ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Simulation und Analyse von topologiebasiertem Gossiping zur Konvergenzoptimierung in unstrukturierten Netzwerken

Maverick Studer

Technical Report – STL-TR-2024-01 – ISSN 2364-7167

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Maverick Studer: Simulation und Analyse von topologiebasiertem Gossiping zur Konvergenzoptimierung in unstrukturierten Netzwerken
Technical report id: STL-TR-2024-01

First published: February 2024
Last revision: October 2023
Internal review: Markus Esch

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Colourbox, https://www.colourbox.de/bild/viele-divers-junge-bild-18018386

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Master-Thesis

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Simulation und Analyse von topologiebasiertem Gossiping zur Konvergenzoptimierung in unstrukturierten Netzwerken

vorgelegt von
Maverick Studer

betreut und begutachtet von
Prof. Dr. Markus Esch

Saarbrücken, 24. Oktober 2023

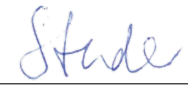
Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 24. Oktober 2023



Maverick Studer

Zusammenfassung

In den letzten Jahren hat sich ein deutlicher Wandel von zentralisierten zu hoch verteilten Systemen vollzogen. Dadurch finden unstrukturierte Netzwerke wie Sensor-, Peer-to-Peer- und Ad-hoc-Netzwerke vermehrt Anwendung. Diese Netze stehen jedoch vor vielen Herausforderungen, wie beispielsweise dem Fehlen einer zentralen Managementeinheit, unbekannten Netzwerktopologien sowie dem ständige Wechsel von Mitgliedern. Um den daraus folgenden Anforderungen gerecht zu werden, haben sich dezentrale Protokolle etabliert.

Gossiping-Verfahren gehören zu dieser Kategorie von Protokollen. Sie ermöglichen eine effiziente Verbreitung und Aggregation von Daten durch periodischen Informationsaustausch zwischen zufällig ausgewählten Partnern. Diese Zufallsauswahl kann bei bestimmten Netzwerktopologien zu Skalierungsproblemen führen.

In dieser Arbeit wird untersucht, ob diese Lücke, durch Verwendung von topologiebasierten Auswahlmechanismen, geschlossen werden kann. Dazu werden optimierte Algorithmen entwickelt, welche Wissen über Community-Strukturen verwenden, um eine gezieltere Auswahl von Kommunikationspartnern durchzuführen. Zur Evaluation der Algorithmen wird eine Simulationsumgebung mit Kubernetes aufgebaut. Diese wird dann verwendet, um verschiedene Simulationen auszuführen und so nachzuweisen, dass eine bessere Konvergenzrate erreicht werden kann.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren kann ein Wandel von zentralisierten zu hoch verteilten Systemen beobachtet werden. Dieser Trend wird von einem enormen Anstieg an vernetzten Geräte begleitet. Infolgedessen steigen auch die Netzwerkgrößen und damit die Konnektivität zwischen Computern. Zu den Gründen für diese Entwicklung zählen die Ausweitung des Internetzugangs und die Zunahme an Internet of Things (IoT)-Devices sowie mobilen Endgeräten wie Smartphones und Tablets. Auch der technologische Fortschritt in vielen industriellen Bereichen führt zum vermehrten Einsatz internetfähiger Geräte. Stand April 2023 gibt es etwa 8.5 Milliarden mobile Endgeräte im Internet, 174 Millionen mehr als im Vorjahr [32]. Die Zahl an IoT-Devices ist sogar um 2.3 Milliarden gewachsen, bis auf 16.7 Milliarden [107]. Aufgrund der Verbreitung von vernetzten Geräten werden insbesondere Peer-to-Peer- und Ad-hoc-Netzwerke sowie Sensornetze immer häufiger eingesetzt. Diese Technologien zeichnen sich durch ihre unstrukturierte Netzwerkarchitektur aus, was jedoch mit verschiedenen Einschränkungen einhergeht. Dieser Paradigmenwechsel hat die Anforderungen an Skalierbarkeit und Fehlertoleranz moderner verteilter Systeme deutlich erhöht [63]. Im Folgenden werden nun die wichtigsten Beschränkungen solcher Netzwerke erläutert [16]:

1. Die Netzwerke verfügen über keine zentrale Managementeinheit.
2. Die Netzwerktopologie bleibt den Mitgliedern unbekannt.
3. Die Netzwerktopologie ist ständigen Veränderungen unterworfen, da Hosts beliebig hinzukommen oder ausscheiden können.
4. Die Stabilität und Geschwindigkiet der Verbindung sowie die Hardwarekapazitäten der einzelnen Mitglieder können stark begrenzt sein.

Um diese Herausforderungen zu bewältigen, entstanden dezentralisierte Protokolle, die auf Einfachheit und Skalierbarkeit setzen. Gossiping ist ein solches dezentrales Verfahren, welches eine effiziente Ausbreitung oder Aggregation von Daten ermöglicht [61]. Anwendungsbeispiele beinhalten das Finden von netzwerkweiten Minima oder Maxima sowie das Bilden einer Summe oder eines durchschnittlichen Werts. Das Vorgehen ist dabei der Verbreitung von Klatsch und Tratsch im gesellschaftlichen Raum nachempfunden. Jeder Teilnehmer sendet periodisch Informationen an zufällig ausgewählte Gossip-Partner. Hierbei erfolgt ein paarweiser Informationsaustausch, wodurch Daten asynchron und epidemisch im Netzwerk verbreitet werden [39]. Gossiping-Protokolle garantieren keine vollständige Informationsverbreitung, sondern bieten stattdessen Wahrscheinlichkeiten für bestimmte Ergebnisse. Man spricht hierbei von *probabilistischen Garantien*. Ein Beispiel für eine solche Garantie ist, dass nach X Runden mindestens 90% der Teilnehmer im Netzwerk eine bestimmte Information erhalten haben.

Die ersten Gossip-Algorithmen waren hinsichtlich ihrer Skalierbarkeit eingeschränkt. Grund war, dass jeder Knoten vollständige Mitgliedschaftsansichten speicherte. Viele moderne Umsetzungen beschränken deshalb die Sicht jedes Netzwerkmitglieds auf die lokalen Nachbarn [3]. Durch diese Begrenzung wird die Skalierbarkeit des Systems erheblich verbessert, da Netzwerkteilnehmer wesentlich weniger Daten im Speicher behalten

müssen. Seitdem wurden viele weitere Verbesserungen entwickelt, und die Forschung im Bereich der Gossip-Algorithmen wird nach wie vor intensiv vorangetrieben. Auch für unterschiedliche Anwendungsszenarien kommen stets neue Optimierungenvorschläge auf. Die Herausforderungen bestehen vor allem darin, die Leistungsfähigkeit für den Einsatz in unstrukturierten Netzwerken zu verbessern. In solchen Netzwerken sind optimale Strategien, die aus der Netzwerktopologie abgeleitet werden können, nicht anwendbar.

1.1 Motivation

Um die Motivation der Thesis nachzuvollziehen, muss ein grobes Verständnis über die Bedeutung von von Community-Strukturen in unstrukturierten Netzwerken vorhanden sein. Unstrukturierte Netzwerke zeichnen sich durch ihre Zufälligkeit und entsprechend fehlende Struktur aus. Trotz der Abwesenheit von Struktur können sie verschiedene topologische Eigenschaften aufweisen. Eine wichtige Eigenschaft ist hierbei die Ausprägung von Community-Strukturen. Communities stellen Gruppen von Netzwerkknoten dar, wobei Knoten innerhalb dieser untereinander stark verbunden sind. Andererseits verfügen Netzwerke mit stark ausgeprägten Community-Strukturen über wenige Verbindungen zwischen Knoten verschiedener Communities. Solche Strukturen können vor allem in sozialen Netzwerken und in Computernetzwerken beobachtet werden. Beispiele beinhalten Local Area Networks (LANs), Data Center Networks [40] und IoT-Netzwerke wie Smart Grids, Smart Cities und Militärsysteme [80]. Auch Autonomous Systems, wie sie bei einem Internet Service Provider (ISP) oder einem Unternehmen mit globaler Internet-Infrastruktur zum Einsatz kommen, weisen häufig Community-Strukturen auf [10].

Das Ziel dieser Thesis ist es, Gossip-Algorithmen für unstrukturierte Netzwerke zu optimieren. Dies soll mithilfe topologischer Eigenschaften erreicht werden, wobei der Schwerpunkt besonders auf den Community-Strukturen liegt. Gegenstand der Optimierung ist hierbei die Auswahl des nächsten Kommunikationspartners. Bei herkömmlichen Gossip-Algorithmen erfolgt diese rein zufällig. Eine Optimierungsmöglichkeit besteht, wenn man hier eine gezieltere Auswahl durchführt. Dazu werden die Informationen, die jeder Knoten über seine direkten Nachbarn, um Zugehörigkeiten zu Communities erweitert. Es folgt die Möglichkeit, eine gewichtete Auswahl des Gossip-Partners durchzuführen, wodurch Informationen schneller innerhalb oder außerhalb der Communities propagiert werden können. Infolgedessen kann eine schnellere Konvergenz erreicht werden. Dies wiederum führt zu einer Beschleunigung der Laufzeit der Algorithmen und einer besseren Skalierbarkeit. Ultimativ wird so ein effizienterer Einsatz in großen Netzwerken möglich. Zeitkritische Systeme können ebenfalls von einer gesteigerten Leistung profitieren, da diese von einer Verbesserung der Datenkonsistenz profitieren. Durch eine Beschleunigung der Konvergenz können Knoten frühzeitig auf Systeminformationen zugreifen, was Verzögerungen minimiert und die Datenverfügbarkeit erhöht.

1.2 Aufbau

Die Arbeit erläutert die wesentlichen Schritte, die zur Konzeption und Implementierung einer Simulationsumgebung durchgeführt wurden. Ebenso wird der Aufbau, die Durchführung und die Evaluation von Testreihen zu neu entwickelten Gossip-Algorithmen beschrieben. Dabei werden die einzelnen Projektphasen erläutert.

In Kapitel zwei werden erstmals wichtige technische Grundlagen erläutert, die zum Verständnis der Thesis notwendig sind. Hier werden Grundlagen zu Netzwerken erklärt, wobei insbesondere auf verschiedenen Arten eingegangen wird. In diesem Kontext erfolgt

auch die Definition von Community-Strukturen sowie von Algorithmen zur Erzeugung und Analyse von Netzwerkgraphen. Ebenfalls wird das erforderliche Hintergrundwissen zu Gossiping-Verfahren vermittelt und die Architektur sowie Funktionsweise von Kubernetes beschrieben. Zuletzt werden dann die verwendeten Technologien zur Datenspeicherung und Kommunikation sowie Bibliotheken zur Implementierung behandelt.

Die verwandten Arbeiten werden im dritten Kapitel beschrieben. Hier werden verschiedene Projekte und Veröffentlichungen vorgestellt, welche die Grundlage für diese Thesis legen. Dies Projekte befassen sich mit Grapherzeugung, dezentraler Community Detection und optimierten Gossiping-Verfahren. Es wird erläutert, inwiefern sie bei der Lösungsfindung und Definition von Evaluationsaspekten verwendet werden konnten. Zudem erfolgt eine Abgrenzung dieser Thesis von den verwandten Arbeiten.

Das vierte Kapitel befasst sich mit der Zielsetzung und der Analyse der Anforderungen. Dabei wird die Vision des Projekts formuliert. Anschließend werden alle Anforderungen an die Simulationsumgebung ermittelt, kategorisiert und priorisiert. Daraufhin erfolgt die Vorstellung des Lösungsansatzes, welcher in den folgenden Kapiteln ausgearbeitet wird.

Im fünften Kapitel erfolgt die Konzeption der Lösung. Zuerst wird das *Top-Level-Design* erläutert, welches sich mit der Abbildung des Systemaufbaus beschäftigt. Es beschreibt die Architektur der Simulationsumgebung und legt die Interaktion der einzelnen Komponenten sowie deren Funktionsumfang fest. Im nächsten Schritt wird das *Low-Level-Design* beschrieben, wobei die Komponenten im Detail dargestellt werden. Dabei werden sowohl Aufbau als auch Funktionalität der Komponententen festgehalten.

In Kapitel sechs wird die Implementierung vorgestellt und die Umsetzung der Anforderungen dargelegt. Zuerst wird die prototypische Entwicklung der Simulationsumgebung präsentiert. Des Weiteren erfolgt die Beschreibung des Deployments des Kubernetes-Clusters sowie der Bereitstellung der Anwendungen. Außerdem werden Implementierungsdetails zu den Applikationen erläutert. Dies beinhaltet eine Beschreibung der Umsetzung des Tools zur Grapherzeugung, des Kubernetes Operators und der Container Applikationen.

Das siebte Kapitel befasst sich mit der Evaluation der Simulationsreihen. Es werden die durchgeführten Testreihen und die dabei gewonnenen Ergebnisse präsentiert. Anhand der Metriken und Daten werden Erkenntnisse über die Performance der Gossip-Algorithmen gezogen.

Abschließend erfolgt im achten Kapitel eine Rekapitulation der wichtigsten Punkte. In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst und bewertet. Ein abschließendes Fazit wird gezogen und ein Ausblick auf mögliche Erweiterungen und zukünftige Forschungsarbeiten gegeben.

1.3 Projektplanung

Für die Projektplanung wurden Arbeitspakete erstellt und in einem Projektstrukturplan organisiert. Die übergeordneten Projektphasen umfassten unter anderem die Planung, die Analyse und den Entwurf, die Implementierung sowie die Bereitstellung der Simulationsumgebung. Ebenso Teil des Projekts war die Konzeption, Durchführung und Analyse von Simulationsreihen verschiedener Netzwerktopologien und Algorithmen. Die einzelnen Arbeitspakete wurden den entsprechenden Projektphasen zugeordnet. Mit Hilfe der Software *GanttProject* [46] wurde aus dem Projektstrukturplan ein Gantt-Diagramm erstellt. Dieses Diagramm ermöglichte es, die Arbeitspakete zeitlich zu ordnen, Abhängigkeiten zu erkennen und Termine für Meilensteine festzulegen.

2 Technische Grundlagen

Es folgt die Erklärung der technischen Grundlagen, wo verschiedene Definitionen, Prinzipien und Konzepte erläutert werden. Das Kapitel beginnt mit einer Einführung in Netzwerke. Dabei werden unstrukturierte und komplexe Netzwerke sowie Modelle zur Erzeugung dieser betrachtet. Außerdem erfolgt die Definition von Community-Strukturen sowie der Louvain-Methode zur Erkennung dieser. Anschließend wird das erforderliche Hintergrundwissen in Bezug auf die Gossip-Algorithmen vermittelt. Ihre Funktionsweise wird beschrieben, wobei besonderes auf das Konvergenzverhalten eingegangen wird. Es ist eine zentrale Eigenschaft der Algorithmen und bestimmt die Qualität der Informationsverbreitung. Da ein Kubernetes-Cluster für die Simulation der Gossiping-Verfahren verwendet werden soll, stellen Containerisierung und Orchestrierung ebenso einen Schwerpunkt dar. Die Simulationen sollen durch einen Operator initialisiert werden. Zum Verständnis des Operator Patterns in Kubernetes ist Wissen über die Cluster-Architektur notwendig. Dementsprechend erfolgt die Beschreibung der verschiedenen Komponententen sowie Ressourcen. Des Weiteren werden Technologien wie der MinIO Object Storage zur Datenspeicherung sowie Kommunikationsprotokolle wie TCP und gRPC erklärt. Schließlich wird noch auf die wichtigsten Python-Bibliotheken eigegangen, die zur Implementierung verwendet wurden.

2.1 Netzwerke

Als Netzwerk ist eine Struktur zu verstehen, die aus einer Menge von miteinander verbundenen Elementen besteht. Ein zentraler Aspekt ist dabei meist der Informationsaustausch. Netzwerke können in vielen Bereichen des täglichen Lebens gefunden werden. Beispiele beinhalten soziale Netzwerke, biologische und elektronische Systeme sowie Transportnetzwerke [62]. Computernetzwerke sind von besonderer Bedeutung, da sie die Grundlage für die moderne Kommunikation darstellen. Hierbei sind sowohl *Local Area Network (LAN)* als auch *Wide Area Network (WAN)* passende Beispiele [21]. Viele dieser realen Netzwerke verfügen über eine sehr hohe Komplexität. Sie bestehen aus mehreren Tausend bis Millionen Knoten und Kanten und verfügen über unregelmäßige und schwer überschaubare Strukturen. Zudem unterliegt ihr Aufbau im Laufe der Zeit dynamischen Veränderungen [6]. Neue Knoten kommen hinzu und bestehende fallen weg, wodurch die Komplexität zusätzlich steigt. Je größer und komplexer die Netzwerke werden, desto schwieriger wird ihre Betrachtung.

Zur Veranschaulichung und Analyse werden Netzwerke häufig als Graphen modelliert. Hierbei werden die Elemente des Netzwerks als Knoten und deren Verbindungen als Kanten abstrakt dargestellt. Je nachdem, ob die Kanten eine Richtung aufweisen oder nicht, erfolgt die Unterscheidung zwischen gerichteten und ungerichteten Graphen. Ein Graph wird als gerichtet bezeichnet, wenn alle Kanten einen Start- und Endknoten haben. Andererseits verfügen ungerichtete Graphen über Kanten, die keine Richtung haben. Hier sind beide verbundenen Knoten sowohl Start als auch Ende. In dieser Arbeit wird das Gossiping durch einen paarweisen Informationsaustausch abgebildet. Zur Durchführung eines solchen Austauschs muss zwischen zwei Knoten immer eine bidirektionale Verbindung bestehen. Dementsprechend stellt jede Kante immer eine beidseitige Verbindung

dar, weshalb alle Netzwerke als ungerichtete Graphen modelliert werden können. Außerdem können mit Hilfe der Graphentheorie Strukturen, Beziehungen und Muster effizient untersucht werden. Anwendungsbeispiele umfassen die Analyse von Netzwerktopologien, Routing-Algorithmen, Verkehrsflüssen und die Erkennung von Anomalien. Auch die Analyse von Gossip-Algorithmen kann so durchgeführt werden.

Im Folgenden müssen verschiedene Arten von Netzwerken betrachtet werden. Auf der obersten Ebene unterscheidet man zwischen strukturierten und unstrukturierten Netzwerken. Hierbei haben die strukturierten Netze eine fest definierte Topologie, wobei Verbindungen zwischen den Knoten ein klares Muster aufweisen. Bekannte Topologien umfassen die Bus-, Linien- und Sterntopologie. In der Praxis weit verbreitet sind ebenso folgende Topologien [21]:

Hierarchische Netzwerke:
Diese Netzwerke haben eine hierarchische Struktur mit mehreren Ebenen. Jeder Knoten in einer höheren Ebene ist mit mehreren Knoten in der niedrigeren Ebene verbunden. Dementsprechend ist die Topologie meist baumähnlich, weshalb man auch alternativ von Baum-Netzwerken spricht. Sie werden häufig in Organisationsstrukturen, Dateisystemen und verteilten Rechnersystemen verwendet.

Ring-Netzwerke:
In einem Ringnetz sind die Knoten in einer geschlossenen Schleife miteinander verbunden. Dabei werden Daten von Knoten zu Knoten im Kreis weitergeben, bis sie das vorgesehene Ziel erreichen. Die Vewendung von Ringnetzen erfolgt häufig in Token-Rings und bei bestimmten Arten von LANs.

Vollvermaschte Netzwerke:
Ist jeder Knoten mit jedem anderen Knoten im Netz verbunden, so spricht man auch von vollvermaschten Netzwerken. Diese Netzwerke bieten Redundanz und Fehlertoleranz und eignen sich daher für Anwendungen, die eine hohe Zuverlässigkeit erfordern. Anwendungsbeispiele sind drahtlose Sensornetzwerke und Peer-to-Peer-Systeme.

Vermaschte Netzwerke, die nicht vollverbunden sind, verfügen über Knoten, die nur mit einer Teilmenge anderer Knoten verbunden sind. Liegt keine klare Struktur der Verbindungen vor, werden sie als unstrukturiert bezeichnet. In Abbildung 2.1 ist außerdem eine grafische Darstellung der zuvor besprochenen Netzwerktopologien zu finden. Die Klasse der unstrukturierten Netzwerke wird im Folgenden genauer betrachtet.

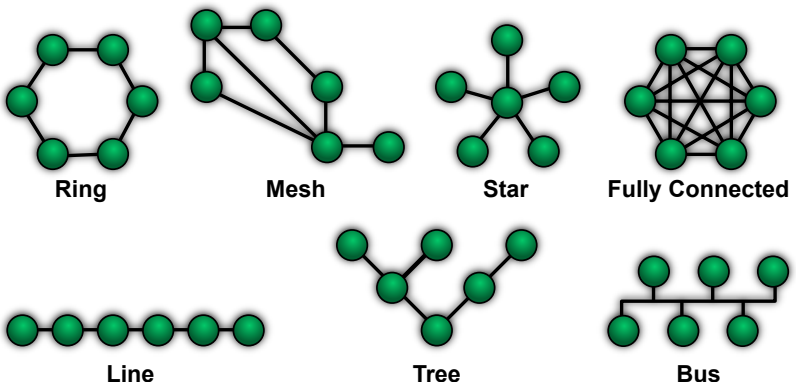


Abbildung 2.1: Netzwerktopologien [73]

2.1.1 Unstrukturierte und komplexe Netzwerke

In strukturierten Netzwerken sind die Verbindungen zwischen den Knoten nach klaren Regeln gebildet. Im Gegensatz dazu gibt es in unstrukturierten Netzwerken keine vordefinierten Beziehungen. Sie zeichnen sich durch ihre fehlende Ordnung und Regelmäßigkeit aus. Verbindungen zwischen den Knoten werden oft auf zufällige Weise hergestellt.

Komplexe Netzwerke werden ähnlich wie unstrukturierten Netzwerke ohne vorab geplante Struktur aufgebaut. Dennoch weisen hier Knotenverbindungen meist nicht-triviale Muster auf. Solche Strukturen können beim Aufbau des Netzwerks durch die Verhaltensweisen beim Hinzufügen von Knoten entstehen. Beispiele für komplexe Netzwerke sind soziale und biologische Netzwerke sowie das World Wide Web [74]. Hier kann die Entstehung von Verbindungen durch verschiedene Faktoren beeinflusst werden. Nimmt man soziale Netzwerke als Beispiel, so entstehen Verbindungen durch die Interaktionen der Teilnehmer. Dasselbe kann auf Computernetzwerke übertragen werden, wo keine zentrale Routing-Infrastruktur vorhanden ist. Als Folge des planlosen Aufbaus entsteht eine Abwesenheit von vorhersehbaren Pfaden [86]. Eine weitere Eigenschaft ist die häufig beobachtete hohe Variabilität der Knotengrade. Man spricht hierbei auch von einer *skalenfreien Gradverteilung*. Bei einer solchen Gradverteilung sind einzelne Knoten stark verbunden und haben dementsprechend viele Nachbarn, während andere Knoten hingegen nur über sehr wenige Verbindungen verfügen. Weitere Beispiele für Muster sind die *Small-World-Eigenschaften* und *Community-Strukturen* [41]. Letztere werden detailliert im folgenden Kapitel erläutert.

Viele Systeme weisen Eigenschaften unstrukturierter beziehungsweise komplexer Netzwerke auf [74]. Aufgrund der undefinierten Struktur sind diese jedoch erst nach tiefgehenden Analysen erkennbar. Nur so können Erkenntnisse über vorhandene Struktureigenschaften und -muster gewonnen werden. Modelle zur Analyse solcher Netzwerke werden in verschiedenen Bereichen angewandt, um die Effektivität der Netzwerke und ihrer Anwendungen zu verbessern. Zum Beispiel können bei Kommunikationsnetzwerken zentrale Knoten als Bottlenecks, Sicherheitsrisiken oder Fehlerquellen identifiziert werden [86]. Infolgedessen können entsprechende Strategien zur Lösung entwickelt werden. Durch die gesammelten Ergebnisse kann dann eine Optimierung von Faktoren, wie Verbindungseigenschaften, Routing und Skalierbarkeit erfolgen.

2.1.2 Community-Strukturen

In Netzwerken werden Communities auch als Gruppen oder Cluster bezeichnet. Community-Strukturen beschreiben Gruppierungen von Knoten in einem Netzwerk, bei denen die Knoten innerhalb der Gruppen eng miteinander verbunden sind, während die Verbindungen zwischen Knoten verschiedener Gruppen weniger stark ausgeprägt sind [94]. Abbildung 2.2 stellt gefundene Cluster in einem Graphen von Co-Autoren von Netzwerkwissenschaftsartikeln dar. Hier sind die Communities durch zusammengehörige Anordnung sowie farbliche Darstellung zu erkennen.

Die Identifizierung von Community-Strukturen ist wichtig, um Netzwerke analysieren zu können und deren Dynamik nachzuvollziehen. Die Anwendungsmöglichkeiten differenzieren sich hierbei, je nachdem, ob es sich um soziale, biologische Netzwerke oder das Internet handelt. Meist haben Knoten innerhalb der Gruppen gleiche Eigenschaften, Funktionen oder Rollen. Als Beispiel können soziale Communities gesehen werden, wo Familien, Arbeitskollegen, Vereine oder sonstige Interessensgruppen Formen solcher Gruppierungen darstellen. Eine Clusterbildung kann so beispielsweise von Verkäufern verwendet werden, um Kundengruppen zu erstellen und so passende Produktvorschläge zu generieren. Für Computer-Netzwerke erschließen sich ebenso viele funktionale Vortei-

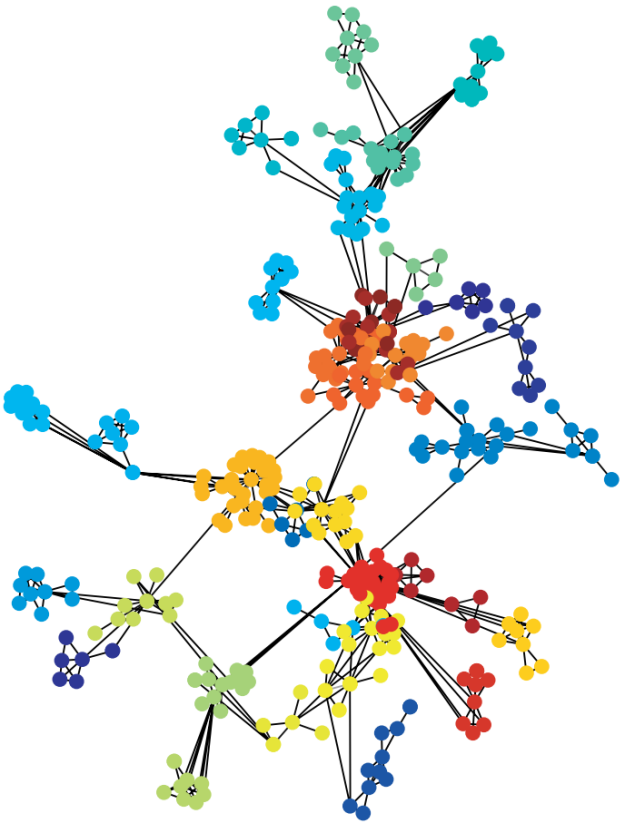


Abbildung 2.2: Clusterbildung im Co-Autoren-Netzwerk von Netzwerkwissenschaftlern [94]

le. So können beispielsweise Konfigurationen einfacher durchgeführt oder Nachrichten durch Routing-Table-Optimierung besser verteilt werden [42].

Die Untersuchung von Community-Strukturen hat zur Entwicklung verschiedener Algorithmen geführt. Diese verwenden verschiedene Ansätze wie Modularitätsoptimierung, hierarchisches Clustering oder stochastische Modelle, um Communities in Netzwerken aufzudecken. Die Modularität eines Netzwerks ist hierbei eine quantitative Metrik, um die Existenz von Communities zu bewerten. Sie ist gemäß [22] wie folgt definiert:

$$Q = \sum_{c=1}^{n} \left[\frac{L_c}{m} - \gamma \left(\frac{k_c}{2m} \right)^2 \right] \tag{2.1}$$

Hierbei ist n die Anzahl an Communities, m die Anzahl an Kanten, L_c die Anzahl an Kanten innerhalb der Community c und k_c ist die Summe der Knotengrade der Community c. Der Auflösungsparameter γ legt das Verhältnis zwischen gruppeninternen und gruppenübergreifenden Kanten fest. Dadurch kann über ihn definiert werden, ob größere oder kleinere Communities bevorzugt werden [85].

Der Modularitätswert von Netzwerken liegt im Bereich von null bis eins. Ein Wert nahe eins sagt aus, dass das Netzwerk hochmodular ist. Das heißt, es gibt viele Verbindungen innerhalb der Communities und wenige Verbindungen zwischen den Communities. Je näher die Modularität gegen null geht, desto schwächer sind die Community-Strukturen im Netzwerk ausgeprägt. Bei einem Wert von null ist die Verteilung der Communities analog einer zufälligen Distribution.

2.1.3 Louvain-Methode

Die Louvain-Methode ist ein populärer Algorithmus zur Identifizierung von Communities. Das Verfahren wurde 2008 von einer Gruppe von Wissenschaftlern entwickelt mit dem Ziel Cluster in komplexen Netzwerken zu erkennen. Zuvor geläufige Algorithmen wie zum Beispiel der Girvan-Newman-Algorithmus [48] waren schlecht skalierbar. Die Motivation hinter der Entwicklung der Louvain-Methode geht auf diese Einschränkung zurück. Die erstmalige Veröffentlichung erfolgte im Paper „Fast unfolding of communities in large networks“ [14]. Hier konnten die Entwickler nachweislich zeigen, dass der neue Algorithmus auch bei sehr komplexen Netzwerken eine hervorragende Laufzeitleistung aufweist.

Die Louvain-Methode verwendet den Ansatz der Modularitätsoptimierung, um Communities in einem Netzwerk zu finden. Anders ausgedrückt, Community-Strukturen werden so ausgewählt, dass der maximale Wert der Modularitätsmetrik erreicht wird. Dabei werden die folgenden zwei Hauptphasen durchgeführt:

(A) *Aggregationsphase:* Zu Beginn wird jeder Knoten im Netzwerk einer eigenen Community zugeordnet. Anschließend erfolgt die Auswahl eines Ausgangsknoten, dessen Nachbarn untersucht werden. Dabei wird die Änderung der Modularität bei Beitritt des Knotens zur Community des jeweiligen Nachbarn bewertet. Der Knoten wird letztendlich in die Community verschoben, bei der die höchste Steigerung der Modularität erzielt wird. Falls die Modularität in keinem Fall erhöht werden kann, bleibt der Knoten in seiner aktuellen Community. Wurde dieser Vorgang für alle Knoten im Netzwerk durchgeführt, so wird mit der nächsten Phase fortgefahren.

(B) *Reduktionsphase:* In der Reduktionsphase werden die Communities, die durch die Aggregationsphase ermittelt wurden, als Knoten behandelt. So erfolgt auch eine Aggregation der Kanten zwischen den ursprünglichen Knoten, wobei gewichtete Kanten entsprechend addiert werden. Nach jeder Reduktionsphase wird eine weitere Aggregationsphase auf dem reduzierten Graphen durchgeführt. Dieser Vorgang wird solange wiederholt bis auch hier keine Erhöhung der Modularität mehr erreicht werden kann.

Durch die iterative Optimierung der Modularität kann die Community-Struktur effizient identifiziert werden. Dazu maximiert das Modell die Verbindungen innerhalb der Communities und minimiert Verbindungen zwischen Communities. In Abbildung 2.3 ist eine grafische Darstellung der Schritte der Louvain-Methode dargestellt. Hierbei kann die initiale Zuweisung sowie die Reihenfolge der Verarbeitung variieren. Vor allem wenn mehrere gleiche Netzwerkpartitionen mit hohen Modularitätswerten im Graphen vorliegen, führen wiederholte Ausführungen zu jeweils verschiedenen Ergebnissen. Die Louvain-Methode ist ein sogenanneter *heuristischer Algorithmus*, was bedeutet, dass sie eine lediglich eine Annäherung an die optimale Community-Struktur vornimmt. Dementsprechend führt das Verfahren nicht zwingend zur optimalen Lösung.

2.1.4 Modelle zur Erzeugung zufälliger Netzwerke

Über die Jahre wurden viele Modelle und Algorithmen entwickelt, um zufällige Netzwerke zu erzeugen. Diese haben zum Verständnis vom Aufbau und der Entstehung komplexer Netzwerke beigetragen. Im Folgenden werden die populärsten Modelle besprochen.

Das erste Modell zur Zufallsgraphgenerierung wurde von den Mathematikern Paul Erdős und Alfréd Rényi in den 1950er Jahren aufgestellt [38]. Es ist benannt nach den Erfindern und trägt den Namen *Erdős–Rényi-Modell*. Das Modell basiert auf einem simplen

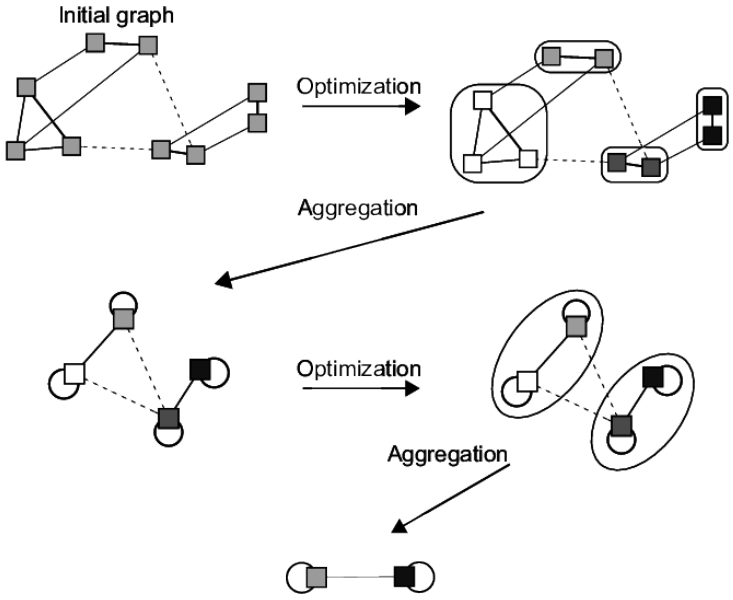


Abbildung 2.3: Schritte der Louvain-Methode auf einem synthetischen Graphen [20]

Vorgehen, wo Graphen initial mit einer festen Anzahl von Knoten erzeugt werden. Anschließend werden die Kanten nacheinander zufällig hinzugefügt. Das exakte Vorgehen ist dabei abhängig von der jeweiligen Variante des Modells:

- **Binomial-Modell:** Das Binomial-Modell ist auch bekannt als das $G(n, p)$ Modell. Hier erfolgt die Erzeugung eines zufälligen Graphen mit n Knoten. Dabei wird jede mögliche Kante zwischen zwei beliebigen Knoten betrachtet. Die Wahrscheinlichkeit p wird verwendet, um zu entscheiden, ob die entsprechende Kante hinzugefügt wird. Der resultierende Graph hat einen durchschnittlichen Knotengrad in Abhängigkeit von p [47].
- **Poisson-Modell:** Das Poisson-Modell ist auch bekannt als das $G(n, m)$ Modell. Hier wird ein Graph mit n Knoten generiert, wobei m die Anzahl an gewünschten Kanten definiert. Anschließend werden alle Kanten gleichmäßig und unabhängig voneinander nach dem Zufallsprinzip hinzugefügt. Der resultierende Graph hat einen variablen Knotengrad [38].

Ein weiteres bekanntes Zufallsgraphmodell ist das *Watts-Strogatz-Modell*. Es wurde 1998 von Duncan J. Watts und Steven H. Strogatz entwickelt [117]. Das Modell versucht das in vielen realen Netzwerken beobachtete *Small-World-Phänomen* abzubilden. Dieses Phänomen findet seinen Ursprung bei einem Experiment von Stanley Milgram aus dem Jahr 1967 [76]. Milgram versucht die Hypothese nachzuweisen, dass jeder Mensch mit jedem anderen auf der Welt über eine kurze Kette von Bekanntschaften verbunden ist. Um dies zu untersuchen, führte er ein Experiment durch, bei dem sechzig zufällig ausgewählte Teilnehmer jeweils ein Paket erhielten. Ihre Aufgabe war es ihr Paket an eine vorab festgelegte Person zu übermitteln. Dabei durften sie das Paket nicht direkt an die Zielperson senden, es sei denn, sie kannten sie persönlich. War ihnen der Empfänger unbekannt, so sollten sie das Paket an eine ihnen bekannte Person senden. Dabei sollte die Person ausgewählt werden, die am wahrscheinlichsten die Zielperson kennt. Die Ergebnisse zeigten, dass die Pakete die Zielpersonen über nur aufgerundet sechs Personen

erreichen konnten. Aus diesem Grund enstand auch die alternative Bezeichnung für das Phänomen, die als *six degrees of separation* bekannt ist [124].

Das Small-World-Phänomen kann auch auf Netzwerke übertragen werden. Das Erdős-Rényi-Modell kann dies jedoch nicht ausreichend modellieren, weshalb verschiedene Weiterentwicklungen durchgeführt wurden. Dem Watts-Strogatz-Modell gelang erstmals eine Erzeugung von Netzwerken, welche die gleichen Eigenschaften aufweisen wie die Small-World-Netzwerke. Hierbei müssen insbesondere zwei Schlüsseleigenschaften erfüllt werden, die auch in realen Netzwerken zu beobachten sind. Netzwerke werden mit einer *starken lokalen Clusterbildung* und *kurzen durchschnittlichen Pfadlängen* erzeugt. Infolgedessen bilden Knoten, die dicht miteinander verbunden sind, Communities. Darüber hinaus wird ein hohes Maß an Konnektivität erzielt, da nur wenige Schritte erforderlich sind, um jeden Knoten von jedem anderen aus zu erreichen. Die Schlüsseleigenschaften werden durch zwei separate Vorgänge bei der Grapherzeugung realisiert. Es folgt eine schrittweise Erklärung des Vorgehens [117]:

1. Es wird mit einer Menge von Knoten, die in einem Ring angeordnet sind, begonnen. Hierbei ist jeder Knoten n mit seinen k nächsten Nachbarn auf beiden Seiten verbunden. Die Ausgangs-Topologie hat eine hohe Konnektivität, jedoch lange Pfade.

2. Anschließend werden die Kanten neu verbunden. Jede Kante des Netzwerks wird hierbei mit einer Wahrscheinlichkeit p modifiziert. Bei einer Modifikation erfolgt die zufällige Auswahl eines neuen Zielknotens.

In Abhängigkeit von p werden mehr oder weniger Kanten neu ausgerichtet. Mit steigender Zufälligkeit des Netzwerks verkürzen sich die Pfade, während gleichzeitig die Clusterbildung abnimmt. Das Modell zeigt, dass bereits eine geringe Menge an Zufälligkeit (Werte von p gegen 0.01) die durchschnittliche Weglänge stark verkürzen kann. In diesem Fall bleiben auch die lokalen Community-Strukturen größtenteils erhalten. Abbildung 2.4 zeigt, wie die Erhöhung der Zufälligkeit die Grapherzeugung beeinflusst.

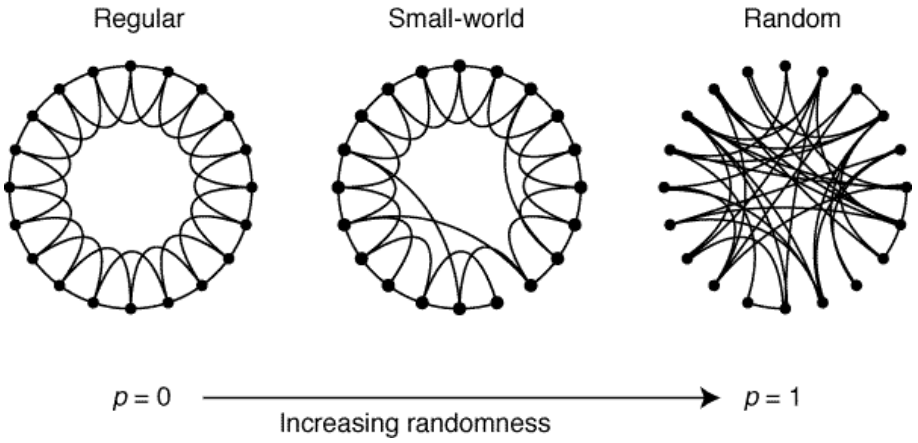


Abbildung 2.4: Watts-Strogatz-Modell für $n = 20$ und $k = 4$ [117]

Das *Barabási-Albert-Modell* verbessert die älteren Modelle von Erdős-Rényi und Watts-Strogatz. Bei dem Erdős-Rényi-Modell steht die Entfernung von Knoten im Mittelpunkt. Das Watts-Strogatz-Modell wiederum ist auf die Small-World-Eigenschaften von Netzwerken konzentriert. Beide Modelle haben die Gemeinsamkeit, dass Netzwerke mit einer festen Anzahl von Knoten erzeugt werden. Diese Knoten werden dann entweder zufällig verbunden (Erdős-Rényi) oder es erfolgt eine Neuausrichtugn bestehender Verbindungen (Watts-Strogatz). Das Barabási-Albert-Modell versucht stattdessen die Dynamik von

realen Netzwerken abzubilden. Dies wird gewährleistet, indem neue Knoten wiederholt eingefügt werden. Das Modell wurde im Jahr 1999 entwickelt und stellte einen wichtigen Meilenstein zur Modellierung realer Netzwerke dar. Eine genaue Betrachtung erfolgt im folgenden Abschnitt.

2.1.5 Skalenfreie Netzwerke

Skalenfreie Netzwerke zeichnen sich durch die *Potenzgesetzverteilung* der Knotengrade aus. Bei einer solchen Verteilung gibt es nur wenige Knoten, die eine sehr hohe Anzahl von Verbindungen haben, während die Mehrheit der Knoten über nur eine geringe Anzahl von Verbindungen verfügt. Die stark verbundenen Knoten werden auch als *Netzwerk-Hubs* bezeichnet. Die Potenzgesetzverteilung unterscheidet sich somit von einer normalen oder Gaußschen Verteilung, wo die meisten Werte um einen Durchschnitt herum gruppiert sind. Es existieren viele Arbeiten, welche Netze in Bezug auf die Potenzgesetzverteilung untersuchen. Dabei wird meist der Grad der Ausprägung anhand des Exponenten untersucht. Diese Arbeiten zeigen, dass die skalenfreien Eigenschaften auch bei verschiedenen realen Netzwerken beobachtet werden kann. Im Paper [115] wurden die Erkenntnisse vorangehender Untersuchungen gesammelt. Diese Ergebnisse sind in Tabelle 2.1 kompakt zusammengefasst. Hieraus ist abzuleiten, dass für viele reale Netzwerke der Exponent γ der Potenzgesetzverteilung zwischen zwei und drei liegt.

Tabelle 2.1: Potenzgesetzverteilungen realer Netzwerke

Netzwerk	**Größe**	∅ **Pfadlänge**	γ
Internet (Domains)	32711	3,56	2,1
Internet (Router)	228298	9,51	2,1
World Wide Web	153127	3,1	$\gamma_{in} = 2{,}1$ $\gamma_{out} = 2{,}45$
E-Mail	56969	4,95	1,81
Software	1376	6,39	2,5
Elektrische Schaltungen	329	3,17	2,5
Math. Co-Autoren	70975	9,5	2,5
Food web	154	3,4	1,13
Stoffwechselsystem	778	3,2	$\gamma_{in} = \gamma_{out} = 2{,}2$
Sprache	460902	2,67	2,7

Für die Untersuchung von Netzwerken ist die Erzeugung synethischen Graphen mit ähnlichen Eigenschaften maßgeblich. Das Barabási-Albert-Modell erzeugt Zufallsgraphen mit einer Potenzgesetzverteilung mit Exponent $\gamma = 3$. Es werden somit Netzwerke generiert, die realitätsnahe skalenfreie Netze approximieren. Das Verfahren wird im folgenden Kapitel detailliert beschrieben. Neben dem Barabási-Albert-Modell können verschiedene weitere Verfahren eingesetzt werden, um skalenfreie Netzwerke zu erzeugen. Viele dieser Modelle setzen dabei auf das zufällige Hinzufügen von Knoten und deren Verbindung.

Im Folgenden werden die Eigenschaften von skalenfreien Netzwerken beschrieben. Neben der Knotengradverteilung entsprechend des Potenzgesetzes gibt es weitere bedeutende Charakteristiken [72]. Zum einen weisen skalenfreie Netze ein hohes Clustering auf. Das heißt, es liegen Communities vor, die eng miteinander verbundene Knoten beinhalten. Des Weiteren sind die durchschnittlichen Pfadlängen im Vergleich zur Netzwerkgröße sehr kurz. Aus diesen Eigenschaften folgt eine hohe Redundanz und dementsprechend

große Robustheit gegenüber Fehlern beziehungsweise Ausfällen. Das Entfernen eines zufälligen Knotens hat wenig Einfluss auf die Integrität des Netzes. Im Gegensatz dazu kann die gezielte Entfernung von Hubs einen großen Einfluss auf die Netzwerkstruktur haben. In skalenfreien Netzwerken verbreitet sich Information von Natur aus epidemisch. Dies ist auf die Hubs zurückzuführen, die aufgrund ihrer zahlreichen Verbindungen eine entscheidende Rolle in der Informationsverbreitung spielen. Bei Computernetzwerken, wo eine schnelle Datenverteilung bevorzugt wird, kann dies als positiv betrachtet werden. Allerdings birgt eine epidemische Verbreitung in anderen Netzwerken ein erhebliches Risikopotenzial. Verfügen soziale Netzwerke über skalenfreie Eigenschaften kann dies beispielsweise die Übertragung von Krankheiten beschleunigen. Neben der Informationsausbreitung werden weitere dynamische Prozesse durch die einzigartige Struktur beeinflusst. Dazu zählen Vorgänge wie die Synchronisation sowie die Bestimmung von Verbindungseigenschaften bei der Entfernung von Knoten.

