiert werden. Neben Programmbibliotheken ist es aber auch möglilch Graphen mittels Graphdatenbanken zu erstellen, zu persistieren und zu analysieren. Bereits in Kapitel 2 dieser Studienarbeit wurden Quellen aus Dokumentationen von neo4j referenziert, um bestimmte Metriken zu erläutern, die u.a. auch innerhalb der Datenbank berechenbar sind. Neben neo4j gibt es aber natürlich auch andere Graphdatenbanken, die verschiedenste Algrotihmen auf Graphen ausführen können. Ein weiterer Vertreter ist z.B. „TigerGraph“.

Wie zu sehen ist, ist die Berechnung der Graphmetriken auf vielfältige Art und Weise möglich. Das Problem besteht jetzt darin, eine Möglichkeit zu schaffen diese Berechnung so zu abstrahieren, dass man theoretisch die verschiedenen Berechnungs-Logiken über eine zentrale einheitliche Schnittstelle ansprechen kann. Diese Schnittstelle vereinheitlicht die Analyse und Erstellung von Graphen, sodass ein potenzieller Nutzer oder

eine potenzielle Nutzerin nur das gewünschte „Backend“ wählen muss, für ihn oder sie allerdings die konkrete Implementierung irrelevant ist.

4.2 Lösungsansätze zur Abstraktion der Graphmetrik-Berechnung

Zur Lösung der Problemstellung ergeben sich zwei Probleme, die eine Softwarelösung adressieren muss. Erstens muss geklärt werden, wie das Programm einkommende Anfragen an die verschiedenen Berechnungs-„Backends“ verteilen kann. Zweitens muss eine einheitliche Graphdarstellung geschaffen werden, da sämtliche Berechnungs-Logiken eigene Darstellungsweis für einen Graphen haben. Im Falle einer Graphdatenbank muss sogar ein Graph persistiert werden, damit auf ihm gearbeitet werden kann.

Ähnlich wie beim Entwurf für die Klassenbibliothek muss die Abstraktion der Graphmetrik-Berechung über eine Schichtenmuster bzw. eine Schichtenarchitketur gelöst werden. Die Schichten sind dabei in einer linearen Ordnung. D.h. eine höhere Schicht kann jeweils nur auf auf die nächst niedrigere Schicht zugreifen. [BL11] Eine untere Schicht kann auch nur auf die nächst höhere zugreifen. Grundsätzlich benötigt man 2 bzw. 3 Schichten zur Lösung des Problems. Die erste Schicht ist die Schnittstellen- oder „Boundary“-Ebene. Die Komponenten hier stellen die beschriebene einheitliche Schnittstelle zur Verfügung, die Anfragen zur Graphverwaltung und vor allem zur Metrikenberechnung entgegennimmt und weiterleitet. In der zweiten Schicht wird die eigentliche Business-Logik umgesetzt („Bontrol“). Hier befinden sich alle Komponenten, die die jeweiligen einzelnen Berechnungs-„Backends“ ansprechen. Zudem muss es eine Komponente geben, die initial alle Anfragen der „Boundary“ annimmt und je nach bearbeitenden Berechnungs-„Backend“ eine Transformation der Graphdatenstruktur vornimmt, damit korrekt weitergearbeitet werden kann. Damit das Architekturprinzip der „Trennung von Zuständigkeiten“ nicht verletzt wird, wird die Transformationslogik wiederum in eine eigene Komponente ausgelagert [BL11]. In einer dritten Schicht kann es nun noch optional eine Persitierung geben („Entity“). Dies ist dann der Fall, wenn das Berechnungs-„Backend“ eine Graphdatenbank ist, die wiederum ein eigenstädiges Programm, bzw. eine eigenständige Komponente ist.