2.1.6 Barabási-Albert-Modell

Das Paper [9] von Albert-László Barabási und Réka Albert untersucht Skalierungseigenschaften in Netzwerken. Reale Netzwerke wie zum Beispiel soziale und biologische Netzwerke sowie das World Wide Web weisen ähnliche Konnektivitätsmuster auf. Es wird erstmals das *Preferential Attachment* Modell eingeführt. Hierbei werden kontinuierlich neue Knoten hinzugefügt, wobei Verbindungen nach dem *rich-get-richer* Prinzip erfolgen. Das heißt, wird ein neuer Knoten n hinzugefügt, so werden Verbindungen zu bereits stark verbundenen Knoten bevorzugt. Die Wahrscheinlichkeit, dass eine Kante zu einem bestehenden Knoten i erzeugt wird, ist proportional zur Summe seiner vorhandenen Kanten. Diese Summe entspricht den aufsummierten Knotengraden seiner Nachbarn, also $\sum_j k_j$. Es folgt die Formel für die Wahrscheinlichkeit, hierbei ist k_i der Grad des Knotens i:

$$p_i = \frac{k_i}{\sum_j k_j} \tag{2.2}$$

Als Folge der bevorzugten Bindung entstehen im Graphen wenige Netzwerk-Hubs und viele Knoten mit vergleichsweise wenigen Verbindungen. Dementsprechend folgen die Knotengrade einer *Potenzgesetzverteilung*. Dabei kann die Wahrscheinlichkeit $P(k)$, dass ein Knoten mit k weiteren Knoten verbunden ist, bestimmt werden als:

$$P(k) \sim k^{-\gamma} \tag{2.3}$$

Der Exponent γ der Potenzgesetzverteilung wird auch als *Skalierungsexponent* bezeichnet. Für das Barabasi-Albert-Modell kann der Wert von γ genau berechnet werden als:

$$P(k) = \frac{2m^2}{k^3} \tag{2.4}$$

Hierbei ist m die Anzahl an Kanten, die für jeden neuen Knoten n hinzugefügt werden, also der initiale Grad des neuen Knotens. Daraus folgt, dass immer $\gamma = 3$ ist, unabhängig vom Wert von m. Beispiele für skalenfreie Graphen, die nach dem Barabási-Albert-Modell erzeugt wurden, können in Abbildung 2.5 betrachtet werden. Dabei werden die Graphen in Abhängigkeit von folgenden konfigurierbaren Parametern erstellt:

- N - Anzahl an Knoten im Graph
- m - Anzahl an Kanten, die neuen Knoten n hinzugefügt werden

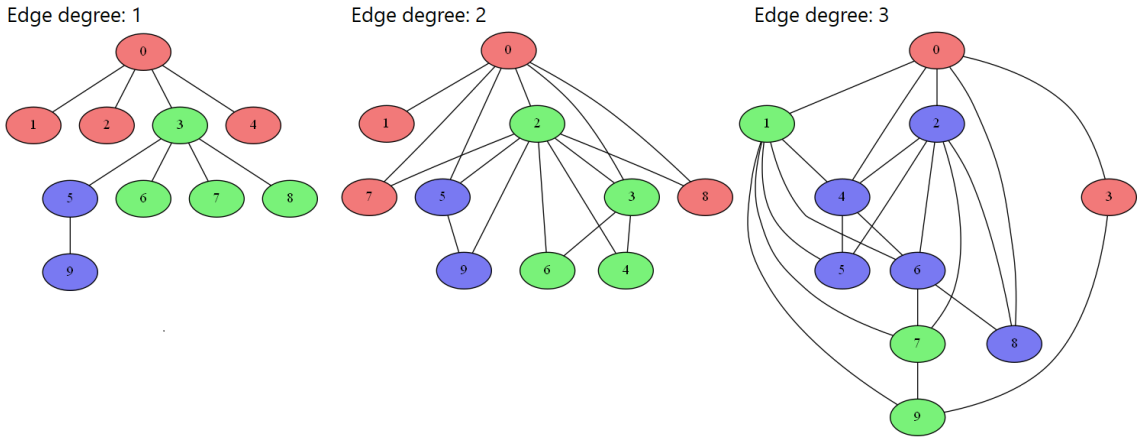


Abbildung 2.5: Skalenfreie Graphen nach Barabási-Albert-Modell

2.1.7 Holme und Kim Grapherzeugung

Der Algorithmus von Holme und Kim [57] baut auf dem Barabási-Albert-Modell auf. Das Barabási-Albert-Modell erzeugt skalenfreie Netzwerke durch Preferential Attachment. Dieses Konzept wird von Holme und Kim durch die zusätzliche Bildung von sogenannten Dreiecken (auch Triaden oder Triangeln) erweitert. Ein Dreieck besteht dabei aus genau drei Knoten, die alle über Kanten miteinander verbunden sind. Das Verfahren beginnt genau wie das Barabási-Albert-Modell mit einem Netzwerk mit einer kleinen Anzahl von Knoten. Anschließend erfolgt eine iterative Vergrößerung, bei der jeder neue Knoten einzeln hinzugefügt und mit m bestehenden Knoten verbunden wird. Im Fall von $m = 2$ werden zwei Knoten, u und v ausgewählt. Falls u und v bereits durch eine Kante verbunden sind, bildet der neue Knoten mit Wahrscheinlichkeit p ein Dreieck mit u und v. Es entsteht eine geschlossene Schleife von Verbindungen. Ist $p = 0$, so ist die Funktionsweise des Algorithmus identisch mit dem Barabási-Albert-Modell. Mit steigenden Werten von p werden mehr Dreiecke gebildet. Die Dreiecksbildung führt zu einer starken Ausprägung von Community-Strukturen. Ebenso wird so die lokale Konnektivität erhöht und die erzeugten Graphen verfügen über eine wesentlich höhere Modularität.

2.1.8 Popularitätsnetzwerke

Popularitätsnetzwerke sind eine Art von Netzwerken, die auf der Beliebtheit von Entitäten basieren. In solchen Netzwerken werden Verbindungen zwischen den Knoten basierend auf ihrer Popularität oder ihrem Einfluss hergestellt. V. Singh erzeugt solche Netze im Rahmen seiner Arbeit zur dezentralen Community-Detection [105]. Zur Generierung wird eine feste Anzahl von Knoten sowie sogenannte Intra-Kanten und Inter-Kanten verwendet. Intra-Kanten beziehen sich auf die Kanten, die zwischen Knoten der gleichen Community bestehen. Währenddessen werden als Inter-Kanten die Kanten bezeichnet, die Knoten zweier verschiedener Communities verbinden. Die Gesamtzahl der Kanten ergibt sich aus der Summe der Inter-Kanten und Intra-Kanten. Hat ein Graph einen hohen Anteil an Intra-Kanten, so sind Community-Strukturen stark ausgeprägt. Mit steigender Anzahl an Inter-Kanten werden Verbindungen zwischen den Knoten verschiedener

Cluster dichter. Dadurch wird es Community Detection Algorithmen erschwert, eine eindeutige Zuordnung von Knoten zu Clustern festzustellen. Die Modularität wird somit durch das Verhältnis der Inter- und Intra-Kanten stark beeinflusst.

2.1.9 Graphmetriken

Im Folgenden werden die für diese Arbeit relevanten Graphmetriken erläutert. Eine umfangreiche Auswahl an Metriken wurde getroffen, um eine breite Flexibilität bei der Evaluation zu gewährleisten. Die Berechnung der Metriken erfolgt nach der Grapherzeugung, wodurch diese dann später mit den jeweiligen Simulationsergebnissen in Kontext gesetzt werden können. So kann eine Begründung unterschiedlicher Leistungen verschiedener Netzwerke durchgeführt werden. Ebenfalls wird es möglich festzustellen, welche Metriken das Konvergenzverhalten am stärksten beeinflussen.

Es folgt die Definition der Metriken in Form einer Formelsammlung. Zudem wird für jede Metrik deren Berechnung sowie Gründe für ihre Verwendung dargelegt. Ein Großteil der Metriken kann mit Hilfe der Python-Bibliothek *NetworkX* berechnet werden. Formeln wurden entsprechend aus dem Code oder dessen Dokumentation [95] entnommen, falls nicht anders angegeben. Dabei wurden zur Generalisierung Änderungen vorgenommen. Ziel war es, eine Gleichheit der verwendeten Symbole sicherzustellen. Für die nachfolgenden Formeln sei definiert:

E = ist die **Kantenmenge**,
V = die **Knotenmenge** und
$\mathcal{C}$ = die **Menge an Communities**

Mittelwerte und Standardabweichungen werden für Metriken berechnet, die für jeden einzelnen Knoten definiert sind. Diese werden berechnet, da bei großen Netzwerken eine Auswertung der einzelnen Knoten sehr zeitaufwendig ist. Die Definition des **Mittelwerts** lautet wie folgt:

$$\overline{x}_N = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.5}$$

Gleichermaßen kann die **Standardabweichung** definiert werden als:

$$\sigma_N = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x}_N)^2} \tag{2.6}$$

Wobei:

N = eine Menge mit n Elementen,
x_i = der Wert eines Elementes ist

Es folgt die Formelsammlung, welche alle Metriken beinhaltet, die im Rahmen der Simulationen dieser Thesis berechnet wurden. Die Evaluation bezieht sich gegebenenfalls nicht direkt auf alle zu Beginn ausgewählten Maße. Für nachfolgende Arbeiten und Analysen stehen die vollständigen Daten jedoch zur Verfügung.

- **Knotenanzahl**
  - **Berechnung:**

$$node\_count = |V| \tag{2.7}$$

  - **Beschreibung:** Die Knotenanzahl im Graphen
  - **Begründung:** Grundlegende Grapheigenschaft
- **Kantenanzahl**
  - **Berechnung:**

$$num\_edges = |E| \tag{2.8}$$

  - **Beschreibung:** Die Kantenanzahl im Graphen
  - **Begründung:** Grundlegende Grapheigenschaft
- **Anzahl Communities**
  - **Berechnung:**

$$num\_communities = |\mathcal{C}| \tag{2.9}$$

  - **Beschreibung:** Die Gesamtanzahl an Communities im Graphen, wobei die Communities der einzelnen Knoten zuvor durch die Louvain-Methode berechnet werden müssen
  - **Begründung:** Grundlegende Eigenschaft modularer Graphen
- **Durchschnittliche Community-Größe**
  - **Berechnung:**

$$avg\_community\_size = \overline{x}_{community\_sizes} \tag{2.10}$$

    mit:

$$community\_sizes = \{|\mathcal{C}_i| \mid \mathcal{C}_i \in \mathcal{C}\} \tag{2.11}$$

  - **Beschreibung:** Durchschnittliche Anzahl an Knoten pro Community
  - **Begründung:** Grundlegende Eigenschaft modularer Graphen
- **Standardabweichung Community-Größe**
  - **Berechnung:**

$$stdev\_community\_size = \sigma_{community\_sizes} \tag{2.12}$$

  - **Beschreibung:** Standardabweichung der Anzahl an Knoten pro Community
  - **Begründung:** Grundlegende Eigenschaft modularer Graphen

- **Gesamtes durchschnittliches Knoten Clustering**
  - **Berechnung:**

$$overall\_avg\_clustering = \overline{x}_{node\_clustering} \tag{2.13}$$

mit:

$$node\_clustering = \{clust(u) \mid u \in V\} \tag{2.14}$$

und:

$$clust(u) = \frac{2T(u)}{deg(u)(deg(u)-1)} \tag{2.15}$$

wobei:

$T(u)$ = die Anzahl von Dreiecken und
$deg(u)$ = der Grad des Knotens u ist

  - **Beschreibung:** Berechnung des Mittelwerts des Clustering-Koeffizienten für alle Knoten
  - **Begründung:** Beschreibt Gruppierung der Knoten in Communities
- **Standardabweichung Knoten Clustering**
  - **Berechnung:**

$$overall\_stdev\_clustering = \sigma_{node\_clustering} \tag{2.16}$$

  - **Beschreibung:** Standardabweichung des Clustering-Koeffizienten für alle Knoten
  - **Begründung:** Rückschlüsse auf Abweichungen, Verteilung der Werte
- **Modularität**
  - **Berechnung:**

$$Q = \sum_{c=1}^{|C|} \left[\frac{L_c}{|E|} - \gamma \Big(\frac{k_c}{2|E|} \Big)^2 \right] \tag{2.17}$$

wobei:

L_c = die Anzahl an Kanten innerhalb der Community$c$,
k_c = die Summe der Knotengrade der Community$c$ und
γ = der Auflösungsparameter ist

  - **Beschreibung:** Quantifiziert die Güte der Community-Strukturen
  - **Begründung:** Lässt direkte Rückschlüsse auf vorliegende Community-Strukturen zu

- **Durchschnittlicher Knotengrad**
  - **Berechnung:**

$$avg\_node\_degree = \overline{x}_{node\_degrees} \tag{2.18}$$

    mit:

$$node\_degrees = \{deg(u) \mid u \in V\} \tag{2.19}$$

  - **Beschreibung:** Berechnung des Mittelwerts der Knotengrade
  - **Begründung:** Grundlegende Grapheigenschaft, beschreibt Konnektivität
- **Durchschnittlicher Knotengrad**
  - **Berechnung:**

$$stdev\_node\_degree = \sigma_{node\_degrees} \tag{2.20}$$

  - **Beschreibung:** Standardabweichung der Knotengrade
  - **Begründung:** Grundlegende Grapheigenschaft, Abweichungen der Konnektivität
- **Annäherung Powerlaw-Exponent**
  - **Berechnung:** Anpassung der Verteilung an die Daten und Schätzung des Skalierungsparameters α [119]
  - **Beschreibung:** Angenäherter Powerlaw-Exponent α nach:

$$P(x) = C \cdot x^{-\alpha} \tag{2.21}$$

    wobei:

    x = die Variable und
    C = die Normalisierungskonstante ist

  - **Begründung:** Stellt skalenfreie Grapheigenschaften dar
- **Niedriger Powerlaw-Bereich**
  - **Berechnung:** Wert x_{min} als geschätzte Untergrenze bei Anpassung der Daten [119]
  - **Beschreibung:** Geschätzte Untergrenze der Powerlaw-Gradverteilung für den Skalierungsparameter
  - **Begründung:** Stellt Abweichungen von skalenfreien Grapheigenschaften dar
- **Durchschnittliche Pfadlänge**
  - **Berechnung:**

$$avg\_path\_length = \frac{1}{|V|(|V|-1)} \sum_{u,v \in V, u \neq v} d(u,v) \tag{2.22}$$

    wobei:

    $d(u,v)$ = die kürzeste Pfadlänge zwischen u und v ist

  - **Beschreibung:** Durchschnittliche Pfadlänge zwischen allen Knotenpaaren
  - **Begründung:** Misst Effizienz des Datenflusses und Erreichbarkeit

- **Durchschnittliche Gradzentralität**
  - **Berechnung:**

$$avg\_degree\_centrality = \overline{x}_{degree\_centralities} \tag{2.23}$$

    mit:

$$degree\_centralities = \left\{ \frac{deg(v)}{|V| - 1} \;\middle|\; v \in V \right\} \tag{2.24}$$

  - **Beschreibung:** Mittelwert des Verhältnisses des Knotengrads zur Gesamtzahl der Knoten
  - **Begründung:** Misst Konnektivität

- **Standardabweichung Gradzentralität**
  - **Berechnung:**

$$stdev\_degree\_centrality = \sigma_{degree\_centralities} \tag{2.25}$$

  - **Beschreibung:** Standardabweichung der Verhältnisse zwischen Knotengrad und Gesamtzahl der Knoten
  - **Begründung:** Abweichungen der Konnektivität

- **Durchschnittliche Betweenness Centrality**
  - **Berechnung:**

$$avg\_betweenness\_centrality = \overline{x}_{betweenness\_centralities} \tag{2.26}$$

    mit:

$$betweenness\_centralities = \left\{ \sum_{s,t \in V} \frac{\mathcal{S}(s,t|v)}{\mathcal{S}(s,t)} \;\middle|\; v \in V \right\} \tag{2.27}$$

    wobei:

$\mathcal{S}(s,t)$ = die Anzahl der kürzesten (s,t)-Pfade und
$\mathcal{S}(s,t|v)$ = die Anzahl dieser Pfade ist, die einen Knoten v ($\neq s$ oder t) traversieren, ist

  - **Beschreibung:** Mittelwert der Summen der Verhältnisse zwischen kürzesten Pfaden, die den Knoten v durchlaufen und allen kürzesten Pfaden
  - **Begründung:** Stellt dar, wie wichtig Knoten für die Erreichbarkeit sind

- **Standardabweichung Betweenness Centrality**
  - **Berechnung:**

$$stdev\_betweenness\_centrality = \sigma_{betweenness\_centralities} \tag{2.28}$$

  - **Beschreibung:** Standardabweichung der Betweenness Centrality der einzelnen Knoten
  - **Begründung:** Kann das Verhältnis zwischen Knoten-Hubs und normalen Knoten darstellen

- **Durchschnittliche Closeness Centrality**
  - **Berechnung:**

$$avg\_closeness\_centrality = \overline{x}_{closeness\_centralities} \tag{2.29}$$

    mit:

$$closeness\_centralities = \left\{ \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)} \;\middle|\; u \in V \right\} \tag{2.30}$$

    wobei:

$d(v,u)$ = die kürzeste Pfaddistanz zwischen v und u und
$n-1$ = die Anzahl an Knoten, die von u erreichbar sind, ist

  - **Beschreibung:** Durchschnittswert der eingehenden Closeness Centrality für einen Knoten u als Kehrwert der durchschnittlichen kürzesten Pfaddistanz zu u von allen $n-1$ erreichbaren Knoten
  - **Begründung:** Maß der Knotennähe im Graph, stellt Konnektivität und Erreichbarkeit dar

- **Standardabweichung Closeness Centrality**
  - **Berechnung:**

$$stdev\_closeness\_centrality = \sigma_{closeness\_centralities} \tag{2.31}$$

  - **Beschreibung:** Standardabweichung der Closeness Centrality einzelner Knoten
  - **Begründung:** Abweichungen von Konnektivität und Erreichbarkeit abgebildet

- **Durchschnittliche Eigenvektorzentralität**
  - **Berechnung:**

$$avg\_eigenvector\_centrality = \overline{x}_{eigenvector\_centralities} \tag{2.32}$$

    mit:

$$eigenvector\_centralities = \{ev(v) \mid v \in V\} \tag{2.33}$$

    und:

$$ev(v) = \frac{1}{\lambda} \sum_{u \in N(v)} ev(u) \tag{2.34}$$

    als schrittweise Annäherung für den Knoten v, wobei:

$N(v)$ = die Nachbarn des Knotens v und
λ = eine Normalisierungskonstante ist

  - **Beschreibung:** Knoten mit hoher Eigenvektorzentralität sind mit weiteren Knoten mit hoher Eigenvektorzentralität verbunden, die Zentralität wird iterativ bis zur Konvergenz berechnet
  - **Begründung:** Stellt ein rekursives Zentralitätsmaß dar, bildet Verbundenheit von Hubs untereinander ab

- **Standardabweichung Eigenvektorzentralität**
  - **Berechnung:**

$$stdev\_eigenvector\_centrality = \sigma_{eigenvector\_centralities} \tag{2.35}$$

  - **Beschreibung:** Standardabweichung von der Eigenvektorzentralität der einzelnen Knoten
  - **Begründung:** Misst die Abweichungen der rekursiven Zentralität der Knoten

- **Assortativität**
  - **Berechnung:**

$$assortativity = \frac{\sum_{i,j} e_{ij} - \sum_i (a_i b_i)}{1 - \sum_i (a_i b_i)} \tag{2.36}$$

    wobei:

    e_{ij} = Kanten, die Knoten mit Grad i und j verbinden,
    a_i = Kanten, die einen Endpunkt mit Grad i haben,
    b_i = Kanten, die den anderen Endpunkt mit Grad i haben, darstellen

  - **Beschreibung:** Quantifiziert die Tendenz von Knoten in einem Netzwerk, sich bevorzugt mit anderen Knoten mit ähnlichen Graden zu verbinden
  - **Begründung:** Kann Rückschlüsse auf die Existenz von Knoten-Hubs geben

- **Dichte**
  - **Berechnung:**

$$density = \frac{2|E|}{|V|(|V| - 1)} \tag{2.37}$$

  - **Beschreibung:** Misst die Kantendichte des Graphen
  - **Begründung:** Grundlegende Graphmetrik

- **Durchmesser**
  - **Berechnung:**

$$diameter = \max(eccentricities) \tag{2.38}$$

    mit:

$$eccentricities = \{\max_{u \in V} d(u, v) \mid v \in V\} \tag{2.39}$$

    wobei:

    $d(u, v)$ = die kürzeste Pfaddistanz zwischen u und v ist

  - **Beschreibung:** Durchmesser, als maximale Exzentrizität des Graphen
  - **Begründung:** Grundlegende Graphmetrik

- **Durchschnittliche Exzentrizität**
  - **Berechnung:**

$$avg\_eccentricity = \overline{x}_{eccentricities} \tag{2.40}$$

  - **Beschreibung:** Durchschnittliche Exzentrizität der Graphknoten, also Mittelwert der maximalen Pfadlängen
  - **Begründung:** Grundlegende Graphmetrik

- **Standardabweichung Exzentrizität**
  - **Berechnung:**

$$stdev\_eccentricity = \sigma_{eccentricities} \tag{2.41}$$

  - **Beschreibung:** Standardabweichung der Exzentrizität der Graphknoten
  - **Begründung:** Grundlegende Graphmetrik, Abweichungen geben Rückschlüsse auf Existenz von Hubs

- **Durchschnittlicher PageRank**
  - **Berechnung:**

$$avg\_pagerank = \overline{x}_{PageRank} \tag{2.42}$$

    mit PageRank-Implementierung von NetworkX nach [90]
  - **Beschreibung:** Mittelwert des Knotenrangs auf der Grundlage der Struktur der eingehenden Links
  - **Begründung:** Metrik für Knotenwichtigkeit und Zentralität

- **Standardabweichung PageRank**
  - **Berechnung:**

$$stdev\_pagerank = \sigma_{PageRank} \tag{2.43}$$

  - **Beschreibung:** Standardabweichung zwischen Knotenrängen
  - **Begründung:** Abweichungen geben Rückschlüsse auf Existenz von Hubs

- **Durchschnittlicher Hub-Score**
  - **Berechnung:**

$$avg\_hub\_score = \overline{x}_{hub\_scores} \tag{2.44}$$

    mit:

$$hub\_scores = \{HITS\_HUBS(v) \mid v \in V\} \tag{2.45}$$

    wobei als Algorithmus *Hyperlink-Induced Topic Search (HITS)* zur Berechnung der Werte für die Hubs verwendet wird und iterativ einkommende und ausgehende Kanten betrachtet [65]

  - **Beschreibung:** Hohe durchschnittliche Hub-Scores deuten auf die Existenz vieler untereinander verbundener Hubs hin
  - **Begründung:** Aussagekraft über Existenz von Hubs

- **Standardabweichung Hub-Score**
  - **Berechnung:**

$$stdev\_hub\_score = \sigma_{hub\_scores} \tag{2.46}$$

  - **Beschreibung:** Standardabweichung der berechneten Hub-Scores für die einzelnen Knoten
  - **Begründung:** Abweichungen stellen Verhältnis zwischen Hubs und normalen Knoten dar

- **Durchschnittlicher Nachbargrad**
  - **Berechnung:**

$$avg\_neighbors\_degree = \overline{x}_{avg\_neighbor\_degrees} \tag{2.47}$$

    mit:

$$avg\_neighbor\_degrees = \{k_{nn,v} \mid v \in V\} \tag{2.48}$$

    mit:

$$k_{nn,v} = \frac{1}{|N(v)|} \sum_{u \in N(v)} deg(u) \tag{2.49}$$

    wobei:

    $N(v)$ = die Nachbarn des Knotens v und
    $deg(u)$ = der Grad des Knotens u ist

  - **Beschreibung:** Mittelwert des durchschnittlichen Knotengrads von Nachbarknoten
  - **Begründung:** Grundlegende Graphmetrik, Erkenntnisse über Konnektivität

- **Standardabweichung Nachbargrad**
  - **Berechnung:**

$$stdev\_neighbors\_degree = \sigma_{avg\_neighbor\_degrees} \tag{2.50}$$

  - **Beschreibung:** Standardabweichung der einzelnen durchschnittlichen Knotengradwerte von Nachbarknoten
  - **Begründung:** Grundlegende Graphmetrik, Erkenntnisse über Konnektivität und Hubs

- **Transitivität**
  - **Berechnung:**

$$T = 3\frac{\#triangles}{\#alltriplets} \tag{2.51}$$

    wobei Dreiecke definiert sind als drei Kanten, die einen Zyklus bilden.
  - **Beschreibung:** Wahrscheinlichkeit, dass zwei Knoten einen gemeinsamen direkt verbundenen Nachbarn haben
  - **Begründung:** Bestimmt die Verbundenheit und Existenz von Clustern

- **Knotenkonnektivität**
  - **Berechnung:**

$$\kappa(G) = \min_{u,v \in V} \kappa(u,v) \tag{2.52}$$

mit:

$$\kappa(u,v) = \min_{X \subseteq V \setminus (u,v) |X|} \tag{2.53}$$

wobei:

$X =$ eine Knotenmenge ist, deren Entfernung die Knoten u und v trennt

  - **Beschreibung:** Mindestanzahl der Knoten, die entfernt werden müssen, um den Graphen zu trennen
  - **Begründung:** Misst Graphkomplexität und Verbundenheit

- **Kantenkonnektivität**
  - **Berechnung:**

$$\lambda(G) = \min_{u,v \in V} \lambda(u,v) \tag{2.54}$$

mit:

$$\lambda(u,v) = \min_{E' \subseteq E |E'|} \tag{2.55}$$

wobei:

$E' =$ eine Kantenmenge ist, deren Entfernung die Knoten u und v trennt

  - **Beschreibung:** Mindestanzahl der Kanten, die entfernt werden müssen, um den Graphen zu trennen
  - **Begründung:** Misst Graphkomplexität und Verbundenheit

- **Durchschnittlicher RichClub-Koeffizient**
  - **Berechnung:**

$$avg\_rich\_club\_coefficient = \overline{x}_{RichClubCoefficients} \tag{2.56}$$

mit:

$$RichClubCoefficients = \{\phi(k) \mid k \in D\} \tag{2.57}$$

mit:

$$\phi(k) = \frac{2E_k}{N_k(N_k - 1)} \tag{2.58}$$

wobei:

D = die Menge aller unterschiedlicher Knotengrade $deg(v)$ von $v \in V$,
N_k = die Anzahl an Knoten mit Grad größer k und
E_k = die Anzahl an Kanten unter diesen Knoten ist

  - **Beschreibung:** Misst die Tendenz, dass Knoten hohen Grades Verbindungen untereinander eingehen
  - **Begründung:** Existenz von untereinander verbundenen Hubs

- **Standardabweichung RichClub-Koeffizient**
  - **Berechnung:**

$$stdev\_rich\_club\_coefficient = \sigma_{RichClubCoefficients} \tag{2.59}$$

  - **Beschreibung:** Standardabweichung der RichClub-Koeffizienten der einzelnen Knoten
  - **Begründung:** Abweichungen zeigen die Existenz von stark verbundenen Gruppen und losen Knoten

2.2 Gossiping-Verfahren

Gossip-Algorithmen entstanden in den späten 1980er Jahren auf dem Gebiet der verteilten Datenverarbeitung. Ebenso sind sie unter dem Namen epidemische Algorithmen bekannt. Dieser Begriff wurde durch die Arbeit von Demers et al. geprägt, in welcher der erste Gossip-Algorithmus eingeführt wurde [33]. Das Paper befasst sich mit der Herausforderung, die Konsistenz von verteilten Datenbank-Replikaten aufrechtzuerhalten. In diesem Kontext muss sichergestellt werden, dass alle Replikate über aktuelle Daten verfügen. Dazu soll aber nicht wie sonst ein schwergewichtiges Modell, welches eine zentralen Kontrolle mit strengen Konsistenzprotokollen einsetzt, verwendet werden. Stattdessen wird ein Prozess vorgeschlagen, bei dem ein wiederholter zufälliger Informationsaustausch durchgeführt wird. Diese Idee findet ihren Ursprung bei den sozialen Netzwerken und ist der Verbreitung von Gerüchten nachempfunden. Systeme, die Gossiping-Protokolle zur Informationsverbreitung einsetzen, führen eine zufällige Kommunikation durch bis eine Konvergenz erreicht wird. Trotz dieser Zufälligkeit kann gewährleistet werden, dass irgendwann ein konsistenten Systemzustand erreicht wird. Hierbei spricht man von *Eventual Consistency*.

Im Bereich der verteilten Systeme sind die Gossip-Algorithmen heutzutage weit verbreitet, da sie eine effiziente und skalierbare Methode zur Informationsverbreitung bieten. Ihre dezentrale Natur ermöglicht es, Informationen in großen komplexen Netzwerken ohne zentrale Steuerungseinheit zu verteilen. Im Paper „Gossip Algorithms“ von Devavrat Shah werden die sogenannten *NextGen Networks* charakterisiert. Dabei handelt es sich um moderne Netzwerke, welche neue Ansprüche an die auszuführenden Algorithmen geltend machen. Shah legt fest, dass die Algorithmen folgende Eigenschaften erfüllen müssen [103]:

- Die Algorithmen müssen auf einem bestimmten Knoten lokal ausgeführt werden. Dabei sind auf die Informationen beschränkt, die auf diesem Knoten verfügbar sind. Es ist nicht gestattet, globale Infrastrukturinformationen zu nutzen.
- Die jeweilgen Aufgaben sind iterativ zu lösen. Die durchgeführten Abläufe können dabei sehr dynamisch sein. Gleichermaßen kann sich die Infrastruktur jederzeit ändern. Aufgrund der losen Kopplung der Netzwerkknoten sind Nachrichten asynchron auszutauschen.
- Um gegenüber der Dynamik des Netzes robust zu sein, sind statische Implementierungen zu vermeiden. Ein dezentraler Ansatz, bei dem jeder Knoten selbstständig agiert, ist zu bevorzugen.

- Algorithmen müssen mit minimalen Rechen- und Kommunikationsressourcen auskommen. Pro Iteration sind möglichst wenige Operationen durchzuführen und nur simple Datenstrukturen einzusetzen.

Es folgt die formale Definition für einen Gossip-Algorithmus nach Shah [103]. Der Algorithmus muss für jede Operation an einem beliebigen Knoten $i \in V$ folgende Eigenschaften erfüllen:

(1) Der Algorithmus nutzt nur die lokal-beschränkten Informationen über seine direkten Nachbarn $\mathcal{N}(i) \triangleq \{j \in V : (i,j) \in E\}$.

(2) Sei d_i der Grad des Knotens i in G. So führt der Algorithmus höchstens $O(d_i \log n)$ Berechnungen pro Zeiteinheit durch.

(3) Sei $|F_i|$ der zur Ausgabe benötigte Speicheraufwand am Knoten i. Dann verbraucht der Algorithmus $O(\text{poly}(\log n) + |F_i|)$ Speicher am Knoten i während der Ausführung.

(4) Es wird keine Synchronisierung zwischen dem Knoten i und seinen Nachbarn $\mathcal{N}(i)$ benötigt.

(5) Das letztendliche Ergebnis des Algorithmus bleibt durch „sinnvolle“ Änderungen der Nachbarschaft $\mathcal{N}(i)$ während der Ausführung unbeeinflusst.

Die Gossip-Algorithmen werden zur Berechnung von generischen Netzwerkfunktionen verwendet. In einem solchen Netzwerk hat jeder Knoten, repräsentiert durch $i \in V$, eine bestimmte Information x_i. Das Ziel besteht darin, eine Funktion $f_i(x_1, \ldots, x_n)$ von einem beliebigen Knoten i im Netzwerk zu berechnen. Dabei können die Arten von Netzwerkfunktionen äußerst vielfältig sein und reichen von einfachen bis hin zu komplexen Berechnungen. Ein Beispiel ist hierbei eine lineare Berechnung, wie beispielsweise die Ermittlung des Durchschnitts. In diesem Fall wird die Netzwerkfunktion definiert als $f_i(x_i, \ldots, x_n) = \frac{1}{n} \sum_{k=1}^{n} x_k$. Andere Beispiele beinhalten die Bestimmung eines Maximal- oder Minimalwerts, eine globale Summe oder das Feststellen von Top-k-Werten.

Gossip-Algorithmen sind eine Klasse verteilter Algorithmen, die für die Informationsverbreitung und Datensynchronisation eingesetzt werden. Es lassen sich zwei Arten von Gossiping-Protokollen unterscheiden [13]. Die erste Art ist die der sogenannten *Dissemination Protocols*, die auf die Verbreitung von Wissen im Netzwerk abzielen. Ebenfalls gibt es Protokolle zur *Aggregation*, also die Berechnung von netzwerkweiten Durchschnitts-, Minimal- oder Maximalwerten [61] [63]. Der Ablauf beider Arten von Algorithmen ist im Grunde aber gleich. Netzwerkknoten wählen periodisch zufällige Nachbarn aus und führen mit diesen einen paarweisen Informationsaustausch durch. Dabei werden neue Informationen beziehungsweise Aktualisierungen verbreitet oder Daten abgefragt.

Neben der populären Anwendung im Bereich der replizierten Datenbanken, gibt es weitere Anwendungsbereiche. Gossip-Algorithmen können für eine effiziente Informationsverbreitung in Peer-to-Peer-, Sensornetzen und Ad-hoc-Netzwerken eingesetzt werden [103]. Zudem werden sie häufig für die Fehlererkennung und -behebung in großen verteilten Systemen verwendet. Durch den Austausch von Heartbeat-Nachrichten und Fehlermeldungen können Ausfälle schnell erkannt werden [110]. Infolgedessen wird eine zeitnahe Einleitung von Wiederherstellungsmechanismen und Fehlerbehandlungsroutinen möglich. Gleichermaßen können die Gossip-Algorithmen verwendet werden, um Informationen über Ressourcenverfügbarkeit und Systemauslastung zu verteilen [122]. Bei einer Anfrage nach verfügbaren Ressourcen wird diese an alle Knoten im Netzwerk verteilt. Das System kann daraufhin dezentral Entscheidungen über die Aufgabenzuweisung, Arbeitslastverteilung und Ressourcenzuweisung treffen. Dementsprechend kann

durch die Anwendung von Gossip-Algorithmen ein effizientes Load Balancing in verteilten Systemen realisiert werden.

Gossip-Algorithmen erfreuen sich einer hohen Popularität, da sie viele Vorteile bieten. Zum einen ermöglicht die dezentrale Ausführung, auf zentrale Steuerungseinheiten zu verzichten. Dadurch verfügen die Systeme auch über keinen Single Point of Failure, der sehr anfällig gegenüber Ausfällen ist. Des Weiteren sind Gossiping-Verfahren sehr leichtgewichtig, da Nachrichten ohne großen Overhead verteilt werden. Von daher weisen die Verfahren weisen auch sehr gute Skalierbarkeitseigenschaften auf. Darüber hinaus sind sie durch das probabilistische Vorgehen bei der Informationsausbreitung sind ebenso sehr fehlertolerant. Selbst wenn Knoten nicht mehr erreichbar sind, läuft der Algorithmus problemlos weiter. Infolgedessen sind sie auch für dynamische Netzwerke geeignet, in denen Ereignisse wie der Beitritt und das Verlassen von Knoten jederzeit auftreten können.

Eine bedeutende Eigenschaft von Gossip-Algorithmen ist die sogenannte Eventual Consistency [33]. Dies bedeutet, dass beim Gossiping letztendlich alle Knoten zu einem gemeinsamen Wert konvergieren. Es besteht jedoch keine Gewissheit darüber, wann genau diese Konvergenz eintritt. Dennoch ist eine Laufzeitabschätzung relevant, um die Performanz der Algorithmen zu bewerten. Das Konvergenzverhalten kann je nach dem verwendeten Protokoll und entsprechender Konfiguration variieren. Dabei können die verschiedenen Verhaltensweisen wie folgt klassifiziert werden [5]:

- *Lineare Konvergenz*: Bei der linearen Konvergenz ist die Anzahl an benötigten Runden direkt proportional zur Anzahl der Knoten. Das bedeutet, dass mit zunehmender Größe des Netzwerks auch die Konvergenzzeit gleichmäßig ansteigt. Lineare Konvergenz wird typischerweise bei einfachen epidemischen Gossiping-Protokollen beobachtet.

- *Sublineare Konvergenz*: Sublineare Konvergenz bedeutet, dass die Konvergenzzeit langsamer wächst als die Anzahl der Knoten. Sublineare Konvergenz ist wünschenswert, um eine gute Skalierbarkeit zu erreichen.

- *Superlineare Konvergenz*: Superlineare Konvergenz tritt auf, wenn die Konvergenzzeit schneller wächst als die Anzahl der Knoten. Dies kann passieren, wenn das Gossiping-Protokoll bestimmte Ineffizienzen aufweist oder wenn Netzwerkbedingungen ungünstig sind. Hierdurch wird die Skalierbarkeit stark eingeschränkt.

In der Praxis gibt es viele verschiedene Aspekte, welche die Konvergenzgeschwindigkeit beeinflussen können. Im Folgenden werden nun die wichtigsten Faktoren besprochen:

- *Fanout*: Mit Fanout wird die Anzahl der Nachbarn, an die ein Knoten in jeder Runde Informationen weitergibt, bezeichnet. Höhere Fanout-Werte führen automatisch zu weniger benötigten Runden. Dafür steigt der Netzwerkverkehr pro Runde multiplikativ mit dem Fanout-Wert. Für diese Arbeit wird sich auf einen Fanout-Wert von eins beschränkt. Dadurch wird eine einheitliche Betrachtung gewährleistet. Es werden also pro Runde so viele Informationen ausgetauscht, wie es Knoten im Netzwerk gibt [64] [104].

- *Nachrichtenverbreitung*: Bei der Nachrichtenverbreitung kann zwischen push- und pull-basiertem Gossiping unterschieden werden. Bei push-basierter Verbreitung senden Knoten Informationen proaktiv an ihre Nachbarn, während bei der pull-basierten Variante Informationen von Nachbarn bei Bedarf angefordert werden [123]. Für die Simulationen in dieser Arbeit wird ein hybrider Ansatz evaluiert. Knoten

fragen aktiv für einen Informationsaustausch einen zufälligen Nachbarn an [104]. Anschließend werden beide Knoten aktiv und führen eine bidirektionale Kommunikation durch.

- *Auswahlstrategie*: Einen großen Einfluss auf die Konvergenzgeschwindigkeit hat das verwendete Auswahlverfahren für den Gossip-Partner. Die zufällige Auswahl ist der einfachste Ansatz und wird in vielen Protokollen verwendet. Komplexere Strategien können die Konvergenzgeschwindigkeit erheblich verbessern. Dabei kann eine Gewichtung basierend auf verschiedensten Faktoren durchgeführt werden. In dieser Arbeit sollen Verfahren entwickelt werden, die dazu Wissen über die Community-Zugehörigkeit von Knoten verwenden.

Um die Laufzeit abzuschätzen, können mathematische Erkenntnisse zum Telefonproblem genutzt werden. Hierbei handelt es sich um eine Fragestellung aus der Informationsverbreitung, welches auf das Gossiping übertragbar ist. Im Kontext des Telefonproblems verfügt jeder Knoten eines Kommunikationsnetzes über eine eigene Information, die im Netz verteilt werden muss. Informationen werden dabei an andere Knoten durch Telefonanrufe, also wechselseitige Kommunikation, weitergeben. Bei einem Anruf erfolgt der Austausch aller zu diesem Zeitpunkt bekannten Informationen. Brenda Baker und Robert Shostak haben eine optimale Lösung für das Problem gefunden und veröffentlicht [7]. Sei hierbei $n > 4$ die Anzahl an beteiligten Knoten. Dann beträgt die Mindestanzahl der Anrufe, die erforderlich ist, um die Verbreitung der gesamten Information zu gewährleisten, $2n - 4$. Daraus folgt eine minimale Rundenanzahl von $2\lceil \log_2 n \rceil - 3$ [70]. Unter der Annahme, dass die Gespräche zwischen sich nicht überschneidenden Knotenpaaren gleichzeitig stattfinden können, beträgt die Mindestzeit, somit [56]:

$$T(n) = \begin{cases} \lceil \log_2 n \rceil, & \text{falls } n \text{ gerade} \\ \lceil \log_2 n \rceil + 1, & \text{sonst} \end{cases} \tag{2.60}$$

Um eine genauere Einschätzung der Konvergenzzeit tätigen zu können, wird eine obere Grenze benötigt. Diese wird im Paper [58] für fehlertolerante Gossiping-Verfahren definiert. Hier wird die obere Grenze anhand folgender Formel für große Werte von n und k auftretende Fehler definiert:

$$\tau(n,k) \leq \frac{2}{3}nk + O(n) \tag{2.61}$$

Hierbei liegt für $k = 0$ der optimale Fall vor, wo keine Fehler auftreten. In diesem Fall ist das lineare Laufzeitverhalten eine enge Obergrenze für die Konvergenzzeit. Das Paper [12] kommt zum gleichen Ergebnis. Hier wird berechnet, dass maximal $\Theta(n^2)$ Nachrichten benötigt werden, um eine Konvergenz zu erreichen. Bei einer rundenbasierten Durchführung, wo in jeder Runde jeder Knoten einmal aktiv wird, kommt man somit ebenfalls auf eine lineare Laufzeit.

2.3 Container

Container stellen eine Form der *Betriebssystem-Virtualisierung* dar. In einem Container kann eine Anwendung beliebiger Größe ausgeführt werden. Diese wird entkoppelt von der unterliegenden Infrastruktur. Dadurch hat die Anwendung immer das gleiche Verhalten, egal wo und wann der Container ausgeführt wird [30].

Der Container beinhaltet den Code der Anwendung sowie alles, was zu deren Ausführung benötigt wird. Dabei kann es sich um die Laufzeitumgebung, Konfigurationen, Bibliotheken und Ähnliches handeln [120][36]. Ein *Container-Image* ist eine Datei oder ein Archiv, welches ein solches Software-Paket enthält. Images erleichtern die Bereitstellung und stellen Portabilität, Plattformunabhängigkeit und Skalierbarkeit sicher. Immer wenn Änderungen an einem Container erfolgen sollen, muss ein neues Image erstellt und der Container neu erzeugt werden [30].

2.3.1 Docker

Docker ist eine *Container-Laufzeitumgebung*. Container-Images werden hier als Dateien, die sogenannten *Dockerfiles*, verwaltet. Dockerfiles können als Schablonen verstanden werden, die zur Erstellung von Containern nutzbar sind. Dabei sind sie portabel speicherbar und mehrfach anwendbar [120]. Eine wichtige Eigenschaft von Docker Images ist der Aufbau in Schichten, wobei jede Schicht einer Instruktion im Dockerfile entspricht [1].

In Listing 2.1 ist ein Beispiel für ein Dockerfile zu sehen. Zuerst erfolgt hier die Initialisierung einer Build Stage, indem als Basis für das Image *Alpine-Python 3.9* ausgewählt wird. Anschließend wird die virtuelle Umgebung für die Python-Anwendung erzeugt. Dabei muss der Pfad entsprechend aktualisiert werden, um die in der virtuellen Umgebung installierten Python-Pakete direkt verwenden zu können. Das Arbeitsverzeichnis wird definiert, um den Kontext für folgende Befehle zu setzen. Danach wird eine Datei mit Anwendungsabhängigkeiten im Build-Kontext in ein entsprechendes Verzeichnis des Containers kopiert. Dann werden alle Abhängigkeiten in der virtuellen Umgebung installiert. Im nächsten Schritt wird der Quellcode der Container-Anwendung kopiert. Mit der letzten Zeile erfolgt die Defitntion des Befehls, mit dem der Container gestartet wird. Eine Übersicht zentraler Dockerfile-Befehle ist in Tabelle 2.2 zu finden [37].

```
1  FROM python:3.9-alpine as base
2
3  ENV VIRTUAL_ENV=/opt/venv
4  RUN python3 -m venv $VIRTUAL_ENV
5  ENV PATH="$VIRTUAL_ENV/bin:$PATH"
6
7  WORKDIR /app
8
9  # install all requirements
10 COPY container-app/requirements.txt requirements.txt
11 RUN pip install -r requirements.txt
12
13 COPY container/service.py service.py
14
15 ENTRYPOINT ["python", "-u", "service.py"]
```

Listing 2.1: Beispiel Dockerfile

2.3.2 Docker Hub

Docker Hub ist ein *Repository*, also ein Verzeichnis, für Container-Images. Es dient der zentralen Verwaltung, Versionierung und Bereitstellung der Images. Hier können zuvor kompilierte Dockerfiles unter einem *Tag* abgelegt werden. Anschließend können sie über diesen Tag abgerufen und von überall als Container gestartet werden. Docker Hub stellt ebenfalls Funktionalität zur Automatisierung (Integration mit CI/CD) und zum Teilen von Images bereit. Darüber hinaus bietet es Statistiken und Analysetools an zur Nutzungs- und Leistungsüberwachung.

Tabelle 2.2: Docker-Befehle

Docker-Befehl	Beschreibung
FROM	Initialisiert eine neue Build Stage, setzt das Standard-Image für nachfolgende Befehle
LABEL	Setzt einen Bezeichner zwecks Verwaltung
RUN	Führt Kommandos in einer neuen Schicht aus
CMD	Dient zur Ausführung der Software des Images
EXPOSE	Definiert die Ports, die zur Laufzeit abgehört werden
ENV	Setzt Umgebungsvariablen
ARG	Setzt eine Variable im Scope des Build-Prozesses
ADD, COPY	Kopiert lokale Dateien in den Container
ENTRYPOINT	Setzt den Hauptbefehl des Images
VOLUME	Macht Speicherbereiche verfügbar

2.4 Kubernetes

Kubernetes (k8s) basiert auf einem *Cluster Management System* namens *Borg* [114]. Es ermöglicht die Verwaltung eines Computerclusters, also einem Zusammenschluss mehrerer Rechner [96]. Dabei wird zur Kubernetes zur Orchestrierung von Containern verwendet, indem die Automatisierung der Verwaltung, Bereitstellung und Skalierung realisiert wird [68]. Im Vergleich zu Docker, das auf einem einzelnen Computer ausgeführt wird, ist Kubernetes eine Anwendung für eine Vielzahl an Maschinen. Kubernetes wurde entwickelt, um das Zusammenwirken einer großen Zahl an Containern auf verschiedenen Computersystemen zu verwalten.

2.4.1 Architektur

Im Folgenden wird die Architektur eines Kubernetes-Clusters besprochen. Eine grafische Visualisierung dieser ist in Abbildung 2.6 zu sehen. Die dargestellten zentralen Komponenten werden nun im Einzelnen erklärt.

(1.) Ein **Kubernetes-Cluster** kann als eine Gruppe von Knoten gesehen werden. Hierbei repräsentiert jeder Knoten eine Arbeitsmaschine, welche zusammenarbeiten, um containerisierte Anwendungen auszuführen. Ein Cluster besteht aus dem *Control Plane*, auch Steuerebene, und einem oder mehreren *Worker Nodes*.

(2.) Das **Control Plane** verwaltet und kontrolliert den Gesamtzustand und das Verhalten des Clusters [67]. Es besteht aus den folgenden Komponenten:

  (a.) Der *kube-apiserver* stellt eine interne und eine externe Application Programming Interface (API)-Schnittstelle bereit. Die interne Schnittstelle ermöglicht die Kommunikation der verschiedenen Komponenten des Clusters, während die externe API eine Interaktion zwischen den Clients und dem Cluster realisiert. Dabei ist die Hauptaufgabe des Servers der Emfpang und die Verarbeitung von Anfragen. Diese validiert er anschließend und aktualisiert dann den gewünschten Zustand des Clusters im Key-Value-Store [127].

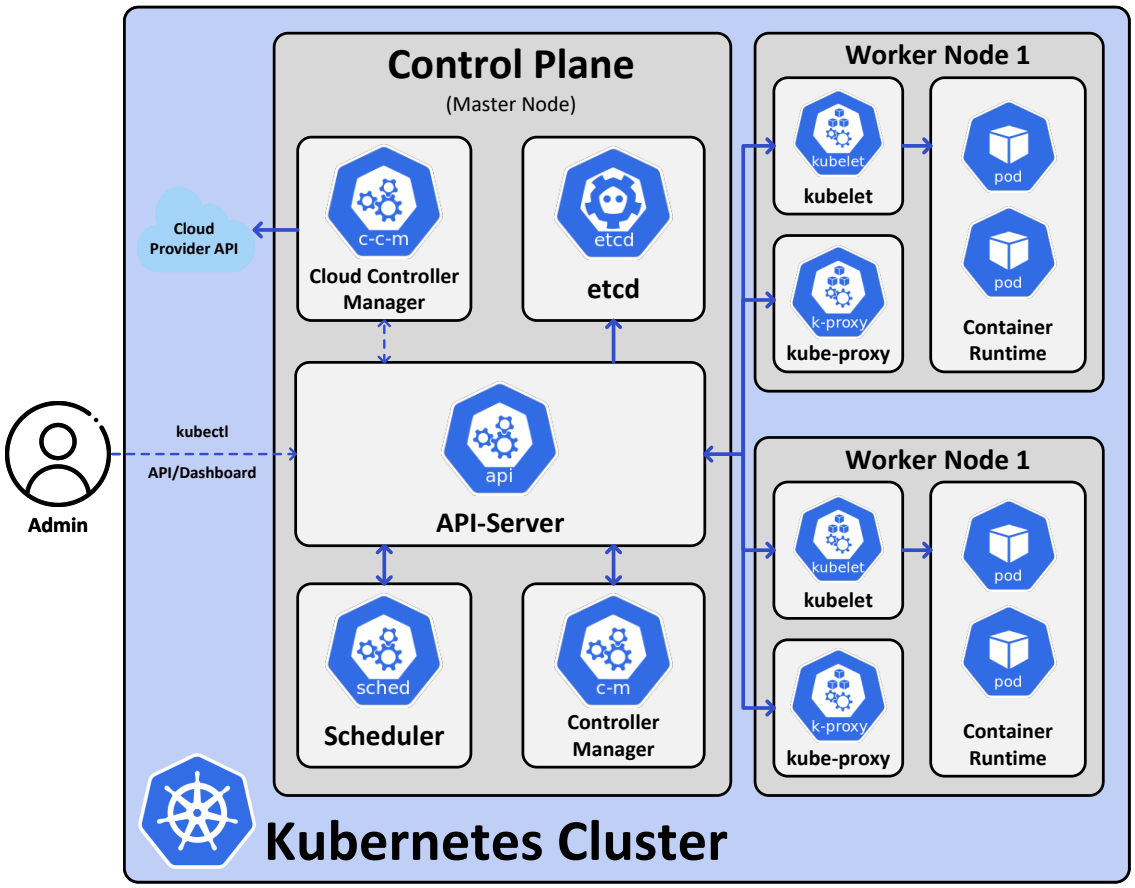


Abbildung 2.6: Kubernetes-Architektur [92]

(b.) Der *etcd* ist ein verteilter Key-Value-Store, der alle Konfigurationsdaten sowie den gewünschten Zustand des Clusters speichert [126]. Die anderen Komponenten der Steuerungsebene greifen auf ihn lesend und schreibend zu. Durch die zentrale Konfiguration kann die Konsistenz des Systems sichergestellt werden. Zudem können die Komponenten so den Zustand des Clusters verfolgen.

(c.) Der *kube-scheduler* ist für die Zuweisung von Pods zu Worker Nodes verantwortlich. Er berücksichtigt Faktoren wie die Ressourcenanforderungen der Container-Anwendungen, Affinitäts- und Anti-Affinitäts-Regeln sowie sonstige Beschränkungen. Der Scheduler verfolgt das Ziel, optimale Planungsentscheidungen zu treffen. Dazu wird der Cluster von ihm kontinuierlich überwacht. Aufgrund dessen kann eine effiziente Ressourcennutzung sichergestellt werden [129].

(d.) Der *kube-controller-manager* führt verschiedene Controller aus, die unterschiedliche Cluster-Funktionalität bereitstellen. Jeder Controller ist hierbei für eine bestimmte Aufgabe zuständig. Dazu zählen die Verwaltung von Knotenlebenszyklen, gewünschten Pod-Replikationen und Service-Endpunkten. Außerdem überwacht der Manager den Zustand des Clusters und stellt sicher, dass der tatsächliche und gewünschte Zustand übereinstimmen [31]. Hierbei folgt er dem zentralen Konzept des *Reconcilation-Loops* [15], welcher aus den folgenden Schritten besteht:

1. *Beobachtung*: Der Controller beobachtet den aktuellen Zustand der von ihm verwalteten Ressourcen, indem er den API-Server abfragt.

2. *Vergleich*: Anschließend vergleicht er den beobachteten mit dem gewünschten Zustand. Der Zielzustand wird dabei durch deklarative Konfiguration, also Deployments oder Custom Resources, definiert.
3. *Unterschiede feststellen*: Abweichungen zwischen dem beobachteten Zustand und dem angestrebten Zustand werden ermittelt, sofern solche vorhanden sind.
4. *Entscheidungsfindung*: Danach trifft der Controller Entscheidungen, wie der beobachtete Zustand in den gewünschten Zustand überführt werden kann. Dazu erstellt, aktualisiert oder löscht er je nach Bedarf Ressourcen.
5. *Ausführung*: In der nächsten Phase werden die notwendigen Aktionen ausgeführt.
6. *Abwarten*: Nach dem Ausführen wartet der Controller darauf, dass der API-Server die Änderungen verarbeitet.
7. *Wiederholung*: Der Reconcilation-Loop wird auf unbestimmte Zeit fortgesetzt. Dabei wird der Zustand der Ressourcen ständig überwacht und angepasst.

(e.) Der optionale *Cloud Controller Manager* integriert Cloud-APIs, um Ressourcen und Dienste in Cloud-Umgebungen zu verwalten. Dadurch wird die Interaktion mit der zugrunde liegenden Infrastruktur möglich [23].

(3.) Die **Worker Nodes** sind physische oder virtuelle Maschinen, die Container ausführen. Die Container laufen in *Pods*, den kleinsten verwaltbaren Einheiten in Kubernetes [87]. Jeder Worker Node besteht aus den folgenden Komponenten:

(a.) Der *kubelet* fungiert als Node Agent. Seine Aufgaben umfassen die Kommunikation mit dem Control Plane und die Verwaltung der laufenden Pods. Der Agent muss hierbei sicherstellen, dass die Container innerhalb der Pods wie gewünscht laufen [130].

(b.) Die *Container-Runtime*, auch Laufzeitumgebung, ist für die Ausführung und Verwaltung der Container zuständig. Dabei unterstützt Kubernetes verschiedene Umgebungen wie Docker, containerd und andere. Ihre Aufgabe ist das Abrufen von Images, das Starten und Stoppen von Containern sowie die Ressourcenzuweisung [29].

(c.) Der *kube-proxy* ist ein Netzwerkproxy. Er ist zuständig für die Suche von Diensten, die Verwaltung von Netzwerkregeln und die Weiterleitung des Datenverkehrs an die Pods. Damit realisiert er die Netzwerkkommunikation zwischen Pods und Services [128].

(d.) *Pods* können eine Gruppe von einem oder mehreren Containern umfassen, die sich dieselben Ressourcen und denselben Namensraum teilen. Sie werden von den Komponenten des Control Plane geplant, konfiguriert und verwaltet. Für ihre Ausführung sind die Worker Nodes verantwortlich. Im Kapitel 2.4.4 ist die Funktionsweise der Pods nochmals ausführlich beschrieben.

(4.) Entwickler und Administratoren können auf das Cluster über *kubectl* zugreifen. Hierbei handelt es sich um ein Command Line Interface (CLI), dass die Interaktion mit dem Control Plane über die Kubernetes-API ermöglicht. Weitere Möglichkeiten auf den API-Server zuzugreifen umfassen andere CLI-Anwendungen sowie grafische Applikationen wie das Kubernetes Dashboard. Endnutzer interagieren direkt mit den Container-Anwendungen, wobei der Zugriff meist über einen Load Balancer oder über Netzwerkproxies erfolgt.

2.4.2 Kubernetes-Cluster Aufbau

Das *Container Runtime Interface (CRI)* und das *Container Networking Interface (CNI)* sind ebenfalls wichtige Bestandteile der Kubernetes-Architektur. Die Steuerungsebene mit dem API-Server, Scheduler und dem Controller-Manager bauen auf ihnen auf. Ebenso greifen die logischen Komponenten auf sie zu, um ihre Funktionalität zu erreichen. Hierbei befähigt das CRI kubelet zur Verwaltung der Container auf jedem Knoten im Cluster. Gleichermaßen wird dem kube-proxy durch das CNI ermöglicht, die Netzwerkkonnektivität für die Dienste innerhalb des Clusters zu realisieren. Die genauen Mechanismen werden nun im Folgenden erläutert.

Das CRI dient als Schnittstelle zwischen kubelet und der Container-Runtime [28]. Es definiert eine API als Schnittstelle und ein Protokoll zur Kommunikation. Dadurch kann kubelet mit Container-Laufzeiten wie beispielsweise *containerd* kommunizieren. Des Weiteren stellt es Funktionen wie die Erstellung von Pods und die Ressourcenverwaltung bereit. Ebenfalls kümmert sich das CRI um die Verwaltung des Lebenszyklus von Containern. Es ist somit verantwortlich für das Starten, Stoppen und Löschen dieser. Aufgrund dessen kann kubelet mit der Container-Runtime interagieren, ohne, dass eine enge Kopplung an eine spezifische Implementierung erfolgen muss. Als Folge dieser Entkopplung können verschiedene Container-Laufzeiten unterstützt werden.

Das CNI ist die Schnittstelle für die Verwaltung von Container-Netzwerken [50]. Sie stellt eine API und Plugins für die Konfiguration der Netzwerkkonnektivität zur Verfügung. Die CNI-Plugins übernehmen Aufgaben wie die IP-Zuweisung, die Einrichtung von Netzwerkschnittstellen, Routing und Netzwerkrichtlinien. Wenn ein Pod erstellt wird, ruft kubelet das entsprechende Plugin auf, um die Netzwerkumgebung für den Pod einzurichten. Die verschiedenen Netzwerklösungen wie Flannel, Calico oder Weave integrieren die CNI-Plugins nahtlos. Aufgrund dessen wird eine Interoperabilität ermöglicht, während die resultierenden Netzwerke für Pods einheitlich bleiben.

Die Interaktion zwischen kubelet, dem CRI (containerd) und dem CNI (Flannel) kann in Abbildung 2.7 nachvollzogen werden. Hier werden die Schritte dargestellt, die kubelet durchführt, wenn ein Pod durch den Scheduler auf einem Worker Node platziert wird. Dabei wird die Erzeugung des Pods durch das CRI erläutert. Ebenso wird beschrieben, wie das Pod Netzwerk durch das CNI Plugin konfiguriert wird.

2.4.2.1 containerd

Containerd ist eine populäre plattformunabhängige Container-Laufzeitumgebung [49]. Container-Laufzeitumgebungen stellen Funktionalität zur Verwaltung und Ausführung von Containern bereit. Der Hauptfokus von containerd liegt hierbei auf Einfachheit, Robustheit und Portabilität. Es ist konzipiert, um in Higher-Level-Container-Anwendungen wie Docker oder Kubernetes eingesetzt zu werden. Für eine alleinstehende Anwendung hingegen ist es nicht geeignet. Dabei fungiert es als Zwischenschicht zwischen den Con©-tainer-Engines und dem Betriebssystem. Die Hauptfunktionalität von containerd ist die Abstraktion von Container-Operationen. Dazu wird eine standardisierte API für die Verwaltung der verschiedenen Funktionen bereitgestellt. Im Folgenden sind weitere wichtige Funktionen von containerd aufgeführt:

- *Image-Verwaltung*: Containerd unterstützt die Verwaltung von Container-Images aus verschiedenen Formaten, einschließlich des weit verbreiteten Docker-Image-Formats.
- *Container-Ausführung*: Es bietet eine Laufzeitumgebung für die Ausführung von Containern, die Verwaltung von Ereignissen und die Prozessverwaltung.

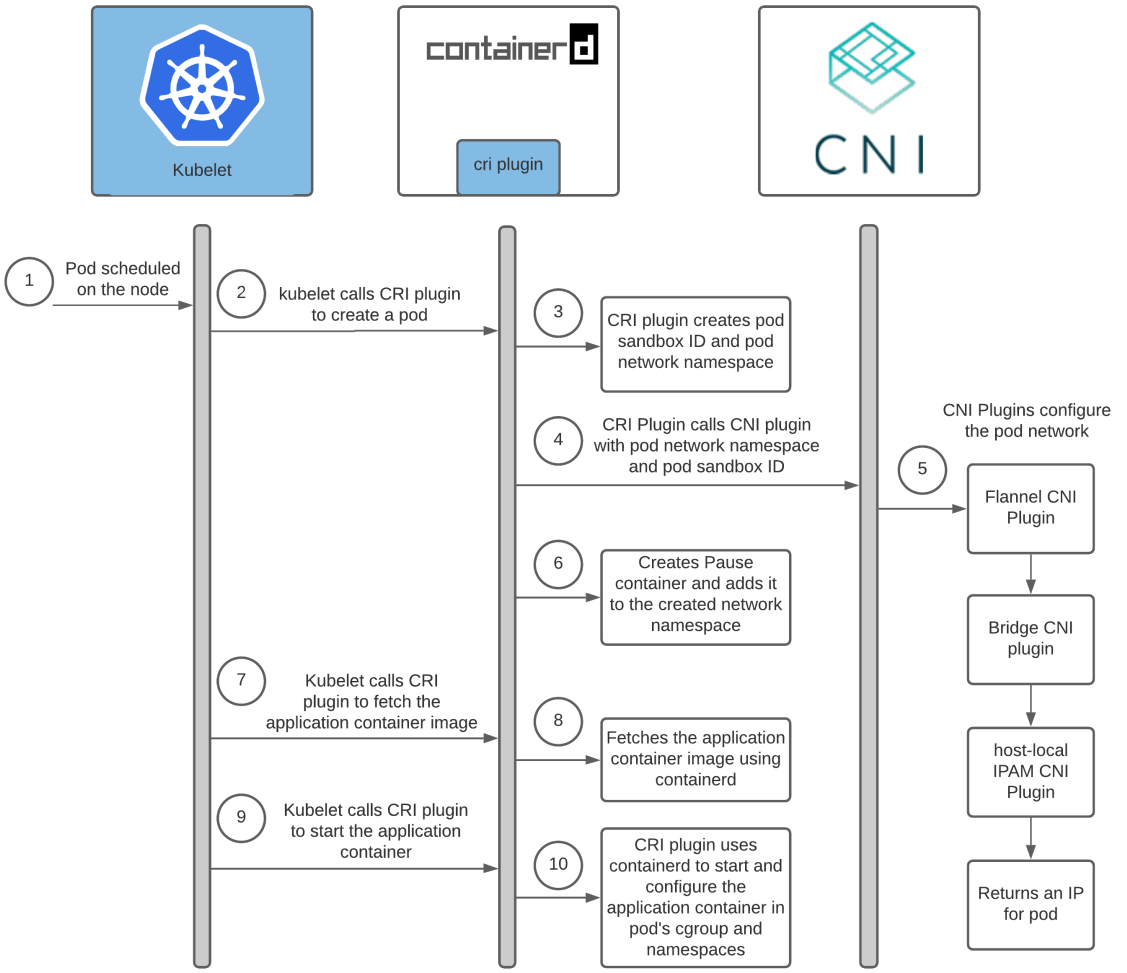


Abbildung 2.7: Zusammenspiel kubelet, CRI, CNI und Plugins [83]

- *Snapshotting und Checkpointing*: Containerd ermöglicht die Erstellung von Container-Snapshots. Dadurch kann der Zustand eines laufenden Containers gespeichert und wiederhergestellt werden. Ebenfalls kann ein solcher Snapshort zur Erstellung neuer Container erfolgen.
- *Ressourcenisolierung und Sicherheit*: Betriebssystemfunktionen wie Namespaces und Kontrollgruppen (cgroups) werden zur Isolierung und Gewährleistung der Sicherheit eingesetzt.

Der Funktionsumfang von containerd umfasst die grundlegenden Operationen für die Verwaltung von Containern. Higher-Level-Container-Anwendungen hingegen haben eine höhere Abstraktionsebene und bauen auf der Funktionalität von containerd auf. Solche Anwendungen umfassen unter anderem Container-Orchestrierungssysteme wie Kubernetes, Container-Registry-Management-Anwendungen und CI/CD-Plattformen. Durch die Verwendung von containerd müssen diese Anwendungen selbst nicht direkt mit den Betriebssystemen der physischen Nodes interagieren. Diese Aufgabe erfüllt containerd, wodurch die betriebssystemspezifischen Abläufe hinter den Container-Operationen verborgen bleiben. In der Praxis wird containerd häufig mit *runc* kombiniert, welches als Tool zur Interaktion mit dem Betriebssystem eingesetzt wird.

2.4.2.2 runc

Runc ist ein weit verbreitetes CLI [52] zum Erzeugen und Ausführen von Containern. Dabei folgt es der Spezifikation der *Open Container Initiative (OCI)*. Es wird von containerd standardmäßig zur Interaktion mit Linux und Windows Containern eingesetzt. Hierbei übernimmt runc die Low-Level-Container-Ausführung und die Verwaltung der Container-Lebenszyklen. Es verwendet dazu direkt die Kernel-Funktionen des jeweiligen Betriebssystem, um beispielsweise auf Ressourcen zuzugreifen. Runc ist als Runtime-Engine weit verbreitet und wird neben containerd von vielen weiteren Container-Laufzeitumgebungen verwendet. Die Laufzeitumgebungen nutzen runc als zentralen Baustein und können so den eigenen Funktionsumfang logisch begrenzen. Dies wird erreicht, indem Low-Level-Operationen abstrahiert werden. Erweiterte Funktionen wie Image-Management, Networking und Storage, können dadurch einfacher abgebildet werden.

2.4.2.3 flannel

Flannel ist eine beliebte Netzwerklösung, die für Kubernetes-Cluster entwickelt wurde [51]. Es realisiert die Kommunikation zwischen Pods über verschiedene physische Knoten hinweg. Dazu wird einfaches und skalierbares Overlay-Netzwerk implementiert, wo jedem Knoten ein eindeutiges Subnetz zugewiesen wird. Der Pod-Verkehr wird dann durch Kapselung der Pakete ermöglicht. Flannel nutzt verschiedene Backend-Mechanismen (häufig *VXLAN*), um die Netzwerkkonnektivität zwischen Knoten herzustellen. Durch den Einsatz von flannel kann Kubernetes die zugrunde liegende Netzwerkinfrastruktur abstrahieren. Als Folge können die Pods unabhängig der physischen Netzwerktopologie nahtlos über Knoten hinweg kommunizieren. Gleichzeitig wird Netzwerkisolierung und Netzwerksicherheit gewährleistet.

2.4.3 Kubernetes Objects

Kubernetes bildet den Zustand des Clusters anhand von Objekten ab. Objekte sind hier Entitäten, die den Zustand und das Verhalten von Anwendungen und Ressourcen definieren. Sie ermöglichen gemeinsam die Konfiguration und Verwaltung von Anwendungen. Zu Beispielen zählen die Bereitstellungseinheiten (Pods, ReplicaSet, Deployments), Services, Konfigurationen (ConfigMaps, Secrets), Speicherressourcen und weitere. Kubernetes stellt sicher, dass die in einem Cluster definierten Objekte vorhanden sind und in dem gewünschten Zustand existieren. Jedes Objekt hat hierbei die Eigenschaften *spec* und *status* [113]. Während spec die erwünschten Eigenschaften des Objekts wird, gibt status den aktuellen Zustand an. Durch Definition und Anwendung von *YAML-Dateien* kann die Erstellung der Objekte durchgeführt werden. YAML Ain't Markup Language (YAML) [111] ist eine auf *Unicode* basierte Sprache zur Datenserialisierung. Neben ihrer populären Anwendung bei der Definition von Konfigurationen, wird sie in vielen weiteren Bereichen eingesetzt. Beispiele beinhalten Datenserialisierung, Automatisierung sowie Orchestrierung und Programmiersprachen.

Darüber hinaus stellt Kubernetes Logik bereit, um die Verwaltung seiner Objekte zu erleichtern. *Namespaces* ermöglichen hierbei die Arbeit von mehreren Nutzern oder Teams an einem oder vielen Projekten [82]. Kubernetes-Objekte können einem solchen Namespace zugeordnet werden und sind dann nur innerhalb diesem sichtbar. Des Weiteren können *Labels* hinzugefügt werden, wodurch eine Einteilung in Gruppen realisierbar wird [71]. Dadurch wird die Identifikation und somit auch die Verwaltung der Objekte erleichtert. Objekte können folglich über Namespaces logisch separiert und durch Labels

über Namespaces hinweg gruppiert werden. Es folgt eine genaue Darstellung verschiedener Kubernetes-Objekte.

2.4.4 Pods

Bei Kubernetes werden Container in *Pods* ausgeführt. Pods wiederum können auf einem virtuellen oder physischen System laufen [88]. Ein Pod stellt die kleinste verwaltbare Einheit in Kubernetes dar und ist als Gruppe von Containern definiert. Alle dem Pod zugehörigen Container haben den gleichen Kontext. Dementsprechend teilen sie sich also System- und Netzwerkressourcen sowie Speicher. In Kubernetes bildet jeder Pod einen logischen Host ab. Dieser beinhaltet einen einzelnen oder mehrere Container, um eng zusammenhängende Anwendungen auszuführen. Der meistverbreitete Anwendungsfall bei der Erstellung von Pods ist das *„one-container-per-pod"-Modell*. Hierbei ist ein Pod als *Wrapper* um einen einzelnen Container zu verstehen. Unter Wrapper versteht man hierbei eine Hülle um den Container, der eine zusätzliche Abstraktionsebene bildet. Das Modell sieht vor, dass jeder Pod nur einen einzigen Container beinhaltet [93].

2.4.5 ReplicaSet

Ein *ReplicaSet* ermöglicht die Verwaltung einer Gruppe von Pods. Hierbei wird insbesondere gewährleistet, dass immer eine bestimmte Anzahl an Pods verfügbar sind [97]. Die direkte Verwendung dieser Objekte ist in der Praxis eher selten, stattdessen werden meist *Deployments* verwendet. Deplozments bauen auf ReplicaSets auf und stellen zusätzliche Funktionalitäten und Verwaltungsmöglichkeiten bereit.

2.4.6 Deployments

Als *Deployment* ist ein Objekt zu verstehen, das den gesamten Lebenszyklus einer Anwendung definiert. Dabei wird der Zielzustand eines Systems deklarativ beschrieben. Kubernetes kümmert sich darum, dass dieser Zustand erreicht wird und erhalten bleibt [34]. In Listing 2.2 ist ein Beispiel zu sehen, welches zur weiteren Erklärung referenziert wird.

Das Deployment definiert über die Eigenschaft *spec* die Spezifikation des Objekts. Dadurch wird das Objekt konfiguriert, in diesem Fall wird ein ReplicaSet erzeugt. Über *replicas* wird die Anzahl an gewünschten Pods festgelegt. Infolgedessen führt das Cluster die Instanzierung von mehreren Replikaten, also Kopien des gleichen Pods, durch. Stürzen Pods ab oder werden auf sonstige Art beendet, so reagiert Kubernetes mit dem Starten neuer Pods. Darüber hinaus kann die Anzahl an Replikaten auch noch nach der Bereitstellung beliebig angepasst werden. Kubernetes macht es damit möglich, eine horizontale Skalierung durchzuführen. Ebenfalls in Teil der Spezifikation ist der *Selektor*, welcher angibt, wie die zugehörigen Pods erkannt werden können. Die Erkennung kann über ein gesetztes Label (hier *kube-operator*) erfolgen. Zuletzt erfolgt mit dem *template-Feld* die Definition der zu erzeugenden Pods. Hier werden Metadaten wie Labels definiert und die Container spezifiziert. In diesem Beispiel wird nur ein Container aus dem referenzierten Image erstellt und ausgeführt.

2.4.7 ConfigMaps und Secrets

Kubernetes stellt zwei verschiedene Objekte zur Verwaltung von Konfiguration bereit. Für nicht vertrauliche Daten sind *ConfigMaps* zu verwenden [26]. *Secrets* hingegen beinhalten sensible Informationen, wie beispielsweise Passwörter oder Zertifikate [101]. Diese

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-operator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kube-operator
  template:
    metadata:
      labels:
        app: kube-operator
    spec:
      containers:
      - name: kube-operator
        image: user/kube-operator:latest
```

Listing 2.2: Beispiel Deployment

Daten benötigen zusätzlichen Schutz und werden deshalb verschlüsselt abgelegt. In Kubernetes erfolgt die Definition von Konfigurationen als Schlüssel-Wert-Paare, auf welche über Referenzen zugeriffen werden kann. Der Vorteil in der Verwendung von Objekten, um Konfiguration zu speichern, liegt in der Entkopplung von den Containern. Konfigurationen werden zentral einmal definiert und können dann beliebig wiederverwendet werden. Dadurch wird die Portabilität gewährleistet. Die Pods weisen die Konfigurationen dann über Umgebungsvariablen zu, indem ausgewählte Inhalte von ConfigMaps und Secrets in der Container-Definition referenziert werden. Umgebungsvariablen bieten hierbei eine Möglichkeit, dynamische Werte an die Container-Anwendungen zu übergeben, ohne dass eine Änderung des Container-Images erfolgen muss. Alternativ kann ein Zugriff über das kubelet beim Image-Pull oder durch Dateien in einem gemounteten Volume erfolgen [101].

2.4.8 Cluster Networking

Im Cluster ordnet Kubernetes jedem Pod eine eigene IP-Adresse zu. Dabei werden Container-Ports von Host-Ports getrennt. Infolgedessen können Container innerhalb eines Pods miteiander kommunizieren. Die Koordination der Kommunikation zwischen verschiedenen Pods wird von Kubernetes selbst verwaltet [25]. Soll ein Zugriff auf die Ressourcen von außerhalb erfolgen, so müssen Services eingesetzt werden. Diese ermöglichen außerdem ein *Load Balancing*, also eine Lastverteilung zwischen den Pods.

2.4.9 Services

Ein Service beschreibt eine logische Gruppe von Pods und die Regeln für den Zugriff auf diese. Der Aufbau eines Services kann anhand eines Beispiels in Listing 2.3 nachvollzogen werden. Dabei wird die Zusammensetzung der Gruppe von Pods über einen Selektor festgelegt. Er basiert auf Schlüssel-Wert-Paaren (z.B. *app: gossip*), die den Pods zugewiesen sind. Wenn ein Service konfiguriert wird, verwendet er den Selektor, um die Pods auszuwählen [102]. Im Beispiel wird ein Zugriff über TCP und gRPC ermöglicht.

In Kubernetes können verschiedener Service Typen definiert werden [102]. Ein als *ClusterIP* definierter Service macht ihn nur von innerhalb des Clusters erreichbar. Legt man einen Service jedoch mit dem Typ *NodePort* an, so ist er über jede Node-IP auf einem spe-