Konkretisiert muss es zur Abstraktion der Graphmetrik-Berechnung also eine Schnittstellenschicht geben mit mindestens einer Komponente, die alle einheitlichen Leistungen definiert, die die Graphmetrik-Berechnung zur Verfügung stellt. In der „Control“-Ebene wird eine Komponente bereitgestellt, die all diese Anfragen annimmt und je nach Berechnungs-„Backend“ den zu untersuchenden Graphen transformiert und anschließend diesen an die eigentlichen Business-Logik-Komponenten weitergibt. Diese Komponenten arbeiten dann mit den jeweiligen Bibliothek und Graphdatenbanken und

sprechen diese mit ihrer API an. Im Falle der Graphdatenbank liegen diese auf der dritten und untersten Schicht. Zusammengefasst kann die Abstraktion der Graphmetrik-Berechnung mit dem Komponentendiagramm in Abbildung 4.1 dargestellt werden. Die

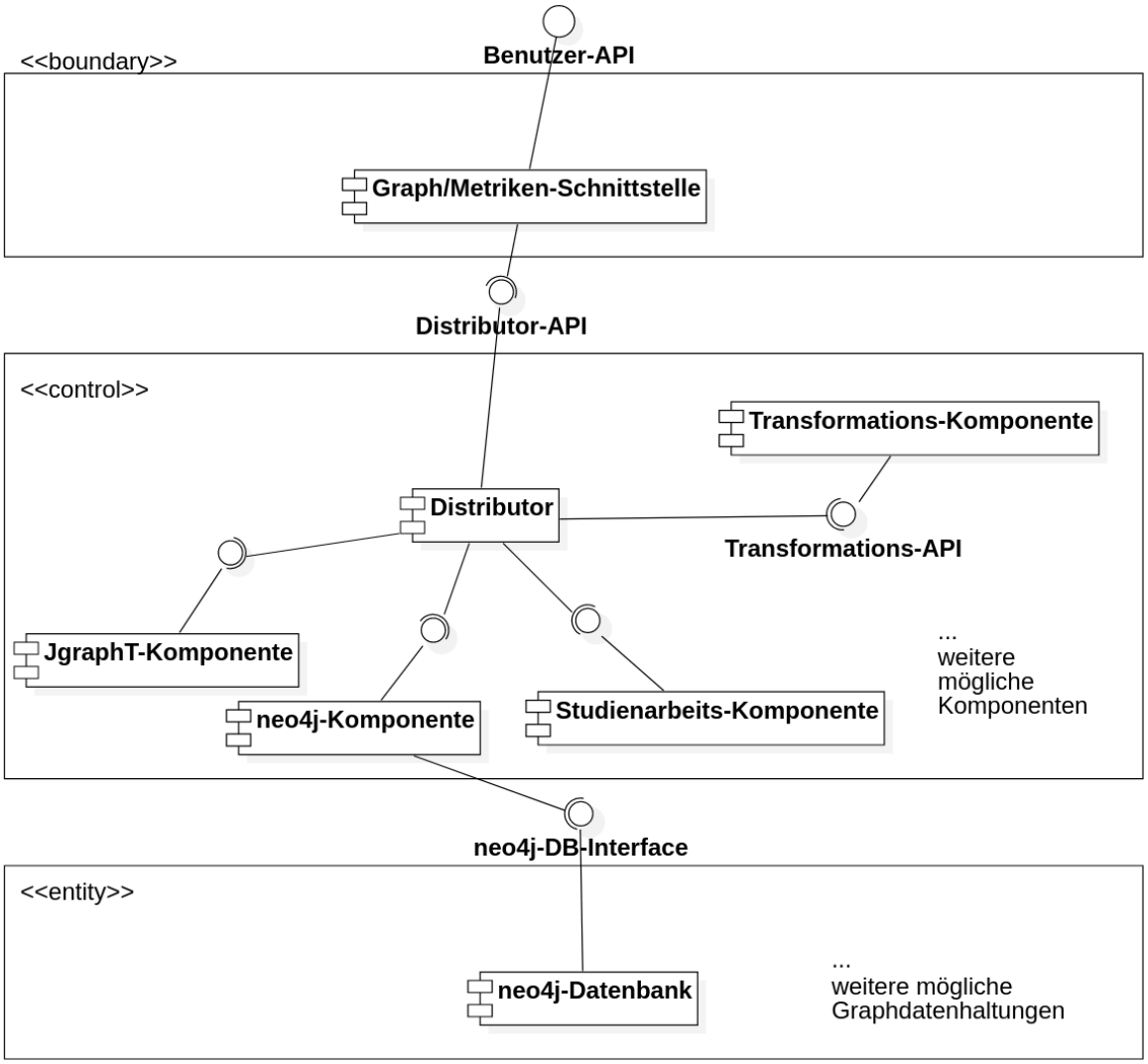

Abbildung 4.1: Komponentendiagramm: Konzeption zur Abstraktion der Graphmetrik-Berechnung

aufgezeigten Komponenten zeigen einzig und allein eine mögliche Konzeption und Aufgabenteilung zur Lösung des Abstraktionsproblems. Wie die konkrete Umsetzung einer solchen Software auszusehen hat, ist schlussendlich davon abhängig, wie sie eingesetzt werden soll. Weiterhin wird durch die Lösung auch nicht geklärt, wie damit umgegangen wird, wenn bestimmte Funktionen nur in machen Berechnungskomponenten vorhanden sind.

Möchte man nun den gegebenen Grobentwurf umsetzten, lassen sich verschiedene Konzepte verfolgen. Die offensichtlichste Umsetzung erfolgt mittels eines normalen Programms, dass alle Komponenten in sich vereint. In diesem Sinne ähnelt die Klassenbibliothek aus Kapitel 3 der gezeigten Abstraktion. Denn auch in der erstellten Bibliothek wurde die Metriken-Berechnung über eine Schnittstelle abstrahiert, sodass

die Implementierung für den Anwender oder die Anwenderin irrelevant ist. Setzt man die Abstraktion der Graphmetrik-Berechnung allerdings als ein zusammenhängendes Programm um, ergeben sich Probleme die adressiert werden müssen. Zunächst können nur Berechnungskomponenten in das Programm z.B. als Bibliothek eingebunden werden, wenn diese mit der Technologie kompatibel sind, in der das Programm geschrieben ist. So kann es sein, dass eine C++-Bibliothek zur Berechnung von Graphmetriken nicht Teil eines Java-Programms sein kann. Des Weiteren muss noch ein Mechanismus zur Verfügung gestellt werden, der eine einheitliche und transformierbare Graphdatenstruktur zur Verfügung stellt, die für die „Benutzer-API“ verwendet werden kann.

Eine weitere Möglichkeit zur Umsetzung des Grobentwurfes wäre es die die jeweiligen Komponenten als eigenständige Programme wahrzunehmen. Dies bietet unter anderem den Vorteil, dass man technisch nicht eingeschränkt ist, solange die Schnittstellen für jeden Teil des Systems klar definiert und ansprechbar sind. So können z.B. Berechnungs-Logiken verschiedenster Sprachen verwendet werden, ohne dass dabei Kompatiblitätsprobleme entstehen. Allerdings würde sich die Komplexität des gesamten Systems deutlich erhöhen. Jede einzelne Komponente benötigt eine eigenständige API, die in der Lage ist, nach außen zu kommunizieren. Infolgedessen müssen vor allem die angebunden Klassenbibliothek erweitert werden, sodass sie als eigenständiger Service angesprochen werden können. Die Kommunikation zwischen den einzelnen Komponenten kann sehr divers sein, solange die Kommunikationswege und die Schnittstellen klar sind. Möglich wäre eine Kommunikation über REST oder SOAP, wobei jeweils das HTTP-Protokoll verwendet wird. Ansonsten sind auch sämtliche andere Arten der Netzwerkkommunikation möglich wie Sokets oder XML-RPC. [BL11] Alternativ kann auch auf eine Message Oriented Middleware (MOM) zurückgegriffen werden. Dies ist eine Software, die speziell dafür designed ist Nachrichten zwischen Applikationen zu verschicken [Cur05]. MOM und Netzwerkkommunikation können auch beliebig miteinander kombiniert werden. Auch bei der Umsetzung mittels eigenständiger Programme kommt aber noch das Problem auf, dass eine zusätzliche einheitliche Graphdatenstruktur braucht, die in die jeweiligen Strukturen der Datenbanken und Graph-Bibliotheken tranformiert werden kann. Zusätzlich muss bei der Kommunikation über die Schnittstellen ein Format gefunden werden, über das kommuniziert wird.