```yaml
apiVersion: v1
kind: Service
metadata:
  app: gossip
  name: node-service
spec:
  type: ClusterIP
  selector:
    app: gossip
    simulation: test
  ports:
  - name: tcp
    port: 90
  - name: grpc
    port: 50051
```

Listing 2.3: Beispiel Service

ziellen Port erreichbar. Dieser Port wird zufällig zugewiesen oder kann explizit gewählt werden, wobei der Bereich zwischen 30000 und 32767 liegt. Somit kann über einen Service vom Typ NodePort auch ein Zugriff von außerhalb des Clusters erfolgen. Alternativ dazu kann eine Definition als *LoadBalancer* erfolgen, wodurch der Service eine Anwendung zugeordnet bekommt, welche die Lastverteilung übernimmt. Schließlich kann durch Zuordnung des *ExternalName-Typs* dem Service ein *DNS-Name* zugewiesen werden. Hierbei realisiert das Kubernetes *Domain Name System (DNS)* die Namensauflösung. Dazu ordnet es den IP-Adressen der Dienste im Cluster sprechende Namen zu.

2.4.10 Kubernetes Operator

Kubernetes eignet sich hervorragend für die Verwaltung zustandsloser Anwendungen. Die Handhabung zustandsabhängiger Anwendungen (*stateful Applications*) ist hingegen meist sehr komplex, da hier individuelle Verwaltungsansätze erforderlich sind. Ein Kubernetes Operator erweitert das Standardverhalten eines Clusters, indem eine maßgeschneiderte Logik für den jeweiligen Workload bereitgestellt wird. Dadurch wird die Verwaltung von zustandsabhängigen Anwendungen möglich. Der Aufbau eines Operators ist in Abbildung 2.8 anhand des Simulation Operators dargestellt. Hierbei gibt es drei Hauptkomponenten [121], welche im Folgenden erläutert werden:

- **Roles und ServiceAccounts:** *Role-Based Access Control (RBAC)* ist ein Sicherheitsmechanismus, der den Zugriff auf Ressourcen reguliert. Mit den Rollen können Berechtigungsprofile definiert werden, während die Serviceaccounts Identitäten für Pods bieten. Die Serviceaccounts werden dabei zur Authentifizierung gegenüber dem API-Server verwendet [27]. Über die RoleBindings kann eine Verküpfung dieser mit Benutzern beziehungsweise Gruppen erfolgen. Kubernetes ermöglicht über RBAC eine feingranulare Rechtegestaltung für beliebige Ressourcen. So kann es realisiert werden, dass der Operator die Rechte erhält, Kubernetes-Objekte zu verwalten. Beispielsweise können so Pods durch den Operator erzeugt, modifiziert und gelöscht werden.
- **Operator Deployment:** Der Operator wird als Container-Anwendung implementiert und dann innerhalb eines Pods bereitgestellt und ausgeführt. Das Deployment beinhaltet die folgenden zwei Komponenten:
  - **API:** Die API kann eine oder mehrere Custom Resource Definitions (CRD) konfigurieren. Ihre Hauptaufgabe ist die Überwachung und Analyse der Custom

Resources. Dabei fängt sie Ereignisse wie das Erstellen, die Modifikation oder Entfernung von Ressourcen ein. Anschließend wird der Controller über die jeweiligen Events informiert.

– **Controller:** Der Controller reagiert auf die von der API erkannten Ereignisse. Anschließend realisiert er dann den jeweiligen Workflow. Dabei hängen die spezifischen Aktionen, die der Operator ausführt, von den Anforderungen der Anwendung ab. So kann der Operator beispielsweise Container-Pods starten, Konfigurationen erstellen und Ressourcen zuweisen.

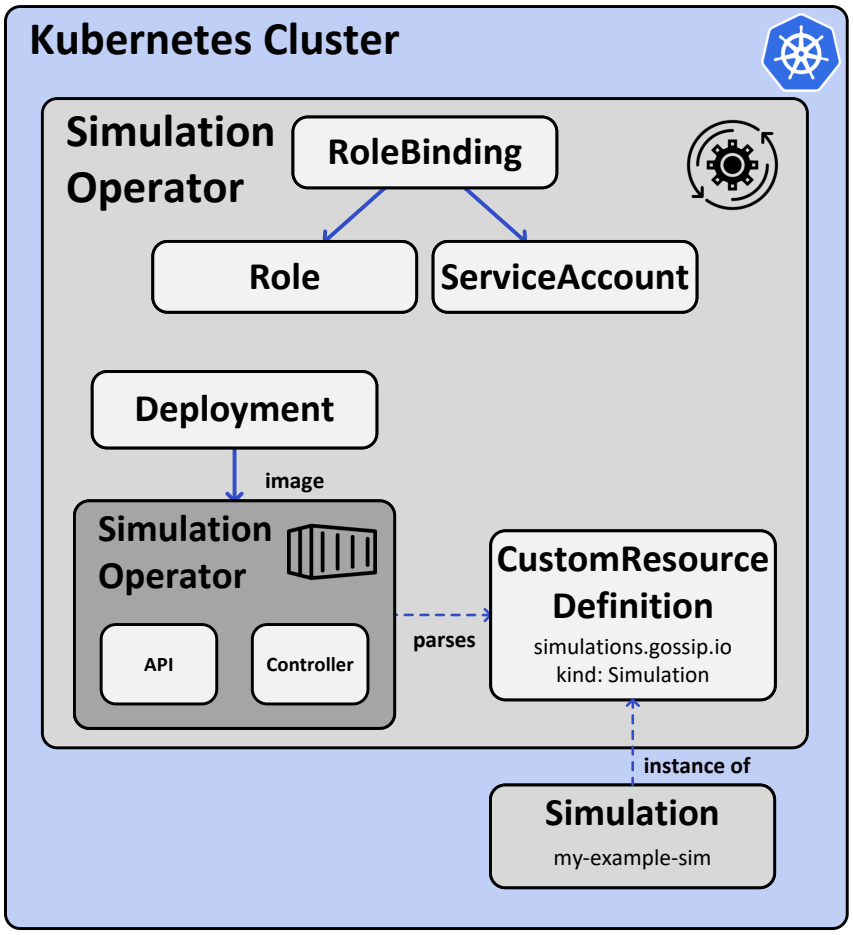


Abbildung 2.8: Aufbau eines Kubernetes Operators (angelehnt an [121])

Durch den Operator können Custom Resources, also eigene Kubernetes-Objekte und deren Komponenten erstellt und verwaltet werden. Ebenfalls können Controller bereitgestellt werden, die anwendungsspezifische Aufgaben automatisieren. Diese Controller interagieren mit dem API-Server, um die Custom Resources zu überwachen und deren Lebenszyklus zu verwalten. Man spricht bei den Ressourcen, die durch den Operator bereitgestellt werden, auch von *Operanden*. Ein Operator knüpft an die Ereignissteuerung an, um auf die Erstellung, Änderungen und das Entfernen von Custom Resources zu reagieren [89]. Die Änderungen an den Custom Resources führen zu entsprechenden Änderungen an den Operanden. Durch die Reaktion auf Änderungen können beispielsweise Selbstheilungsfunktionen ermöglicht werden. Wird ein bestimmter Anwendungsstatus festgestellt, so können automatisch Korrekturmaßnahmen durchgeführt werden.

Ein Operator stellt einen Workload anders bereit als dies ein Administrator über das CLI tut [121]. Über das CLI kann der Admin direkt mit der Kubernetes-API interagieren.

Er definiert hierbei stets den gewünschten Cluster Zustand, z.B. durch Hinzufügen eines Deployments. Die Kubernetes-Controller realisieren dann diesen Zustand, indem sie die entsprechenden Pods erzeugen. Der Operator hingegen läuft als Pod auf einem Worker Node. Er wird erst aktiv, wenn eine Custom Resource seines verwalteten Definitionstyps erstellt, modifiziert oder gelöscht wird. Diese Änderung kriegt er über die Operator-API mit. Der Operator-Controller führt dann seine Logik aus, wodurch der Zielzustand von Kubernetes-Objekten über die Server-API festgelegt wird. Der weitere Ablauf ist für Operator und Administrator analog.

Die erzeugten Kubernetes-Objekte werden vom Operator anhand der Custom Ressource verwaltet. Infolgedessen bleibt die Komplexität der Definition und Konfiguration der jeweiligen Objekte dem Administrator verborgen. Durch das Operator-Pattern wird eine höhere Abstraktion ermöglicht und das Management von komplexen Anwendungen realisierbar [89]. Der Kubernetes Operator befähigt Entwicklern, sich auf die Anwendungslogik und nicht auf Details der Infrastruktur konzentrieren zu können. Zudem bieten Operatoren durch die frei definierbaren Custom Resources deklarative Spezifikationen für die Anwendungsverwaltung. Dadurch wird es Benutzern ermöglicht, die gewünschten Zustände zu definieren und die Implementierungsdetails dem Operator zu überlassen. Insbesondere zustandsbehaftete Anwendungen wie Datenbanken, Message Queues und Big-Data-Frameworks, die über viele Abhängigkeiten verfügen, können so effizient verwaltet werden.

Außerdem greift ein Operator auf eine andere Weise in den Reconcilation-Loop ein als ein Kubernetes-Controller [121]. Die Kubernetes-Controller werden im Control Plane ausgeführt, somit laufen sie auf dem Master Node direkt im Cluster-System. Die Operator-Controller hingegen werden als Pods direkt auf den Worker Nodes ausgeführt. Darüber hinaus stößt der Manager die Reconcilation im Control Plane an. Dabei erfolgt zuerst die Information der Kubernetes- und anschließend der Operator-Controller. Durch die höhere Abstraktion der Operator-Controller benötigen diese einen zusätzlichen Transformationsschritt. CRDs werden zuerst in eingebaute Typen wie Deployments oder Services transformiert, welche dann im zweiten Schritt in die kleinsten Bausteine umgewandelt werden. Das unterschiedliche Vorgehen zur Erreichung des Zielzustands ist in Abbildung 2.9 dargestellt. Dabei wird das Verhalten des Operators mit dem Standarvorgehen von Kubernetes verglichen. Hier ist ebenfalls der Reconcilation-Loop dargestellt.

Das Operator Framework kann genutzt werden, um flexibel eigene Kubernetes Operator zu entwerfen. Hierbei handelt es sich um ein Open-Source-Projekt, welches ein Software Development Kit (SDK) beinhaltet. Es wird ein Set von Tools bereitgestellt, welches Implementierung, Test und Bereitstellung von Operatoren vereinfacht. Zudem implementiert das Framework eine Controller-Laufzeitumgebung. Dadurch realisiert es die Ereignissteuerung, Reconcilation und die Interaktion mit dem API-Server. Da die Kubernetes-Kernfunktionalität abstrakt verwendet werden kann, können sich Entwickler auf die Implementierung der Anwendungslogik des Operators konzentrieren.

2.4.11 Minikube

Minikube ist eine Implementierung eines Kubernetes-Clusters, die als lokale Testumgebung zum Entwickeln und Experimentieren dienen soll [118]. Minikube führt ein Single-Node-Cluster innerhalb einer virtuellen Maschine aus. Es unterstützt die meisten Kubernetes Features, wodurch eine umfangreiche Emulation ermöglicht wird. Gleichzeitig bleibt das Aufsetzen eines ressourcenintensiven Rechner-Clusters erspart.

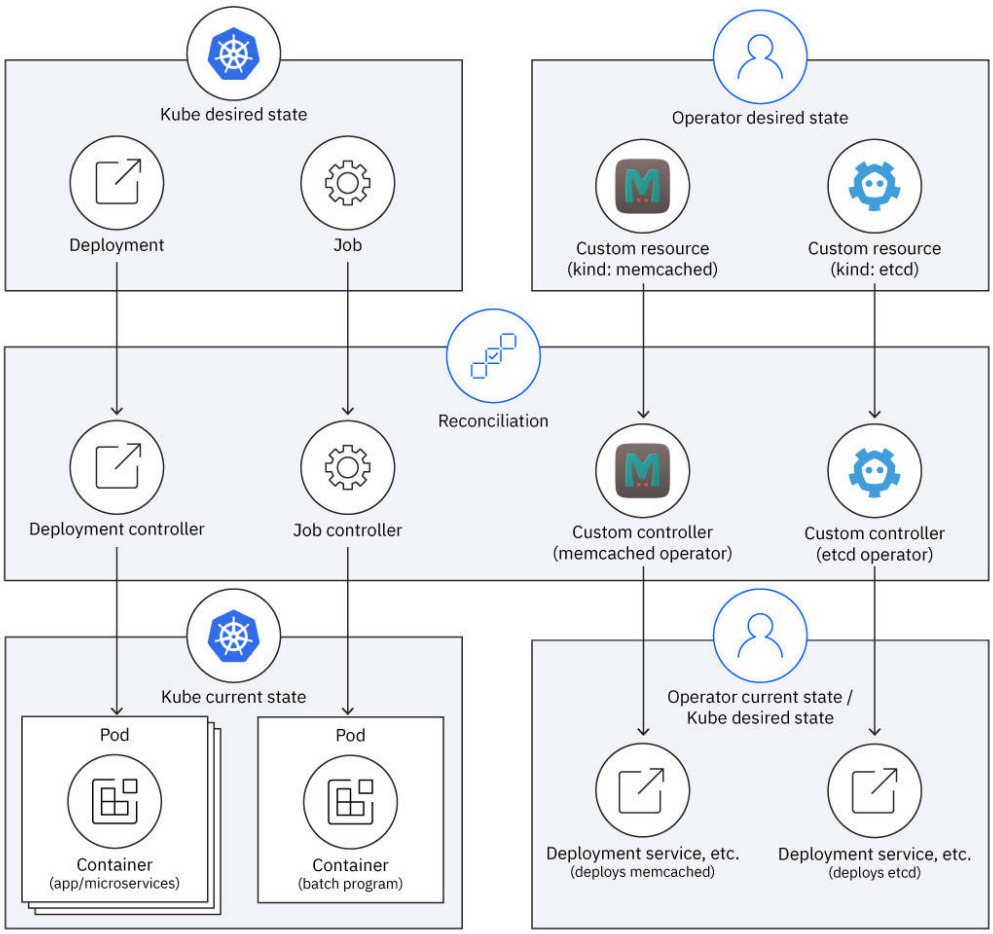


Abbildung 2.9: Operator und Controller Reconcilation [121]

2.5 MinIO Object Storage

MinIO ist eine Kubernetes-native *Object Storage-Lösung* zur Speicherung von unstrukturierten Daten [77] Die Lösung stellt eine *API* bereit, die *Amazon S3* kompatibel ist [79]. Amazon Simple Storage Service (Amazon S3) zählt zum Umfang der *Amazon Web Services (AWS)* [24]. Hierbei handelt es sich um eine skalierbare Object Storage-Lösung, welche über einen *Web Service* genutzt werden kann. MinIO stellt außerdem einen Server, einen Client und ein *SDK* bereit. Der Server ist so konzipiert, dass er hardwareunabhängig und skalierbar ist. Damit ist er optimal für das Deployment als Container geeignet. Er lässt sich somit auch durch Orchestrierungssoftware wie Kubernetes bereitstellen und verwalten [78]. Darüber hinaus steht das SDK in vielen Programmiersprachen bereit und macht den Zugriff auf den Object Storage für Anwendungen möglich. Man kann die angelegten Objekte über das SDK auslesen oder über das bereitgestellte Webinterface betrachten. Des Weiteren ermöglichen Object Storages das Speichern von Objekten beliebiger Art. Im Gegensatz zu herkömmlichen Dateisystemen gibt es bei der Objektspeicherung keine Beschränkungen für Dateitypen oder -formate. So können zum Beispiel Bilder, Animationen und Videos direkt gespeichert werden. Ebenso können Konfigurationen oder Ergebnisse in strukturierter Form, z.B. als JSON-Dateien persistiert werden.

JavaScript Object Notation (JSON) ist ein menschenlesbares, leichtgewichtiges Datenformat zum Austausch von Informationen zwischen Anwendungen. Es besteht aus einer Sammlung von Schlüssel-Wert-Paaren, welche auch in Listen beziehungsweise Feldern

organisiert sein können [60]. JSON findet Einsatz bei der Serialisierung, also der Abbildung von strukturierten Daten in ein serielles Format. So kann es zum Beispiel verwendet werden, um Objekte in einem Object Storage als Dateien abzulegen. Client-Anwendungen müssen die Serialisierung durchführen. Bei erneutem Zugriff können die Dateien von den Clients wieder in die ursprünglichen Objekte deserialisiert werden.

2.6 Vorangehende Arbeiten zur dezentralen Community Detection

Im Folgenden werden zwei Arbeiten besprochen, die Verfahren zur dezentralen Erkennung von Community-Strukturen vorgestellt. Mit diesen können Knoten eine lokale Sicht der Netzwerktopologie erhalten. Es werden Techniken präsentiert mit denen Wissen über die direkten Nachbarn gewonnen werden kann. Als Folge kann eine Optimierung der Partnerwahl beim Gossiping erfolgen.

2.6.1 Decentralized Cluster Detection in Distributed Systems Based on Self-Organized Synchronization

Im Jahr 2016 wurde das Paper [106] von Vikramjit Singh, Markus Esch und Ingo Scholtes veröffentlicht. Thema der Arbeit ist die Konzeption und Implementierung einer Methode zur dezentralen Detektion der Clusterstruktur in unstrukturierten Netzwerken. Knoten in solchen Netzwerken haben keine globale Sicht auf den topologischen Aufbau. Aus diesem Grund können viele populäre Verfahren wie zum Beispiel heuristische Optimierungsalgorithmen (Louvain-Methode, Modularitätsmaximierung), agglomerative oder divisive Algorithmen, spektrale Methoden sowie statistische Inferenztechniken (stochastisches Blockmodell) nicht eingesetzt werden. Diese Methoden nutzen alle globale Informationen über die Struktur des Netzwerkes, welche in dezentralen Anwendungsszenarien nicht erhoben werden können.

Mit dem vorgeschlagenen Verfahren wird es jedoch möglich, auch in dezentralen Anwendungsfällen Community-Strukturen zu erkennen. Nachdem diese ermittelt wurden, können sie für verschiedene Zwecke eingesetzt werden. Hierbei ist die Optimierung von Gossiping-Verfahren eine zentrale Anwendungsmöglichkeit, welche sich vor allem für unstrukturierte Netzwerke anbietet. Das Wissen über Community-Strukturen kann hier zur Beschleunigung der Konvergenz verwendet werden. Dadurch kann die Verbesserung der Skalierbarkeit und Effizienz der Gossip-Algorithmen erfolgen. Im Folgenden wird auf das dargestellte Verfahren zur dezentralen Detektion der Clusterstruktur sowie die daraus folgenden Erkenntnisse eingegangen.

Die vorgestellte Methode basiert auf dem *Kuramoto-Modell*. Hierbei handelt es sich um ein zeitkontinuierliches Modell für die selbstorganisierte Synchronisation gekoppelter Phasenoszillatoren. Das Kuramoto-Modell ermöglicht es verschiedene Rückschlüsse aus den oszillierenden Signalen zu ziehen. Das Modell wurde hier angewandt, um ein zeitdiskretes, verteiltes Synchronisationsprotokoll zu entwickeln. Dabei werden vor allem in den Anfangsstadien der Synchronisation die durchgeführten Kommunikationen beobachtet. Die Analyse des Synchronisationsprozesses ermöglicht es Rückschlüsse auf die Netzwerktopologie zu ziehen. Aus diesem Wissen können die Knoten dann die Community-Mitgliedschaften ihrer nächsten Nachbarn ableiten.

Eine erste Version des Protokolls wurde bereits in der vorangehenden Arbeit „Epidemic Self-synchronization in Complex Networks“ im Jahr 2012 vorgestellt. Hier arbeiteten Ingo Scholtes, Jean Botev, Markus Esch und Peter Sturm an einer epidemischen selbstorganisierten Synchronisationsroutine. Die Routine führt einen periodischen Austausch der

Phasensignale zwischen Nachbarknoten durch. Während der Kommunikation verrechnet ein Knoten sein eigenes Signal mit dem seines Nachbarn. Daraufhin kann der Knoten die Signaldifferenzen berechnen. Anhand der Differenzen kann der Knoten erkennen, welche Nachbarn sich in der eigenen Community befinden. Dazu wird das Wissen verwendet, dass Knoten innerhalb desselben Clusters schneller synchronisieren als Knoten in verschiedenen Clustern. Ein Clustering erfolgt anschließend durch die Berechnung der Area under the Curve (AUC) der Signaldifferenzen. Es wird angenommen, dass die Mehrheit der Nachbarn eines Knotens im gleichen Cluster sind. Liegt der AUC-Wert nahe am Mittelwert, so gehört der Nachbar der gleichen Community an. Ist eine Abweichung über einem bestimmten Schwellenwert, so gehört der Knoten zu einer anderen Community.

Der Kommunikationsaufwand der Synchronisation bleibt konstant, da jeder Knoten nur periodisch Phasensignale mit seinen Nachbarn austauschen muss. Eine experimentelle Bewertung der Methode wurde mit synthetisch erzeugten Netzwerken durchgeführt. Dabei wurde die Erkennung der Communities mit der Community-Struktur verglichen, die durch Anwenden der Louvain-Methode entsteht. Bei Netzwerken mit einer Modularität über 0.6 erreichte die Erkennungsrate etwa 80 Prozent. Zudem nahm die Genauigkeit weiter mit steigender Netzwerkmodularität zu. Die genauen Ergebnisse sind in Abbildung 2.10 visualisiert. Hier ist die Performanz der Methode auf Netzwerken mit Modularitätswerten zwischen 0.2 und 0.9 in Abhängigkeit von der AUC-Schwelle dargestellt. Der Ansatz zeigte somit eine vielversprechende Erkennungsgenauigkeit.

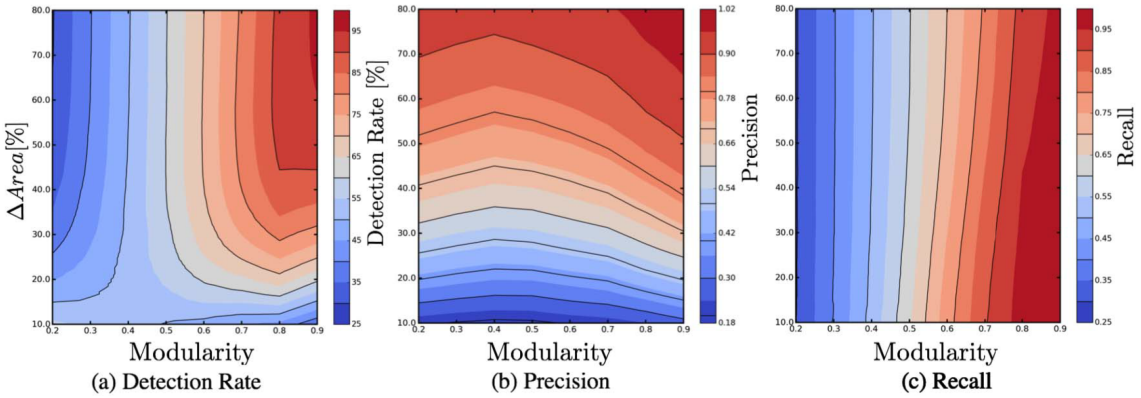


Abbildung 2.10: Erkennungsgenauigkeit in Abhängigkeit der Modularität [106]

2.6.2 Decentralized Community Detection in Unstructured Networks

Das Systemtechniklabor der Hochschule für Technik und Wirtschaft des Saarlandes veröffentlicht ausgewählte technische Berichte. In diesem Rahmen wurde auch die Thesis „Decentralized Community Detection in Unstructured Networks“ [100] von Niklas Schütz publiziert. Diese Arbeit befasst sich mit der Entwicklung, Implementierung und Bewertung von neuen Synchronisationsmethoden für die Erkennung von Communities. Sie baut somit auf den Erkenntnissen der im vorigen Kapitel beschriebenen Arbeit auf. Es werden neue Methoden entwickelt, die auf Variationen des Potts-Modells und Verbesserungen des Kuramoto-Modells basieren. Der erste Ansatz orientiert sich an dem ursprünglichen Verfahren aus dem Paper von Singh[106]. Hier berechnet und vergleicht jeder Knoten die AUC-Werte seiner Nachbarn. Wenn die Werte eines Nachbarn innerhalb von einem bestimmten Prozentsatz des Mittelwerts liegen, gilt er als Mitglied der gleichen Community. Bei der zweiten Methode wird ein neuer Ansatz verwendet, wobei die Differenz der Werte an einem bestimmten Punkt berechnet wird. Beide Varianten erlauben es, je nach

Netzwerktopologie Rückschlüsse auf das Vorhandensein von Communities im Netz zu ziehen. Dabei unterscheiden sie sich jedoch in ihren Synchronisationsgeschwindigkeiten.

Die Arbeit legt einen erheblichen Schwerpunkt auf die Simulation der neuen Methoden und deren Bewertung. Für die Simulation erfolgt der Aufbau des Netzwerks anhand eines Graphen. Dabei wird für jeden Knoten eine Anwendung gestartet, welche die Synchronisationsfunktionalität bereitstellt. Die Ausführung wird über ein Kubernetes-Cluster realisiert, wobei die Anwendungen als Container ausgeführt werden. Abbildung 2.11 stellt den Aufbau des entwickelten Systems dar. Die Software kann dementsprechend in drei Hauptteile unterteilt werden:

- Der erste Teil ist die *Netzwerkerstellung und -konfiguration*. Zuerst werden die Netzwerke auf Grundlage von Graphmodellen erzeugt. Anschließend erfolgt die Konfiguration der Knoten. Die Netzwerke werden dabei mitsamt ihrer Knotenkonfiguration durch entsprechende Dateien abgebildet.
- Der zweite Teil ist die *Simulation*, bei der die Knoten des Netzwerks in einem Kubernetes-Cluster ausgeführt werden. Die Initialisierung der Knoten wird von einem Controller-Dienst anhand der Konfigurationsdatei durchgeführt. Sie führen dann eine Synchronisierung entsprechend der ausgewählten Methode durch. Dabei protokollieren sie kontinuierlich ihren aktuellen Zustand. Diese Protokolle werden zur späteren Auswertung von einem Logstorage-Dienst gesammelt.
- Der letzte Teil ist die *Community Detection*. Sie wird zentral von Skripten unter Verwendung der vom Logstorage Service gesammelten Logs durchgeführt. Zur Auswertung der Protokolle werden zwei Methoden verwendet, nämlich eine AUC-Berechnung und eine Wertedifferenz zu einem bestimmten Zeitpunkt. Hierbei werden ebenso zwei Methoden zur Verbesserung der Community Detection vorgestellt.

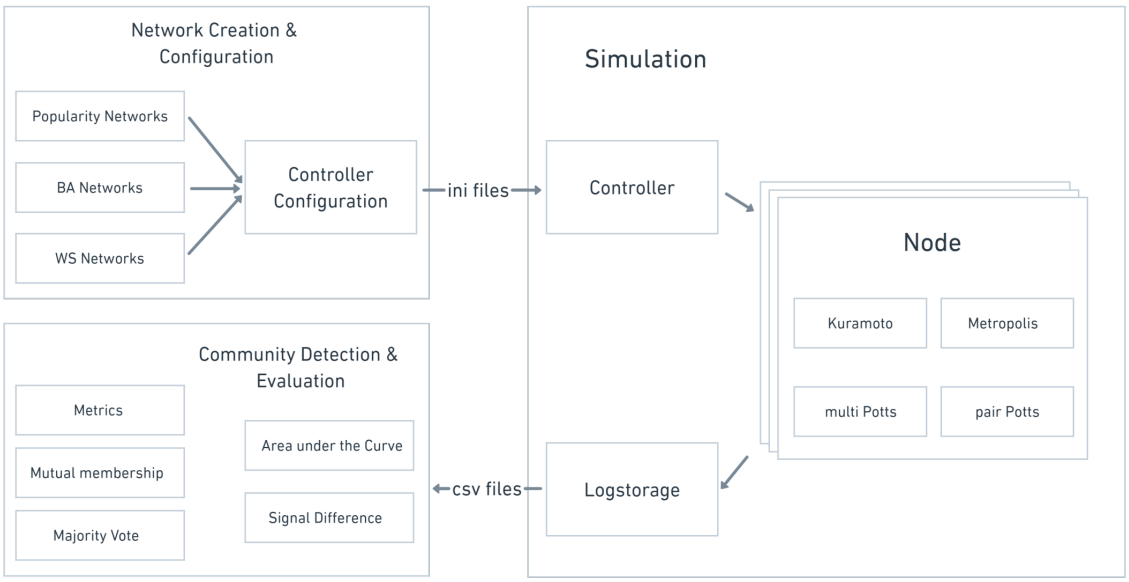


Abbildung 2.11: Systemaufbau für Synchronisationsmodelle und Community Detection [100]

Das System ermöglicht es, Modelle zur Erkennung von Communities in größeren Netzwerken zu untersuchen. Dabei können Experimente mit unterschiedlichen Synchronisationsmethoden auf wiederum verschiedenen Netzwerktopologien durchgeführt werden. Diese Flexibilität wird unter anderem durch die Verwendung eines Kubernetes-Clusters

und die damit verbundenene Entkopplung von Netzwerkerstellung, Simulation und Evaluation erreicht.

Die durchgeführten Simulationsreihen zeigten, dass die Wahl des Synchronisationsalgorithmus die Leistung der Community Detection nur um weniger als 5 Prozent beeinflusst. Der Hauptunterschied zwischen den Synchronisationsmethoden liegt in ihrer Eignung für verschiedene Netzwerke mit unterschiedlichen Parametern. Es bleibt eine Herausforderung, eine optimale Konfiguration zu finden. Keine Methode wurde gefunden, die unabhängig von der Größe und Struktur des Netzwerks eine Synchronisierung mit der gleichen Geschwindigkeit ermöglicht.

Während in der verwandten Arbeit Synchronisationsmethoden untersucht werden, zielt diese Arbeit auf die Evaluation von Gossiping-Verfahren ab. Dennoch gibt es Parallelen, was die Prozessgliederung in Netzwerkerzeugung, Simulation und Evaluation betrifft. Zudem ist die Zielplattform beider Projekte ein Kubernetes-Cluster, was die Entwicklung containerisierter Anwendungen mit sich bringt. Es konnte sich bei der Lösungsfindung an einigen Aspekten des Systemaufbaus der verwandten Arbeit orientiert werden. Hierbei muss beachtet werden, dass der Simulationsaufbau trotz ähnlicher Semantik architekturell sehr unterschiedlich ist. Diese Thesis verwendet einen Kubernetes Operator als Kernstück der Simulationsumgebung. Dabei wird die Durchführung von Simulationen durch einen Simulation Runner Service kontrolliert. Hingegen verwendet die Arbeit von Schütz einen Controller-Service zum Auf- und Abbau der Simulationen. Der Operator realisiert eine automatisierte Verwaltung der Simulationen, wodurch dynamischere Workflows umgesetzt werden können. Ebenfalls sinkt der operative Aufwand bei der Erstellung, Ausführung und dem Abschluss von Simulationen auf ein Minimum. Zu weiteren Unterschieden zählen die Grapherzeugung sowie sowie die betrachteten Graphytpen. Für diese Arbeit sind insbesondere unstrukturierte komplexe Netzwerke zu untersuchen, da hier ein großes Optimierungspotential für Gossiping-Protokolle besteht. Gleichermaßen bietet es sich an, verschiedene Arten von skalenfreien Netzwerken sowie zu untersuchen.

Verfahren zur Community Detection in unstrukturierten Netzwerken sind für diese Arbeit als Voraussetzung zu sehen. Sie gewährleisten die Verfügbarkeit der benötigten topologischen Informationen. In unstrukturierten Netzwerken stehen den Knoten meist keine globalen Daten zur Verfügung, da zentrale Steuerungseinheiten fehlen. Bei den dezentralen Ansätzen sammelt jeder Knoten im Netzwerk Informationen über seine direkten Nachbarn. Anschließend können die Knoten diese verwenden, um die Community-Zugehörigkeit für sich und ihre Nachbarn zu bestimmen. Nachdem die Community-Strukturen ermittelt wurden, können sie eingesetzt werden, um beispielsweise das Gossip-Verhalten im Netzwerk zu optimieren. Für die Auswahl des Gossip-Partners reichen die lokalen Informationen über die direkten Nachbarn aus. Um eine optimierte Informationsausbreitung zu erzielen, werden statt einer zufälligen Weiterleitung die Community-Strukturen berücksichtigt. So kann je nach Ziel die Kommunikation innerhalb der Communities oder zwischen Communities verstärkt werden, um eine lokale oder globale Konvergenz zu beschleunigen.

2.7 Kommunikationsprotokolle

Kommunikationsprotokolle stellen die wesentliche Grundlage für den Informationsaustausch in Computernetzwerken dar. Sie definieren die Regeln und Verfahren in Bezug auf die Übertragung und Interpretation von Daten. Im Laufe der Jahre wurden zahlreiche Kommunikationsprotokolle entwickelt, die für jeweils spezifische Anwendungsfälle optimiert sind. *Transmission Control Protocol (TCP)* wurde bereits in den 1970er Jahren

entwickelt und ist auch heute noch ein Kernprotokoll des Internets. In den letzten Jahren ist mit dem Aufkommen von hoch-verteilten Systemen eine neue Generation von Kommunikationsprotokollen entstanden. *Google Remote Procedure Call (gRPC)* zählt zu diesen neuen Protokollen und erfüllt die Anforderungen, die moderne Systeme mit sich bringen.

2.7.1 Transmission Control Protocol

TCP ist ein weit verbreitetes Protokoll der Transportschicht in Netzwerken. Es ist das Kernstück vieler Internetanwendungen und wird unter anderem in Browsern, Netzwerkdiensten wie E-Mail oder zur Dateiübertragung verwendet. Das Protokoll wird als verbindungsorientiert bezeichnet, da es zuverlässige, geordnete und fehlergeprüfte Zustellung von Datenpaketen über Internet Protocol (IP)-Netze gewährleistet. Zuerst wird dabei eine Verbindung zwischen zwei Hosts hergestellt. Hierbei spricht auch vom Aufbau einer Session. Anschließend beginnt die Datenübertragung zwischen den Kommunikationspartnern. Die Session endet erst, wenn alle Daten erfolgreich zugestellt wurden. TCP garantiert hierbei, dass die Pakete alle in der richtigen Reihenfolge ankommen. Falls notwendig, kann es auch die erneute Übertragung verlorener Pakete übernehmen.

2.7.2 Google Remote Procedure Call

gRPC ist ein Open-Source-Framework. Es realisiert eine effiziente und sichere Kommunikation zwischen verteilten Anwendungen über *Remote Procedure Calls (RPC)*. Ein RPC ermöglicht es einem Client, eine Prozedur eines Servers auf einem entfernten System aufzurufen, als wäre es ein lokaler Aufruf. Im Folgenden wird der Funktionsumfang von gRPC erläutert:

- gRPC nutzt das *HTTP/2-Protokoll* zum Nachrichtentransport. HTTP/2 bietet zahlreiche Verbesserungen gegenüber seinem Vorgänger HTTP/1. Neue Funktionen wie Multiplexing, Header-Kompression und Server-Push werden einführt. Zudem wird die Ressourcennutzung effizienter und Latenzzeiten werden geringer, wodurch die Gesamtleistung steigt [69].

- Unter *Protocol Buffers* (kurz *protobuf*) ist ein Datenserialisierungsformat zu verstehen. Es ist unabhängig von der verwendeten Programmiersprache [59]. gRPC verwendet protobuf als *Interface Definition Language (IDL)*. Es ermöglicht Entwicklern, die Struktur ihrer Daten und Dienste in einem eindeutigen und für Menschen lesbaren Format festzuhalten. Protobuf bietet Funktionen wie effiziente Serialisierung, Abwärtskompatibilität und die Möglichkeit, Code für mehrere Programmiersprachen zu erzeugen. In der Praxis kommt heutzutage meist *proto2* oder auch das neuere *proto3* zum Einsatz. Konkret kann es verwendet werden, um Schnittstellen und Nachrichten zu definieren. Diese werden zwischen den Komponenten des verteilten Systems ausgetauscht. Die Definitionen dienen als Vertrag zwischen Client und Server. Sie legen für beide Seiten die Dienstmethoden, Nachrichtentypen und ihre jeweiligen Felder fest. Mit dem *Protocol Buffer Compiler* kann Client- und Servercode in verschiedenen Programmiersprachen aus den Definitionen generiert werden. Diese Codegenerierung vereinfacht den Entwicklungsprozess und gewährleistet eine konsistente und interoperable Kommunikation. Die Interoperabilität wird unter anderem auch durch die Verwendung von *Stubs* ermöglicht. Diese verhalten sich wie Proxies, sind also Platzhalter für die eigentliche Implementierung, welche die Komplexität der Netzwerkkommunikation abstrahieren. Die Stubs beinhalten clientseitige Methoden, die den Dienstmethoden in der protobuf-Definition entsprechen. Stub-Methoden können anschließend wie normale Funktionsaufrufe verwendet

werden. Der Stub ist hierbei verantwortlich für die Serialisierung der Parameter, das Senden der Request und die Bearbeitung der Antwort. Die Implementierungen der Remote-Prozeduren sind auf dem Server zu finden. Diese haben den gleichen Namen und die Parameterstruktur wie die Stub-Methoden und können so ausgeführt werden [69].

- Weitere Features umfassen die Flusskontrolle, wellche die Datenübertragung verwaltet und eine optimale Ressourcennutzung sicherstellt. Ebenso bietet gRPC Echtzeitkommunikation, Authentifizierungs- und Sicherheitslösungen, Load Balancing und weiteres an.

Der Ablauf, wie ein gRPC-Aufruf über das Netzwerk verarbeitet wird, ist zusätzlich in Abbildung 2.12 dargestellt.

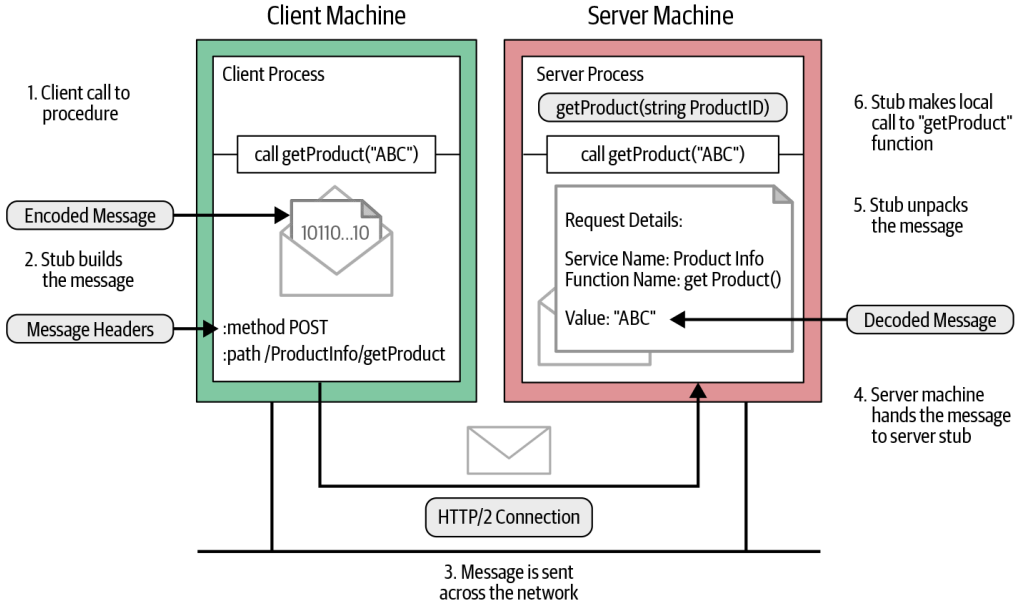


Abbildung 2.12: Ablauf gRPC-Aufruf über Netzwerk [69]

2.8 Python-Bibliotheken

Im Folgenden werden spezielle Python-Bibliotheken vorgestellt. Hierbei wird sich auf die umfangreichen Bibliotheken begrenzt, die große Aspekte der Anwendungslogik abbilden. Von zentraler Bedeutung ist *NetworkX*, welches den programmtechnischen Umgang mit Graphen realisiert. Ebenfalls wird *PyGraphViz* verwendet, um hochwertigere Visualisierungen zu erzeugen. Das *Kopf Framework* erleichtert die Implementierung eines Kubernetes Operators.

2.8.1 NetworkX

NetworkX ist eine Python-Bibliothek für die Erstellung, Bearbeitung und Analyse komplexer Graphen beziehungsweise Netzwerke [55]. Die Bibliothek ist sehr populär aufgrund der vielseitigen Anwendungsmöglichkeiten, dem großen Funktionsumfang und der Nutzerfreundlichkeit. Sie stellt einen umfassenden Umfang an Werkzeugen bereit, wodurch unter anderem verschiedenste Arten von Graphen erzeugt werden können. Des Weiteren ermöglicht NetworkX die Bearbeitung von Graphen. So können zum Beispiel

beliebig Knoten und Kanten hinzugefügt, modifiziert und entfernt werden. Ebenso sind viele Funktionen zur Analyse von Graphen verfügbar. Es können zum Beispiel zahlreiche Netzwerkmetriken berechnet werden. Auch komplexere Analysealgorithmen aus Bereichen wie Clustering, Zentralitätsanalyse und Community-Detection können ausgeführt werden. Außerdem unterstützt NetworkX verschiedene Arten der Darstellung von Graphen.

2.8.2 PyGraphViz

PyGraphViz ist eine Python-Schnittstelle für die Visualisierungssoftware Graphviz [131]. Graphviz ist ein leistungsstarkes Werkzeug zur Visualisierung von Graphen. Die Schnittstelle bietet Funktionalität, um Darstellungsobjekte zu erzeugen. Diese verfügen über vielseitige Möglichkeiten zur Definition und Anpassung der Visualisierung. Auch die Darstellung von großen Netzwerke mit hoher Komplexität kann erfolgen. Dazu kann zwischen verschiedenen Layout-Techniken ausgewählt werden. Somit wird die Bibliothek verwendet, um hochwertige Graphvisualisierungen zu automatisieren und in Python-Anwendungen zu integrieren.

2.8.3 Kopf Framework

Das Kopf Framework ist eine Python-Bibliothek, mit der Kubernetes Operatoren erstellt werden können [66]. Ein Kubernetes Operator ist eine Software-Erweiterung, welche die Verwaltung komplexer Anwendungen auf Kubernetes-Clustern automatisiert. Kopf vereinfacht die Entwicklung von Operatoren, indem es einen deklarativen Ansatz bietet. Hierbei können Custom Resources direkt über den Code definiert werden. Ebenso erfolgt die Bereitstellung einer Reihe von Dienstprogrammen und Konventionen. Es wird der Reconciliation Loop unterstützt, welcher sicherstellt, dass der aktuelle und gewünschte Zustand der Ressourcen übereinstimmt. Bei Fehlern können Mechanismen zu deren Behandlung und Wiederherstellung von ursprünglichen Zuständen verwendet werden. Des Weiteren wird die Erstellung von benutzerdefinierte Controller möglich. Diese Controller überwachen den Zustand der Kubernetes-Ressourcen und senden Ereignisse, sobald bestimmte Bedingungen eintreten. Dazu interagiert Kopf eng mit dem Operator Framework und dem Kubernetes-API-Server. Entwickler können festlegen, wie diese Controller auf Änderungen der Ressourcen reagieren sollen. Infolgedessen wird es möglich dynamisch Workflows und Automatisierungsroutinen abzubilden.

3 Verwandte Arbeiten

Diese Thesis verbindet verschiedene Fachbereiche, da sie einen komplexen Prozess abbildet, der die Erzeugung von Graphen, den Aufbau von Simulationen und deren Durchführung integriert. Im Folgenden wird auf verwandte Arbeiten aus unterschiedlichen Bereichen eingegangen. In diesem Kontext werden Forschungsarbeiten und Projekte präsentiert, die inhaltlich eng mit dieser Thesis verbunden sind. Diese verwandten Arbeiten dienen dazu, die Konzeption zu erleichtern und wichtige Erkenntnisse für die Untersuchungen bereitzustellen. Zuerst erfolgt die Beschreibung von Arbeiten, die Methoden zur dynamischen Grapherzeugung diskutieren. Anschließend wird auf Projekte eingegangen, die gewichtete Gossiping-Verfahren vorstellen. Dabei werden Parallelen und Unterschiede zur eigenen Thesis erläutert.

3.1 Generating graphs that approach a prescribed modularity

Das Paper [112] wurde von S. Trajanovski et al. im Jahr 2013 vorgestellt. Es beschreibt einen Algorithmus zur Erzeugung von Graphen, die eine Zielmodularität approximieren. Dazu wird ein Graphgenerator, der *Tunable Modularity Graph Generator (TMGG)* verwendet. Dieser konstruiert Graphen mit einer bestimmten Anzahl von Verbindungen und verschiedenen topologischen Eigenschaften. Hierbei beginnt der Algorithmus mit einem Ausgangsgraphen. Zur Veränderung der Modularität werden dann Verbindungen neu ausgerichtet. Dabei können die folgenden Transformationen durchgeführt werden:

- **Transformation 1**: Inter-Community-Links zwischen zwei Communities werden durch Intra-Community-Links innerhalb einer der Communities ersetzt.
- **Transformation 2**: Wenn es zwei Communities mit signifikant unterschiedlichen Verzweigungsgraden gibt, werden Intra-Community-Links von der Community mit mehr Kanten zur Community mit weniger Kanten verschoben.
- **Transformation 3**: Inter-Community-Links zwischen zwei Communities werden durch Intra-Community-Links innerhalb einer dritten Community ersetzt. Diese Transformation wird durch Wiederholung von Transformation 1 und 2 ausgeführt.

Die Transformationen bieten verschiedene Varianten, um die Modularität zu erhöhen. Die Modularität kann jedoch auch durch Umkehroperationen verringert werden. Dazu muss in jeder Transformation die Ersetzung der jeweils anderen Kantentypen erfolgen, also statt Inter- Intra-Community-Links und umgekehrt. Des Weiteren hat der Algorithmus drei Varianten, die sich in der Reihenfolge der Anwendung der Transformationen unterscheiden:

- **StartReplacing**: Transformation 1 wird zuerst ausgeführt. Anschließend wird, falls notwendig, mit Transformation 2 fortgeführt.
- **StartShifting**: Transformation 2 wird zuerst ausgeführt. Anschließend wird, falls notwendig, mit Transformation 1 fortgeführt.

- **Random**: In jeder Iteration wird zufällig zwischen Transformation 1 und 2 ausgewählt.

Mit der gewählten Variante ändert sich die Ausführungsdauer sowie die Genauigkeit. Ebenfalls unterscheiden sich die topologischen Eigenschaften der erzeugten Graphen. StartReplacing erreicht die Zielmodularität in der geringsten Anzahl von Iterationen. Die resultierenden Graphen haben eine vergleichsweise höhere Anzahl von Inter-Community-Links. Dabei sind erzeugte Communities meist vergleichbarer Größe. Die Variante Start-Shifting generiert tendenziell Graphen mit sehr wenigen Verbindungen zwischen Communities. Zudem haben die resultierende Graphen eine baumartige Struktur. Mit Start-Replacing wird eine geringe Laufzeit einer hohen Präzision bevorzugt. ShiftReplacing hingegen verteilt die Prioritäten genau umgekehrt, eine hohe Genauigkeit steht im Vordergrund. Die Random-Variante ist eine ausgeglichene Alternative zu den beiden anderen Verfahren.

Die Logik des Graphgenerators entspricht den Anforderungen zur Erzeugung von Graphen mit Approximation einer Zielmodularität. Der eingesetzte Algortithmus ist detailliert dargestellt, jedoch liegen keine Evaluationsergebnisse vor. Dadurch kann die tatsächliche Genauigkeit der Approximation nicht bewertet werden- Außerdem wurde der Algorithmus erst in einer späteren Projektphase entdeckt und für den Einsatz in dieser Arbeit evaluiert. Während dieser Evaluation wurde keine bestehende Implementierung des Verfahrens gefunden. Aufgrund zeitlicher Beschränkungen konnte keine eigenständige Neuentwicklung des Algorithmus mehr durchgeführt werden. Der Algorithmus stellt dennoch eine sinnvolle Erweiterung für die Grapherzeugungsroutine dieser Arbeit dar. Graphen mit einer Zielmodularität wurden für die Testreihen aus einer großen Menge von erzeugten Graphen ausgewählt. Die direkte Generierung von Graphen mit einer festen Modularität stellt hier eine zeitsparende Alternative dar.

3.2 Spectral Graph Forge: A Framework for Generating Synthetic Graphs With a Target Modularity

Das Paper [8] von L. Baldesi et al. wurde im Jahr 2019 veröffentlicht und stellt eine Methode namens *Spectral Graph Forge (SGF)* vor. Diese Methode dient der Erzeugung von modularen Zufallsgraphen, welche ähnliche Community-Strukturen wie reale Netzwerke aufweisen. Hierbei wird die Modularität als Metrik zur Charakterisierung von Community-Strukturen betrachtet. Zunächst erfolgt die Darstellung eines Graphen G in Form einer Adjazenzmatrix A. Anschließend wird die Modularitätsmatrix B aus dieser abgeleitet. Die Moduarlitätsmatrix ermöglicht es nun, die modulare Struktur des Graphen zu quantifizieren. Im nächsten Schritt nutzt die SGF-Methode die spektrale Struktur der Matrixdarstellung aus, um eine Klassifizierung der Knoten durchzuführen. Dazu wird die Adjazenzmatrix A in eine Menge von Eigenwerten λ_i und entsprechenden Eigenvektoren v_i zerlegt. Die Eigenvektoren können hierbei positive oder negative Eigenwerte annehmen. Eigenvektoren, die mit positiven Eigenwerten ($\lambda_i > 0$) assoziiert sind, beziehen sich auf Paare von *Kern-Peripherie-Strukturen*. Dabei sind Kernknoten untereinander dicht verbunden und bilden eine Community. Peripherieknoten hingegen sind weniger stark verknüpft und haben meistens nur Verbindungen zu Kernknoten. Eigenvektoren mit negativen Werten ($\lambda_i < 0$) stehen im Zusammenhang mit *Bipartitionen*. Man versteht darunter eine Partitionierung der Knoten eines Graphen in zwei disjunkte Gruppen. Die SGF-Methode bevorzugt Verbindungen zwischen Knoten mit Eigenvektorwerten entgegengesetzten Vorzeichens. Dadurch kann sichergestellt werden, dass erzeugte Graphen

sowohl Kern-Peripherie-Strukturen als auch Bipartitionen aufweisen. Dies ermöglicht die Generierung von modularen Zufallsgraphen mit vielfältigen Community-Strukturen.

Die Eigenvektoren korrelieren mit der Ausprägung der Community-Strukturen. Es ist möglich, die Eigenvektoren von B zur Identifizierung von Partitionen zu verwenden. Dabei werden die Knoten anhand der Vorzeichen der Eigenwerte in Klassen eingeteilt. Der SGF-Algorithmus umfasst mehrere Schritte:

1. Zunächst wird eine Approximation der Modularitätsmatrix mit niedrigem Rang berechnet. Ziel dabei ist es, globale Strukturinformationen zu erfassen und gleichzeitig lokale Variationen zu entfernen. Der Benutzer kann den Grad der Annäherung mit einem Parameter festlegen.

2. Anschließend wird die Approximation zurück in eine Adjazenzmatrix transformiert. Eine angenäherte Modularitäts- und Adjazenzmatrix wird erzeugt.

3. Im nächsten Schritt wird eine Normalisierungsfunktion auf die Matrixwerte angewandt. Dadurch kann sichergestellt werden, dass die Werte in den Bereich $[0, 1]$ fallen und gleichzeitigt die Struktur erhalten bleibt.

4. Schließlich erfolgt die Erzeugung eines synthetischen Graphen. Hierbei wird eine *inhomogene Bernoulli-Graphenverteilung* verwendet, um so die Entropie der synthetischen Verteilung exakt bestimmt zu können. Die Entropie ist eine Metrik, die zur Bewertung der Qualität der Zufallsgrapherzeugung verwendet wird. Sie misst den Grad der Variation und stellt somit die Vielfalt der erzeugten Graphen fest.

Die Arbeit beinhaltet ebenso eine tiefgreifende Evaluation des Verfahrens. Hierbei wird der SGF-Algorithmus anhand der Genauigkeit bei der Ausrichtung auf die Modularität bewertet. Ebenfalls geht die Zufälligkeit der entstehenden Graphen in die Bewertung ein. Es wird gezeigt, dass der Algorithmus andere moderne Techniken übertrifft.

Das Verfahren eignet sich sehr gut zur Erzeugung von Graphen mit einer gewünschten Modularität. Es ist jedoch wesentlich aufwendiger einzusetzen als andere Verfahren zur Zufallsgrapherzeugung, da zuerst spezifische Ausgangsgraphen generiert werden müssen. Tests mit der NetworkX-Implementierung des Verfahrens zeigten, dass für beliebige Ausgangsgraphen meist nicht die gewünschten Modularitätswerte erreicht werden können. Es könnte dennoch als Erweiterung in dieser Arbeit eingesetzt werden. Dazu müsste ein Algorithmus entwickelt werden, der entsprechende Ausgangsgraphen erzeugt, die wiederum durch SGF verarbeitet werden.

3.3 Geographic Gossiping and Path Averaging

Das Paper [35] von Alexandros G. Dimakis et al. aus dem Jahr 2007 stellt erstmals ein Gossiping-Verfahren vor, welches geografische Informationen zur Optimierung einsetzt. Ziel ist es, den Durchschnitt aller Sensormessungen auf verteilte und fehlertolerante Weise zu berechnen. Dadurch sollen diverse Probleme bei der verteilten Mittelwertbildung in Sensornetzen gelöst werden. Die Standardverfahren basieren auf paarweisem Austausch von Werten zwischen Knoten. Sie weisen Nachteile bei der Anwendung in komplexen Sensornetzwerken auf. Die Ineffizienz geht auf langsame *Mischzeiten (shuffle times)* von *Random Walks* zurück. Ein Random Walk in einem Graphen ist ein zufälliger iterativer Prozess. Dieser beginnt bei einem Ausgangsknoten und endet, wenn ein Zielknoten erreicht wurde. In jedem Schritt wird eine ausgehende Kante auf Grundlage eines stochastischen Prozessees ausgewählt. Dieser Prozess ist durch die Konnektivität des Graphen bestimmt.

Die Mischzeit eines Random Walks ist die Zeit, die zur Annäherung der stationären Verteilung des Graphen benötigt wird. Sie gibt an, wie schnell der Random Walk verschiedene Knoten im Graphen besucht. Im Idealfall sollte ein Random Walk eine schnelle Mischzeit haben. Das bedeutet, dass eine nahezu gleichmäßige Wahrscheinlichkeitsverteilung über alle Knoten erreicht wird. Dies gewährleistet eine effektive Durchführung von Aufgaben wie Zufallsstichproben oder Informationsverbreitung. Haben Graphen eine spezifische Struktur, kann ein Random Walk wesentlich mehr Zeit in Anspruch nehmen. Zum Beispiel kann dieser bei hochmodularen Graphen über längere Zeiträume innerhalb desselben Clusters verweilen. Für die Informationsverbreitung bedeutet dies, dass oft redundante Daten weitergeben werden. Aber auch bei Graphen mit vielen Verzweigungen kann es durch entlegenen Knoten zu hohen Laufzeiten kommen. Insbesondere bei Sensornetzwerken, wo Energie- und Ressourcenknappheit besteht, ist eine große Effizienz anzustreben. Dementsprechend ist ein wiederholtes Senden der gleichen Daten zu vermeiden.

Das Paper schlägt einen optimierten Gossip-Algorithmus vor, welcher geografisches Routing und Resampling verwendet. Es wird nachgewiesen, dass der Kommunikations-Overhead deutlich abnimmt. Ebenso erfolgt die Untersuchung der Performance des Algorithmus gegenüber Standard-Algorithmen. Dabei werden reguläre Graphen wie Ring- oder Gittergraphen sowie zufällige geometrische Graphen evaluiert. Die zufälligen geometrischen Graphen stellen hierbei realistische Modelle für Sensornetzwerke dar. Es wird festgestellt, dass eine signifikante Beschleunigung der Konvergenz erreicht wird. Der geografische Gossip-Algorithmus kann auch auf verschiedene Klassen von Zufallsfeldern, wie beispielsweise Markov-Zufallsfelder, angewandt werden. Dazu werden experimentelle Ergebnisse dargestellt, um die theoretische Analyse zu untermauern und die Leistung des Algorithmus zu demonstrieren.

Auf dem Paper aufbauend wurde im Jahr 2008 von F. Benezit et al. eine weitere Arbeit veröffentlicht [11]. Diese untersucht ebenfalls das Konsens-Problem in Netzwerken, wo nur lokalisierte Nachrichtenübermittlungen erlaubt sind. Es wird ein verbesserter Algorithmus vorgeschlagen, der auf dem geografischen Gossiping-Verfahren aufbaut und zusätzlich ein Path Averaging anwendet. In diesem Zusammenhang erfolgt die Berechnung einer globale Funktion, wobei dazu Daten verwendet werden, die über die über die Knoten verteilt sind. Zur Erprobung des Algorithmus wird als Funktion die Mittelwertbildung verwendet. Der Algorithmus setzt Standortinformationen ein, um Informationen schneller und effizienter verbreiten zu können. Dadurch kann die Konvergenz beschleunigt werden, ohne dass der Kommunikationsaufwand steigt. Die Effizienz des Path Averaging beruht hierbei auf der opportunistischen Kombination von Routing und Mittelwertbildung. Dabei werden Informationen in Richtung eines zufällig ausgewählten Ziels weitergeleitet. Gegebenenfalls können zusätzliche Verarbeitungen durch Knoten auf den gerouteten Pfaden erfolgen.

Die Arbeit untersucht ebenso primär Gittertopologien und zufällige geometrische Graphen. Dabei wird vorausgesetzt, dass jeder Netzwerkknoten seinen eigenen Standort kennt. Darüber hinaus muss gegeben sein, dass Knoten auch die Standorte ihrer nächsten Nachbarn durch Anfragen ermitteln können. Zudem müssen die Knoten die Größe des geometrischen Raums kennen, in dem sie sich befinden. Sind diese Voraussetzungen erfüllt, so kann das geometrische Gossiping ausgeführt werden. Es folgt nun eine schrittweise Beschreibung des Algorithmus zu einer Zeiteinheit:

- Wird ein zufälliger Knoten aktiviert, so wählt dieser ein zufälliges Ziel (geometrische Position).
- Der Knoten erstellt dann ein Paket mit seiner aktuellen Schätzung des Durchschnitts, seiner Position, der Anzahl der bisher besuchten Knoten und der Zielposition.

- Dieses Paket versendet er an den Nachbarn, der dem Ziel am nächsten ist.
- Sobald ein Knoten das Paket erhält, leitet er es gierig in Richtung des Ziels weiter. Dabei addiert er seinen Wert zur Summe und erhöht den Zähler für die bisherigen Besuche.
- Als Zielknoten gilt der Knoten mit der kleinsten Distanz zur Zielposition. Dieser berechnet den Durchschnitt, sobald das Paket ihn erreicht.
- Anschließend wird der Durchschnittswert auf demselben Weg zurückgeführt.

Der gesamte Prozess der Weiterleitung bis zur Mittelwertbildung wird als eine Runde bezeichnet. Eine solche Runde wird auf einem zufälligen Pfad ausgeführt und hat eine Dauer von einer Zeiteinheit. Nach jeder Runde ersetzen die Knoten auf dem Pfad ihre Schätzungen durch den ermittelten Durchschnitt. Zur Vereinfachung wird ein modifiziertes Routing-Schema verwendet, das sogenannte ($\leftrightarrow$, $\updownarrow$)-Box-Routing. Hier erfolgt eine Unterteilung des geometrischen Raums in Quadrate gleicher Größe, wobei in jedem Quadrat mit hoher Wahrscheinlichkeit eine bestimmte Anzahl von Knotenpunkten erwartet wird. Beim ($\leftrightarrow$, $\updownarrow$)-Box-Routing werden die Nachrichten von Quadrat zu Quadrat weitergegeben. Dabei erfolgt die Weitergabe entweder zuerst horizontal und dann vertikal oder umgekehrt. In Abbildung 3.1 ist ein Beispiel für das Routing dargestellt. Hier wählt der Knoten mit Initialwert 3 eine zufällige Position als Ziel aus. Die Weitergabe erfolgt dann von einer Box zur nächsten, wobei hier zuerst die Boxen ausgewählt werden, die horizontal am nächsten liegt. Danach wird die Nachricht an die nächsten vertikalen Boxen versandt, bis schließlich die Box der Zielposition erreicht ist. Die Knoten innerhalb der jeweiligen Boxen werden hierbei zufällig ausgewählt. Zum Schluss wird der Mittelwert von 2,5 bestimmt und alle Knoten auf dem Pfad ersetzen ihre Werte durch diesen. Das Box-Routing dient zu Analysezwecken, um die Beweisführung zu vereinfachen. Als logisches Konzept ist es für eine praktische Umsetzung ungeeignet. Stattdessen wird es verwendet, um entsprechende Beweise sowie experimentelle Ergebnisse zu liefern und somit die Optimierung nachzuweisen.

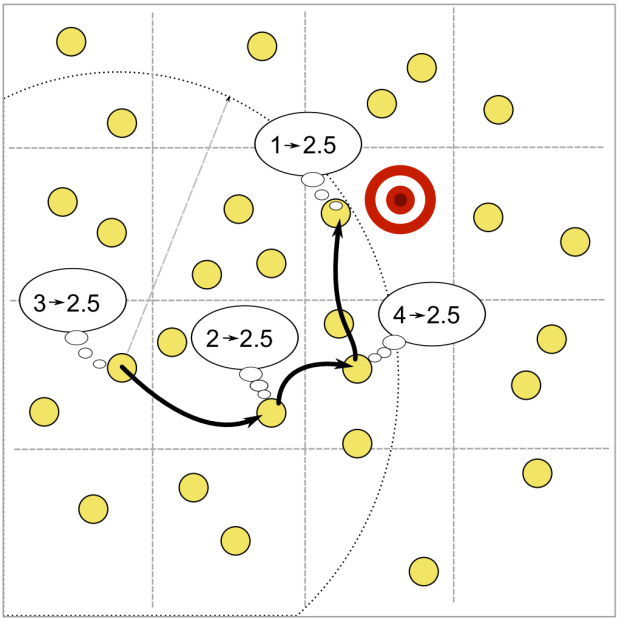


Abbildung 3.1: Geografisches Gossiping mit Path Averaging [11]

Die oben beschriebenen Gossip-Algorithmen führen eine Nachrichtenverteilung anhand der geografischen Nähe durch. Dabei ist das Ziel, Informationen effizient an einen Zielort zu verbreiten. Die Algorithmen dieser Arbeit sollen topologische Eigenschaften verwenden, um eine gezieltere Verbreitung innerhalb des Netzwerks zu ermöglichen. Dabei sollen insbesondere Informationen über die Community-Strukturen eingesetzt werden. Somit werden nur die Verbindungen zwischen Knoten betrachtet, während die physische Netzwerkstruktur unberücksichtigt bleibt. Folglich wird ein abstrakter Ansatz verwendet als bei der verwandten Arbeit. Außerdem werden keine geografischen, sondern topologische Infrastrukturinformationen verarbeitet. Die Anwendbarkeit wird dadurch größer, auch logische beziehungsweise virtuelle Netzwerke können von den Erkenntnissen profitieren.

Eine wichtige Erkenntnis, die aus dieser Arbeit gewonnen wurde, ist die Wichtigkeit der entlegenen Knoten. Solche Knoten werden bei einem zufälligem Informationsaustausch erheblich seltener aktiv. Dies kann zu einer Verlangsamung einer Nachrichtenverteilung führen. Netzwerke, die viele entlegene Knoten haben, bieten somit ein großes Optimierungspotential. Hier kann eine Informationsausbreitung durch gezieltere Auswahl dieser Knotten siginifikant beschleunigt werden.

3.4 Andere optimierte Gossiping-Verfahren

Neben den geografischen und pfadbasierten Gossip-Algorithmen, gibt es weitere optimierte Verfahren. Im Folgenden werden mehrere Algorithmen vorgestellt, welche sich auf spezifische Anwendungsbereiche ausrichten. Sie ermöglichen es, Leistung, Skalierbarkeit und Effizienz der Informationsverbreitung in verschiedenen verteilten Systemen zu verbessern. Dabei stellen sie maßgeschneiderte Lösungen für bestimmte Anwendungsbereiche dar. Sie berücksichtigen die besonderen Anforderungen und Einschränkungen der Zielsysteme. Beispiele beinhalten Sensornetzwerke, IoT-Anwendungen, Rechenzentren und Multicast-Protokolle.

3.4.1 Accelerating Distributed Averaging in Sensor Networks: Randomized Gossip over Virtual Coordinates

Das Paper [44] aus dem Jahr 2016 stellt ein optimiertes Gossiping-Verfahren für Sensornetze vor. Es wird das Ziel gesetzt, die Anzahl an Nachrichtenübertragungen zu reduzieren und somit eine schnellere Konvergenz zu erreichen. Um dies zu erreichen, wird der zuvor dargestellte geografische Ansatz auf Netze ausgeweitet, deren Knoten ihren geografischen Standort nicht kennen. Dazu werden den Knoten virtuelle Koordinaten zugeordnet.

Der vorgeschlagene Algorithmus trägt den Namen *Randomized Gossip over Virtual Coordinates (RGVC)*. Auch er beschäftigt sich mit der verteilten Mittelwertbildung in Ad-hoc- und drahtlosen Sensornetzen. Hierbei wird die Kommunikation über ein virtuelles Koordinatensystem realisiert, wodurch die Notwendigkeit einer geografischen Lokalisierung entfällt. Dazu wird eine bestimmte Anzahl von Knoten als *Beacons* ausgewählt. Bei einem Beacon handelt es sich um einen Knoten, der eine feste Position hat. Dadurch kann er den anderen Knoten als Referenzpunkt dienen.

Der Algorithmus ermöglicht die Berechnung von Mittelwerten zwischen zwei entfernten Knoten. Das virtuelle Koordinatensystem stellt die Grundlage für die Kommunikation dar. Mit Hilfe von *Reverse-Path-Tree-Algorithmen* können alle Knoten die Sprungdistanzen zu den Beacons berechnen. Infolgedessen können die Koordinaten jedes Knotens als Vektor dieser Sprungdistanzen dargestellt werden. Während der Berechnungsphase wählt jeder Knoten einen zufälligen Zielpunkt aus. Das Routing erfolgt dann schrittweise ent-

lang des Pfads zu diesem Ziel. In jedem Schritt wählt der Knoten seinen Nachbarn so aus, dass die verbleibende Distanz zum Zielpunkt minimal ist. Der Pfad endet, wenn kein weiterer Fortschritt in Richtung des Zielpunkts möglich ist. Anschließend sendet der letzte Knoten auf dem Pfad seine aktuelle Schätzung des Mittelwerts zurück. Dadurch können alle Knoten auf dem Rückweg ihre Schätzungen ebenfalls aktualisieren.

Die dargestellte Methodik zeichnet sich durch ihre Einfachheit aus. Dennoch kann anhand experimenteller Ergebnisse die gute Performance gezeigt werden. Die Konvergenzrate ist im Vergleich zum randomisierten Verfahren deutlich besser. Das Problem der verteilten Mittelwertbildung wird für Netzwerke gelöst, wo Knoten über keine geografischen Kenntnisse verfügen. Trotz dieser Einschränkung ist das dargestellte Verfahren effizient und performant.

Der Gossip-Algorithmus der verwandten Arbeit verteilt Informationen entlang von Pfaden. Dabei werden Routen für die Weiterleitung von Nachrichten verwendet. Durch diesen Aufbau wird eine gezielte Verbreitung sehr einfach durchführbar. Die Anwendung ist somit für Point-to-Point-Übertragungen optimiert. Diese Arbeit hingegen baut auf dem paarweisen, randomisierten Gossiping-Verfahren auf. Hierbei wählen Knoten Nachbarn zufällig für einen Informationsaustausch aus. Im Mittelpunkt steht die Herausforderung Informationen möglichst schnell im gesamten Netzwerk zu verteilen. Der Anwendungsfall beider Algorithmen ist dementsprechend unterschiedlich. Zudem bleiben beim randomisierten Gossiping konkreten Pfade im Netzwerk unbeachtet, wodurch man simplere Vorgänge und eine größere Flexibilität erhält. Des Weiteren kann so der Verbreitungsprozess vollkommen dezentral durchgeführt werden.

3.4.2 Context-Aware Gossip-Based Protocol for Internet of Things Application

Im Jahr 2018 wurde das Paper [4] veröffentlicht, was ein Gossiping-Protokoll vorstellt, das für IoT-Anwendungen optimiert ist. Hierbei verwendet das Protokoll eine *Multi-factor weighting function (MFWF)* zur Berechnung der Gewichte bei der Kommunikationsauswahl. Diese berücksichtigt verschiedene Parameter wie Restenergie, Chebyshev-Distanzen zu Nachbarknoten, Knotendichte und Nachrichtenpriorität. Die MFWF ermöglicht die Anpassung des Protokolls an wechselnde Kontexte. Als Folge können Nachrichten entsprechend priorisiert werden.

Der Algorithmus folgt einem schrittweisen Verfahren zur Datenverbreitung. Wenn ein Knoten neue Daten zu senden hat, sammelt er Informationen von seinen Nachbarknoten. Auf Grundlage dieser Informationen ordnet er den Nachbarknoten mit der MFWF Gewichte zu. Dabei wird zur Berechnung der Funktion die folgende Formel verwendet:

$$w = \sum_{i=1}^{n} f_i \cdot p_i \tag{3.1}$$

Hierbei ist n die Anzahl an verwendeten Parametern und f_i der Gewichtungsfaktor des Parameters p_i. Der Faktor kann entweder 0 oder 1 sein, je nachdem, ob der Parameter für die Berechnung berücksichtigt werden soll oder nicht. Die verwendeten Faktoren und ihr Mapping zu den Parametern ist in Tabelle 3.1 dargestellt. Zudem sind hier die Berechnungsformeln für die Parameter gegeben. Jeder Knoten berechnet mit diesen Formeln die Gewichte seiner direkten Nachbarn. Bei der Weiterleitung einer Nachricht wählt ein Knoten den Nachbarn mit dem höchsten Gewichtungswert aus. Wenn mehrere Knoten die gleiche Gewichtung haben, wird einer nach dem Zufallsprinzip selektiert. Dieser Prozess wird fortgesetzt, bis die Nachricht den Zielknoten erreicht.

Tabelle 3.1: Mapping zwischen Faktoren und Parametern [4]

Faktor	Parameter	Formel
$f1$	Restenergie	$\text{Energy} = \frac{ResidualEnergy}{InitialEnergy}$
$f2$	Nächster-Knoten-Distanz	$\text{NeighbourDist} = \frac{1}{\max(\lvert X-nX\rvert,\lvert Y-nY\rvert)}$
$f3$	Zieldistanz	$\text{SinkDist} = 1 - \frac{neighbourToSinkDist}{sourceToSinkDist}$
$f4$	Knotendichte	$\text{NodeDensity} = \frac{localDensity}{\#Nodes}$
$f5$	Nachrichtenpriorität	$\text{Msg} = \begin{cases} 0 \ low \\ 1 \ high \end{cases}$

Das Paper präsentiert ebenso eine Evaluation des Algorithmus. Hier wird die Leistung im Vergleich zu traditionellen und geografischen Gossiping-Protokollen anhand experimenteller Ergebnisse bewertet. Dabei werden verschiedene Metriken analysiert wie die Nachrichtenverzögerung, die Bandbreite oder auch die Anzahl an versendeten und wiederholten Nachrichten. Das Protokoll erreicht eine gute Performance und hohe Skalierbarkeit. Durch die MFWF kann eine flexibele Erweiterung mit beliebigen Parametern erfolgen. Dadurch wird die Erstellung von maßgeschneiderten Lösungen für den jeweiligen IoT-Anwendungsfall möglich.

Die verwandte Arbeit nutzt einen gewichteten Ansatz zur Selektion des nächsten Nachbarn. Dabei erfolgen Datenübertragungen von einem Start- zu einem Zielknoten. In dieser Arbeit hingegen werden Verfahren analysiert, die eine netzwerkweites Gossiping über paarweisen Informationsaustausch realisieren. Außerdem ist der Gossip-Algorithmus der verwandten Arbeit für einen spezifischen IoT-Anwendungsfall konzipiert. Es werden zur Gewichtung Informationen aus dem wie Energie und geografische Nähe genutzt. Hierbei erfolgt die Entscheidung bei der Auswahl des Nachbarn nicht mehr probabilistisch. Stattdessen wird der Knoten mit höchstem Gewicht gezielt ausgewählt. Diese Arbeit hingegen befasst sich mit einem generischen Szenario, was auf keinen spezifischen Anwendungsfall beschränkt ist. Zudem sollen zur Entscheidung topologische Informationen eingesetzt werden, welche in jedem Netzwerk verfügbar sind. Und ebenso soll die Auswahl weiterhin zufällig und nicht deterministisch erfolgen. Die verwandte Arbeit zeigt, dass sich die Verwendung einer MFWF anbietet, wenn komplexe Gewichtungsfaktoren zu berechnen sind. Müssen mehrere Aspekte für die Entscheidung berücksichtigt werden, so kann dieser Ansatz verwendet werden. Hierauf kann gegebenenfalls bei der Weiterentwicklung der gewichteten Algorithmen zurückgekommen werden.

3.4.3 Efficient Epidemic-style Protocols for Reliable and Scalable Multicast

I. Gupta et al. veröffentliche im Jahr 2002 das Paper [54], welches adaptives Verbreitungsprotokoll vorstellt. Das Protokoll besteht aus zwei Unterprotokollen, nämlich *Tree Dissemination* und *Gossip*. Mit Tree Dissemination werden dynamische Multicast-Bäume erzeugt, die eine *Leaf-Box-Hierarchy* bilden. Hierbei erfolgt eine gleichmäßige Verteilung der Knoten auf Ebenen beziehungsweise Boxen durch eine Mapping-Funktion. Die Auswahl einer ausreichenden Anzahl von Mitgliedern auf jeder Ebene der Hierarchie muss hierbei gewährleistet werden. Als Gossiping-Protokoll wird entweder *Flat Gossiping* oder *Hierarchical Gossiping* verwendet. Während des Vorgangs erfolgt die Verteilung von Nachrichten über Multicasts. Das bedeutet, dass jedes Mitglied eine festgelegte Anzahl von

Nachrichtenkopien sendet, wobei das Protokoll hierfür eine Obergrenze definiert.

In experimentellen Studien erfolgt ein Vergleich der Skalierbarkeit von Hierarchical Gossiping mit Flat Gossiping. Dabei wird die Wahrscheinlichkeit des Empfangs der Multicast-Nachrichten in verschiedenen Runden betrachtet. Der Netzwerk-Overhead wird berechnet, indem die Kopien eines Multicasts gezählt werden, die interne Knoten und Domänengrenzen durchlaufen. Hierbei werden Metriken wie die Belastung pro Mitglied, die Zuverlässigkeit und die Latenzzeiten untersucht. Insbesondere die Auslastung der Domänengrenzen wird evaluiert, da hier kritische Engpässe entstehen können. Die Analyse zeigt, dass die Zuverlässigkeit mit zunehmender Gruppengröße abnimmt.

Das hierarchische Gossiping ist auf Netzwerke beschränkt, wo die Topologie einer hierarchischen Struktur unterliegt. Auf unstrukturierte Netzwerke, wo alle Knoten gleichgestellt sind, kann es nicht angewandt werden. Diese Netzwerke sind dynamisch und folgen keiner vordefinierten Struktur. Somit existiert kein Muster nach dem eine Einordnung in Boxen beziehungsweise Ebenen erfolgen kann.

3.4.4 Topology-aware Gossip Dissemination for Large-scale Datacenters

M. Branco hat im Jahr 2012 eine Masterthesis erstellt, welche ein Konzept für ein optimiertes Gossiping-Protokoll beschreibt [18]. Das Verfahren wurde dann im Jahr 2013 unter dem Namen *Bounded Gossip* vorgestellt [19]. Die Anwendung des Protokolls zielt auf auf Daten- und Rechenzentren ab. In diesem Kontext werden verschiedene Netzwerke und populäre Topologien analysiert. Insbesondere wird die dreistufige Architektur analysiert, bei der Core-, Aggregations- und Edge-Switches eingesetzt werden. Hier variieren Latenzzeiten und Kommunikationskosten zwischen den verschiedenen Switches und deren Knoten. In diesen Architekturen führt der Einsatz von herkömmlichen Gossiping-Verfahren oft zu Skalierungsproblemen. Im schlimmsten Fall kann es sogar zu einer Sättigung der Switches auf der höchsten Ebene kommen, wodurch Engpässe und hohe Latenzzeiten entstehen. Alternative Topologien wie *Triton* und *CamCube* werden vorgeschlagen. Sie zielen darauf ab, Unterschiede bei den Kommunikationskosten auszugleichen und die gesamten Hardwarekosten zu senken. Triton führt dazu modulare Gruppen von Switches ein, die den Lastausgleich verbessern. CamCube wiederum implementiert eine physische Distributed-Hashtable-Architektur zur Optimierung des Informationsflusses. Ebenfalls wird die Modellierung der Netzwerkeigenschaften als Optimierungsmöglichkeit dargelegt. Als Folge soll die Latenz, Link-Unabhängigkeit und Konnektivität verbessert werden. Es wird außerdem eine gewichtete Graphdarstellung vorgeschlagen, um die Aufteilung der Knoten in Regionen zu erfassen. Dadurch sollen insbesondere Latenznachteile minimiert werden.

Der Kern der Arbeit ist das topologiebasierte Gossiping-Verfahren, welches für große Rechenzentren optimiert ist. Bounded Gossip baut hierbei auf den Prinzipien von Hierarchical Gossip und *CLON* auf. Bei CLON handelt es sich um ein Gossiping-System für Cloud-Umgebungen [75]. Das System um Bounded Gossip führt jedoch neue Ansätze und Mechanismen ein. Im Folgenden werden die drei Hauptkomponenten beschrieben, die zusammenarbeiten, um die Funktionalität zu realisieren:

- Der *Peer-Sampling-Service* beziehungsweise Mitgliedschaftsdienst ist dafür verantwortlich, jedem Knoten eine Teilansicht des Systems zur Verfügung zu stellen. Diese Ansichten spiegeln Teile der Netzwerktopologie wider und sind in hierarchischen Ebenen organisiert. Hierbei stellt jede Ebene eine andere Schicht der Netztopologie dar. Die Ansichten werden in regelmäßigen Abständen zwischen den Knoten ausgetauscht. Dabei hängt sowohl Größe als auch Inhalt der Ansicht von der Position des Knotens in der Netzhierarchie ab.

- Das *Gossip-basierte Verbreitungsschema* nutzt die bereitgestellten Teilansichten. Hier erfolgt die Verbreitung der Nachrichten auf teilweise deterministische Weise unter Berücksichtigung der Hierarchie der Topologie. Die Nachrichten werden zunächst an entfernte Knoten übertragen und dann auf weitere lokale Ebenen weitergeleitet. Dabei verarbeitet jeder Knoten eine Nachricht nur einmal und sendet sie an eine bestimmte Anzahl von Nachbarn weiter. Diese Anzahl wird durch den Fanout-Parameter auf jeder Hierarchiestufe festgelegt. Die Nachrichtenverbreitung unterscheidet sich je nach Alterszähler sowie Hierarchieebene und Rolle des Knotens. Hierbei werden verschiedene Rollen zu Beginn definiert und den Knoten zugewiesen. Es wird dann eine deterministische Auswahl sowie eine Synchronisation zwischen Knoten mit gleichen Rollen durchgeführt. Dadurch können redundante Nachrichten minimiert werden. Der Algorithmus gewährleistet durch Replikation ebenso eine Fehlertoleranz und hohe Zuverlässigkeit.

- Es wird ein verteilter *Flusskontrollmechanismus* eingesetzt, der auf der Topologie basiert. Dieser begrenzt den Kommunikations-Overhead, indem Quoten für die Nachrichtenverbreitung definiert werden. Der Mechanismus stellt sicher, dass die Knoten diese während des Verbreitungsprozesses einhalten. Somit wird ein übermäßiger Datenverkehr vermieden und eine bessere Gesamtsystemleistung gewährleistet.

Auch hier wird die Leistung des Protokolls in Bezug auf Latenz, Durchsatz, Ressourcenverbrauch und Zuverlässigkeit untersucht. Die Optimierung wird anhand experimenteller Ergebnisse nachgewiesen. Dabei werden die Simulationen mit *PeerSim* auf einer exemplarischen Topologie eines Rechenzentrums ausgeführt. PeerSim ist eine Plattform, die speziell für die Simulation von Peer-to-Peer-Systemen entwickelt wurde, um verschiedene Netzwerktopologien und Routing-Protokolle in einer kontrollierbaren Umgebung zu testen und zu evaluieren. Die Netzwerktopologie besteht hierbei aus einem Core-Switch, der sich in 8 Aggregations-Switches verzweigt. Jeder Aggregations-Switch führt wiederum zu 10 Edge-Switches mit Clustern zu je 32 Knoten. Aufgrund der Ähnlichkeit zu existierenden Systemen können die erzielten Ergebnisse auf reale Anwendungsfälle übertragen werden.

Die verwandte Arbeit zielt auf die Optimierung für die Anwendung in Daten- beziehungsweise Rechenzentren ab. Es werden dabei häufig verwendete Topologien betrachtet, wie die hierarchischen Netzwerkstrukturen. Diese verfügen über feste topologische Eigenschaften, die bereits bei der Netzplanung festgelegt werden. Eine Optimierung des Gossipings kann erfolgen, indem dieses spezifische Wissen eingesetzt wird. In dieser Thesis werden jedoch unstrukturierte Netze betrachtet, welche keine vorab festgelegte Struktur aufweisen. Das Verbreitungsschema kann dementsprechend kein Vorwissen einsetzen, stattdessen muss ein generischer Ansatz verfolgt werden. Außerdem bestimmt die verwandte Arbeit den besten Kommunikationspartner und wählt diesen deterministisch aus. Im Rahmen dieser Thesis soll ein zufälliges gewichtetes Gossiping durchgeführt werden. Des Weiteren erfolgt in der verwandten Arbeit der Einsatz von Techniken wie Flusskontrolle und Synchronisation. Hierbei handelt es sich um aufwendige Verfahren, die einen erheblichen Rechenaufwand mit sich bringen. Im Gegensatz dazu werden in diesem Projekt leichtgewichtige Gossiping-Verfahren betrachtet, die einen netzwerkweiten paarweisen Austausch durchführen. Diese Verfahren können mit minimalem Rechenaufwand in jedem Netzwerk verwendet werden.

3.5 Community-Based Gossip Algorithm for Distributed Averaging

Christel Sirocchi and Alessandro Bogliolo veröffentlichten am 09. Juni 2023 das Konferenzpapier „Community-Based Gossip Algorithm for Distributed Averaging“ [108]. Das Paper ist die erste Publikation, die Community-Strukturen als Mittel zur Optimierung von Gossip-Algorithmen einsetzt. Das Konzept dieser Thesis wurde zum Veröffentlichungszeitpunkt der verwandten Arbeit bereits festgelegt. Ebenfalls wurden schon Prototypen erstellt und weiterentwickelt, wodurch Erkenntnisse und Rückschlüsse nur begrenzt genutzt werden konnten. Große Änderungen waren aufgrund der fortgeschrittenen Projektphase und dem zeitlichen Rahmen nicht mehr umsetzbar. Es war jedoch zu prüfen, ob die Ergebnisse der verwandten Arbeit im weiteren Projektverlauf Anwendung finden könnten. Dabei war zu verifizieren, ob die Schlussfolgerungen der Arbeit bei der Definition der eigenen Simulationsreihen Anwendung finden können. In diesem Kontext wurden die untersuchten Topologien und deren Eigenschaften ausgewertet. Auch fand eine Analyse der verwendeten Metriken statt, um zu verifizieren, ob diese eine sinnvolle Erweiterung darstellen. In diesem Kontext wurden auch die Evaluationsergebnisse ausführlich studiert. Im Nachgang konnten einige Schlussfolgerungen auf diese Thesis übertragen werden. Zusätzlich boten die Methoden und Ergebnisse der Referenzarbeit eine solide Grundlage für Vergleiche. Des Weiteren konnte sogar eine direkte Gegenüberstellung erfolgen, bei welcher der Algorithmus der verwandten Arbeit auf den in dieser Arbeit generierten Graphen getestet wurde. Im Folgenden wird die nun Arbeit von Sirocchi und Bogliolo detailliert beschrieben. Anschließend erfolgt eine Abgrenzung zur eigenen Thesis, die Unterschiede zwischen den Arbeiten herausstellt.

Die Arbeit befasst sich mit den Auswirkungen der Modularität auf die Konvergenz von Gossip-Algorithmen zur verteilten Mittelwertbildung. Dabei gelingt es den Autoren nachzuweisen, dass sich die Modularität nachteilig auf die Konvergenzrate auswirkt. Es wird festgestellt, dass hierbei den sogenannten *Boundary Nodes*, also Grenzknoten, eine besondere Wichtigkeit zukommt. Sie sind entscheidend für die Kontrolle des Informationsflusses im Netzwerk, da sie mehrere Communities verbinden. Die Autoren bieten eine angepasste Modularitätsmetrik an, welche eine höhere Aussagekraft hat und zudem verteilt berechnet werden kann. Im letzten Schritt wird ein optimierter Gossip-Algorithmus vorgeschlagen, der Wissen über lokale Community-Strukturen nutzt, um den Informationsfluss zwischen den Communities zu verbessern.

Der Community-Based-Algorithmus baut auf einem Netzmodell auf, wo Netzwerke durch Graphen $G = (V, E)$ dargestellt werden. Ein Graph besteht hierbei aus einer Menge von Knoten V und Kanten E. Zudem hat jeder Knoten v_i einen Wert x_i sowie eine Menge von Nachbarn Ω_i. Dabei kann der Knotenwert unterschiedliche Bedeutungen haben. So kann es sich um einen Zustand, eine Meinung oder eine physikalische Größe wie eine Position oder ein Temperaturwert handeln. Der Vektor $x(k)$ bildet den Zustand des Netzwerks ab, indem er alle Knotenwerte $x = (x_1, \ldots, x_n)^T$ zum Zeitpunkt k darstellt. Ziel des Algorithmus ist, dass alle Knoten asymptotisch auf denselben Wert $x*$ konvergieren, sodass:

$$\lim_{t \to +\infty} x_i(t) = x*, \forall i \in I. \tag{3.2}$$

Dieser Wert soll bei der Mittelwertbildung der Durchschnitt der summierten Anfangswerte sein. Je nach Anwendungsfall kann das Gossiping verschiedene Berechnungsfunktionen implementieren. Alternativ kann so auch die Bestimmung von einem Minimum, Maximum oder von Top-k-Werten durchgeführt werden.

Die Autoren haben das Ziel einen optimierten Algorithmus zu entwickeln, der bessere Skalierbarkeitseigenschaften aufweist. Dazu müssen die Probleme der Standardverfahren betrachtet werden. Der Gossip-Algorithmus nach Boyd et al. [17] kann als Standard-Gossiping bezeichnet werden. Knoten wählen hierbei asynchron einen zufälligen Nachbarn aus und tauschen sich mit diesem aus. Auch hier wird eine lokale Mittelwertbildung durchgeführt. Der Ansatz der randomisierten Auswahl hat für Netzwerke mit Community-Strukturen nachweisliche Einschränkungen. Solche Netzwerke haben eine ungleichmäßige Kantendistribution, wo Knoten innerhalb von Communities stark verbunden sind. Im Gegensatz dazu sind Verbindungen zwischen Knoten verschiedener Communities seltener. Bei einer zufälligen Auswahl wählen Grenzknoten tendenziell häufiger Knoten aus ihrer eigenen Community aus. Dadurch erfolgt ein häufig wiederholter, oft auch redundanter Informationsaustausch innerhalb der Communities.

Die Probleme der herkömmlichen Algorithmen sollen durch ein neues Verfahren namens *Community Selection* gelöst werden. Bei der Auswahl werden zunächst nur die angrenzenden Communities betrachtet. Es wird zufällig eine Community ausgewählt, unabhängig davon, wie viele Nachbarknoten dieser angehören. Anschließend wird ein Knoten aus dieser selektiert, wobei die Auswahl auch hier zufällig erfolgt. Zur Auswahl einer Community müssen Knoten die Zugehörigkeiten ihrer direkten Nachbarn kennen. Dementsprechend müssen sie eine lokale Sicht auf die Netzwerktopologie erhalten. Dazu wird ein Algorithmus zur dezentralen Community Detection vorgestellt, nämlich der *Label Propagation Algorithm (LPA)*. Dieser ist in das Gossip-Schema integriert. Das kombinierte Verfahren wird als *LPA Community Selection* bezeichnet. Im Nachgang erfolgt nun eine formale Beschreibung des Verfahrens. Jeder Knoten v_i speichert über Zeit seinen Wert x_i, sein Label l_i sowie Informationen seiner Nachbarn $l_j^i \ \forall v_j \in \Omega_i$. Eine Partition der Nachbarschaft Ω_i ist definiert als $\mathcal{C}^i = \left\{C_1^i, C_2^i, \ldots, C_m^i\right\}$. Zwei Knoten der Nachbarschaft gehören zur gleichen Community C_n^i aus $\mathcal{C}^i$, wenn sie die gleichen Labels haben, also $l_j^i = l_k^i$. Zu Beginn des Verfahrens setzt jeder Knoten v_i ein eindeutiges Label $l_i(0)$ und weist seinen Nachbarn entsprechende Labels zu. Die Wahrscheinlichkeit zur Zeit k einen Nachbarn auszusuchen berechnet sich dann wie folgt:

$$p_{ij}(k) = \frac{1}{|\mathcal{C}^i(k)| \cdot |\left\{v_h : v_h \in \Omega_i \wedge l_h^i(k) = l_j^i(k)\right\}|} \forall v_j \in \Omega_i, \text{ sonst } 0 \tag{3.3}$$

Die vorhandenen Communities in der Nachbarschaft werden zu gleichen Wahrscheinlichkeiten ausgewählt. Dasselbe gilt für die Knoten innerhalb dieser Communities. Die Community-Informationen werden zeitgleich zum Gossiping gewonnen. Bei einer Interaktion zwischen zwei Knoten v_i und v_j werden die geschätzten Mittelwerte sowie die aktuellen Labels ausgetauscht. Im Anschluss führt jeder Knoten entsprechende Aktualisierungen durch. Daraufhin aktualisiert er sein eigenes Label, falls sich dieses vom häufigsten Label seiner Nachbarn unterscheidet. Dabei wird die *Prec-Max-Strategie* angewandt, sodass bei Gleichstand das aktuelle Label vor dem bestimmten Maximum ausgewählt wird.

Es wird ebenfalls ein Verfahren namens *Known Community Selection* vorgestellt. Das Verfahren nimmt an, dass Community-Zugehörigkeiten bereits bekannt sind. Hierbei muss also kein LPA ausgeführt werden, um die Labels zu bestimmen. Somit werden während dem Gossiping konstante Labels L_i verwendet. Ebenso gibt es eine feste Partition $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$. Die LPA Community Selection degeneriert zu diesem Verfahren, sobald die Labels aller Knoten konvergiert sind. Die Wahrscheinlichkeit der Nachbarauswahl lässt sich dann entsprechend zeitunabhängig definieren als:

$$p_{ij} = \frac{1}{|C_n : C_n \cap \Omega_i \neq 0| \cdot |\{v_h : v_h \in \Omega_i \wedge L_h = L_j\}|} \forall v_j \in \Omega_i, \text{ sonst } 0 \tag{3.4}$$

Die verwandte Arbeit nutzt verschiedene Metriken, um die strukturellen Eigenschaften des Graphen zu beschreiben. Diese werden bei der Evaluation berücksichtigt, um unter anderem auch die Korrelation zur Konvergenzrate zu bestimmen. Im Nachgang erfolgt die Klassifizierung der Metriken sowie die Beschreibung ihres Verwendungszwecks:

- **Gradmetriken** messen die Gradverteilung und das Mischungsmuster. Berechnet werden der Mittelwert und die Standardabweichung der Knotengrade sowie die Assortativität.
- **Clustering-Metriken** messen die Tendenz von Knoten, gemeinsame Nachbarn zu haben. Hier wird Durchschnitt und Standardabweichung des Clustering-Koeffizient auf Knotenebene berechnet. Ebenso wird die Transitivität als Metrik auf Netzwerkebene verwendet.
- **Community-Metriken** liefern Statistiken zu Community-Merkmalen. Dazu zählen die Anzahl der Communities sowie Durchschnitt und Standardabweichung ihrer Größen.
- **Modularitätsmetriken** bewerten die Stärke der Netzwerkunterteilung in Communities. Neben dem Modularitätsmaß, werden zudem zwei zusätzliche Metriken eingeführt. Die Erste stellt das Verhältnis zwischen der Anzahl an Grenzkanten und der Gesamtanzahl an Kanten dar. Die zweite Metrik misst den Anteil von Knoten, die zu Knoten anderer Communities verbunden sind.
- **Leistungsmetriken** können zur Bewertung des Konvergenzverhaltens genutzt werden. Hier wird der normalisierte Fehler gemessen, durch die die Konvergenzrate dann als zeitunabhängiger Koeffizient definiert werden kann.

Der vorgeschlagene Algorithmus wird sowohl in synthetischen Netzwerken als auch in realen Peer-to-Peer-Netzwerken evaluiert. Dabei erfolgt ein Nachweis der Verbesserung der Leistung gegenüber dem Standardverfahren. Modulare Netzwerke werden mit dem FARZ-Benchmark [132] generiert. Das Tool ermöglicht eine flexible Erzeugung unter Angabe verschiedener Parameter. Konkret erfolgt die Generierung von tausend zusammenhängenden Graphen der Größe $n = 1000$. Um möglichst variable, aber modulare Netzwerke zu erhalten, werden folgende Werte verwendet:

- Anzahl der Communities $h \in [3, 25]$
- Wahrscheinlichkeit der Kantenbildung innerhalb von Communities $\beta \in [0.5, 0.99]$
- Anzahl der pro Knoten erzeugten Kanten $m \in [2, 20]$
- Gradähnlichkeitskoeffizient $\alpha \in [0.1, 0.9]$
- Koeffizient der gemeinsamen Nachbarn $\gamma \in [0.1, 0.9]$

Bei der Evaluation werden Vergleiche mit Nullmodellen gezogen. Dazu zählen das *Erdős-Rényi-Modell*, das *Konfigurationsmodell* und das *stochastische Blockmodell*. Hierzu werden Graphen mit 1000 Knoten, einem Grad von 13, mit jeweils 15 Communities erzeugt, wobei die Modularität mit 0, 0.5 und 0.9 angesetzt wird. Anschließend wird randomisiertes Gossiping auf den Netzwerken durchgeführt. Schließlich werden die Simulationsergebnisse gesammelt und das Konvergenzverhalten bewertet. Die Ergebnisse zeigen, dass ein Einfluss von Gradverteilung und Modularität auf die Konvergenzrate besteht. Eine positive Korrelation zwischen Modularität und Konvergenzzeit kann nachgewiesen werden.

Das heißt, dass das Gossiping langsamer auf Graphen höherer Modularität konvergiert. Ebenso haben die Gradverteilung, das Clustering und die Gradassortativität einen großen Einfluss. Im nächsten Schritt werden auch die verschiedenen Metriken für die strukturellen Eigenschaften bewertet. Über den Pearson Korrelationskoeffizienten wird ihr Einfluss auf die Konvergenzrate bestimmt. Es wird festgestellt, dass Modularitätsmaße, die auf Grenzknoten basieren, die Leistung besser vorhersagen. Randknoten mit hoher Zentralität können hierbei genauere Vorhersagen für den globalen Durchschnitt liefern. Insbesondere die Grenzknoten spielen eine wichtige Rolle bei der Informationsausbreitung.

Zur abschließenden Auswertung wird das LPA Community Selection Verfahren mittels *Normalized Mutual Information (NMI)* bewertet. Die Resultate zeigen, dass die Methode eine hohe Genauigkeit bei der Erkennung von Communities erreicht. Verglichen mit der Zufallsauswahl konvergiert das Gossiping wesentlich schneller, insbesondere bei hochmodularen Netzwerken. Bei Netzwerken mit einer Modularität unter 0.7 ist die Konvergenzrate im Durchschnitt nur 10 Prozent höher als bei der Zufallsauswahl. Steigt die Modularität jedoch bis auf 0.7 so nimmt die Leistung bereits um 20 Prozent zu. Bei einer Modularität über 0.9 kann eine Leistungssteigerung von 100 Prozent verzeichnet werden. In Abbildung 3.2 sind die exakten Ergebnisse dargestellt.

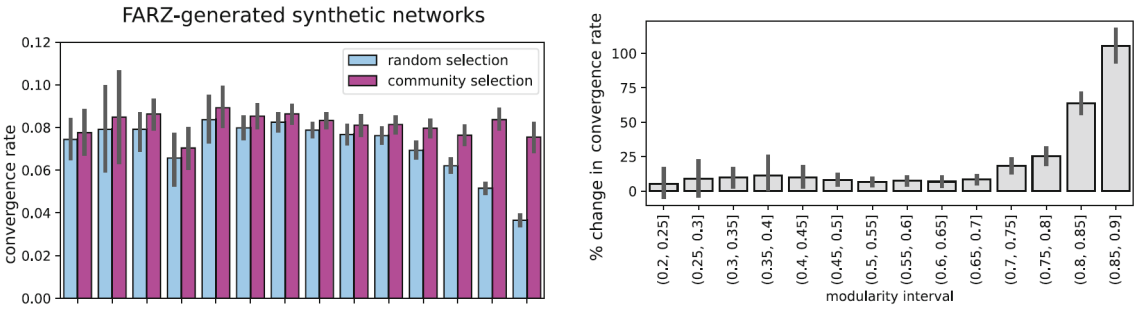


Abbildung 3.2: Konvergenzraten bei Zufalls- und Community-Auswahl [108]

Abschließend wird auch die Performance des Verfahrens auf realen Netzwerken analysiert. Hier werden Snapshots von Gnutella-Netzwerken untersucht, die an verschiedenen Tagen im August 2002 aufgenommen wurden. Gnutella ist ein Peer-to-Peer-Protokoll für Netzwerke, welches primär im Bereich von File Searching und Sharing eingesetzt wird. Auch für diese Netzwerke kann eine schnellere Konvergenz festgestellt werden. Je nach Snapshot variieren die Beschleunigungen jedoch sehr stark. Diese Auswertung bestätigt, dass das Verfahren auch in der Praxis angewandt werden kann.

Nun folgt die Abgrenzung der verwandten Arbeit von der eignen Thesis. Beide Arbeiten verwenden einen Simulator, um Ergebnisse zur Performance der jeweiligen Algorithmen zu sammeln. Die verwandte Arbeit setzt hier ein Python-basiertes Tool ein, welches die Erkennung von Communities, die Generierung von Netzwerken und die Berechnung von Metriken integriert. Mit dem Tool werden dann das randomisierte Standard-Verfahren und der Community-Based-Algorithmus simuliert. Dabei findet keine tatsächliche Kommunikation bei der Simulation statt, stattdessen sind die Abläufe experimentell realisiert. Im Gegensatz dazu verwendet dieses Projekts einen Kubernetes Operator zur Simulation. Hier wird für jeden Netzwerkknoten ein Pod erstellt, welcher die tatsächliche Netzwerkkommunikation zum Gossiping durchführt. Die Simulation ist dementsprechend wesentlich komplexer, aber auch realitätsnaher. Als Folge können die entwickelten Algorithmen ohne Anpassungen praktisch eingesetzt werden. Die erstellte Simulationsumgebung ist zudem flexibel erweiterbar. Realistisches Verhalten wie beispielsweise das Wegfallen und Hinzukommen von Knoten kann ebenfalls simuliert werden.

Neben den großen Unterschieden, was die Simulation betrifft, gibt es weitere nennens-

werte Differenzen. Als erstes sind die untersuchten Graphen anzumerken, welche unterschiedliche Eigenschaften aufweisen. Sirocchi und Bogliolo führen viele Simulationen durch, wo Graphen mit hohen Knotengraden (oft über dreizehn) untersucht werden. Diese Arbeit hingegen evaluiert Graphen mit niedrigeren Knotengraden (meist zwischen zwei und vier) wie sie in vielen realen Netzwerken zu finden sind. Ebenfalls wird ein großer Fokus auf skalenfreie Graphen gelegt. Die Unterschiede in den betrachteten Graphen macht die Gegenüberstellung der Evaluationsergebnisse schwierig. Aus diesem Grund ist ein direkter Vergleich durch Ausführen des Community-Based-Algorithmus auf den generierten Graphen sinnvoll.

Des Weiteren unterscheiden sich die Algorithmen in Bezug auf das Umfang des verwendeten Wissens. Bei der verwandten Arbeit wird Wissen außerhalb der loaklen Sicht eines Knotens verwendet, um Boundary Nodes zu identifizieren. Zur Identifikation eines Nachbarn als Grenzknotens muss der Knoten auch dessen Nachbarn kennen. Für die grundlegenden Algorithmen dieser Arbeit sollen jedoch nur Informationen über direkte Nachbarn verwendet werden. Lediglich komplexere Verfahren sollen eine Gewichtung basierend auf zusätzlichen Daten durchführen. Beim Vergleich der Ergebnisse muss berücksichtigt werden, welcher Umfang an Wissen verwendet wird. In dieser Arbeit steht der Nachweis einer Optimierung sowie die Gegenüberstellung der Verfahren im Mittelpunkt. Es wird versucht sinnvolle Verfahren für unterschiedliche Anwendungsszenarien zu konzipieren. Dabei kann zur Evaluation eine Bewertung der unterschiedlichen Metriken erfolgen. Es sollen zusätzliche Metriken neben der Modularität hinzugezogen werden. Priorität haben hierbei die Metriken, für die eine hohe Korrelation zur Konvergenzrate nachgewiesen wurde. Diesbezüglich dienen die Ergebnissen der verwandten Arbeit zur Orientierung. Es muss jedoch verifiziert werden, ob die Erkenntnisse auch für die im Rahmen dieses Projekts generierten Graphen gelten. Des Weiteren ist zu prüfen, ob der Community-Based-Algorithmus ebenfalls gute Ergebnisse auf diesen Graphen erzielt. Dabei ist die Variante namens Known Community Selection zu verwenden, da das Wissen über Community-Strukturen vorausgesetzt wird. Im Rahmen dieser Thesis wird angenommen, dass dezentrale Verfahren wie die Synchronisationsprotokolle diese bereits zuvor ermittelt haben. Abschließend kann also ein Vergleich des Community-Based-Algorithmus mit den Algorithmen dieser Arbeit durchgeführt werden.

4 Analyse

Die Projektanforderungen wurden in den folgenden Schritten erarbeitet. Dabei wurde zunächst das übergeordnete Ziel des Projekts festgelegt. Anschließend wurden die spezifischen Anforderungen definiert und ein erster Lösungsansatz entwickelt. In der ersten Implementierungsphase erfolgte dann die Erstellung der ersten Prototypen, um die Machbarkeit der Lösung zu verifizieren. Zum Schluss wurden diese evaluiert und als Grundlage für die weitere Konzeption verwendet.

4.1 Zielsetzung

Das Ziel der Thesis ist die Entwicklung von gewichteten Gossip-Algorithmen, die eine schnellere Konvergenz in unstrukturierten Netzwerken erreichen. Da diese Netzwerke keine deterministische Struktur aufweisen, sind hier Operationen wie das Suchen, Verbreiten und die Aggregation von Informationen allgemein ineffizienter als in strukturierten Netzwerken. Unstrukturierte Netzwerke bieten jedoch zahlreiche andere Vorteile, welche sich durch das Fehlen einer zentralen Steuerung ergeben. Dazu zähleen unter anderem ein geringerer Management-Overhead und eine bessere Skalierbarkeit, weshalb die Netzwerke im Bereich von Peer-to-Peer- und Sensornetzwerken sowie bei verteilten Systemen bevorzugt werden. Ebenso verfügen sie über eine natürliche Redundanz und sind meist robuster gegenüber Ausfällen und Verbindungsproblemen. In unstrukturierten Netzwerken werden häufig Gossip-Algorithmen angewandt. Hierbei handelt es sich um dezentrale Verfahren, die einen effizienten Informationsaustausch ermöglichen. Dabei erfolgt eine Verteilung der Daten im Netzwerk, indem randomisiert paarweise Kommunikationen zwischen Knoten durchgeführt werden. Weisen die Netzwerke bestimmte Strukturen auf, so kann die Konvergenzgeschwindigkeit sehr langsam werden. In dieser Thesis wird untersucht, ob und in welchem Umfang eine Verbesserung der Konvergenzrate durch den Einsatz von Topologieinformationen erreicht wird. Als Topologieinformationen werden dabei primär Community-Strukturen des Netzwerks verwendet. Beim Gossiping wird dann eine Gewichtung bei der Auswahl der Nachbarn eingesetzt, die das Wissen über die Communities berücksichtigt. Dadurch kann eine gezieltere Auswahl von Nachbarknoten erfolgen, wodurch eine Anpassung an die Netzwerkstruktur erfolgt.

Die wesentlichen Schritte umfassen hierbei die Entwicklung einer entsprechenden Simulationsumgebung sowie die Definition und Durchführung von Simulationsreihen. Hierbei ist die Simulationsumgebung für die Erstellung, den Aufbau und die Ausführung der Simulationen zuständig. Im ersten Schritt müssen Graphmodelle unstrukturierter Netzwerke erzeugt werden können. Diese ermöglichen einen Simulationsaufbau, wo jeder Netzwerkknoten als Host mit Gossip-Funktionalität abgebildet wird. Die Simulationsumgebung ermöglicht dann die Ausführung verschiedener Gossiping-Verfahren. Hier sollen umfangreiche Simulationen zur Evaluation verschiedener gewichteter Algorithmen erfolgen. Die verschiedenen Verfahren werden dabei untereinander und mit herkömmlichem Gossiping verglichen. Ebenfalls sollen Netzwerke mit unterschiedlichen Eigenschaften untersucht werden, um Auswirkungen auf das Konvergenzverhalten zu analysieren. Schließlich können die Erkenntnisse zur Optimierung der Leistung und Skalierbarkeit in realen Netzwerkszenarien angewandt werden.

4.2 Anforderungen

Die Anforderungen wurden über den Zeitraum der Analysephase gesammelt und ausgearbeitet. Dabei wurden im ersten Schritt grobe *Use Cases* erstellt. Dazu wurden zuerst die Akteure definiert, die in verschiedenen Rollen innerhalb der Simulationsumgebung agieren. Anschließend wurden ihre Aufgaben festgelegt. Die finalen Use Cases sind im Anhang A.1 ausgelagert.

Im nächsten Schritt wurden aus den Use Cases funktionale und nichtfunktionale Anforderungen abgeleitet. Diese wurden wiederum in *Must-*, *Should-* und *Nice-To-Have-Anforderungen* unterteilt. Dadurch konnten die Anforderungen priorisiert und in eine zeitliche Abfolge gebracht werden.

4.2.1 Funktionale Anforderungen

Die funktionalen Anforderungen bilden die angestrebte Kernfunktionalität der Lösung ab und sind wie folgt festgehalten worden.

4.2.1.1 Must-Have-Anforderungen

Simulation von Gossiping:
Ein zentraler Bestandteil der Arbeit ist die Simulation von Gossiping. Bei Gossiping handelt es sich um ein dezentrales Verfahren, das effiziente Operationen zur Aggregation und Verteilung von Informationen ermöglicht. Hierbei muss ein einfacher Informationsaustausch zwischen benachbarten Knoten umgesetzt werden. Als praktische Anwendung ist die Ermittlung eines netzwerkweiten minimalen Werts zu verwenden.

Gewichtete Algorithmen basierend auf Netzwerktopologie:
Es sind Gossip-Algorithmen zu implementieren, die topologische Informationen einsetzen. Die Gossip-Algorithmen selektieren dabei den nächsten Kommunikationspartner unter Berücksichtigung von Informationen über die Netzwerkstruktur. Zur Auswahl werden insbesondere Clusterstrukturen genutzt, um die lokale oder globale Konvergenz des Gossipings zu beschleunigen.

Simulation von Netzwerken basierend auf Graphen:
Die Netzwerksimulation erfolgt auf Basis von Graphen, die komplexe Netzwerkstrukturen beliebiger Größe modellieren können. Dabei werden die Netzwerk-Hosts als Knoten abgebildet und verfügen über die Funktionalität, Gossiping durchzuführen. Zudem kennt jeder Host lediglich seine direkten Nachbarn, mit denen er kommunizieren kann.

Abbildung verschiedener topologischer Eigenschaften:
Ziel der Testreihe ist die Analyse, inwieweit Strukturinformationen zur Optimierung des Gossiping-Verfahrens eingesetzt werden können. Dafür muss die Erzeugung der simulierten Netzwerke so erfolgen, dass sie verschiedene topologische Eigenschaften erfüllen. Es werden unterschiedliche Ausprägungen von Clusterstrukturen sowie verschiedene Arten von Graphen untersucht. Dazu zählen skalenfreie Graphen und Popularitätsgraphen.

Dynamische Generierung von Graphen:
Ziel ist es, variable Netzwerkstrukturen zu testen und zu vergleichen. Graphen werden als Grundlage zur Abbildung des Netzwerks bei der Simulation eingesetzt.

Dementsprechend müssen Netzwerke als Graphen generiert werden können. Hierfür wird eine möglichst dynamische Grapherzeugungsroutine benötigt, welche Graphen anhand von Parametern wie der Anzahl der Knoten, der Anzahl der Cluster und der Modularität erzeugt.

Rundenbasierte Simulation:
Um den Simulationsablauf effektiv zu überwachen, wird das Gossiping rundenbasiert durchgeführt. Das bedeutet, dass in jeder Runde jeder Host im Netzwerk mit einem ausgewählten Nachbarn Informationen austauscht. Hierbei läuft die Simulation so lange, bis eine Konvergenz erreicht ist.

Dokumentation der Ergebnisse:
Die Ergebnisse und der Verlauf der durchgeführten Simulation sind zur späteren Auswertung persistent zu speichern. Dabei ist insbesondere das Konvergenzverhalten eine wichtige Metrik, da es zum direkten Vergleich der Performanz der Algorithmen eingesetzt werden kann.

Evaluation verschiedener gewichteter Algorithmen:
Es sind Testreihen durchzuführen, deren Ergebnisse verglichen werden müssen. Dadurch können Erkenntnisse über das Konvergenzverhalten der verschiedenen gewichteten Algorithmen gesammelt werden. Verschiedene Möglichkeiten zur optimalen Auswahl des Kommunikationspartners sind zu vergleichen. Zur Gewichtung sind primär topologische Strukturinformationen zu verwenden. Diese können simpel oder in Form von Wahrscheinlichkeiten von Clusterzugehörigkeiten verwendet werden. Andere Informationen wie eine Historie vergangener Kommunikationen kann in komplexeren Algorithmen zum Einsatz kommen.

Vergleich mit Baseline:
Die neuen gewichteten Algorithmen sind mit dem Baseline-Verfahren, also der klassischen zufälligen Auswahl, zu vergleichen. Dies macht eine Bewertung der Leistungssteigerung möglich, wodurch Erkenntnisse gewonnen werden können.

Schlussfolgerungen und Vergleich mit aufgestellten Annahmen:
Final sind logische Schlussfolgerungen aus den Testreihen zu ziehen, die durch die gesammelten Daten bestätigt werden. Diese Schlussfolgerungen werden dann mit den zu Beginn aufgestellten Annahmen verglichen, um festzustellen, inwieweit diese zutreffen.

4.2.1.2 Should-Have-Anforderungen

Gegenüberstellung der Anwendbarkeit der Algorithmen und ihrer Konfigurationen:
Anhand der Ergebnisse der Testreihen sollen mögliche Anwendungsbereiche für die erstellten Algorithmen abgeleitet werden können. Es soll möglich sein, basierend auf den Vor- und Nachteilen der gewichteten Algorithmen und ihrer Konfigurationen Optimierungen für Gossiping-Protokolle vorzunehmen. Je nach Anwendungsgebiet könnte beispielsweise eine schnellere lokale oder globale Konvergenz angestrebt werden.

4.2.1.3 Nice-to-have-Anforderungen

Erfassung und Auswertung genauer Messwerte aus umfangreichen Simulationen:
Mit Hilfe von umfangreichen Simulationen können detaillierte Statistiken erfasst werden, die zu präzisen und verlässlichen Schlussfolgerungen führen.

4.2.2 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen stellen sinnvolle Einschränkungen für das Design dar. Sinn und Zweck ist hier die Einhaltung von Qualitätsstandards.

4.2.2.1 Must-Have-Anforderungen

Simulation mit Kubernetes:
Das Kubernetes-Cluster des Distributed Systems Lab der Hochschule für Technik und Wirtschaft des Saarlandes wird für die Durchführung der Simulationsreihen verwendet. Dadurch ist die Zielplattform festgelegt.

Nutzung eines Kubernetes Operators:
Zur effizienten Konfiguration und Ausführung der Simulationen wird das Operator-Pattern eingesetzt. Ein Kubernetes Operator ermöglicht die flexible Erzeugung von Pods, die als Netzwerkhosts fungieren und mit entsprechender Logik ausgestattet werden können. Außerdem sorgt er dafür, dass die erzeugten Pods nach dem Löschen der Simulationsressource wieder entfernt werden.

Effizienter Einsatz der Cluster-Ressourcen:
Bei Simulationen mit vielen Netzwerkhosts entsteht eine hohe Ressourcenlast. Es wird angestrebt, die Clusterressourcen effizient zu nutzen und nur so lange wie nötig in Anspruch zu nehmen. Folglich ist wichtig, die Ressourcen nach Abschluss der Simulation schnellstmöglich freizugeben. Zudem müssen die Anwendungen der Netzwerkknoten sowie des Simulation Runners, die CPU- und Speicherressourcen effizient nutzen. Auch große Simulationen mit hunderten von Knoten müssen durchführbar sein.

Trennung der Generierung von Graphen und der Simulationsdurchführung:
Durch selbstdefinierte Ressourcen im Kubernetes-Cluster wird eine einheitliche Repräsentation der Netzwerke erreicht. Die Umsetzung der Grapherzeugung und der Simulationsdurchführung erfolgt als separate logische Schritte. Hierbei kann zuerst ein Graph generiert und dann als Objekt angelegt werden. Die Graphobjekte können dann zur Simulation verwendet werden. Dadurch wird eine logische Entkopplung erreicht, wodurch auch Graphen beliebig wiederverwendet werden können.

Object Storage zur Datenspeicherung:
Für die Speicherung von Simulationsergebnissen wird ein Object Storage verwendet. Die Ergebnisse werden in einer entsprechenden Ordnerstruktur als JSON-Dateien persistiert.

4.2.2.2 Should-Have-Anforderungen

Minimierung des administrativen Aufwands bei der Simulationsdurchführung:
Ziel soll es sein, den administrativen Aufwand für die Durchführung einer Simulation so gering wie möglich zu halten. Simulationen sollen möglichst effizient und flexibel erstellt, überwacht und ausgewertet werden. Mit der Minimierung des administrativen Aufwands wird es möglich Zeit zu sparen, wodurch zusätzliche und umfassendere Simulationen durchgeführt werden können.

4.2.2.3 Nice-to-have Anforderungen

Einsatz einer Schnittstelle zur Generierung von Graphen:
Eine nützliche Erweiterung wäre die Implementierung einer flexiblen Schnittstelle. Über diese können Graphen in einem einheitlichen Format (z.B. Adjazenzliste) oder als Kubernetes-Ressourcen erzeugt werden. Dadurch könnten Graphen schnell generiert werden, was den Aufwand für die Simulationsdurchführung weiter reduzieren würde.

Parallele Durchführung mehrerer Simulationen:
Es wäre wünschenswert, die Möglichkeit zu haben, beliebig viele Simulationen parallel durchzuführen, um Daten so schnell wie möglich zu sammeln. Dadurch können Wartezeiten beim Ausführen von Simulationen vermieden werden.

4.3 Lösungsansatz

Der Lösungsansatz sieht die Implementierung einer vierteiligen Architektur vor. Diese umfasst eine Graphgenerierungsroutine und einen Kubernetes Operator zur Verwaltung und Initialisierung von Simulationen. Des Weiteren ist das Kubernetes-Cluster, wo die Simulationen durchgeführt werden, und die Evaluationsroutine Teil des Konzepts. Es folgt eine Beschreibung der einzelnen Komponenten:

1. Grapherzeugungsroutine:
Die Routine soll Graphen verschiedener Arten mit unterschiedlichen Algorithmen erzeugen. Dabei sollen sowohl bestehende Implementierungen verwendet als auch eigene Generierungsmethoden entwickelt werden. Die Graphen sollen alle Netzwerkinformationen beinhalten, sodass sie direkt zur Abbildung von logischen Netzwerkstrukturen eingesetzt werden können. Dabei sollen die Graphen in einem einheitlichen Format speicherbar sein, z.B. als Adjazenzliste oder -matrix. Darüber hinaus soll eine Custom Resource Definition für ein Graphobjekt zusätzliche Metadaten verfügbar machen. Zur möglichst flexiblen Erstellung von Graphen bietet sich hier die Entwicklung einer Schnittstelle an.

2. Simulation Operator:
Entsprechend dem Operator-Pattern wird komplexe Logik in eine separate Komponente ausgelagert. Das Operator-Pattern wird häufig eingesetzt, wenn viele Schritte durchgeführt oder verschiedene Objekten koordiniert werden müssen [43]. Hierbei übernimmt der Operator die Initialisierung und die Verwaltung der Operation beziehungsweise des Workflows. Es wird eine Entkopplung von den erzeugten Ressourcen erzielt, wodurch der Funktionsumfang der einzelnen Ressourcen reduziert werden kann. Infolgedessen können Änderungen am Operator und an den Ressourcen separat voneinander erfolgen. Vorteile sind somit eine verbesserte Code-Modularität, Wartbarkeit, Änderbarkeit und Lesbarkeit.

In diesem Anwendungsfall überwacht und verwaltet der Operator CRDs für die Graph- und Simulationsobjekte. Über die Simulationsressource werden hierbei Informationen festgelegt, wie zum Beispiel der verwendete Algorithmus, die Anzahl an Durchläufen und weitere Metadaten. Zudem referenziert sie ein Graphobjekt, auf dem die Simulation ausgeführt werden soll. Der Operator baut bei der Neuanlegung eines Simulationsobjekts die entsprechende Netzwerkstruktur anhand des referenzierten Graphen auf. Dabei erzeugt er Pods für die Netzwerkknoten und Services zur Kommunikation innerhalb des Netzwerks. Ebenso erzeugt er Ressourcen

für eine zentrale Simulationssteuerungskomponente, den Simulation Runner. Darüber hinaus gewährleistet der Operator, dass alle Pods zum Applikationsstart über die notwendigen Daten verfügen. Dies soll durch Verwendung von Umgebungsvariablen sichergestellt werden. Aus diesen können die Anwendungen können die entsprechende Konfiguration flexibel auslesen. Gleichermaßen entfernt der Operator bei Löschen von Simulationsressourcen alle angelegten Pods und Services erneut.

3. Simulation im Kubernetes-Cluster:
Im Cluster wird das Gossiping durch die Ausführung der Node Pods simuliert. Jeder Node Pod stellt einen Knoten im Netz dar und wendet den simulierten Gossip-Algorithmus an. Innerhalb eines Pods wird eine Service-Applikation ausgeführt, welche eine Kommunikation mit den anderen Knoten ermöglicht. Hierbei wählt jeder Knoten bei seiner Erstellung zufällig einen Wert oder bekommt diesen zugewiesen. Für diese Thesis soll ein Gossiping-Algorithmus zur Minimum-Findung eingesetzt werden. Eine netzwerkweite Aggregation stellt eine einfache, aber typische Anwendung dar. Dabei müssen mehrere Varianten des Algortihmus entwickelt werden, die eine unterschiedliche Gewichtung zur Partnerwahl durchführen. Je nach Verfahren muss gegebenenfalls die Erweiterung der lokalen Ansicht eines jeden Knotens erfolgen. Zusätzliche Daten wie Informationen über die Zugehörigkeit von Nachbarn zur eigenen Community bis hin zu Zugehörigkeitswahrscheinlichkeiten können eingesetzt werden. Zur Ausführung von mehreren Gossiping-Runden wird eine zentrale Steuerungseinheit verwendet. Sie kann jede Runde einzeln starten und im Anschluss wichtige Daten von den Teilnehmern einsammeln. Hierbei kann eine Kommunikation mit den Node Pods über *Remote Procedure Calls RPC* durchgeführt werden. Darüber hinaus wird es möglich, das Konvergenzverhalten anhand des Netzwerkstatus zu beobachten. Abschließend kann die Steuerungseinheit dann alle Ergebnisse speichern und die Beendigung der Simulation einleiten.

4. Evaluationsroutine:
Die Evaluationsroutine soll die Sammlung, Visualisierung und den Vergleich von Ergebnissen erleichtern. Hierbei ist das Ziel, die Bewertung verschiedener Algorithmen und ihrer Leistung. Dazu müssen Algorithmen auf verschiedenen Netzwerkstrukturen verglichen werden.

Dieses Lösungskonzept zielt darauf ab, Gossip-Algorithmen in unstrukturierten Netzwerken mit Hilfe einer Kubernetes-Infrastruktur zu simulieren und zu bewerten. Administrative Aspekte wie die Grapherzeugung und Orchestrierung der Simulationsressourcen werden dargestellt. Diese werden mit der Umsetzung von Gossip-Algorithmen und deren Simulation kombiniert. Es wird ein Konzept dargelegt, welches den Workflow einer Simulation skizziert mit dem Ziel möglichst große Automatisierung zu gewährleisten. Ebenso wird ein Rahmen zur Analyse und Evaluation des Gossipings geboten. Hier steht insbesondere die Feststellung des Konvergenzverhaltens im Mittelpunkt.

5 Konzeption

Die Konzeption baut auf dem Lösungansatz der Analysephase auf. Hier wurde festgehalten, dass Kubernetes zur Simulation von unstrukturierten Netzwerken verwendet werden soll. Dabei erfolgt die Automatisierung des Simulationsworkflows durch einen Kubernetes Operator. Im nächsten Schritt musste der Aufbau des Systems gestaltet werden. Ebenfalls mussten Konzepte zur Umsetzung der Anwendungsfälle erstellt werden. Zuerst wurde hierzu das Top-Level Design durchgeführt, wobei das Gesamtsystem in einzelne Komponenten zerlegt wird. Ziel ist es, das Zusammenspiel der Abläufe festzulegen und die Aufgabenbereiche abzugrenzen. Die einzelnen Funktionen der Systemteile werden hierbei nur grob skizziert. Anschließend wurde das Low-Level Design erstellt, wo die Systemkomponenten im Detail beschrieben werden. Im ersten Schritt wird hier der Ablauf der Grapherzeugung spezifiziert. Danach erfolgt die Definition der Funktionsweise des Simulation Operators. In diesem Zusammenhang wird auch der Simulationsablauf konkretisiert und der Node-Service sowie der Simulation Runner beschrieben. Des Weiteren beinhaltet das Konzept einige Forschungsfragen und eine erste Spezifikation der Testreihen. Diese stellen einen Rahmen für die Evaluation dar, wobei die Forschungsfragen durch die Simulationen beantwortet werden sollen.

5.1 Top-Level Design

Die *Top-Level-Architektur* kann in Abbildung 5.1 nachvollzogen werden. Ein großer Fokus wurde hierbei auf die Interaktion des Benutzers mit dem System gelegt. Der administrative Aufwand zur Ausführung von Simulationen sollte möglichst gering ausfallen. Die Grundlage für das Architekturdesign wurde aus den Anwendungsfällen entnommen. In der Abbildung sind die drei Systemkomponenten dargestellt, welche die verschiedenen Anwendungen beinhalten. Zum Verständnis der Gesamtarchitektur werden diese im Folgenden erläutert.

Graph Configuration and Generation:
Im ersten Schritt interagiert ein Nutzer mit der Schnittstelle zur Grapherzeugung. In Abhängigkeit des Erzeugungsalgorithmus werden unterschiedliche Eingaben gefordert. Beispiele beinhalten die Anzahl Knoten, Kanten und Communities. Das komplexe Verfahren fordert die Angabe einer Zielmodularität für die generierten Graphen. Als Schnittstelle sieht das Konzept den Einsatz eines CLI vor, wobei es sich um eine textbasierte Benutzerschnittstelle handelt. Diese kann durch Eingabe von Befehlen in einen Kommandozeileninterpreter verwendet werden. Damit kann der Nutzer dann den Programmablauf durch Inputs steuern. Es wurde sich für ein CLI entschieden, da zur flexiblen Konfiguration von Graphen verkettete Nutzereingaben benötigt werden. Durch spezifische Befehle und Abkürzungen wird eine schnelle und einfache Bedienung ermöglicht. Die Entwicklung einer API, die über ein Graphical User Interface (GUI) bedient werden kann, wäre eine Alternative gewesen. Hierbei stellt das CLI jedoch die bessere Option dar. Gründe dafür sind die Leichtgewichtigkeit der Anwendung sowie der geringere Entwicklungsaufwand. Ebenso spricht der Anwendungsfall für die Auswahl, denn die Applikation ist ein

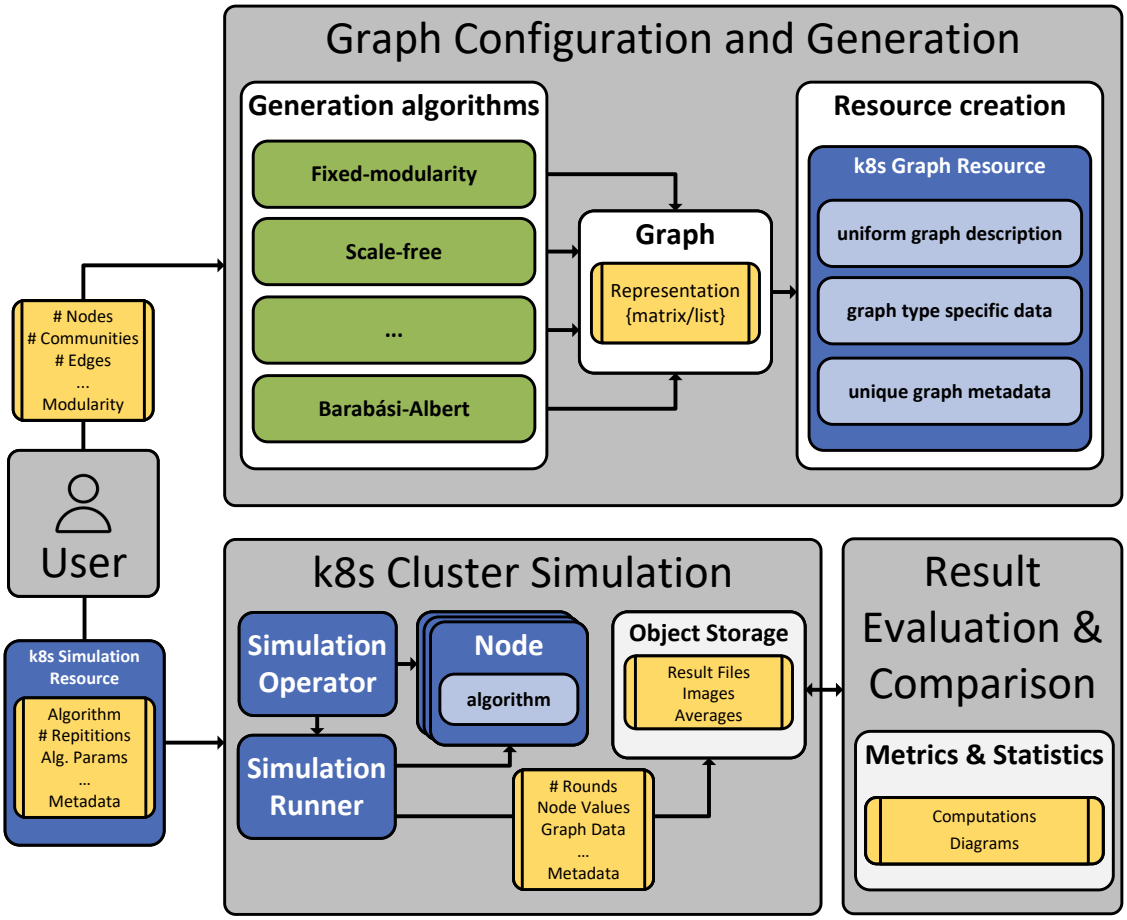


Abbildung 5.1: Top-Level Architektur der Gossiping Simulationsumgebung

administratives Hilfstool zu Forschungszwecken. Das Tool soll lokal zur Erstellung von Graphen oder CRDs genutzt werden. Eine Nutzung von mehreren Nutzern ist nicht erforderlich, weshalb die Anwendung nicht skalierbar oder verteilt ausführbar sein muss. Das CLI soll nach Verarbeitung der Nutzereingaben den Graphen generieren und in einer einheitlichen Graphdarstellung speichern. Alternativ kann eine direkte Erzeugung eines Kubernetes-Objekts entsprechend der CRD durchgeführt werden. Das Objekt verfügt über die Beschreibung des Graphen, typspezifische Daten, Metriken sowie spezielle Metadaten.

Kubernetes-Cluster Simulation:

Der zweite Schritt umfasst die Erstellung und das Deployment der Simulationsressource. Eine Simulation wird definiert durch den ausgeführten Algorithmus, die Anzahl an Wiederholungen und den referenzierten Graphen. Zudem müssen gegebenenfalls noch Parameter für den Algorithmus definiert werden und zusätzliche Metadaten festgelegt werden. Wird die Ressource im Cluster angelegt, so startet der Simulation Operator automatisch den Simulationsworkflow. Der Operator stellt ein festes Cluster-Deployment dar, welches logisch entsprechend des Operator-Patterns von den Simulationskomponenten getrennt ist. Im ersten Schritt erzeugt er die Pods für die Simulationsanwendungen. Anschließend wird die Simulation gestartet, wobei die Nodes das Gossiping entsprechend der Anleitung des Runners durchführen. Dabei verwenden die Netzwerkknoten den definierten Gossip-Algorithmus. Sobald eine Konvergenz erreicht wird, endet die Simulation. Der Runner persistiert dann

die Ergebnisse im Object Storage. Dazu zählen die Anzahl an durchgeführten Runden, die Wertänderungen der jeweiligen Knoten sowie alle Konfigurations- und Metadaten. Der Object Storage ist ein statisches Deployment im Kubernetes-Cluster und somit stets erreichbar. Alle durchgeführten Simulationen sind hier in einer entsprechenden Ordnerstruktur über Namen und Datum zu finden. Infolgedessen kann eine flexible Analyse der Ergebnisdateien, Visualisierungen und Durchschnittswerte bei wiederholten Ausführungen erfolgen.

Result Evaluation and Comparison:
Verschiedene Simulationen können über Metriken und Statistiken verglichen werden. Es sollen im letzten Schritt einige Routinen verfügbar sein, die eine Analyse der Simulationen vereinfachen. Dabei kann es sich um Skripte handeln, die Berechnungen automatisiert durchführen und Diagramme erzeugen.

Die Simulationsumgebung ist für die Zielplattform Kubernetes konzipiert. Dabei soll zur lokalen Entwicklung minikube eingesetzt werden. Nach Abschluss der Entwicklung sollen die Simulationsreihen dann im DSL-Cluster erfolgen. Hier können auch große Netzwerke mit vielen Knoten simuliert werden.

Im Folgenden werden die Konzepte der statischen Kubernetes-Ressourcen, wie dem Kubernetes Operator und dem Object Storage, besprochen. Anschließend erfolgt die Beschreibung der dynamischen Simulationskomponenten. Dazu zählen die Nodes und der Simulation Runner, welche als Pod Applikationen nur während der Simulationsausführung aktiv sind. Der Operator verwaltet ihren Lebenszyklus und stellt dabei sicher, dass keine Ressourcen länger als notwendig blockiert werden.

5.1.1 Simulation Operator

Die Funktionswiese des Simulation Operators kann in Abbildung 5.2 nachvollzogen werden. Er wird über ein statisches Deployment im Cluster angelegt. Dazu wird ein Docker-Image erstellt und über Docker Hub bereitgestellt, welches dann über das Repository von jeder Maschine abgerufen werden. Die Implementierung soll mit dem *Python Kopf Framework* erfolgen [66]. Hierbei handelt es sich um eine Schnittstelle zum Kubernetes Operator Framework. Das Framework bietet viele hilfreiche Tools und vereinfacht die Interaktion mit der Kubernetes-API. Dadurch kann flexibel die Erstellung neuer Objekte wie Pods und Services erfolgen. Auch Mechanismen zur Erkennung der Erstellung, Änderung und Entfernung von CRDs werden bereitgestellt. Zudem hat sich das Framework bei der prototypischen Entwicklung bewährt. Im Folgenden werden nun die Hauptfunktionen des Operators dargestellt:

- *Erstellung von Ressourcen*: Sobald ein neues CRD-Objekt erstellt wird, erkennt der Operator die entsprechenden Spezifikationen. Er kann das Graphobjekt ebenfalls auslesen, welches über einen Selector referenziert wird. Damit verfügt er über alle relevanten Spezifikationen, um die Pods und Services der Nodes und des Simulation Runners zu initialisieren.
- *Löschen von Ressourcen*: Die Löschung eines CRD-Objekts wird ebenfalls erkannt. Erfolgt die Entfernung des Objekts nach einer Simulation, so baut der Operator alle erzeugten Ressourcen ab. Ebenso kann ein Abbruch der Simulation durchgeführt werden, wenn das Objekt während der Ausführung gelöscht wird.

Ein Kubernetes Operator kann auch Änderungen an CRDs verarbeiten, was jedoch in diesem Anwendungsfall nicht sinnvoll ist. Eine laufende Simulation mit anderen Parametern zu versehen oder Änderungen am angelegten Graphen durchzuführen, ist nicht zu

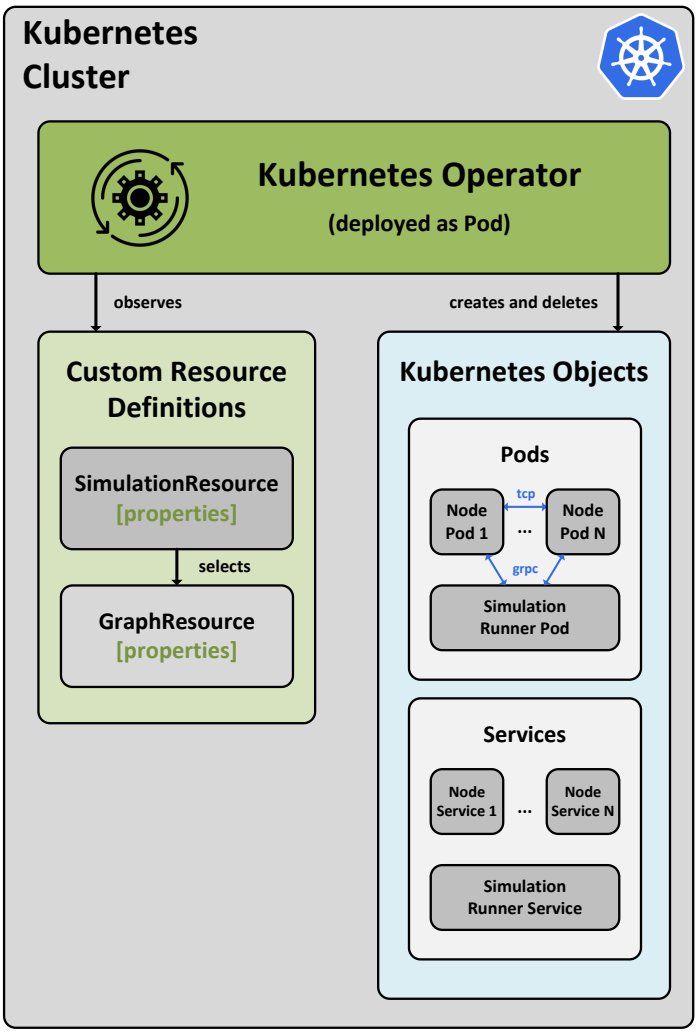


Abbildung 5.2: Funktionsweise Simulation Operator

unterstützen. Dementsprechend werden Änderungen an der Simulationsressource ignoriert. Dadurch kann sichergestellt werden, dass laufende Simulationen unveränderlich sind und somit kein zusätzliches Fehlerpotential entsteht. Um umgehend Anpassungen geltend zu machen, kann jederzeit ein Neustart der Simulation erfolgen.

Mit der Erzeugung aller Pods und Services ist die Aufgabe des Operators erledigt. Jeder Node Pod initialisiert seinen Container und damit die Service Applikation. Anschließend wird auf eingehende Befehle gewartet, welche von der Simulation Runner Anwendung versendet werden, sobald die erste Gossip-Runde startet. Fehler aufgrund von Nichterreichbarkeit müssen hierbei vermieden werden. Dementsprechend erfolgt die Erzeugung des Simulation Runner Pods erst, wenn alle Node Pods gestartet sind. Dadurch wird sichergestellt, dass alle Anwendungen laufen bevor eine Kommunikation stattfindet.

5.1.2 Data Storage

Die Ergebnisse sollen im *MinIO Object Storage* persistiert werden. Die Verwendung eines Object Storages bietet hierbei gegenüber einer herkömmlichen relationalen Datenbank zahlreiche Vorteile. Ein Object Storage ist besser zur Verarbeitung unstrukturierter Daten geeignet. In diesem Fall produzieren die Simulationen verschiedene Ergebnisse in Abhängigkeit der Algorithmen, Graphen und deren Konfigurationen. Diese Ergebnisse

sind als JSON- oder Bilddateien zu speichern. Ein weiterer Vorteil ist die unbeschränkte Skalierbarkeit, welche durch eine einfache Struktur der Datenhaltung erreicht wird [2]. Die Skalierbarkeit ist für diesen Anwendungsfall besonders wichtig, da parallele beziehungsweise ressourcenintensive Simulationen eine hohe Schreiblast verursachen können. Zudem ist ein Object Storage nicht nur einfach zu installieren und zu nutzen, sondern auch äußerst leistungsstark. Durch simple Zugriffe und flache Hierarchien sind Lese- und Schreibvorgänge sehr performant.

Eine Alternative zur Verwendung eines *Amazon S3 Object Storages* wäre die Nutzung von *NoSQL-Lösung*. Diese Lösungen ermöglichen die Aktualisierung und Änderung von Objekten beziehungsweise Dateien. Dieser Vorteil kann jedoch für das Projekt ignoriert werden, denn Simulationsergebnisse stellen Momentaufnahmen dar. Eine spätere Änderung dieser ist weder sinnvoll noch erlaubt. Der Object Storage bietet bei den gegebenen Anforderungen weitere Vorteile, mit denen NoSQL-Lösungen nicht dienen können. Zur Visualisierung von Simulationen oder auch bei der Evaluation ist auch die Erzeugung von Bilddateien oder Animationen variabler Größe denkbar. Komplexe Simulationen können potenziell erhebliche Datenmengen erzeugen, die hier direkt gespeichert werden können.. Außerdem bleiben alte Ergebnisse stets relevant, wodurch eine Archivierung dieser sinnvoll ist. Ein solcher Vorgang kann einfach mit inem Object Storage abgebildet werden. Darüber hinaus ist eine Skalierung simpler und effizienter durchführbar als bei anderen Lösungen. Aus diesen Gründen fiel die Wahl zur Haltung der Simulationsergebnisse auf einen Object Storage. Konkret wurde MinIO als Open-Source-Server ausgewählt. Für MinIO spricht die Performanz, Skalierbarkeit und die weite Verbreitung. Hier steht außerdem eine ausführliche Dokumentation und eine umfangreiche SDK zur Verfügung. Ein weiterer Grund ist die bereits erlangte Expertise aus früheren Projekten sowie positive Erfahrungen bei der Anwendung.

Die Ergebnisse sollen als Objekte mit zeitlichem Kontext im JSON-Format abgelegt werden. Die Daten beinhalten dann unter anderem Konfigurationen, Metadaten, Konvergenzverhalten und Werthistorien. Ziel ist es möglichst alle anfallenden Daten zu speichern, um später möglichst viel Informationen auswerten zu können. Optional sind grafische Visualisierungen, z.B. in Form von Momentaufnahmen des Graphen in den jeweiligen Gossiping-Runden. Mit MinIO kann man Objekte ebenfalls in Verzeichnissen strukturieren. Dadurch erhält man die Möglichkeit, Simulationen in Reihen zu strukturieren. Für jede ausgeführte Testreihe können dann verschiedene Ergebnisdateien in einem Ordner gruppiert werden. Eine hierarchische Organisation entsprechend dem Zeitpunkt der Erhebung oder dem Namen der Simulation ermöglicht eine schnelle Navigation bei der Suche nach Datensätzen. Ebenfalls wird die Umsetzung einer Evaluationsroutine zum Vergleich verschiedener Testreihen erleichtert und die Übersichtlichkeit steigt.

5.1.3 Pod Applikationen

Die Node Pods und der Simulation Runner Pod werden dynamisch durch den Operator verwaltet. Ihr Lebenszyklus deckt sich mit der Simulationsausführung. Sie werden vom Operator über ein Container-Image erzeugt und über Umgebungsvariablen konfiguriert. Die verwendeten Container-Images werden hierbei ebenfalls über das Docker Hub Repository bereitgestellt. Für den jeweiligen Entwicklungsstand wird ein Tag gesetzt, über den die Images dann refernziert werden können.

Beim Start der Container der Node Pods führen diese die Node-Service-Anwendung aus. Sobald diese startet, erfolgt die Initialisierung eines *TCP-Sockets*. Durch Abhören des TCP-Port wird auf einkommende Verbindungen gewartet. Wird eine Verbindung zwischen zwei Nodes hergestellt, so können beide Nachrichten empfangen (*recv*) und senden (*send*). Eine paarweise Kommunikation und ein entsprechender Datenaustausch

kann ermöglicht werden. Jeder Node kann dann den erhaltenen mit seinem lokalen Wert vergleichen und das Minimum als neuen Wert setzen.

Der Simulation Runner wird gleichermaßen über ein eigenes Image erzeugt und konfiguriert. Er steuert die Nodes über RPCs. Dazu wird das *gRPC Framework* eingesetzt, welches das RPC-Protokoll weiterentwickelt. gRPC basiert auf HTTP/2 und Protocol Buffers. Protocol Buffers ist ein Werkzeug sowie eine Sprache zur Datenserialisierung. Das Werkzeug ermöglicht es, Schnittstellen programmiersprachenunabhängig in einer IDL (hier proto 3) zu spezifizieren [69]. Infolgedessen können Datenstrukturen, Dienstmethoden sowie deren Ein- und Ausgaben definiert werden. In Listing 5.1 ist der Aufbau der proto-Datei dargestellt, die im Rahmen der prototypischen Entwicklung verwendet wurde, um einen Gossip-Austausch abzubilden. Mit dem Protocol Buffer Compiler *protoc* sowie dem entsprechenden Plugin von gRPC kann aus dieser Spezifikation der Client- und Server-Code generiert werden. Bei gRPC sind die Clients Anwendungen, die Anfragen an einen Server senden, um Dienste aufzurufen. Wird dies auf die Nodes und den Simulation Runner übertragen, so sind die Nodes die Server-Anwendungen, während der Simulation Runner als Client agiert. Die Generierung des Codes erzeugt eine Schnittstelle, den GossipServicer, welche durch die Node-Service-Anwendung implementiert wird. Sie definiert letztendlich die konkrete Logik, deren Ausführung bim Aufruf der Servicemethoden erfolgt. Für den Client wird mit sogenannten *Stubs* gearbeitet, welche analog zu Proxies funktionieren. Das heißt, sie agieren als Vermittler zwischen Clients und Servern und erleichtern die Kommunikation. Also realisieren sie die Weiterleitung der Aufrufe und die Rückgabe der Ergebnisse. Der Simulation Runner muss eine Liste solcher Stubs verwalten. Hierbei hat die Liste jeweils einen Eintrag pro Knoten im Netzwerkgraphen und der Runner muss vom jeweiligen Knoten auf den Stub schließen können.

Folglich muss einerseits für den paarweisen Informationsaustausch eine Kommunikation zwischen den Node Pods realisiert werden. Andererseits muss der Simulations-Runner mit den Node-Pods kommunizieren, um die Steuerung der Gossiping-Runden zu ermöglichen. Da sowohl die Nodes als auch der Simulation Runner in Pods ausgeführt werden, muss diese Kommunikation über Services abgebildet werden. Dazu erzeugt der Simulation Operator die jeweiligen Services, welche die entsprechenden Ports für TCP und gRPC freigeschalten. Die Kommunikationskanäle sind in Abbildung 5.3 grafisch dargestellt.