5 Fazit und Ausblick

5.1 Fazit zu den Ergebnissen der Studienarbeit

5.2 Ausblick

Glossar

Literatur

[Aig15] Martin Aigner. *Graphentheorie: eine Einführung aus dem 4-Farben Problem*. 2., überarbeitete Auflage. Springer Studium Mathematik Bachelor. OCLC: 927721160. Wiesbaden: Springer Spektrum, 2015. 196 S. ISBN: 978-3-658-10322-4 978-3-658-10323-1.

[Alo88] N. Alon. "The linear arboricity of graphs". In: *Israel Journal of Mathematics* 62.3 (Okt. 1988), S. 311–325. ISSN: 0021-2172, 1565-8511. DOI: 10.1007/BF02783300. URL: http://link.springer.com/10.1007/BF02783300.

[And77] Lars Dovling Andersen. "On edge-colorings of graphs." In: *MATHEMATICA SCANDINAVICA* 40 (1. Dez. 1977), S. 161. ISSN: 1903-1807, 0025-5521. DOI: 10.7146/math.scand.a-11685. URL: http://www.mscand.dk/article/view/11685 (besucht am 24. 10. 2020).

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements-Engineering*. 3. Aufl. Lehrbücher der Informatik. OCLC: 488675080. Heidelberg: Spektrum, Akad. Verl, 2009. 624 S. ISBN: 978-3-8274-1705-3.

[Bal97] V. K. Balakrishnan. *Schaum's outline of theory and problems of graph theory*. Schaum's outline series. New York: McGraw-Hill, 1997. 293 S. ISBN: 978-0-07-005489-9.

[Bet19] Tyler Elliot Bettilyon. *Implementations of Graphs*. Medium. 6. Feb. 2019. URL: https://medium.com/tebs-lab/implementations-of-graphs-92eb7f121793 (besucht am 17. 03. 2021).

[Bha19] Jatin Bhasin. *Graph Analytics — Introduction and Concepts of Centrality*. Medium. 19. Aug. 2019. URL: https://towardsdatascience.com/graph-analytics-introduction-and-concepts-of-centrality-8f5543b55de3 (besucht am 26. 01. 2021).

[BK79] Frank Bernhart und Paul C Kainen. "The book thickness of a graph". In: *Journal of Combinatorial Theory, Series B* 27.3 (Dez. 1979), S. 320–331. ISSN: 00958956. DOI: 10.1016/0095-8956(79)90021-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0095895679900212 (besucht am 24. 10. 2020).

[BL11] Helmut Balzert und Peter Liggesmeyer. *Lehrbuch der Softwaretechnik. 2: Entwurf, Implementierung, Installation und Betrieb*. 3. Aufl. Lehrbücher der Informatik. OCLC: 750951360. Heidelberg: Spektrum, Akad. Verl, 2011. 596 S. ISBN: 978-3-8274-1706-0.

[Bla03] Robin Leigh Blankenship. “Book embeddings of graphs”. Diss. Louisiana State University, 2003. URL: https://digitalcommons.lsu.edu/gradschool_dissertations/3734?utm_source=digitalcommons.lsu.edu%2Fgradschool_dissertations%2F3734&utm_medium=PDF&https://digitalcommons.lsu.edu/gradschool_dissertations/3734/utm_campaign=PDFCoverPages.

[BP15] Anand Bihari und Manoj Pandia. “Eigenvector centrality and its application in research professionals’ relationship network”. In: *2015 1st International Conference on Futuristic Trends in Computational Analysis and Knowledge Management, ABLAZE 2015*. 2015. DOI: 10.1109/ABLAZE.2015.7154915.

[BP98] Sergey Brin und Lawrence Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. 1998. URL: http://infolab.stanford.edu/~backrub/google.html (besucht am 28. 01. 2021).

[Bra01] Ulrik Brandes. “A faster algorithm for betweenness centrality*”. In: *The Journal of Mathematical Sociology* 25.2 (Juni 2001), S. 163–177. ISSN: 0022-250X, 1545-5874. DOI: 10.1080/0022250X.2001.9990249. URL: http://www.tandfonline.com/doi/abs/10.1080/0022250X.2001.9990249 (besucht am 26. 01. 2021).

[CB15] Paul Chiusano und Rúnar Bjarnason. *Functional programming in Scala*. OCLC: ocn823712614. Shelter Island, NY: Manning Publications, 2015. 300 S. ISBN: 978-1-61729-065-7.

[Chv06] V. Chvátal. “Tough graphs and hamiltonian circuits”. In: *Discrete Mathematics* 306.10 (Mai 2006), S. 910–917. ISSN: 0012365X. DOI: 10.1016/j.disc.2006.03.011. URL: https://linkinghub.elsevier.com/retrieve/pii/S0012365X06001397 (besucht am 09. 12. 2020).

[CKL10] Marek Cygan, Łukasz Kowalik und Borut Lužar. “A Planar Linear Arboricity Conjecture”. In: *Algorithms and Complexity*. Hrsg. von Tiziana Calamoneri und Josep Diaz. Bd. 6078. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 204–216. ISBN: 978-3-642-13072-4 978-3-642-13073-1. DOI: 10.1007/978-3-642-13073-1_19. URL: http://link.springer.com/10.1007/978-3-642-13073-1_19 (besucht am 22. 01. 2021).

[CLR87] Fan R. K. Chung, Frank Thomson Leighton und Arnold L. Rosenberg. “Embedding Graphs in Books: A Layout Problem with Applications to VLSI Design”. In: *SIAM Journal on Algebraic Discrete Methods* 8.1 (Jan. 1987), S. 33–58. ISSN: 0196-5212, 2168-345X. DOI: 10.1137/0608002. URL: https://epubs.siam.org/doi/10.1137/0608002 (besucht am 29. 01. 2021).

[Coh+14] Edith Cohen u. a. “Computing Classic Closeness Centrality, at Scale”. In: *Proceedings of the Second ACM Conference on Online Social Networks*. COSN ’14. event-place: Dublin, Ireland. New York, NY, USA: Association for Computing Machinery, 2014, S. 37–50. ISBN: 978-1-4503-3198-2. DOI: 10.1145/2660460.2660465. URL: https://doi.org/10.1145/2660460.2660465.

[Cun85] William H. Cunningham. “Optimal Attack and Reinforcement of a Network”. In: *J. ACM* 32.3 (Juli 1985). Place: New York, NY, USA Publisher: Association for Computing Machinery, S. 549–561. ISSN: 0004-5411. DOI: 10.1145/3828.3829. URL: https://doi.org/10.1145/3828.3829.

[Cur05] Edward Curry. “Message-Oriented Middleware”. In: *Middleware for Communications*. 2005, S. 1–28. ISBN: 978-0-470-86206-3. DOI: 10.1002/0470862084.ch1.

[Die00] Reinhard Diestel. *Graphentheorie*. 2., neu bearb. und erw. Aufl. Springer-Lehrbuch. OCLC: 247312585. Berlin: Springer, 2000. 314 S. ISBN: 978-3-540-67656-0.

[EK13] W. Ellens und R. E. Kooij. *Graph measures and network robustness*. _eprint: 1311.5064. 2013.

[Eve12] Shimon Even. *Graph algorithms*. Unter Mitarb. von Guy Even. 2nd ed. Cambridge, NY: Cambridge University Press, 2012. 189 S. ISBN: 978-0-521-51718-8 978-0-521-73653-4.

[Fre78] Linton C. Freeman. “Centrality in social networks conceptual clarification”. In: *Social Networks* 1.3 (1978), S. 215–239. ISSN: 0378-8733. DOI: https://doi.org/10.1016/0378-8733(78)90021-7. URL: http://www.sciencedirect.com/science/article/pii/0378873378900217.

[Gep17] Gephi. *Supported Graph Formats*. 2017. URL: https://gephi.org/users/supported-graph-formats/ (besucht am 14. 04. 2021).

[GIT14] GITTA. *Durchmesser eines Graphen*. Durchmesser eines Graphen. 2014. URL: https://www.gitta.info/Accessibiliti/de/html/StructPropNetw_learningObject2.html (besucht am 18. 11. 2020).