```
1 syntax = "proto3";
2 package gossip;
3
4 service Gossip {
5   rpc Gossip(GossipRequest) returns (GossipResponse) {}
6 }
7
8 message GossipRequest {}
9 message GossipResponse {}
```

Listing 5.1: Beispiel proto3 anhand Prototyp

5.2 Low-Level-Design

Mit dem *High-Level-Design* wird der Gesamtaufbau in die jeweiligen Bausteine zerlegt und deren Funktionalität festgelegt. Anschließend müssen diese Bausteine im Detail konzipiert werden. Das *Low-Level-Design* befasst sich mit der Beschreibung des inneren Aufbaus der Komponenten. Im Folgenden wird das Konzept der einzelnen Systemteile erläutert und auf wichtige Designentscheidungen eingegangen.

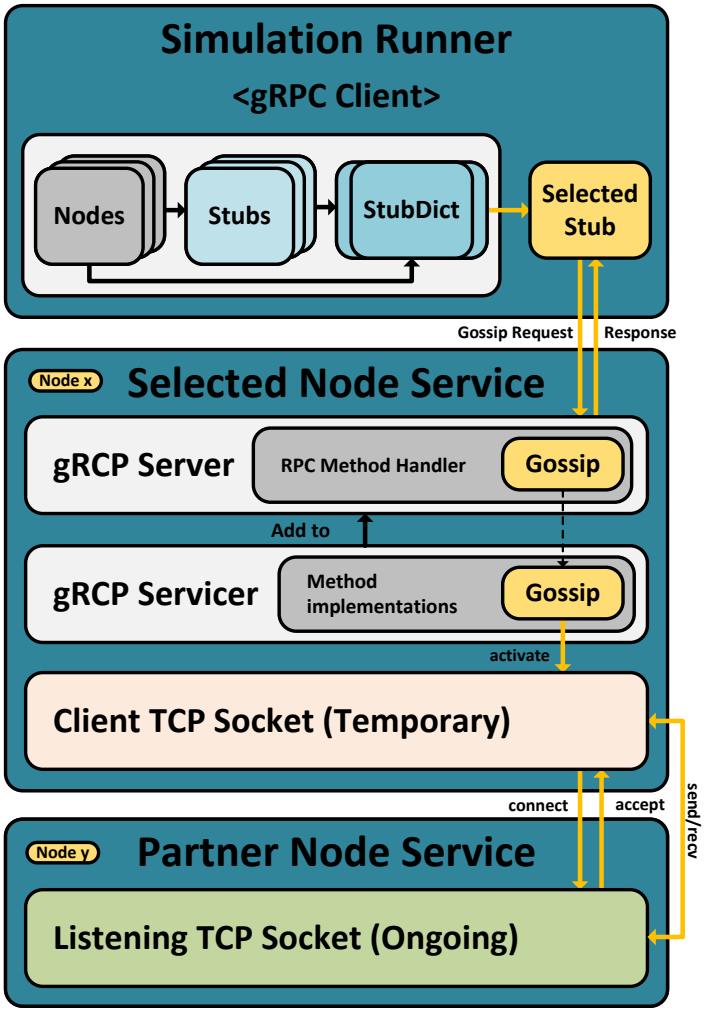


Abbildung 5.3: Kommunikationsabläufe der Simulation Pods

5.2.1 Ablauf der Generierung von Graphen

Zur Generierung von Graphen soll die Python Library *NetworkX* eingesetzt werden. Die Bibliothek verfügt über eine Vielzahl von Erzeugungsalgorithmen für unterschiedliche Graphtypen. Zudem ermöglicht sie einen einfachen Umgang mit den generierten Graphen. Es gibt eine große Auswahl an Algorithmen und Funktionen zur Graphanalyse, wodurch unter anderem die Bestimmung von Communities und die Berechnung der Modularität durchgeführt werden kann. Bei der prototypischen Entwicklung hat sich die Bibliothek bewährt. Hier konnte auf viele benötigte Funktionen zur Erzeugung, Modifikation, Analyse und Visualisierung von Graphen zurückgegriffen werden. Eine Darstellung der Graphen konnte dabei direkt über NetworkX oder mit Hilfe von anderen Bibliotheken erfolgen. Der Operator muss die Graphen auslesen und anschließend die Netzwerktopologie aufbauen. Dabei muss er außerdem Operationen wie die Erkennung von Communities durchführen, um den Node-Services alle notwendigen Informationen bereitstellen zu können. Dementsprechend muss auch er auf die Graphbibliothek zugreifen. Da Python bereits als Programmiersprache für den Operator ausgewählt wurde, bietet es sich somit an für die Grapherzeugung auch Python zu verwenden. Dadurch kann die gesamte Logik zur Verwaltung der Netzwerkgraphen in allen Anwendungen mit der gleichen Bibliothek realisiert werden. Ein weiterer Vorteil der Bibliothek ist die nahtlose Integration von anderen wissenschaftlichen Bibliotheken wie *NumPy* und *SciPy*. Diese stellen erweitere

Funktionalität zur Array-Verwaltung sowie für komplexe mathematische Berechnungen bereit. Zudem erleichtern sie den Umgang mit großen Datenmengen. Infolgedessen steht zur Abbildung der Grapherzeugungsroutinen ein sehr großer Funktionsumfang bereit.

Die Generierung von Graphen soll über ein CLI-Tool erfolgen. Dabei müssen verschiedene Routinen umgesetzt werden, um so eine hohe Flexibilität bei der Erzeugung zu erreichen. Ebenso soll die Bedienung des CLI möglichst simpel sein. Folgende Ansprüche müssen durch das Tool abgedeckt werden:

- *Erzeugung verschiedener Graphen:* Es erfordert die Implementierung verschiedener Routinen, um unterschiedliche Graphtypen zu generieren. Abhängig vom Typ müssen anschließend spezifische Parameter zur Erzeugung festgelegt werden.
- *Erweiterbarkeit*: Andere Graphtypen können gegebenenfalls im späteren Projektverlauf hinzukommen. Eine solche Erweiterung muss mit minimalem Aufwand möglich sein.
- *Erzeugung mehrerer Graphen*: Für die Simulationsreihen müssen zahlreiche Graphen generiert werden können. Effizient ist deshalb die Möglichkeit zu schaffen mehrere Graphen mit gleichen Parametern zu generieren. Zudem soll eine Parameterstaffelung, also eine gleichzeitige Angabe verschiedener Parameter, erfolgen können. Ein Anwendungsbeispiel wäre hier die dynamische Erzeugung von Popularitätsgraphen. Dabei ist die Betrachtung des Verhältnisses zwischen Intra- und Inter-Community-Kanten maßgeblich. Es sollen dementsprechend Graphen mit aufsteigender Anzahl an Intra- und absteigender Anzahl an Inter-Community-Kanten erzeugt werden. Ziel ist es, eine Reihe solcher Graphen mit unterschiedlichen Parametern in einem Schritt zu generieren.
- *Graphrepräsentation*: Ein Graph soll sowohl als Adjazenzliste als auch direkt als Graphobjekt entsprechend der CRD speicherbar sein. Dadurch kann der Aufwand der manuellen Erstellung vermieden werden. Dies ist sinnvoll, da die Graphressource alle Parameter beinhalten muss, die zuvor über das CLI abgefragt wurden.
- *Einfache Navigation und Bedienung*: Der Prozess soll so gestaltet werden, dass der Nutzer möglichst wenig Eingaben tätigen muss.

Um die Erweiterbarkeit und die User Experience (UX) zu gewährleisten, wurden detaillierte Flussdiagramme erstellt. Mit Hilfe dieser Diagramme konnte das Konzept zur Graphgenerierung optimiert werden. Bei der Erzeugung mehrerer Graphen konnten dadurch mehrere Nutzerinteraktionen im Vergleich zum ersten Prototyp eingespart werden. In Abbildung 5.4 ist der generelle Ablauf des Erzeugungsprozesses visualisiert. Es erfolgt dabei eine Abbildung von größeren Teilprozessen als Subprozesse, welche wiederum in separaten Sequenzdiagrammen dargestellt sind. Bei der Entwicklung werden die einzelnen Programmteile voneinander getrennt betrachtet. Jeder Teil realisiert eine abgegrenzte logische Aufgabe, man spricht hierbei von *Separation of Concerns (SoC)* [98]. Infolgedessen werden lange komplexe Funktionen vermieden, was somit auch die Gesamtkomplexität beschränkt. Der resultierende Code wird folglich auch lesbarer und wartbarer.

Das CLI wird mit einem Befehl gestartet, der den Algorithmus festgelegt. Zudem erfolgt hier auch die Angabe der Anzahl an zu erzeugenden Graphen sowie des Graphtyps. Anschließend folgt die initiale Auswahl der Parameter, die für den ausgewählten Algorithmus benötigt werden. Im Nachgang wird die Erzeugung aller Graphen durchgeführt. Der Nutzer kann sich diese dann direkt darstellen lassen oder zur Visualisierung exportieren. Dadurch kann er prüfen, ob die Graphen so aussehen, wie er sich das bei Eingabe der Parameter vorgestellt hat. Falls dies nicht der Fall ist, so kann der Prozess

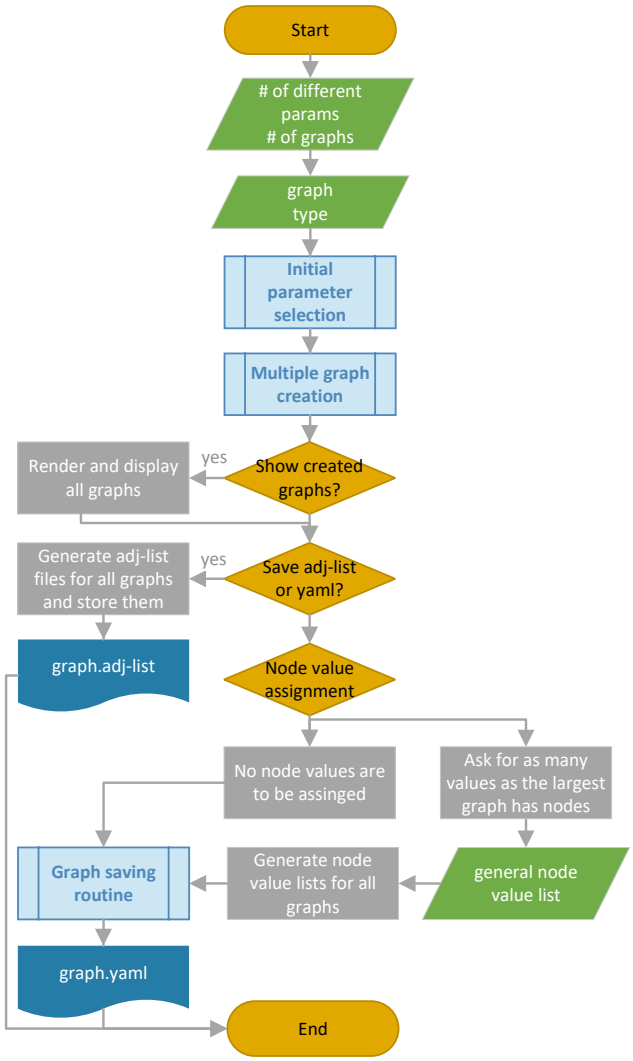


Abbildung 5.4: Ablauf der Generierung von Graphen

jederzeit abgebrochen werden. Ist der Nutzer hingegen zufrieden mit den Graphen, so kann er fortfahren. Er muss sich entscheiden, ob die Graphen lediglich als Adjazenzlisten zu speichern sind oder er Graphressourcen in Form von YAML-Dateien erzeugen möchte. Mit der Speicherung als Adjazenzliste endet die Routine. Für die Erzeugung der YAML-Dateien, kann noch eine Zuweisung von Knotenwerten erfolgen. Feste Knotenwerte können für die Testreihen eingesetzt werden, wenn verschiedene Algorithmen auf den gleichen Graphen zu testen sind. Standardmäßig werden zufällige Knotenwerte von den Node Pods selbst gewählt. Dadurch hat man, wenn ein direkter Vergleich zwischen Algorithmen erfolgen soll, eine weitere zufällige Komponente neben der Auswahl der Gossip-Partner. Die Zufälligkeit kann jedoch durch eine feste Zuteilung eliminiert werden, wodurch eine bessere Vergleichbarkeit erzielt wird. Dabei muss der Nutzer so viele Werte eingeben, wie der größte Graph Knoten hat. Anschließend erfolgt die Zuweisung der Werte zu den jeweiligen Graphen. Der Prozess endet schließlich mit der Erzeugung der YAML-Dateien durch die Speicherroutine. Im Folgenden werden nun die einzelnen Teilprozesse beschrieben.

5.2.1.1 Grapherzeugungsalgorithmen

Der erste Subprozess beinhaltet die Auswahl des Graphtyps, wodurch für alle weiteren Abläufe der verwendeten Erzeugungsalgorithmus festgelegt wird. Je nach Algorithmus müssen verschiedene Parameter abgefragt werden. Ist nur ein Graph zu generieren, so muss nur eine Abfrage erfolgen. Ist eine Mehrfacherzeugung gewünscht, so müssen die Parameter wiederholt abgerufen werden. Der Ablauf der Parameterauswahl ist in Abbildung 5.5 visualisiert. Die spezifischen Parameterabfragen werden in separate logische Blöcke aufgeteilt. So ist eine beliebige Wiederholung möglich, wenn mehrere Graphen auf einmal zu erzeugen sind.

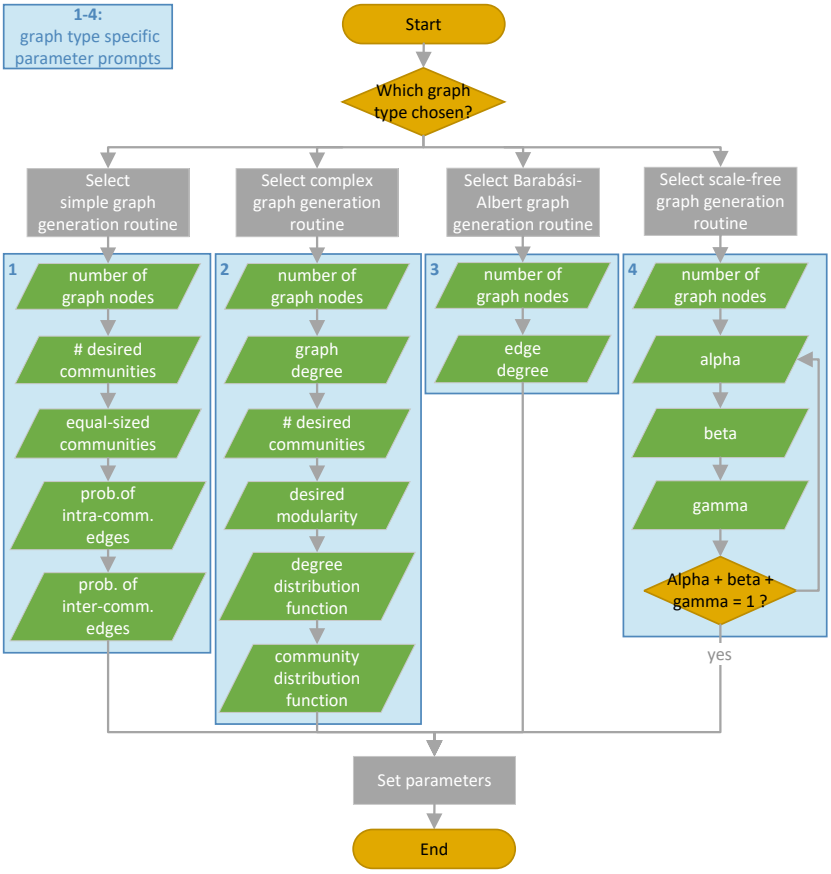


Abbildung 5.5: Auswahl von Graphtypen

Zur Selektion der Routinen zur Grapherzeugung musste eine Evaluation erfolgen. Dabei war es notwendig festzulegen, welche Netzwerke simuliert werden sollen. Unter Berücksichtigung dieser konnten dann entsprechende Verfahren ausgewählt werden. Die Auswahl der Erstellungsverfahren wurde initial auf folgende vier Methoden festgelegt:

Simple Graph Generation Routine:
Der Nutzer muss die Gesamtzahl der Knoten im Graphen und die gewünschte Anzahl der Communities im Graphen eingeben. Ebenso muss er festlegen, ob die Communities gleich groß sein sollen. Des Weiteren muss er die Wahrscheinlichkeit von Kanten innerhalb und außerhalb der Communities definieren. Zur Erzeugung werden in Abhängigkeit der Anzahl gewünschter Communities Knoten logisch gruppiert. Kanten innerhalb der logischen Gruppen werden entsprechend der definierten Wahrscheinlichkeiten hinzugefügt. Gleichermaßen erfolgt auch das Hinzufügen von Kanten zwischen Mitgliedern von verschiedenen Gruppen. Das

Verfahren ähnelt der Erzeugung von Popularitätsnetzwerken nach Singh [105]. Singh verwendet zur Generierung eine feste Vorgabe der Anzahl von Inter- und Intra-Community-Kanten. Dieser Erzeugungsalgorithmus nutzt stattdessen prozentuelle Wahrscheinlichkeiten. Durch dieses Verfahren können also Graphen mit variabel stark ausgeprägten Community-Strukturen generiert werden.

Complex Graph Generation Routine:
Dieses Verfahren wurde im Rahmen des Papers [99] veröffentlicht, wo Community-Strukturen in biologischen Netzwerken untersucht werden. Die Routine erzeugt komplexe synthetische Graphen, die realen biologischen Netzen nachempfunden sind. Dabei werden verschiedene Parameter verwendet, um Graphen zu erzeugen, die eine möglichst zufällige Struktur haben. Als Parameter werden die Anzahl an Knoten sowie an Communities, der Grad des Graphen, die Zielmodularität sowie Distributionsfunktionen für den Grad des Graphen und die Communities abgefragt. Der zugehörige Code zu diesem Paper ist als Python Skript öffentlich verfügbar. Das Skript soll in die eigene Anwendung eingebunden und zur Grapherzeugung genutzt werden. Maßgeblich ist hierbei die Definition einer Zielmodularität, da diese eine wichtige Metrik für die Testreihen darstellt.

Scale-Free Graph Generation:
Es soll eine Methode zur Erzeugung von skalenfreien Graphen eingesetzt werden. Diese weisen eine Potenzgesetzverteilung der Knotengrade auf. Das heißt, es gibt viele Knoten mit wenigen Kanten und wenig Knoten mit vielen Kanten. Zur Erzeugung von skalenfreien Graphen wird die Angabe der Knotenanzahl sowie verschiedener Wahrscheinlichkeiten benötigt. Definiert werden müssen hierbei α, β und γ. Durch α wird beim Hinzufügen eines neuen Knotens dessen Verbindung zum Graphen bestimmt. Dabei erfolgt die Auswahl nach der Eingangsgradverteilung. Mit steigenden Werten von α wird ein höheres Clustering erzielt, da Hubs aufgrund ihres hohen Eingangsgrades bei der Auswahl präferiert werden. Man spricht hierbei von *Preferential Attachment* [109]. Die Wahrscheinlichkeit β dient der Bestimmung des Start- und Endknotens beim Hinzufügen einer zusätzlichen Kante. Hierbei werden die Knoten nach der Eingangs- und Ausgangsgradverteilung selektiert. In Korrelation mit β steigt die Zufälligkeit des Graphen. Die Verbindung eines neuen Knotens zum Graphen wird auch durch den Parameter γ festgelegt. Dabei werden die Kanten nach der Ausgangsgradverteilung ausgesucht. Je höher γ wird, desto stärker verbunden ist der Graph.

Barabási-Albert Graph Generation:
Die Barabási-Albert-Methode wird ebenso zur Erzeugung skalenfreier Graphen verwendet. Auch hier wird Preferential Attachment angewandt, um eine Potenzgesetzverteilung der Knotengrade zu erreichen. Neue Knoten werden mit hoher Wahrscheinlichkeit mit Knoten mit bereits vielen Kanten verbunden. Bei den erzeugten Graphen manifestieren sich natürliche Community-Strukturen, welche sich um die Hub-Knoten herum bilden. Dabei sind die Community-Strukturen je nach Gradverteilung unterschiedlich stark ausgeprägt. Die Eigenschaften von skalenfreien Netzwerken können auch in realen Netzwerken wie beispielsweise Computernetzen beobachtet werden.

Alle Verfahren müssen garantieren, dass die erzeugten Graphen vollständig miteinander verbunden sind. Um dies zu gewährleistet, kann ein entsprechender Algorithmus entwickelt werden. Der Algorithmus fügt dabei zufällig Kanten zwischen den Mitgliedern der separierten Graphteile ein. Dies wird rekursiv, solange wiederholt bis der Graph

vollständig miteinander verbunden ist. Pseudocode für eine solche Implementierung ist in 5.2 zu finden. Alle Graphalgorithmen, die nicht immer vollständig verbundene Graphen erzeugen, können durch diese Funktion erweitert werden.

```
def make_fully_interconnected(graph):
    components = get_connected_components(graph)
    num_components = length(components)
    if num_components < 2:
        return graph

    component1 = randomly_select_component(components)
    component2 = randomly_select_component(components)
    while component1 == component2:
        component2 = randomly_select_component(components)

    node1 = randomly_select_node(component1)
    node2 = randomly_select_node(component2)
    new_graph = create_copy_of(graph)
    add_edge(new_graph, node1, node2)

    return make_fully_interconnected(new_graph)
```

Listing 5.2: Pseudocode Graph vollständig verbinden

Die Implementierung der Auswahl der Algorithmen soll dynamisch erfolgen. Dementsprechend soll ein Design-Pattern angewandt werden, das einen Austausch der graphtypenspezifischen Logik während dem Programmablauf ermöglicht. Anbieten würde sich hier das *Strategy-Pattern* [45]. Es wird eingesetzt, wenn Algorithmen, Funktionen oder generell Verhalten während der Laufzeit austauschbar sein muss. Diese werden durch die Strategien als separate Abläufe dargestellt. Das Pattern ermöglicht die Trennung zwischen der Auswahllogik und der Programmausführung. Zu Beginn der Routine erfolgt die Auswahl des Erzeugungsalgorithmus durch den Nutzer, wodurch die Strategie festgelegt wird. Im weiteren Verlauf werden generische Funktionen und Objekte genutzt, welche unabhängig von der gewählten Strategie sind. Erst zur Laufzeit erfolgt die Konkretisierung dieser, wodurch die Logik des ausgewählten Algorithmus ausgeführt wird.

5.2.1.2 Mehrfacherzeugung von Graphen

Der zweite Subprozess ist die Mehrfacherzeugung von Graphen. Der Ablauf dieses Prozesses ist in Abbildung 5.6 grafisch dargestellt. Hier wird die Generierungsstrategie entsprechend des ausgewählten Graphtyps gesetzt. Dabei können verschiedene Eingaben erfolgen, um die Anzahl an zu erstellenden Graphen festzulegen. Die Eingabe einer einzelnen Zahl N führt zu Erzeugung von insgesamt N Graphen. Diese werden alle mit den gleichen Parametern erstellt, die zu Beginn eingeben wurden. Ebenso kann die Verwendung von verschiedenen Parametern für unterschiedliche Graphen erfolgen. Dazu müssen zwei Zahlen *MxN* übergeben werden. M steht für die Anzahl an verschiedenen Parameterkombinationen, die der Nutzer angeben möchte. Es werden dann für jede Kombination N Graphen erzeugt. Dies MxN-Erzeugung unterstützt zwei Vorgänge, nämlich die Einzeleingabe und die Interpolation. Beim ersten Vorgang gibt der Nutzer alle M Parameterkombinationen nacheinander ein. Möglich ist es aber auch nur die Parameter für die erste und letzte Gruppe an Graphen zu definieren. Die Parameter der weiteren N-2 Graphen werden dann interpoliert. Schließlich endet der Vorgang mit der Erzeugung der Graphen. Dabei werden die verwendeten Parameter für die Benamung der Graphen und als Eigenschaften für die Graphressourcen verwendet.

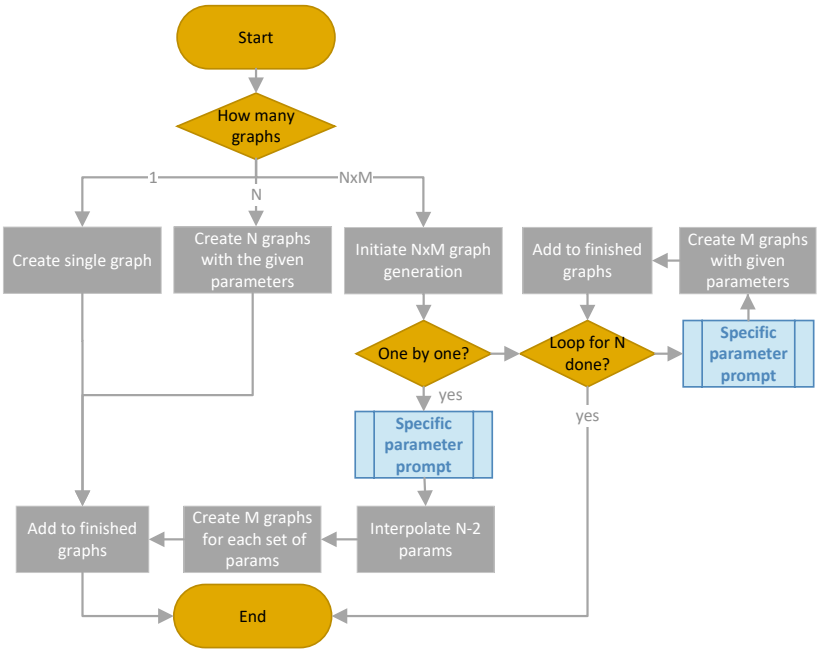


Abbildung 5.6: Mehrfacherzeugung von Graphen

5.2.1.3 Erzeugen von Kubernetes Graphressourcen

Eine administrative Erleichterung stellt die Erzeugung von Kubernetes Graphressourcen dar. Dazu wurde zuerst die CRD für die Graphobjekte erstellt. Diese wurde in der prototypischen Entwicklung bereits festgelegt und nun entsprechend erweitert. Der Aufbau der CRD kann in Listing 5.3 nachvollzogen werden. Sie definiert den Aufbau von Graphobjekten und legt die Typen der Eigenschaften fest. Mit den *additionalProperties* können beliebige zusätzliche Metadaten definiert werden. Lediglich die Adjazenzliste muss bei jedem Graphen zwingend definiert werde, denn ohne diese kann keine Simulation durchgeführt werden.

Listing 5.4 beinhaltet ein Minimalbeispiel eines Graphobjektes. Ähnliche Graphobjekte sollen auch durch die Routine zur Generierung von Graphen erzeugt werden können. Hierbei soll es möglich sein, Informationen wie der Graphtyp sowie Metadaten basierend auf den Eingabeparametern festzulegen. Außerdem wird die Adjazenzliste bestimmt, indem der Erzeugungsalgorithmus verwendet wird. Diese Daten können somit direkt zur Definition des Graphobjektes verwendet werden. Dabei können die Metriken wie beispielsweise die Modularität in den Eigenschaften (*graphProperties*) hinterlegt werden. Bei einer statischen Zuweisung der Knotenwerte muss eine manuelle Angabe dieser erfolgen (*nodeValueList*). Da die Graphobjekte entsprechend der CRD einen festen Aufbau haben, kann die YAML-Struktur programmtechnisch aufgebaut werden.

In Abbildung 5.7 wird der Ablauf der Speicherung der generierten Graphen als YAML-Dateien dargestellt. Die Graphen erhalten entsprechend der eingegebenen Parameter Namen. Dabei soll die Routine die Angabe eines Namenspräfixes oder auch eines völlig neuen Namens ermöglichen. Bei einer mehrfachen Erzeugung von Graphen werden die duplizierten Namen durch Nummerierungen erweitert. Es muss hierbei beachtet werden, dass Kubernetes-Ressourcen maximal 63 Zeichen lang sein dürfen. Dementsprechend muss gegebenenfalls eine Kürzung des Namens erfolgen. Nach der Festlegung der Namen folgt die Erzeugung der Adjazenzlisten. Anschließend kann der Inhalt der YAML-Dateien mit allen notwendigen Daten befüllt werden. Der Prozess endet mit der Abspeicherung der Dateien in einem entsprechenden Ordner.

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
    name: graphs.gossip.io
spec:
    group: gossip.io
    versions:
    - name: v1
      served: true
      storage: true
      schema:
      openAPIV3Schema:
          type: object
          properties:
              spec:
                  type: object
                  properties:
                      adjacencyList:
                        type: array
                        items:
                          type: string
                      graphType:
                        type: string
                        default: 'undefined'
                      valueList:
                        type: string
                      graphProperties:
                        additionalProperties:
                            type: string
                        type: object
                  required: ['adjacencyList']
    scope: Namespaced
    names:
        plural: graphs
        singular: graph
        kind: Graph
        shortNames:
        - g
```

Listing 5.3: Graph Custom Resource Definition

```yaml
apiVersion: gossip.io/v1
kind: Graph
metadata:
  name: my-simple-graph
spec:
  adjacencyList:
  - "1 2 4,"
  - "2 3 4,"
  - "3 4,"
  - "4 5,"
  - "5 6,"
  - "6,"
  graphType: "simple"
  valueList: "11,22,33,44,55,66"
  graphProperties:
    nodeCount: "6"
    numEdges: "7"
    modularity: "0.62"
```

Listing 5.4: Minimalbeispiel Graphobjekt

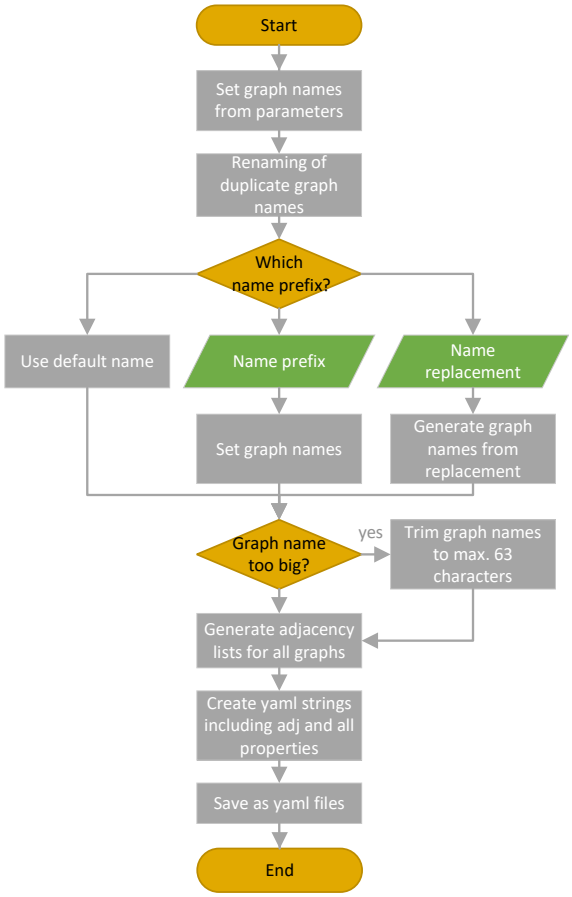


Abbildung 5.7: Speichern von Graphen als YAML-Dateien

5.2.2 Abstraktion durch Simulation

Der Hauptaspekt der Arbeit ist die Simulation von Gossiping, wobei das Protokoll die Ermittlung eines netzwerkweiten Minimums durchführen soll. Die Simulationen sollen der Untersuchung des Konvergenzverhaltens in Bezug auf verschiedene Algorithmen und Arten von Netzwerken dienen. Dabei soll zur Ausführung der Simulationen ein Kubernetes-Cluster verwendet werden. Hier werden die Netzwerke anhand von Graphen aufgebaut, wobei jeder Knoten durch einen Pod repräsentiert wird. Die Abbildung der Netzwerkknoten als Pods ermöglicht eine realitätsnahe Darstellung eines echten Netzwerks. Dadurch können die unterschiedlichen Gossip-Algorithmen in einer kontrollierten Umgebung untersucht werden. Um das Konvergenzverhalten des Gossiping-Protokolls genau bewerten zu können, erfolgt die Ausführung der Simulation in Runden. Innerhalb einer Runde wird jeder Knoten nur einmal aktiviert, wodurch dieser einen Nachbarn auswählt und mit diesem kommuniziert. Durch eine rundenbasierte Ausführung kann eine detaillierte Analyse des Verhaltens und der Konvergenzgeschwindigkeit erfolgen. Außerdem gewährleistet dieser Ansatz eine präzise Beurteilung und Bewertung der Protokolle.

Bei der Simulation des Gossiping-Protokolls werden verschiedene Richtlinien festgelegt, um die Komplexität zu reduzieren. Durch diese Beschränkungen steigt die Verständlichkeit, Überprüfbarkeit und Vergleichbarkeit der Ergebnisse. Ebenso kann ein größerer Fokus auf die Analyse der Konvergenzgeschwindigkeit gelegt werden, indem eine gemeinsame Grundlage für alle Untersuchungen gelegt wird. Im Anschluss werden nun die einzelnen Bestimmungen erläutert:

Datenveränderungen:
Jeder Knoten erhält zu Beginn einen Wert zugeteilt oder wählt diesen aus. Änderungen der Knotenwerte können ausschließlich über das Gossiping-Verfahren vorgenommen werden, externe Modifikationen sind nicht gestattet.

Abwechselnde Kommunikation:
Mit dem rundenbasierten Ansatz wird eine genaue Feststellung des Konvergenzverhaltens ermöglicht. Zudem sollen in jeder Runde die Node Pods einzeln in der gleichen Reihenfolge zum Durchführen des Gossipings aufgerufen werden. Der Simulation Runner führt die zentrale Steuerung durch und ermöglicht eine direkte Auswertung. Bei dezentralen Gossiping-Verfahren ist es schwerer, den Ablauf sowie die Konvergenz auszuwerten.

Statische Netzwerkstrukturen:
Während der Ausführung des Protokolls bleiben die Netzwerke statisch. Das heißt es kommen keine neuen Knoten hinzu und es fallen auch keine Knoten aus. Außerdem bleiben die Verbindungen zwischen den Knoten konsistent. Dadurch wird sichergestellt, dass sich die Anwendungen ausschließlich auf die Kommunikation und den Informationsaustausch konzentrieren können.

Keine Ausfälle:
In der Praxis gibt es verschiedene Verfahren zum erneuten Senden oder auch zur Wiederaufnahme ausgefallener Knoten ins Netz. Es wird bei der Simulation jedoch von einem optimalen Ablauf ausgegangen. Dementsprechend werden Nachrichten im Netzwerk immer erfolgreich ausgetauscht und es kommt zu keinen Verbindungsabbrüchen oder Paketverlusten. Darüber hinaus fallen keine Knoten aus und jede Anwendung verfügt über ausreichend Ressourcen, um ihrer Funktionalität nachzukommen.

Homogene und aktive Knoten:
Alle Knoten werden als gleich angesehen und erhalten die gleiche Funktionalität. Zudem ist ein Großteil der benötigten Parameter zur Initialisierung gleich. Ausnahmen beinhalten hierbei den Startwert und die Informationen über die Nachbarn. Sobald die Steuerungskomponente aktiv wird müssen alle Knoten stets bereit sein, ihre Funktionalität auf Abruf auszuführen.

Ausführungsdauer:
In der Praxis erfolgt die Leistungsbewertung anhand der Ausführungsdauer. Diese wird gemessen vom Start des Gossipings bis eine Konvergenz eintritt. Zur Auswertung der Simulationen soll die Anzahl an benötigten Runden verwendet werden. Dadurch können die Ergebnisse besser verglichen werden. Bei großen Netzwerken wird die Konvergenz meist an Schwellenwerten (z.B. 95 Prozent der Knoten) festgelegt. Für diese Arbeit erfolgt die Beendigung erst bei vollständiger Konvergenz, da die netzwerkweite Informationsverbreitung simuliert wird.

Der Zweck dieser Richtlinien besteht darin, die Simulationsausführung zu vereinfachen. Ebenso wird die Auswertung simpler und die Vergleichbarkeit der Ergebnisse steigt. In der Praxis können diese Bestimmungen jedoch kaum gewährleistet werden. Die Vereinfachung wirkt sich aber nicht negativ auf die Güte der gewonnenen Erkenntnisse aus. Bei der Durchführung von Simulationen wird oft auf einer höheren Abstraktionsebene gearbeitet. Details und Komplexitäten von realen Systemen oder Methoden werden

bewusst vernachlässigt. Diese Abstraktion ermöglicht die Konzentration auf die wesentlichen Aspekte. Außerdem dienen Simulationen der Untersuchung bestimmter Verhaltensweisen und Dynamiken. Trotz Veränderungen an der konkreten Ausgestaltung können Rückschlüsse auf allgemeine Verhaltensmuster oder Trends gezogen werden. Falls eine Beschleunigung des Konvergenzverhaltens durch eine Optimierung der Partnerwahl erfolgen kann, kann dies durch die Simulation nachgewiesen werden. Auch wenn die Optimierung anhand von Gossiping-Algorithmen mit globaler Berechnung von Gewichten nachgewiesen wurde, können die Erkenntnisse auf die dezentralen Verfahren übertragen werden. Des Weiteren dienen die Simulationen zum Vergleich von verschiedenen Alternativen. unterschiedliche Netzwerke sowie Algorithmen und deren Konfigurationen können gegenübergestellt werden. Relevante Erkenntnisse können gewonnen werden, solange die Untersuchungen in derselben Umgebung unter den gleichen Umständen erfolgen. Wenn Auswirkungen von Änderungen (z.B. verschiedene Konfigurationen der Algorithmen) verwendet werden, um die Ergebnisse von Modellen zu bewerten, spricht man von *Sensitivitätsanalysen*.

5.2.3 Ablauf einer Simulation

Der Simulationsablauf ist im Anhang B.2 in Abbildung B.3 dargestellt. Dabei werden Nutzerinteraktionen sowie Systeminteraktionen abgebildet, um sowohl den Informationsfluss als auch das Systemverhalten zu veranschaulichen. Im Folgenden wird der dargestellte Ablauf ausführlich beschrieben.

Im ersten Schritt erzeugt der Nutzer einen oder mehrere Netzwerke. Die Netzwerke werden mit dem CLI-Tool generiert und durch Graphen repräsentiert. Dabei erfolgt die Erstellung von Kubernetes-Ressourcen für die Graphen, welche neben der Netzwerktopologie auch Parameter und Metadaten beinhalten. Diese Graphen werden dann als Objekte in Kubernetes angelegt und können so in beliebigen Simulationen verwendet werden.

Anschließend wird die Ressource für die Simulation in Kubernetes erzeugt, welche zuvor erstellte Graphobjekte referenziert. Die Simulationsressource beinhaltet spezifische Parameter zur Konfiguration sowie Metadaten zur Auswertung. Der Simulation Operator überwacht die Kubernetes-API und reagiert automatisch auf die Erzeugung der Simulationsressource. Seine Verantwortung besteht in der Erstellung der Knoten als Node Pods, wobei jeder Pod einen voll funktionsfähigen Netzwerkknoten darstellt. Dazu implementiert er einen Kubernetes-Client und kann so Befehle an die Kubernetes-API senden. Darüber hinaus verwaltet Kubernetes die Ressourcenzuteilung und den Lebenszyklus der Pods. Durch Kubernetes kann somit eine Skalierung von Simulationen erfolgen. Die Größe wird dabei nur durch die maximale Anzahl an Pods, die das Cluster gleichzeitig ausführen kann, limitiert. Nach der Initialisierung der Pods werden die Container-Anwendungen ausgeführt, welche dann über die Umgebungsvariablen konfiguriert werden. Die Umgebungsvariablen umfassen verschiedene Daten wie den ausgewählten Algorithmus sowie Informationen über die Nachbarknoten, die für den Gossip-Austausch benötigt werden. Wenn alle Node Pods gestartet sind, also sich im Ready-Status befinden, wird der Simulation Runner Pod bereitgestellt. Der Runner ist verantwortlich für die Koordination der rundenbasierten Ausführung. Zuletzt werden alle erforderlichen Dienste zur Kommunikation zwischen den Knoten eingerichtet.

Bei Simulationsstart fragt der Simulation Runner alle initialen Werte über gRPC-Calls ab. Anschließend beginnt der Runner mit der Routine und gibt nacheinander den Befehl an jeden Host das Gossiping durchzuführen. Auch dieser Befehl wird über eine gRPC-Methode abgebildet. Wird ein Knoten aufgerufen, so wählt er entsprechend des festgelegten Algorithmus einen Nachbarn aus. Danach tauschen sich beide Knoten über TCP aus, wodurch eine realstische Netzwerkkommunikation abgebildet werden. Beide

Knoten können nun feststellen, ob der Wert des Partners niedriger ist und gegebenenfalls den eigenen Wert überschreiben. Die Simulation läuft dabei so lange, bis eine Konvergenz erreicht ist. Dies ist der Fall, wenn die Knoten im Netzwerk sich auf ein gemeinsames Minimum geeinigt haben. Sobald die Konvergenz erreicht ist, trägt der Runner Pod die Ergebnisse der Simulation zusammen, welche anschließend im Object Storage persistiert werden. Danach sendet er Stoppbefehle an alle Node Pods über gRPC und beendet dann die eigene Anwendung. Nach kurzer Zeit wechseln alle Simulations-Pods in den Completed-Status, wodurch die genutzten Ressourcen freigegeben werden. Die Logs der Anwendungen bleiben jedoch erhalten, wodurch der Nutzer noch die durchgeführte Simulation evaluieren kann. Zu einem beliebigen späteren Zeitpunkt kann der Nutzer die Simulationsressource löschen. Der Simulation Operator löscht daraufhin alle Dienste und Pods, die im Zusammenhang mit der Simulation erstellt wurden. Dieser Prozess stellt sicher, dass alle Ressourcen entfernt werden und keine unerwünschten Auswirkungen auf das Kubernetes-Cluster haben.

5.2.4 Definition verschiedener Algorithmen

Ein Ziel dieser Arbeit ist es, gewichtete Gossip-Algorithmen zu entwickeln. Informationen über Community-Strukturen sollen hier eingesetzt werden, um das Konvergenzverhalten zu optimieren. Eine Evaluierung der Algorithmen ist erforderlich, inklusive eines Vergleichs untereinander. Um einen Vergleich zu ermöglichen, wird ein Baseline-Algorithmus verwendet, der die Auswahl des Gossip-Partners vollkommen zufällig durchgeführt. Dieser Algorithmus dient als Referenzpunkt für die anderen Algorithmen und kann so verwendet werden, um die erreichte Optimierung festzustellen.

Zusätzlich wird ein Algorithmus namens *WeightedFactor* realisiert. Dieser verwendet eine zusätzliche Gewichtung bei der Auswahl des Gossip-Partners. Als Gewichte werden feste Werte verwendet, die mit den zufällig ausgewählten Wahrscheinlichkeit multipliziert werden. Dabei erfolgt eine Multiplikation nur für Nachbarn die sich innerhalb derselben Community befinden. Dadurch kann die Verteilung von Informationen innerhalb oder außerhalb der eigenen Community bevorzugt werden.

Der zweite neue Algorithmus nennt sich *CommunityProbabilities*. Er berechnet für jeden Nachbarknoten Wahrscheinlichkeiten für dessen Zugehörigkeit zur eigenen Community. Diese Wahrscheinlichkeiten können dann zu einer gezielten Gewichtung eingesetzt werden. Dabei sind sie auch als Faktoren bei der zufälligen Auswahl einzusetzen.

Die Algorithmen können durch Einsatz einer Gedächtnislogik erweitert werden. Durch diese erhalten die Knoten die Fähigkeit sich an vorherige Kommunikationspartner zu erinnern. Hierbei soll eine simple Logik (*Memory*) eingesetzt werden, die jeden Nachbarn, mit dem sich bereits ausgetauscht wurde, benachteiligt. Alternativ kann die Verwendung der *ComplexMemory*-Logik erfolgen. Hier wird für jede abgelaufene Kommunikation eine weitere Verringerung der Wahrscheinlichkeit einer erneuten Auswahl durchgeführt. Ebenso werden alle modifizierten Gewichte vergessen, wenn ein neuer Wert empfangen wird, wodurch die Gewichtungen dann auf anfängliche Werte zurückgesetzt werden. Die Idee dahinter ist, dass ein Knoten bei Erhalt eines neuen Werts diesen wieder allen anderen Knoten mitteilen soll. Dabei erfolgt eine erneute Berücksichtigung der initialen Prioritäten, was dazu führt, dass die anfängliche Bevorzugung von Knoten erneut greift.

Die Algorithmen werden durch eindeutige Namen definiert, über die eine Initialisierung erfolgen kann. Der Simulation Operator erhält diesen Namen aus der Definition der Simulationsressource. Er initialisiert die Pods und kann so über die Umgebungsvariablen auch den anzuwendenden Algorithmus festlegen. Der Simulation Runner hingegen benötigt den Namen nur zum Speichern der Ergebnisse, da er wie alle Parameter zur späteren Auswertung persistiert werden muss. Für die Steuerung des Gossipings ist der Algorith-

mus irrelevant. Alle Node Pods führen den Node-Service über das selbe Container-Image aus. Der Node Service erhält die Information über den anzuwendenden Algorithmus vom Operator. Er implementiert die verschiedenen Algorithmen und führt diese während der Simulation aus. Ähnlich wie bei der Grapherzeugung kann auch hier das Strategy-Pattern eingesetzt werden. Dadurch kann zur Laufzeit der Algorithmus zur Selektion des Gossip-Partners anhand der Strategie eingesetzt werden.

In Abbildung 5.8 wird der Aufbau der Node-Service-Applikation dargestellt. Der *GossipServicer* stellt eine vom Protocol Buffer Compiler *protoc* generierte gRPC-Schnittstelle bereit, welche direkt aus der proto-Datei erzeugt wird. Entsprechend dem oben genannten Ablauf werden folgende Methoden benötigt:

- **CurrentValue** - Diese Methode dient der Abfrage des aktuellen Werts. Sie wird zur initialen Abfrage der Werte und nach jeder Gossip-Runde eingesetzt.
- **Gossip** - Diese Methode wird zum Starten einer Gossip-Kommunikation genutzt. Dabei wählt der Knoten einen Nachbarn aus und tauscht sich mit diesem über TCP aus.
- **StopApplication** - Mit dieser Methode kann der Node-Service beendet werden.

Der *GossipService* implementiert das GossipServicer-Interface und definiert die konkrete Funktionalität der gRPC-Methoden. Hier wird auch das erste Konzept der Algorithmen dargestellt. Der GossipService hat einen aktiven Algorithmus, der für die Simulation eingesetzt wird. Die Algorithmus-Klasse stellt die Basisfunktionalität bereit, wodurch die Auswahl eines Knotens aus einer Liste von Nachbarn erfolgen kann. Über die Algorithmus-Klasse kann auch direkt das Baseline-Verfahren abgebildet werden, wobei eine zufällige Auswahl durchgeführt wird. Die WeightedFactor- und Community-Probabilities-Algorithmen erweitern den Standardalgorithmus und überschreiben hierbei dessen Funktionalität. Sie können ebenfalls über Vererbung mit den Memory- und ComplexMemory-Erweiterungen kombiniert werden. Dieses Konzept macht eine flexible Erweiterung der Logik möglich. Wenn während späteren Phasen der Arbeit Erkenntnisse gewonnen werden, kann eine Anpassung der Algorithmen erfolgen. Zudem können durch den modularen Aufbau beliebig neue Verfahren entwickelt und hinzugefügt werden.

Für die Konzeption ist auch die Entscheidung relevant, wie die Erkennung von Communities in den Simulationsprozess integriert wird. Hier waren im Wesentlichen zwei verschiedene Abläufe möglich. Die erste Option ist die Erkennung von Communities durch den Simulation Runner abzubilden, da er die Simulationssteuerung übernimmt. Der Nachteil dieses Vorgehens ist jedoch, dass mehr Kommunikationsaufwand benötigt werden würde, da die gewonnenen Informationen auf den Node Pods benötigt werden. Der Simulation Runner selbst müsste das Wissen über die Community-Zugehörigkeiten also entsprechend aufbereiten und an die Node Pods verteilen. Die zweite Option ist, die Erkennung durch den Simulation Operator selbst durchzuführen, wodurch diese zusätzliche Kommunikation nicht erfolgen muss. Ebenfalls hat der Operator über die Graphressource zu Beginn alle notwendigen Informationen ohnehin vorliegen und kann diese den Node Pods über Umgebungsvariablen bereitstellen. Ein Nachteil besteht darin, dass der Operator zusätzliche Komplexität erhält. Die Entscheidung fiel unter Abwägung der Vor- und Nachteile auf die zweite Option. Insbesondere trug zur Entscheidung die Tatsache bei, dass so die Komplexität des Simulation Runners beschränkt werden kann. Seine Aufgabe ist die Ablaufsteuerung, also das die Verwaltung der Runden sowie die Ermittlung und Speicherung der Ergebnisse. Der Operator hingegen ist für den Auf- und Abbau der Netzwerktopolgie zuständig. Semantisch passt die Erkennung der Communities hierzu ebenfalls besser, da diese für die Initialisierung der Netzwerkknoten benötigt wird.

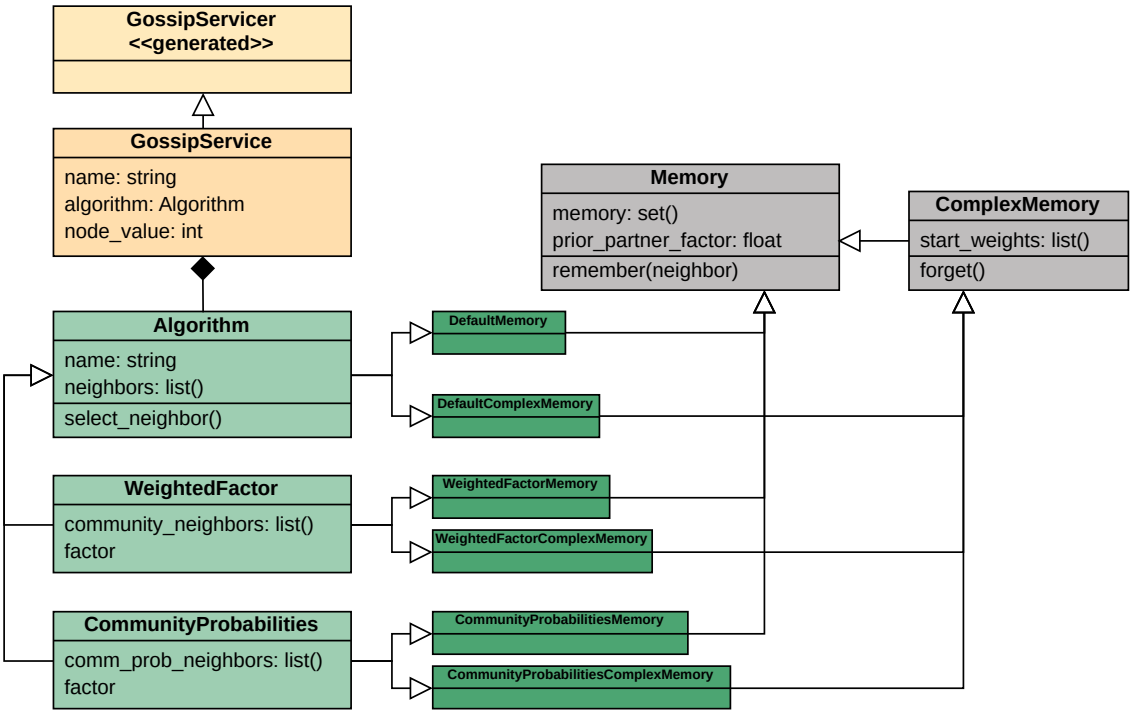


Abbildung 5.8: Klassendiagramm Algorithmen

Zur Erkennung der Community-Strukturen soll die *Methode nach Louvain* eingesetzt werden. Die Louvain-Methode ist eine der beliebtesten Methoden zur Erkennung von Community-Strukturen. Als Gründe für ihren Einsatz sind unter anderem die hohe Skalierbarkeit und schnelle Geschwindigkeit zu nennen. Sehr große Graphen können effizient durch die gierige Optimierungsstrategie ausgewertet werden. Zudem optimiert die Louvain-Methode die Modularitätsmetrik, welche die Qualität von Community--Strukturen angibt [14]. Im Rahmen dieser Arbeit sind Graphen mit unterschiedlichen Ausprägungen dieser Strukturen zu untersuchen. Weitere Vorteile ist die gute Parametrisierbarkeit sowie die weite Verfügbarkeit der Implementierungen. Die Louvain-Methode wird vom Operator verwendet, um die Community-Strukturen global zu bestimmen. Anschließend wird das Wissen so aufbereitet, dass jeder Knoten nur über lokale Informationen verfügt. Dadurch kann die Simulation so erfolgen, wie es bei einer praktischen Anwendung der Fall wäre. Bei einer dezentralen Bestimmung der Community-Strukturen wie zum Beispiel durch die Synchronisationsprotokolle verfügt jeder Knoten nur über eine Sicht auf seine direkte Nachbarschaft. Dementsprechend erhält auch jeder Knoten nur die Informationen über die Community-Zugehörigkeiten seiner Nachbarn.

5.2.5 Forschungsfragen und Testreihendefinition

Zuerst wurden einige Forschungsfragen aufgestellt, welche mit den Testreihen beantwortet werden sollen. Anschließend wurde ein grobes Konzept für die ersten Testreihen entworfen. Es folgt nun zuerst die Auflistung der Forschungsfragen:

- Kann die Leistung von Gossip-Algorithmen durch Berücksichtigung der Netztopologie und insbesondere der Community-Strukturen verbessert werden?
- Wie wirken sich die verschiedenen Gossip-Algorithmen auf die Konvergenzzeit aus?

- Welche Faktoren beeinflussen die Effektivität der jeweiligen Algorithmen?
- Wie wirken sich speicherbasierte Erweiterungen auf die Effizienz und das Konvergenzverhalten aus?
- Welche Grapheigenschaften haben den meisten Einfluss auf die Konvergenzzeit?
- Gibt es alternative gegebenenfalls bessere Metriken zur Gewichtung als Wissen über die Community-Strukturen?
- Welche Anwendungsszenarien ergeben sich für die Gossip-Algorithmen?
- Kann mit anderen Konfigurationen eine Beschleunigung einer lokalen Konvergenz erzielt werden? In welchen Szenarien kann dies sinnvoll sein?
- Welche Schlussfolgerungen lassen sich für unstrukturierte Netzwerke ziehen? Inwiefern können die Erkenntnisse auf andere Netze übertragen werden?

Auf Grundlage der Forschungsfragen wurden dann folgende Testreihendefinitionen aufgestellt:

- **Einstiegssimulationsreihe zum Vergleich Baseline mit WeightedFactor (naiver Ansatz) auf Graphen mit verschiedenen Modularitäten:** Es sind Graphen mit Modularitäten zwischen 0 und 1 zu erzeugen. Für verschiedene Modularitätsniveaus sollen dann Simulationen auf den jeweiligen Graphen durchgeführt werden. Dabei soll nachgewiesen werden, dass hochmodulare Graphen das größte Optimierungspotential haben.
- **Einstiegssimulationsreihe zum Vergleich Baseline mit WeightedFactor (naiver Ansatz) auf Popularitätsgraphen:** Es sind Graphen mit verschiedenem Verhältnis zwischen Inter- und Intra-Community-Kanten zu generieren. Anschließend sollen Simulationen für die verschiedenen Verhältnisse ausgeführt werden. Dabei ist zu zeigen, dass Graphen mit einem höheren Anteil an Inter-Community-Kanten ein höheres Verbesserungspotential haben.
- **Simulationen zur Optimierung der gewichteten Algorithmen:** Anknüpfend an die vorangehenden Simulationen sollen die besten Graphtypen ausgewählt werden. Für die anschließenden Simulationsreihen ist sich auf diese zu beschränken. Im Folgenden wird dann eine tiefergehende Auswertung sowie Weiterentwicklung der Algorithmen durchgeführt. Hierbei sollen zum Beispiel verschiedene Faktoren des gewichteten Algorithmus verglichen werden. Ebenfalls soll eine Evaluation des dynamischen CommunityProbabilities-Algorithmus erfolgen. Im nächsten Schritt sind dann auch die Memory-Erweiterungen auszuwerten.
- **Simulationen zu fortgeschrittenen Algorithmen:** Auf den Erkenntnissen kann optional eine Neuentwicklung eines oder mehrerer optimierter Algorithmen erfolgen. Dies erfordert dann gegebenenfalls weitere Simulationsreihen.
- **Simulationen auf Partitionen des Gnutella-P2P-Netzwerks:** Final sollen die Ergebnisse am Gnutella-P2P-Netzwerk erprobt werden. Dieses zeigt durch die Zufälligkeit zwar keine stark ausgeprägten Modularitätsmerkmale ist aber als reales Netzwerk relevant für die Auswertung.

Diese erste grobe Definition der Simulationsreihen wurde iterativ weiterentwickelt. Die finalen Definitionen sind im Evaluationskapitel 7.2 zu finden, wo der detaillierte Aufbau der Reihen erläutert wird. Hier werden die Netzwerkgraphen und deren Eigenschaften beschrieben sowie die verwendeten Algorithmen und ihre Parameter genannt.

5.3 Zusammenfassung des Konzepts

Das Konzept stellt die Architektur einer Simulationsumgebung für Gossiping-Verfahren vor. Als Anwendungsszenario wird das Feststellen des netzwerkweiten Minimums verwendet. Dabei wird rundenbasiertes paarweises Gossiping durchgeführt. Verschiedene Netzwerke sollen über eine dynamische Erzeugungsroutine generiert und als Graphen dargestellt werden. Die Erzeugung der Netzwerkstruktur übernimmt anschließend der Simulation Operator. Dieser folgt dem gleichnamigen Pattern und orchestriert die Graphen und Simulationen. Beim Simulationsaufbau erzeugt er für jeden Knoten einen Pod, welcher eine Container-Anwendung beinhaltet, den Node-Service. Zwischen den Pods erfolgt der Austausch über TCP, um eine realistische Netzwerkkommunikation zu simulieren. Außerdem erstellt er den Simulation Runner, welcher die Simulationssteuerung übernimmt. Er initialisiert die verschiedenen Runden, überwacht die Konvergenz und speichert die Ergebnisse. Zur Steuerung der Knoten kommuniziert er mit diesen über gRPC.

Mit der Simulationsumgebung sollen dann verschiedene gewichtete Algorithmen untersucht werden. Hier soll zur Gewichtung vor allem Wissen über Community-Strukturen im Netzwerk berücksichtigt werden. Zur Ermittlung dieser Strukturen ist dabei die Louvain-Methode zu verwenden. Die Evaluation soll mit naiven Algorithmen beginnen und dann zunehmend komplexere Verfahren untersuchen. Dabei sollen die anfänglichen Testreihen die Netzwerke mit dem höchsten Optimierungspotential identifizieren. Somit ermöglicht das Konzept die Untersuchung der Leistung von verschiedenen Gossip-Algorithmen auf unterschiedlichen Netzwerken.

6 Implementierung

Das Implementierungskapitel beschäftigt sich mit der praktischen Umsetzung des Konzepts. Hierbei wird die Entwicklung der verschiedenen Komponenten der Simulationsumgebung beschrieben. Zunächst erfolgt die Erläuterung der Implementierung der Prototypen, welche dem Nachweis der Machbarkeit dient. Die Prototypen legten den Grundstein für die nachfolgenden Entwicklungsphasen und wurden fortlaufend weiterenwickelt, bis sämtliche Anforderungen umgesetzt waren. Anschließend wird der Aufbau des Kubernetes-Clusters für die Simulationsumgebung dargestellt. Hierbei wird auf die einzelnen Schritte zur Installation und Konfiguration eingegangen. Außerdem werden erste Testsimulationen erläutert, welche mit den Prototypen durchgeführt wurden. Diese konnten zur Identifikation von Limitierungen und Verbesserungspotentialen verwendet werden. Im Anschluss wird erklärt, wie die Weiterentwicklung der einzelnen Anwendungen abgelaufen ist. Hier werden dann die finalen Applikationen beschrieben. Dazu zählt das CLI zur Erzeugung von synthetischen Netzwerken, welches die Erstellung verschiedener Graphen ermöglicht, die für den Aufbau von Simulationen genutzt werden können. Des Weiteren wird der Kubernetes Operator erläutert. Dieser realisiert die Automatisierung und Verwaltung von Simulationen. Außerdem erfolgt die Vorstellung der verschiedenen Containerapplikationen. Während der Node-Service einen einzelnen Knoten im Netzwerk darstellt, koordiniert der Simulation Runner die Durchführung und Überwachung der Simulation. Schließlich werden noch die Routinen zur Evaluation der Simulationsergebnisse erklärt.

6.1 Prototypische Entwicklung

Der erste Implementierungsschritt war eine prototypische Entwicklung der Simulationsumgebung. Das Ziel bestand darin, die Grundfunktionalität zu erreichen und so die Durchführbarkeit zu prüfen. Es wurden Prototypen für den Simulation-Operator und die Container-Applikationen erstellt. Für den Node-Service wurden Klassen implementiert, mit denen verschiedene Algorithmen abgebildet werden können. Dabei werden die Basisfunktionen des Gossipings unterstützt, wie die Initialisierung, die Auswahl eines Nachbarn und der Informationsaustausch. Für die Nachbarauswahl wurde erstmals nur das Baseline-Verfahren verwendet. Gleichermaßen minimalsitisch erfolgte die Umsetzung der Simulation Runner Applikation. Der Ursprung des CLI-Tools zur Grapherzeugung war ein einfaches Skript. Es diente der Erstellung von Adjazenzlisten basierend auf den verschiedenen Algorithmen. Diese wurden dann manuell in erste simple YAML-Dateien für Graphressourcen eingetragen.

Die Anwendungen wurden dann in einer kontrollierten Umgebung getestet. Dazu wurde ein Testsystem mit minikube als Zielplattform gewählt. Minikube ist eine leichtgewichtige Kubernetes-Lösung, welche die Erstellung und Verwaltung eines lokalen Clusters ermöglicht. Diese Umgebung läuft auf einem Knoten, dem Entwicklerrechner. Sie wurde für den Test und die spätere Weiterentwicklung der Prototypen verwendet.

Die Tests wiesen auf einige Einschränkungen im Ablauf der Simulation hin und verdeutlichten die erforderlichen Änderungen am Konzept. Iterativ wurden Anpassungen und Erweiterungen vorgenommen. Die nachfolgenden Testbereitstellungen dienten der Be-

wertung und konnten als Ausgangspunkt für den nächsten Entwicklungszyklus genutzt werden. Dabei wurden die Prototypen kontinuierlich verbessert und um neue Features erweitert. So wurde beispielsweise das anfängliche Skript zur Grapherzeugung zu einem umfassenden CLI-Tool. Ebenso kamen neue Grapherzeugungsroutinen hinzu und bestehende fielen weg. Auch die Container-Anwendungen wurden kontinuierlich verbessert. Hier wurde ein besonderer Aufwand in die Integration der gewichteten Algorithmen gesteckt, da für diese zahlreiche Parameter hinzugefügt werden mussten. Ebenso wurden viele Anpassungen an der bestehenden Logik durchgeführt, so wurde zum Beispiel Logik zur Erkennung von Community-Strukturen hinzugefügt. Darüber hinaus wurde auch Auswertungslogik (Metriken und Visualisierung) ausgearbeitet und Änderungen an den Simulationsserien (mehrere Graphen und verschiedene Parameterkonfigurationen) vorgenommen. Ebenfalls fanden Erweiterungen an beiden Container-Applikationen statt, um den fehlerfreien Ablauf des Gossipings sicherzustellen. Hier wurden die Synchronisierungsvorgänge überarbeitet, um eine Parallelisierung von Simulationen zu ermöglichen. Die Prototypen durchliefen viele Änderungszyklen und wurden so inkrementell bis zur finalen Version weiterentwickelt. Im Folgenden werden die einzelnen Anwendungen beschrieben, wobei insbesondere auf Abweichungen vom Konzept eingegangen wird.