[GS17] Shekhar Gulati und Rahul Sharma. *Java Unit Testing with JUnit 5*. Berkeley, CA: Apress, 2017. ISBN: 978-1-4842-3014-5 978-1-4842-3015-2. DOI: 10.1007/978-1-4842-3015-2. URL: http://link.springer.com/10.1007/978-1-4842-3015-2 (besucht am 20. 04. 2021).

[GW92] Harold N. Gabow und Herbert H. Westermann. "Forests, frames, and games: Algorithms for matroid sums and applications". In: *Algorithmica* 7.1 (Juni 1992), S. 465–497. ISSN: 0178-4617, 1432-0541. DOI: 10.1007/BF01758774. URL: http://link.springer.com/10.1007/BF01758774 (besucht am 21. 01. 2021).

[GY04] Jonathan L. Gross und Jay Yellen, Hrsg. *Handbook of graph theory*. Discrete mathematics and its applications. Boca Raton: CRC Press, 2004. 1167 S. ISBN: 978-1-58488-090-5.

[Har01] Frank Harary. *Graph theory*. 15. print. OCLC: 248770458. Cambridge, Mass: Perseus Books, 2001. 274 S. ISBN: 978-0-201-41033-4.

[IEE98] IEEE. "IEEE Recommended Practice for Software Requirements Specifications". In: *IEEE Std 830-1998* (1998), S. 1–40. DOI: 10.1109/IEEESTD.1998.88286.

[Jun13] Dieter Jungnickel. *Graphs, networks and algorithms*. 4th ed. Algorithms and computation in mathematics 5. OCLC: 821566132. Berlin: Springer, 2013. 675 S. ISBN: 978-3-642-32278-5 978-3-642-32277-8.

[Kai90] Paul Kainen. "The book thickness of a graph, II". In: *Congressus Numerantium* 71 (1990), S. 127–132.

[Kar96] Richard M. Karp. "Reducibility among combinatorial problems". In: *INFORMS Jounal on Computing* 8.4 (1996), S. 344–354.

[Kne19] Helmut Knebl. *Algorithmen und Datenstrukturen: Grundlagen und probabilistische Methoden für den Entwurf und die Analyse*. OCLC: 1123167896. 2019. ISBN: 978-3-658-26511-3.

[Kol21] Andreas Kollegger. *Gram: a data graph format*. DEV Community. 2021. URL: https://dev.to/neo4j/gram-a-data-graph-format-3mi2 (besucht am 14. 04. 2021).

[Lov12] László Lovász. *Large networks and graph limits*. American Mathematical Society colloquium publications volume 60. Providence, Rhode Island: American Mathematical Society, 2012. 475 S. ISBN: 978-0-8218-9085-1.

[Mat20a] Matlab. *Directed and Undirected Graphs - MATLAB & Simulink - MathWorks Deutschland*. Directed and Undirected Graphs. 2020. URL: https://de.mathworks.com/help/matlab/math/directed-and-undirected-graphs.html (besucht am 10. 11. 2020).

[Mat20b] Matlab. *Graph and Network Algorithms*. Graph and Network Algorithms. 2020. URL: https://de.mathworks.com/help/matlab/graph-and-network-algorithms.html?searchHighlight=graph&s_tid=srchtitle (besucht am 12. 01. 2021).

[Mat20c] Matlab. *Measure node importance*. centrality. 2020. URL: https://de.mathworks.com/help/matlab/ref/graph.centrality.html#bu86660 (besucht am 26. 01. 2021).

[Mat20d] Matlab. *Shortest path distances of all node pairs*. Distances. 2020. URL: https://de.mathworks.com/help/matlab/ref/graph.distances.html (besucht am 18. 11. 2020).

[Mat87] D. W. Matula. "Determining edge connectivity in 0(nm)". In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. ISSN: 0272-5428. Okt. 1987, S. 249–251. DOI: 10.1109/SFCS.1987.19.

[Meg15] Natarajan Meghanathan. "Use of Eigenvector Centrality to Detect Graph Isomorphism". In: *Computer Science & Information Technology* 5 (2015). DOI: 10.5121/csit.2015.51501.

[MW17] Heinrich Müller und Frank Weichert. *Vorkurs Informatik: der Einstieg ins Informatikstudium*. 5., erweiterte und überarbeitete Auflage. Lehrbuch. OCLC: 993704399. Wiesbaden: Springer Vieweg, 2017. 392 S. ISBN: 978-3-658-16140-8 978-3-658-16141-5.

[Nas61] C. St.J. A. Nash-Williams. "Edge-Disjoint Spanning Trees of Finite Graphs". In: *Journal of the London Mathematical Society* s1-36.1 (1961), S. 445–450. ISSN: 00246107. DOI: 10.1112/jlms/s1-36.1.445. URL: http://doi.wiley.com/10.1112/jlms/s1-36.1.445 (besucht am 21. 01. 2021).

[neo20a] neo4j. *Betweenness Centrality - Centrality algorithms*. Graph Data Science. 2020. URL: https://neo4j.com/docs/graph-data-science/current/algorithms/betweenness-centrality/ (besucht am 27. 01. 2021).

[neo20b] neo4j. *Closeness Centrality - Centrality algorithms*. Graph Data Science. 2020. URL: https://neo4j.com/docs/graph-data-science/current/algorithms/closeness-centrality/ (besucht am 27. 01. 2021).

[neo20c] neo4j. *Eigenvector Centrality - Centrality algorithms*. Graph Data Science. 2020. URL: https://neo4j.com/docs/graph-data-science/current/algorithms/eigenvector-centrality/ (besucht am 27. 01. 2021).

[Nie10] Jakob Nielsen. *Usability engineering*. Nachdr. OCLC: 760142137. Amsterdam: Kaufmann, 2010. 362 S. ISBN: 978-0-12-518406-9.

[Ora17] Oracle. *PageRank and variants*. Oracle PGX 2.4.0 Documentation. 2017. URL: https://docs.oracle.com/cd/E56133_01/2.4.0/reference/algorithms/pagerank.html (besucht am 28. 01. 2021).

[Ora21] Oracle. *Java Platform, Standrad Edition 8 API Specification*. 2021. URL: https://docs.oracle.com/javase/8/docs/api/ (besucht am 16. 04. 2021).

[Pér84] B. Péroche. "NP-completeness of some problems of partitioning and covering in graphs". In: *Discrete Applied Mathematics* 8.2 (Mai 1984), S. 195–208. ISSN: 0166218X. DOI: 10.1016/0166-218X(84)90101-X. URL: https://linkinghub.elsevier.com/retrieve/pii/0166218X8490101X (besucht am 22. 01. 2021).

[Pla83] Michael J. Plantholt. "The chromatic index of graphs with large maximum degree". In: *Discrete Mathematics* 47 (1983), S. 91–96. ISSN: 0012365X. DOI: 10.1016/0012-365X(83)90074-2. URL: https://linkinghub.elsevier.com/retrieve/pii/0012365X83900742 (besucht am 17. 01. 2021).

[Res15] Wolfram Research. *FindClique*. Publisher: Wolfram Research. 2015. URL: https://reference.wolfram.com/language/ref/FindClique.html (besucht am 29. 01. 2021).

[RN16] Stuart J. Russell und Peter Norvig. *Artificial intelligence: a modern approach*. Third edition, Global edition. Prentice Hall series in artificial intelligence. Boston: Pearson, 2016. 1132 S. ISBN: 978-1-292-15396-4.

[Sag] SageMath. *Generic graphs (common to directed/undirected) — Sage 9.2 Reference Manual: Graph Theory*. Sage Math Reference Manual. URL: https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html (besucht am 10. 11. 2020).

[Sag20a] SageMath. *Graph coloring*. 2020. URL: https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/graph_coloring.html (besucht am 14. 01. 2021).

[Sag20b] SageMath. *Graph Theory*. Sage Math Reference Manual. 2020. URL: https://doc.sagemath.org/html/en/reference/graphs/index.html (besucht am 25. 10. 2020).

[Sar+13] Ahmet Erdem Sariyuce u. a. "Incremental algorithms for closeness centrality". In: 2013, S. 487–492. DOI: 10.1109/BigData.2013.6691611.

[Sri11] Srirangan. *Apache Maven 3 cookbook: over 50 recipes towards optimal Java software engineering with Maven 3*. Quick answers to common problems. OCLC: 838117329. Birmingham: Packt Publ, 2011. 208 S. ISBN: 978-1-84951-244-2 978-1-84951-245-9.

[Sry20] Said Sryheni. *Number of Shortest Paths in a Graph — Baeldung on Computer Science*. 9. Dez. 2020. URL: https://www.baeldung.com/cs/graph-number-of-shortest-paths (besucht am 30. 03. 2021).

[SU11] Edward R. Scheinerman und Daniel H. Ullman. *Fractional graph theory: a rational approach to the theory of graphs*. Dover books on matehmatics. OCLC: ocn721885660. Minola, N.Y: Dover Publications, 2011. 211 S. ISBN: 978-0-486-48593-5.

[Tit19] Peter Tittmann. *Graphentheorie: Eine anwendungsorientierte Einführung*. 3., aktualisierte Auflage. München: Hanser, Carl, 2019. 168 S. ISBN: 978-3-446-46052-2 978-3-446-46503-9.

[Tru93] V. A. Trubin. "Strenght of a graph and packing of trees and branchings". In: *Cybernetics and Systems Analysis* (1993).

[Tur04] Volker Turau. *Algorithmische Graphentheorie*. Oldenbourg Wissenschaftsverlag, 1. Jan. 2004. ISBN: 978-3-486-59377-8. DOI: 10.1524/9783486593778. URL: https://www.degruyter.com/view/title/310250 (besucht am 24. 10. 2020).

[Ull14] Christian Ullenboom. *Java SE 8 Standard-Bibliothek: das Handbuch für Java-Entwickler*. 2., aktualisierte und erw. Aufl. Galileo Computing. OCLC: 882980507. Bonn: Galileo Press, 2014. 1448 S. ISBN: 978-3-8362-2874-9.

[Ull16] Christian Ullenboom. *Java ist auch eine Insel: Einführung, Ausbildung, Praxis*. 12., aktualisierte und überarbeitete Auflage. Rheinwerk Computing Standardwerk. OCLC: 934810648. Bonn: Rheinwerk Verlag GmbH, 2016. 1312 S. ISBN: 978-3-8362-4119-9 978-3-8362-4121-2.

[Vöc+08] Berthold Vöcking u. a., Hrsg. *Taschenbuch der Algorithmen*. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-76393-2 978-3-540-76394-9. DOI: 10.1007/978-3-540-76394-9. URL: http://link.springer.com/10.1007/978-3-540-76394-9 (besucht am 06. 12. 2020).

[Weia] Eric W. Weisstein. *Arboricity*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/Arboricity.html (besucht am 20. 01. 2021).

[Weib] Eric W. Weisstein. *Chromatic Number*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ChromaticNumber.html (besucht am 15. 01. 2021).

[Weic] Eric W. Weisstein. *Fractional Edge Chromatic Number*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/FractionalEdgeChromaticNumber.html (besucht am 03. 02. 2021).

[Weid] Eric W. Weisstein. *Independence Number*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/IndependenceNumber.html (besucht am 29. 01. 2021).

[Weie] Eric W. Weisstein. *Maximum Independent Vertex Set*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/MaximumIndependentVertexSet.html (besucht am 29. 01. 2021).

[Wol20a] Wolfram. *Graph Measures & Metrics*. Wolfram Language Documentation. 2020. URL: https://reference.wolfram.com/language/guide/GraphMeasures.html (besucht am 25. 10. 2020).

[Wol20b] Wolfram. *Wolfram Function Repository*. 2020. URL: https://resources.wolframcloud.com/FunctionRepository/ (besucht am 14. 01. 2021).

[Wol20c] Wolfram. *Wolfram Language & System Documentation Center*. Documentation Center. 2020. URL: https://reference.wolfram.com/language/ (besucht am 06. 12. 2020).

[XN17] Mingyu Xiao und Hiroshi Nagamochi. “Exact algorithms for maximum independent set”. In: *Information and Computation* 255 (2017), S. 126–146. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2017.06.001. URL: http://www.sciencedirect.com/science/article/pii/S0890540117300950.