6.2 Kubernetes-Cluster Deployment

Zu Beginn war geplant, das Kubernetes-Cluster des Distributed Systems Lab zum Ausführen der Simulationsreihen zu nutzen. Aufgrund unvorhergesehener Probleme mit dem Cluster musste jedoch im Projektverlauf eine alternative Umgebung gefunden werden. Der erste Gedanke fiel dabei auf *Managed-Kubernetes-Lösungen*. Hierbei handelt es sich um Dienste, die Kubernetes-Containerorchestrierung in einer Cloud-Umgebung bereitstellen. Bei den Managed-Lösungen werden die Cluster vorkonfiguriert und von den Dienstleistern betreut. Entwickler haben somit keinen Aufwand und können direkt auf den fertigen Cluster zugreifen. Eine Alternative dazu ist das Anmieten von Cloud-Infrastruktur und das Aufsetzen eines eigenen Clusters. Schließlich fiel die Auswahl unter Abwägung der Vor- und Nachteile auf ein eigenes Cluster-Deployment. Die Gründe für diese Entscheidung umfassen insbesondere folgende Aspekte:

- *Kosteneffizienz*: Die Managed-Lösungen sind wesentlich kostenintensiver, wodurch weniger leistungsstarke Hardware erworben werden kann. Abstriche bei der Leistung der Server beeinträchtigen die Simulationsgröße. Ziel soll es sein, mindestens tausend Knoten simulieren zu können. Da Gossip-Algorithmen auf kleineren Netzwerken in wenigen Runden konvergieren, können Verbesserungen bei kleineren Netzwerkgrößen nicht mehr genau festgestellt werden. Zur Simulation von Netzwerken mit tausend Knoten wird bei Managed-Lösungen ein Cluster-Setup mit mehreren Servern benötigt. Die Kosten sind in diesem Fall vergleichbar hoch, wobei für denselben Preis eine erheblich leistungsstärkere Cloud-Infrastruktur gemietet werden kann.

- *Kontrolle und Anpassbarkeit*: Ein eigener Cluster kann beliebig konfiguriert und angepasst werden. Dies ist insbesondere wichtig, da für die Anwendung eine Erhöhung der sinnvoll ist. Die Services der einzelnen Node Pods sind sehr leichtgewichtig. Kubernetes hat eine standardmäßige Begrenzung von 110 Pods pro Worker Node, welche jedoch für diesen Anwendungsfall weit überschritten werden kann. Dabei wird ein Overhead für den Konfigurationsaufwand in Kauf genommen. Das heißt das Control Plane benötigt gegebenenfalls mehr Zeit zum Erstellen, Verteilen und Initialisieren der Container. Die Maximalanzahl an Pods pro Knoten kann nur beim

Cluster Setup erhöht werden. Managed-Lösungen werden vorkonfiguriert bereitgestellt, wodurch nicht durchführbar ist. Infolgedessen würde viel Hardware-Leistung bei der Simulationsausführung ungenutzt bleiben.

Als Infrastrukturanbieter wurde *Hetzner* ausgewählt [53]. Hetzner zeichnet sich durch seine wettbewerbsfähigen Preise für Cloud-Hosting-Dienste aus. Außerdem ist die angebotene Infrastruktur sehr leistungsstark und modern. Damit stellt der Anbieter besonders für kleinere Projekte eine optimale Wahl dar. Hetzner hat viele Rechenzentren in Deutschland, wodurch die Verfügbarkeit sehr hoch ist und Latenzzeiten gering bleiben. Darüber hinaus erfolgt beim Anmieten eine sofortige Bereitstellung und zusätzlich kann jederzeit eine flexible Skalierung durchgeführt werden. Auch die Verwaltung von Servern kann über angebotene Schnittstellen intuitiv erfolgen. Des Weiteren wird das Erstellen von Backups und Snapshots angeboten, wodurch der Zustand des Clusters speicherbar wird. Dadurch kann im Falle eines schwerwiegenden Fehlers mit minimalem Zeit- und Arbeitsaufwand ein funktionaler Zustand wiederhergestellt werden. Konkret wurden fünf *ARMx64 CAX31-Server* angemietet, wobei jeder Server mit 8 vCPUs, 16 GB RAM und 160 GB Festplattenspeicher ausgestattet ist. Als Betriebssystem ist *Ubuntu 22.04* installiert. Alle Server sind so konfiguriert, dass sie im privaten Hetzner-Netzwerk laufen. Zusätzlich wird eine öffentliche IP-Adresse für den Control Plane Node gemietet. Dies ist notwendig, um flexibel von Remote auf den Server zugreifen zu können.

Im ersten Schritt mussten die Server konfiguriert werden. Dabei war es notwendig mehrere Konfigurationen vorzunehmen, um das Netzwerk einzurichten. Zuerst wurden das Routing und die Namensauflösung eingerichtet, damit erstmals eine Kommunikation zwischen den Knoten stattfinden kann. Das Routing ermöglicht den Worker Nodes Zugang zum Internet über die Steuerungsebene. Dadurch haben allen Knoten Internetzugang, was den den Installationsprozess der Cluster-Anwendungen erheblich vereinfacht. Ebenfalls musste die Konfiguration der IP-Tabellen, Standard-Gateways und Routen sowie der Namensauflösung erfolgen. Sobald ein ordnungsgemäßer Netzwerkbetrieb erreicht wurde, konnte mit der Installation der Anwendungen fortgefahren werden.

Bei der Anwendungsinstallation wurde zuerst der Control Plane Node aufgesetzt. Dabei wurde mit der Installation der Container-Engine begonnen. Als Engine wurde der *runc-containerd-Stack* verwendet, da hier die Anbindung an das CRI sehr simpel ist. Außerdem handelt es sich hierbei um eine weitverbreitete Wahl, wodurch viel Dokumentation verfügbar ist. Für das korrete Setup musste zuerst *runc* installiert werden. Es ist ein CLI-Tool, das mit dem Linux-Kernel direkt interagiert, um Container zu erzeugen, auszuführen und zu verwalten. Anschließend wurde die Container-Laufzeitumgebung *containerd* aufgesetzt, welche als Brücke zwischen runc und der Kubernetes fungiert. Während runc die konkreten Low-Level-Container-Operationen realisiert, kapselt containerd diese und stellt einen erweiterten Funktionsumfang bereit. Dazu zählt beispielsweise Imageverwaltung, Checkpointing, Isolierung und Sicherheit. Im Anschluss wurden die *CNI-Plugins* installiert. Sie führen IP-Zuweisungen durch, richten die Netzwerkschnittstellen ein und konfigurieren Routing sowie Netzwerkrichtlinien. Des Weiteren mussten die Komponenten zur Verwaltung des Kubernetes-Clusters installiert werden. Diese beinhalten *kubeadm, kubectl* und *kubelet* und erleichtern die Bereitstellung, Verwaltung und Interaktion mit dem Cluster. Mit dem Initialisierungsbefehl (*kubeadm init*) wurde der Control Plane Node eingerichtet. In diesem Schritt konnte durch Modifikation der Konfigurationsdateien dann das Limit an Pods pro Worker Node erhöht werden. Dabei mussten die Subnetzmasken so geändert werden, dass die Pod-Netzwerke eine ausreichende Größe unterstützen. Zuletzt wurde die Installation von *flannel* durchgeführt, wobei es sich um eine beliebte Netzwerklösung für Kubernetes handelt. Flannel stellt ein ordnungsgemäßes Cluster-Networking sicher, indem es die Kommunikation der Container und Pods innerhalb des Clusters, also

zwischen den physischen Knoten, realisiert.

Im nächsten Schritt wurden ähnliche Konfigurationsschritte auf den Worker Nodes durchgeführt. Ein entsprechendes Installationsskript wurde erstellt und ausgeführt. Nach erfolgreicher Einrichtung wurde die korrekte Funktionalität der Control Plane und Worker Nodes geprüft. Zuletzt wurden die Knoten mit dem Control Plane über das Join-Kommando (*kubeadm join*) verbunden. Dieser Schritt finalisiert den Aufbau der Cluster-Infrastruktur.

Anschließend wurde der Remote-Zugriff auf den Cluster über kubectl eingerichtet. Hierbei muss die Cluster-Konfiguration exportiert und dann als Kontext in kubectl hinzugefügt werden. Das Tool ermöglicht über Kontextwechsel die parallele Nutzung verschiedener Cluster. Dadurch wird ein flexibler Wechsel zwischen der Testumgebung mit minikube und der Simulationsumgebung möglich. Zur Verbindung mit dem Cluster werden Zertifikate verwendet, welche lokal zu installieren sind. Die Verwendung von Zertifikaten sichert die Kommunikation mit dem Cluster ab. Zusätzlich musste die lokale Hosts-Datei um die öffentliche IP-Adresse erweitert werden, sodass eine erfolgreiche Namensauflösung erfolgen kann.

Als Nächstes wurden alle Anwendungen bereitgestellt, um die Prototypen der Simulationsumgebung zu evaluieren. Da auf dem Cluster kein dynamischer StorageProvider verfügbar ist, musste MinIO mit manuellen PV und PVC bereitgestellt werden. Der Mehraufwand dieser Konfigurationsänderung war jedoch im Vergleich zum zusätzlichen Aufsetzen eines StorageProviders minimal. Während des Deployments wurde dann festgestellt, dass die zuvor erstellten Container-Images auf dem ARM64-Befehlssatz nicht ohne Weiteres verwendet werden können. Die Container-Images müssen entsprechend der Architektur der Zielplattform erstellt werden. Daher ist es erforderlich, Wheels für die verwendeten Bibliotheken zu nutzen, die mit der Architektur kompatibel sind. Die Images mussten deshalb neu erstellt werden, wobei *docker buildx* verwendet wurde. Dabei handelt es sich um ein Drop-In-Ersatz für den Standard-Build-Client, der das Buildkit erweitert. Ein Builder-Container wird dann verwendet, um Multi-Architektur-Images zu erstellen. Das hießt, es können Images erstellt werden, die sowohl auf der Architektur des Host-Systems als auch auf ARM64-Hosts laufen. Diese können dann sowohl in der Test- als auch in der Simulationsumgebung eingesetzt werden.

Nach erfolgtem Deployment aller prototypischen Anwendungen wurde der erste Testlauf durchgeführt. Hier fiel auf, dass der Cluster bei der Durchführung von Simulationen mit einer großen Anzahl von Pods nicht alle Container-Images laden kann. Grund dafür ist, dass das Pull Rate Limit von Docker erreicht wird. Dadurch können ab diesem Zeitpunkt keine weiteren Images mehr von dem jeweiligen Host beziehungsweise Profil heruntergeladen werden. Als Lösung wurde eine private Docker-Registry im Cluster eingerichtet. Containerd musste so umkonfiguriert werden, dass der Abruf der Images von dieser Registry erfolgt. Dazu muss jedoch eine Authentifizierung mit dem Registry-Zertifikat erfolgen. Um dies zu realisieren wurde das Zertifikat im Builder-Container von Docker installiert. Ebenfalls musste eine Installation auf den Worker Nodes erfolgen, damit Pod-Images von der privaten Docker-Registry abgerufen werden können. Die Verbindungsinformationen zur Kommunikation mit der Registry wurden dann zusätzlich als Secret im Cluster hinterlegt. Über den Eintrag *imagePullSecrets* im Deployment des Simulation Operators können die Pods sich entsprechend authentifizieren. Gleichermaßen wurde die Pod-Spezifikation der Node-Services und des Simulation Runners um diesen Eintrag erweitert. Um den Push von Container-Images in die Cluster-Registry möglichst einfach zu gestalten, wurde der Service-Type des Registry Services auf NodePort geändert. Infolgedessen wird der Service auf dem statisch definierten Port verfügbar und kann über diesen angesprochen werden. Dazu musste das Zertifikat auch Remote installiert

werden. Analog wurde die MinIO Konsole (Object Browser GUI) konfiguriert, um auch hier einen externen Zugriff zu ermöglichen. Infolgedessen können die Ergebnisse mit möglichst geringem Aufwand geprüft und weiterverarbeitet werden.

Bei der ersten Simulation wurde festgestellt, dass der Kubernetes-Scheduler dazu neigt, Pods demselben Worker Node zuzuweisen. Dies erfolgt aufgrund der gleichen Pod-Konfigurationen. Der Scheduler versucht, gleiche beziehungsweise ähnliche Anwendungen logisch zusammenzubringen. Er führt dabei die Anwendungen auf einem physischen System aus, wodurch diese auf den gleichen Speicher zugreifen können. Dadurch kann das Cluster den Speicherverbrauch durch Vermeidung von redundanter Persistenz von Variablen und Instruktionen reduzieren. Infolgedessen gelingt es ihm, Gesamtressourcen zu sparen im Vergleich zu einer verteilten Ausführung, wo die Speicherinhalte repliziert werden müssten. Hierbei entsteht jedoch in diesem Anwendungsfall eine unausgewogene Verteilung der Arbeitslast. Um dies zu beheben, werden Topologieverteilungsbeschränkungen (*topology spread constraints*) implementiert. Jedem Worker Node wird ein Label (*simulation-node*) mit einem eindeutigen Werten zugewiesen. Der Operator wird dann so modifiziert, dass er Pods mit der Topologieverteilungsbeschränkung für diesen Schlüssel bereitstellt. Er verfolgt die Verteilung der Anwendungen auf den Nodes und zählt mit. Dabei stellt er sicher, dass eine Differenz von eins nicht überschritten wird. Das heißt, dass diese Beschränkung gewährleistet, dass der Scheduler eine gleichmäßige Bereitstellung durchführt. Es wird eine ausgewogene Verteilung der Arbeitslast über alle Worker Nodes aufrechterhalten.

6.3 Command Line Interface zur Grapherzeugung

Die Entwicklung des Tools zur Generierung von Graphen erfolgte entsprechend dem konzipierten Ablauf (siehe Kapitel 5.2.1). Es wurde großer Wert auf Änderbarkeit und Erweiterbarkeit gelegt, wodurch das Hinzukommen beziehungsweise der Austausch von Algorithmen zur Grapherzeugung flexibel möglich sein sollte. Hierzu wird das Strategie-Pattern verwendet, welches eine Entkopplung von der Auswahl der Erzeugungsroutine und der Grapherzeugung realisiert. Alle Strategien zur Generierung von Graphen haben gemeinsame Schnittstellen. Dabei werden gleiche Funktionssignaturen genutzt, um die verschiedenen Schritte abzubilden. Jede Strategie hat spezifische Parameter, welche vei der Erzeugung des jeweiligen Graphtyps entgegengenommen werden. Die Funktionen geben dann *NetworkX-Graphobjekte* zurück, welche die erzeugte Netzwerktopolgie darstellen. Gleichermaßen wird weitere Funktionalität über das Pattern abgebildet. Beispiele beinhalten die Festlegung des Graphnamens, die Eingabe der Parameter und die Ausgabe der Grapheigenschaften. Die Generierungsstrategien sind von diesen Abläufen der Grapherzeugung logisch entkoppelt. Abstrakte Funktionen werden eingesetzt, die erst nach Auswahl des Graphtyps spezifiziert werden. Eine Erweiterung, um neue Algorithmen erfordert dann ledsiglich die Implementierung weiterer Strategien. Diese werden den Schnittstellen in einem Dictionary-Eintrag hinzugefügt. Dadurch wird ebenso das Risiko minimiert, dass bei neuen Implementierungen Fehler eingefügt werden. In Listing 6.1 ist dargestellt, wie die Schnittstelle für die Funktionen der Grapherzeugung aussieht.

Der Ablauf der Grapherzeugung wurde während der Implementierung erweitert und angepasst. Die in der Konzeption beschriebene Generierung von *N* oder *MxN* Graphen wurde analog umgesetzt (siehe Kapitel 5.2.1.2). Es wurde aber eine weitere Erzeugungsroutine hinzugefügt. Hier muss der Nutzer *C/N* oder *C/MxN* als Eingabe wählen. Im späteren Verlauf wird dann noch eine der berechneten Metriken abgefragt. Von der gewählten Metrik werden die Wertebereiche der *N* beziehungsweise *MxN* Graphen angezeigt. Dann muss ein Wert zur Approxmimation ausgewählt werden. Anschließend wird

```
def get_creation_func(graph_type):
  create_graph_funcs = {
    GRAPH_TYPE_FIXED_POPULARITY_SHORT: generate_fixed_popularity_graph,
    GRAPH_TYPE_SCALE_FREE_SHORT: generate_scale_free_graph,
    GRAPH_TYPE_BARABASI_ALBERT_SHORT: generate_barabasi_albert_graph,
    GRAPH_TYPE_HOLME_KIM_SHORT: generate_holme_kim_graph,
  }
  func = create_graph_funcs.get(graph_type)
  return func

# set the functions for the chosen graph type
params_func = get_params_func(graph_type)
graph_properties_func = get_graph_properties_func(graph_type)
end_params_func = get_end_params_func(graph_type)
create_graph_func = get_creation_func(graph_type)
graph_properties = {}
graphs = {}
name_counts = {}
# create N graphs for the given params
# add them and their properties to the arrays
def create_for_N(params):
    for _ in range(0, N):
        name = get_graph_name(graph_type, params)
        print(f'Creating graph {name}...')
        graph = create_graph_func(*params)
        if name not in name_counts:
            # First occurrence of the name
            name_counts[name] = 1
        else:
            # Duplicate name found
            count = name_counts[name]
            name_counts[name] += 1
            name = f'{name}_{count}'
        graph_properties[name] = graph_properties_func(*params)
        graphs[name] = graph

# get the graph parameters from user input
graph_params = params_func()

# in case of "N" graph generation
if sameParams is True:
    # create all graphs with the same params
    create_for_N(graph_params)
```

Listing 6.1: Strategie-Design-Pattern zur Grapherzeugung

mit den C Graphen fortgefahren, deren Werte am nächsten an dem ausgewählten Wert liegen, während die anderen verworfen werden. Diese Variante ermöglicht Graphen zu generieren, bei denen eine Metrik einen bestimmten Wert approximiert. Dadurch wird beispielsweise die Aufgabe erleichtert, Netzwerke mit einer Zielmodularität zu erstellen.

Die verwendeten Erzeugungsalgorithmen wurden mehrfach anhand von Prototypen evaluiert und anschließend angepasst. Mit nur geringfügigen Änderungen wurden die folgenden Algorithmen aus dem Konzept (siehe Kapitel 5.2.1.1) übernommen:

- *Duales Modell nach Barabási-Albert zur Erzeugung skalenfreier Graphen:* Graphen werden auf Grundlage der Knotenanzahl *node_count*, sowie m_1, m_2 und p erzeugt. Der Nutzer gibt nur einen Float-Wert m für die Anzahl an hinzugefügten Kanten ein. Aus m wird m_1, m_2 und p berechnet, sodass $m = m_1 \cdot p + m_2 \cdot (1 - p)$.

- *Modifizierte Version einer skalenfreien Grapherzeugung:* Die Eingaben α, β, γ werden verwendet, wobei $\alpha + \beta + \gamma = 1$ sein muss. Je höher der Wert von α, desto stärker ist das Clustering des erzeugten Graphen. Der Wert von β hingegen bestimmt die Zufälligkeit bei dem Setzen von Kanten, während γ die Graphverbundenheit beeinflusst. Ein Parameter *edge_multiplier* wurde ergänzt, welcher die Anzahl an Kanten im Endgraphen beeinflusst. Dazu werden, falls *edge_multiplier* größer als eins ist, ähnliche Subgraphen erzeugt und deren Kanten hinzugefügt. Infolgedessen steigt Gesamtkantenzahl des erzeugten Graphen multiplikativ mit dem Wert des Parameters. Das Verfahren ermöglicht die Generierung stark unmodularer skalenfreier Graphen bei hohen Werten von β und geringen Werten von α.

- *Das simple Verfahren zur Erzeugung von Popularitätsgraphen:* Hierbei können die Parameter *p_inter* und *p_intra* angegeben werden. Diese legen die Wahrscheinlichkeiten für Inter- und Intra-Community-Kanten fest. Ebenfalls lässt sich die Anzahl an Knoten *node_count* sowie Communities *comm_count* parametrisieren. Auch kann mit *equal_sized* angegeben werden, ob die Communities gleicher Größe sein sollen.

Die Verwendung des komplexen Grapherzeugungsalgorithmus mit fixer Modularität (siehe Kapitel 5.2.1.1) wurde früh verworfen. Aufgrund möglicher Kompatibilitätsprobleme wegen alter Python-Versionen führten viele Parametereingaben zu Skriptabstürzen. Zudem wurden die Modularitäten meist nicht ausreichend gut approximiert. Die Differenz zwischen Zielmodularität und erreichter Modularität lagen oft bei über 25 Prozent. Außerdem nimmt die Generierung von Graphen sehr viel Zeit in Anspruch, während oft sogar keine Ausgabe zustande kommt. Grund dafür ist, dass die Grapherzeugung solange wiederholt wird bis ein Graph gefunden ist, für den die Modularität angemessen approximiert ist. Die Wiederholung erfolgt hierbei aber nur so oft bis eine definierte Obergrenze erreicht ist. Wenn diese Grenze überschritten ist, erfolgt die Ausgabe eines Fehlers, und das Programm wird abgebrochen.

Neu hinzugefügt wurde der *Holme-Kim-Algorithmus*, welcher eine Erweiterung des Barabási-Albert-Modells darstellt. Der Algorithmus nutzt Dreieckskanten, um höhere Modularitäten zu erzeugen. Dabei bleibt der Exponent der Powerlaw-Verteilung erhalten. Mit dem Barabási-Albert-Modell können sehr hohe Modularitätswerte (im Bereich 0.8 bis 1.0) schwer bis gar nicht erreicht werden, weshalb der Holme-Kim-Algorithmus verwendet wurde. Durch das Hinzufügen der Dreieckskanten eignet sichdas Verfahren besonders gut zur Erzeugung skalenfreier Graphen mit hoher Modularität.

Ebenfalls wurde auf der simplen Erzeugungsroutine (siehe Kapitel 5.2.1.1) aufbauend ein weiterer Graphgenerierungsalgorithmus erstellt. Dieser ist dem Algorithmus von Singh zu Popularitätsgraphen nachempfunden [105]. Er nimmt fixe Zahlen von Inter- und Intra-Community-Kanten als Parameter. Dabei stellt er sicher, dass die Communities untereinander verbunden sind. Der Algorithmus ist in Listing 6.2 dargestellt.

```
1 def generate_fixed_popularity_graph(node_count, comm_count, num_intra,
      num_inter):
2   # Create an empty graph
3   graph = nx.Graph()
4   # Generate a list of community sizes
5   sizes = [node_count // comm_count] * comm_count
6   sizes[0] += node_count % comm_count
7   factor = num_intra / node_count
8   num_intra_arr = [math.ceil(size * factor) for size in sizes]
```

```
  # Generate a random modular graph
  for i in range(comm_count):
    nodes = list(range(sum(sizes[:i]), sum(sizes[:i + 1])))
    size = sizes[i]
    subgraph = nx.Graph()
    subgraph.add_nodes_from(nodes)
    node_range = range(nodes[0], nodes[size - 1] + 1)
    dim = 2
    edges = combinations(node_range, dim)
    num_edges = 0
    for _, node_edges in groupby(edges, key=lambda x: x[0]):
      node_edges = list(node_edges)
      random_edge = random.choice(node_edges)
      subgraph.add_edge(*random_edge)
      num_edges += 1

    def random_combination(iterable, r):
      pool = tuple(iterable)
      n = len(pool)
      indices = sorted(random.sample(range(n), r))
      return tuple(pool[i] for i in indices)

    tries = 0
    while num_edges < num_intra_arr[i] and tries < 1000:
      random_edge = random_combination(node_range, dim)
      if random_edge not in subgraph.edges:
        subgraph.add_edge(*random_edge)
        num_edges += 1
        tries = 0
      else:
        tries += 1
    graph.add_nodes_from(subgraph.nodes())
    graph.add_edges_from(subgraph.edges())

  # Set the community dictionary
  community_dict = {}
  for i in range(comm_count):
    nodes = list(range(sum(sizes[:i]), sum(sizes[:i + 1])))
    for node in nodes:
      community_dict[node] = i
  # Set the community attribute for each node in the graph
  nx.set_node_attributes(graph, community_dict, 'desired_community')

  if comm_count > 1:
    # Generate random numbers for each pair of tuples
    pair_numbers = [random.randint(0, num_inter) for _ in range(
        comm_count * (comm_count - 1) // 2)]
    total_numbers = sum(pair_numbers)

    # Normalize the numbers to ensure the sum is equal to num_inter
    normalized_numbers = [int(num * num_inter / total_numbers) for num
        in pair_numbers]
    remaining = num_inter - sum(normalized_numbers)

    # Distribute the remaining difference to the first few pairs
    for i in range(remaining):
      normalized_numbers[i] += 1
  else:
    normalized_numbers = [num_inter]
```

```
66    index = 0
67    # Add additional edges between communities based on the assigned
          numbers
68    for i in range(comm_count):
69      for j in range(i + 1, comm_count):
70        num_edges = normalized_numbers[index]
71        index += 1
72        add_fixed_inter_community_edges(graph, i, j, num_edges)
73    # Make sure that the whole graph is interconnected
74    interconnected_graph = make_fully_interconnected(graph)
75    return interconnected_graph
```

Listing 6.2: Grapherzeugung mit fixer Anzahl Intra- und Inter-Community-Kanten

Neben der Interpolationsroutine wurde eine weiteres Verfahren zur Erzeugung mehrerer Graphen ohne einzelne Eingabe der Parameter definiert. Diese erfordert genau wie die Interpolation die Eingabe einer weiteren Reihe Parameter. Dabei werden aber die Werte nicht interpoliert, sondern zufällig zwischen den eingegebenen Start- und Endparametern ausgewählt. Diese Routine ermöglicht eine Zufälligkeit der Graphen zu erreichen. Insbesondere bei der Approximation verschiedener Werte von Metriken wird so sichergestellt, dass nicht alle Graphen die gleichen Eigenschaften haben.

Für die Visualisierungsroutine wurde zuerst eine Darstellung der Graphen durch *PyGraphViz* entsprechend dem Konzept durchgeführt. Hierbei muss das Graphlayout im Programm statisch definiert werden. Die Darstellung in einem statischen Format kann zu einer eingeschränkten Erkennbarkeit führen, insbesondere bei Graphen mit unterschiedlicher Größe oder Konnektivität (Kanten- und Knotengrade). Um zumindest zwei verschiedene Basisfälle darstellen zu können, wurde die Visualisierungsroutine angepasst. Dabei wird je nach Knotenanzahl ein unterschiedliches Layout verwendet. Infolgedessen können sowohl kleine als auch große Graphen sinnvoll visualisiert werden. Allerdings erfordert das Rendern von Graphen mit vielen Knoten einen erhebliche Zeitaufwand, weshalb ein alternativer Ansatz hinzugefügt wurd. Dieser Ansatz nutzt Gephi, ein Open-Source-Softwaretool zur Visualisierung und Analyse komplexer Netzwerke und Graphen. NetworkX ermöglicht eine flexible Speicherung eines Graphen im Gephi-Format. Mit Gephi diese Dateien dann importiert und anschließend dargestellt werden. Hierbei kann für den jeweiligen Graphen die Auwahl eines passenden Layout-Algorithmus erfolgen, wodurch eine übersichtliche Darstellung möglich wird. Außerdem werden verschiedene Tools zur Untersuchung der Graphen bereitgestellt, wie zum Beispiel Statistiken, Metriken und Analysealgorithmen. Die Nutzung von Gephi zur Visualisierung wurde bei der Simulationsausführung bevorzugt, da so der Zeitaufwand reduziert wird. Wird PyGraphViz verwendet, so muss jeder Graph gerendert werden, was je nach Graph sehr viel Zeit in Anspruch nehmen kann. Eine Speicherung im Gephi-Format hingegen kann sofort erfolgen und die spätere Visualsierung erfordert lediglich einen geringen zusätzlichen Aufwand.

Nach der Visualisierung wurde weitere Logik zur Berechnung verschiedener Graphmetriken hinzugefügt. Diese Metriken werden einerseits für die *C/MxN*-Grapherzeugung verwendet (siehe Kapitel 5.2.1.2). Andererseits werden sie den YAML-Dateien der Graphressourcen hinzugefügt. Dadurch müssen der Operator und Runner keine eigenen Berechnungen der Metriken mehr durchführen. Alle auszuwertenden Daten liegen bei Ausführung vor und können ausgelesen werden. Für jeden Graphen erfolgt die Berechnung von allen im Kapitel 2.1.9 genannten Metriken. Diese werden den Grapheigenschaften mit entsprechenden Namen hinzugefügt. Mit dem Setzen der berechneten Metriken ist die Erstellung der YAML-Datei fast beendet. Es erfolgt lediglich noch eine Abfrage nach einem Serien-Label (*series*), welches dann den Graphressourcen zugewiesen wird. Dadurch wird es möglich, die Graphen einer Serie von Testreihen zuzuordnen. Die Serien

sind ein Konzept, das die flexible Ausführung von Simulationen über mehrere Graphen ermöglicht, wie in den folgenden Kapiteln erläutert wird.

Im Zuge der Evaluation von realen Netzwerken musste ein zusätzliches Tool entwickelt werden. Analog zur verwandten Arbeit von Sirocchi [108] sollte hier das Gossiping auf dem Gnutella Peer-to-Peer-Netzwerk ausgeführt werden. Bei Gnutella handelt es sich um ein File-Sharing-Netzwerk mit über sechzig tausend Knoten. Aufgrunddessen musste eine Aufteilung des Netzwerkgraphen in Partitionen, also zusammenhängende Subgraphen einer festen Größe, erfolgen. Das Tool ermöglicht die Erzeugung von Partitionen von beliebigen Ausgangsnetzen, wobei die Anzahl und Größe der Subgraphen frei konfiguriert werden kann. Hierbei müssen die Ausgangsnetzwerke als Kantenliste eingelesen werden. Das Format entspricht hierbei dem Graphformat von *Network Repository* [84]. Hier sind viele Graphen von realen Systemen, sozialen Netzwerken sowie aus vielen anderen Bereichen wie beispielsweise Infrastruktur, Biologie und Geografie zu finden. Das Tool wendet anschließend dieselbe Logik wie das CLI an, um Kubernetes-Graphressourcen für die Partitionen zu erzeugen. Diese können dann gleichermaßen erstellt und für Simulationen verwendet werden.

6.4 Kubernetes Operator

Der Kubernetes Operator ist das Kernstück der Simulationsumgebung. Seine Aufgabe ist die Orchestrierung der Simulationsressourcen und die Abbildung des Workflows. Dabei realisiert er den Auf- und Abbau der Netzwerke sowie die Konfiguration der Knoten sowie der Steuerungskomponente.

Der Prototyp des Simulation Operators wurde entsprechend der Konzeption weiterentwickelt. Die Anwendung ist ein Python-Skript, das den Lebenszyklus einer Gossiping-Simulation auf einem Netzwerkgraphen verwaltet. Pods und Services werden im Kubernetes-Cluster erstellt und überwacht. Dazu wird das Kopf-Framework verwendet, wobei es sich um ein populäres Operator-Framework handelt. Kopf macht es möglich, Handler für Ereignisse wie das Erstellen und Löschen benutzerdefinierter Ressourcen zu definieren. Die Custom Resource Definition (CRD) der Simulationsressource ist in Listing 6.3 dargestellt und wird im Folgenden erläutert.

Die Simulationsressource wählt über den *graphSelector* zugehörige Graphressourcen aus. Dabei kann mit *matchLabels* die Selektion von einzelnen Graphen direkt über den Namen erfolgen. Wenn stattdessen das Serien-Label (*series*) verwendet wird, können gleich mehrere Graphen referenziert werden. Dazu werden alle Graphen einer Simulationsserie mit dem gleichen Tag versehen. Anschließend können sie über diesen Tag ausgewählt und einer Simulation zugeordnet werden. Darüber hinaus definiert die CRD den Algorithmus und die Anzahl an Wiederholungen der Ausführung. Es kann außerdem festgelegt werden, ob die Initialisierung der Graphen zufällig erfolgen soll oder nicht. Bei einer zufälligen Initialisierung wählt jeder Knoten seinen anfänglichen Wert selbst. Im anderen Fall wird, falls vorhanden, die Liste der Knotenwerte am Graphen verwendet, um statische Initialwerte zuzuordnen. Ist diese Liste nicht vorhanden, so setzt jeder Knoten seine Knotennummer als Wert. Mit *visualize* kann entschieden werden, ob grafische Darstellungen zu den Graphzuständen exportiert werden sollen. Des Weiteren beinhaltet die CRD algorithmenspezifische Daten, wie den *factor* und den *priorPartnerFactor*. Zu Beginn der Implementierung waren diese Parameter nur einfache Zahlenwerte. Im späteren Verlauf wurde ein Weg gesucht, wie flexibel Simulationsserien gestaltet werden können, die verschiedene Algorithmenparameter erproben. Als Anwendungsbeispiel ist die Simulation des WeightedFactor-Algorithmus mit aufsteigendem Faktor zu nennen. Zur Simulation von verschiedenen Faktoren auf den selben Graphen muss die Angabe mehrerer Para-

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: simulations.gossip.io
spec:
  group: gossip.io
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                graphSelector:
                  type: object
                  properties:
                    matchLabels:
                      type: object
                      additionalProperties:
                        type: string
                algorithm:
                  type: string
                  default: 'default'
                repetitions:
                  type: number
                  minimum: 1
                  default: 1
                randomInitialization:
                  type: boolean
                  default: true
                visualize:
                  type: boolean
                  default: false
                factor:
                  type: array
                  items:
                    type: number
                priorPartnerFactor:
                  type: array
                  items:
                    type: number
                simulationProperties:
                  additionalProperties:
                    type: string
                  type: object
              required: ['graphSelector']
  scope: Namespaced
  names:
    plural: simulations
    singular: simulation
    kind: Simulation
    shortNames:
    - sim
```

Listing 6.3: CRD der Simulationsressource

meterwerte erfolgen können. Deshalb werden bei der Definition nun statt Zahlenwerten Arrays verwendet. Wenn unterschiedliche Werte für verschiedene Paramter angegeben sind, so werden die Simulationen für jede Parameterkombination ausgeführt. Die Details zur Simulationssteuerung sind in Kapitel 6.5.1 dargestellt. Zuletzt unterstützt die CRD die Angabe von beliebigen Metadaten. Dazu können die Simulationseigenschaften (*simulationProperties*) durch Schlüssel-Wert-Paare (über den Eintrag *additionalProperties*) erweitert werden.

Im Folgenden wird die Funktionsweise des Simulation Operators erläutert. Der Operator reagiert auf die Erstellung und Löschung von Objekten der Simulationsressource. Dazu muss er mit dem Kubernetes-API-Server interagieren, wobei diese Interaktion durch die Verwendung des Kopf-Frameworks abstrahiert wird. Die Bibiliothek ermöglicht die Erstellung der verschiedenen Kubernetes-Clients, welche dann zur Interaktion verwendet werden können. Zuerst muss jedoch eine Authentifizierung erfolgen, welche den Kontext für die darauffolgende Kommunikation setzt. Dazu wird im Cluster ein ServiceAccount angelegt und mit einem Token versehen. Der *ServiceAccount* gewährt dem Operator die erforderliche Identität, um auf Ressourcen im Cluster zuzugreifen. Außerdem ist der ServiceAccount mit einem *Token* verknüpf, welcher Authentifizierung gegenüber dem API-Server dient. Zuletzt erfolgt die Zuordnung der Admin-Rolle über ein *ClusterRoleBinding*. Infolgedessen wird der Operator unter anderem befähigt Pods und Services zu erstellen und zu löschen. Das Deployment des Operators und die Ressourcen zur Authentifizierung sind in Listing 6.4 dargestellt.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simulation-operator
spec:
  replicas: 1
  selector:
  matchLabels:
    app: simulation-operator
  template:
    metadata:
      labels:
        app: simulation-operator
  spec:
    serviceAccountName: simulation-operator-sa
    containers:
      - name: simulation-operator
        image: gossip-registry:32652/simulation-operator:v0
    imagePullSecrets:
      - name: reg-cred-secret
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: simulation-operator-sa
---
apiVersion: v1
kind: Secret
metadata:
  name: operator-service-account-secret
  annotations:
    kubernetes.io/service-account.name: simulation-operator
type: kubernetes.io/service-account-token
---
```

```
35 apiVersion: rbac.authorization.k8s.io/v1
36 kind: ClusterRoleBinding
37 metadata:
38   name: simulation-operator-cluster-role-binding
39 roleRef:
40   apiGroup: rbac.authorization.k8s.io
41   kind: ClusterRole
42   name: cluster-admin
43 subjects:
44   - kind: ServiceAccount
45     name: simulation-operator-sa
46     namespace: default
```

Listing 6.4: Simulation Operator Deployment

Ein erfolgreiches Deployment des Simulation Operators stellt sicher, dass dieser über sämtliche erforderliche Berechtigungen verfügt. Bei Applikationsstart lädt der Operator dann die cluster-interne Konfiguration (*Incluster-Config*), um die Kubernetes-Clients zu erstellen. Für die Interaktion mit dem API-Server werden zwei unterschiedliche Client-Typen benötigt. Es wird zwischen dem *Standard-API-* und dem *Customs-API-Client* unterschieden. Der Standard-API-Client dient der Interaktion mit den grundlegenden Kubernetes-Objekten, wie Pods und Services. Im Gegensatz dazu wird der Custom-API-Client speziell für die Verwendung mit Custom Resources eingesetzt. Die Custom Resources sind auf die Funktionsweise des Operators zugeschnitten und erweitern die Standard-API. Dadurch ermöglichen sie die Abbildung von spezifischen Workflows wie der Durchführung von Simulationen auf Netzwerkgraphen.

Im nächsten Schritt wird auf Ereignisse (z.B. Erstellung eines Simulationsobjekts) der Kubernetes-API reagiert. Dazu erfolgt die Registrierung von *Handlern*, welche beim Eintritt der Ereignisse aufgerufen werden. Funktionen werden mit einem sogenannten *Decorator* (z.B. *@kopf.on.create('gossip.io', 'v1', 'simulations')*) versehen. So kann bei Erstellung eines Simulationsobjekts die Routine zum Aufbau des Netzwerks gestartet werden. Gleichermaßen erfolgt die Dekoration der Logik zum Abbau der Ressourcen beim Löschen eines Objekts. Die Modifikation von laufenden Simulationen wird nicht behandelt, da durch Veränderung der Parameter die Ergebnisse nicht mehr ausgewertet werden können.

Wird ein Simulationsobjekt erstellt, so erfolgt die Initialisierung einer entsprechenden Simulation beziehungsweise Serie von Simulationen. Der Hauptbestandteil der Initialisierung ist die Routine zum Aufbau des Netzwerks, wobei Pods und Services anhand des Graphen erzeugt werden. Das Vorgehen wird nun im Folgenden schrittweise dargestellt:

- Zuerst wird die Spezifikation der Simulationsressource analysiert. Hier werden die zu simulierenden Graphen auf der Grundlage der angegebenen Labels bestimmt.
- Dann erfolgt die Auswertung der einzelnen Graphen:
  - Im ersten Schritt werden graphenspezifische Daten aufbereitet. Dazu zählt zum Beispiel die Erstellung eines Dictionaries für die Nachbarn eines jeden Knotens.
  - Ebenfalls werden algorithmenspezifische Daten erhoben und verarbeitet. Ein Beispiel hierfür ist die Ausführung der Methode von Louvain zur Erkennung von Communities. Anschließend werden Dictionaries erzeugt, welche zur Zuordnung von Nachbarn der gleichen Community dienen.
  - Optional kann eine feste Zuteilung von Knotenwerten erfolgen. Dabei werden entweder die am Graphen festgelegten Werte oder die Knotenummern entsprechend der Adjazenzliste verwendet.

  - Im nächsten Schritt wird der Service für die Graphknoten erzeugt. Dieser ermöglicht die TCP-Kommunikation zwischen den Knoten und die gRPC-Kommunikation mit dem Runner.
  - Im Anschluss wird die Erstellung der Node Pods durchgeführt. Umgebungsvariablen werden auf Grundlage der Simulationseinstellungen und Grapheigenschaften definiert. Eine Zuordnung erfolgt bei der Container-Definition. Es hat sich gezeigt, dass bei der gleichzeitigen Bereitstellung von vielen Pods das System überlastet wird. Das Control Plane stellt hierbei ein Bottleneck dar. Die Leistung kann deutlich verbessert werden, wenn die Pod-Erzeugung in Chargen durchgeführt wird. Dabei wartet die Applikation, bis sich alle Pods im aktuellen Batch im Status *Running* befinden. Listing 6.5 bietet einen Auszug aus diesem Prozess.
  - Danach wird der Runner-Service erzeugt, welcher die gRPC-Kommunikation mit den Knoten zwecks Simulationssteuerung realisiert.
  - Zuletzt wird der Runner-Pod erstellt, wobei auch hier entsprechende Umgebungsvariablen gesetzt werden. Damit sind schließlich alle Applikationen bereitgestellt und die Simulation beginnt.
  - Dier Runner überwacht die Simulation auf einem Graphen anhand der Ressourcen. Dabei wartet er bis alle Pods den Status *Succeeded* erreichen, also die Anwendungen erfolgreich terminieren. Es kann jedoch vorkommen, dass einzelne Container nicht beendet werden, da in seltenen Fällen einzelne gRPC-Aufrufe fehlschlagen. Für die Ausführung der Gossiping-Befehle ist dies vernachlässigbar, da die Konvergenzgeschwindigkeit aufgrund der Seltenheit des Phänomens unbeeinflusst bleibt. Wird jedoch der Befehl zum Stoppen der Anwendung nicht ausgeführt, so musste der einzelne Pod manuell beendet werden, damit die Simulation fortfährt. Anstatt stets auf alle Pods zu warten, wird nur solange unterbrochen bis ein Großteil der Pods (99 Prozent) über einen längeren Zeitraum (20 Sekunden) terminiert ist.
  - Die Simulation auf einem Graphen wird beendet, indem alle Ressourcen gelöscht werden.

- Die Ressourcen der zuletzt ausgeführten Simulation bleiben bis zur Entfernung der Simulationsressource bestehen. Als Folge können die Pod-Logs inspiziert werden, welche ausführliche Informationen zum Simulationsablauf beinhalten. Dabei wird das Logging-Modul verwendet, um Meldungen während des Erstellungs- und Löschungsprozesses zu protokollieren.

```
1  num_nodes = len(nodes)
2  num_batches = math.ceil(num_nodes / batch_size)
3
4  # Create a Pod for each node in the graph
5  for batch_index in range(num_batches):
6    start_index = batch_index * batch_size
7    end_index = min((batch_index + 1) * batch_size, num_nodes)
8    batch_nodes = nodes[start_index:end_index]
9
10   for node in batch_nodes:
11     pod_name = get_resource_name(graph_index, name, node)
12     labels = {
13       'app': 'gossip',
14       'simulation': name,
15       'simulation_id': simulation_id,
```

```
16        'graph': graph_name,
17        'node': str(node)
18      }
19      # Create the container for the Pod
20      container = client.V1Container(
21        name=DOCKER_NODE_NAME,
22        image=DOCKER_NODE_IMAGE,
23        env=env,
24        ports=[
25            client.V1ContainerPort(container_port=TCP_PORT, name='tcp'),
26            client.V1ContainerPort(container_port=GRPC_PORT, name='grpc')
27        ]
28      )
29      # define the pod
30      pod = client.V1Pod(
31        metadata=client.V1ObjectMeta(
32          name=pod_name,
33          namespace=namespace,
34          labels=labels
35        ),
36        spec=client.V1PodSpec(
37          restart_policy='OnFailure',
38          containers=[container],
39          image_pull_secrets=[client.V1LocalObjectReference(name=
                REGISTRY_SECRET_NAME)],
40          topology_spread_constraints=[
41            client.V1TopologySpreadConstraint(
42              max_skew=1,
43              topology_key="simulation_node",
44              when_unsatisfiable="DoNotSchedule",
45              label_selector=client.V1LabelSelector(
46                match_labels={
47                  "app": "gossip"
48                }
49              )
50            )
51          ]
52        )
53      )
54      try:
55        # create the pod in the current namespace
56        api.create_namespaced_pod(namespace=namespace, body=pod)
57        log.info(f'Pod {pod_name} created.')
58        pods.append(pod_name)
59      except ApiException as e:
60        log.error(f'Error creating pod: {e}')
61
62  # Wait for the pods in this batch to start
63  log.info(f'Waiting for node pods in batch {batch_index + 1} to start.')
64  while True:
65    # List all pods matching the specified labels
66    node_pods = api.list_namespaced_pod(namespace=namespace,
          label_selector=','.join(
67      [f"{k}={v}" for k, v in labels.items()]))
68    # Check if all pods in this batch are in the "Running" state
69    if all(pod.status.phase == 'Running' for pod in node_pods.items):
70      log.info(f'All node pods in batch {batch_index + 1} are now running.
            ')
71      break
72  log.info(f'Finished creating Pods for graph {graph_name}.')
```

Listing 6.5: Erzeugung der Node Pods durch den Simulation Operator

Alle erstellten Ressourcen werden mit Labels versehen, wodurch die zugehörige Simulation klar erkennbar wird. Hier erfolgt neben der dem Setzen eines Tags entsprechend der Benamung auch die Vergabe einer ID zur eindeutigen Kennzeichnung. So kann bei wiederholter Ausführung der gleichen Simulation eine Unterscheidung erfolgen. Bei Serien beziehungsweise Reihen werden Tags entsprechend der Graphnamen gesetzt. So ist bei der jeweiligen Ausführung erkennbar, welche Simulation gerade durchgeführt ist. Des Weiteren wird es möglich, den Fortschritt der Simulation abzuschätzen. Außerdem können durch die Verwendung von eindeutigen Namen, Labels sowie den Einsatz von Namespaces parallele Simulationen ausgeführt werden.

6.5 Container Applikationen

Im Rahmen der Implementierung wurden zwei Arten von Container-Applikationen entwickelt, der Simulation Runner und der Node-Service. Der Simulation Runner verwaltet das Gossiping. Er ruft alle Knoten in jeder Runde auf, analysiert die Konvergenz und speichert die Ergebnisse. Jeder simulierte Netzwerkknoten führt den Node-Service aus, welcher die Funktionalität für das Gossiping bereitstellt. Beide Applikationen werden mit Hilfe des Kubernetes Operators als Container in Pods ausgeführt. Im Folgenden wird nun die Implementierung der Anwendungen besprochen.

6.5.1 Simulation Runner

Der Simulation Runner ist die Steuerungskomponente der Simulationsumgebung. Dabei umfassen seine Aufgaben die Verwaltung der Gossiping-Runden und die Ergebnisspeicherung. Er beinhaltet eine Runner-Klasse sowie einige Datenklassen für die Graph-, Algorithmus- und Ergebnisdaten. Bei Anwendungsstart werden zuerst die Umgebungsvariablen ausgelesen, welche zuvor durch den Operator aufbereitet wurden. Dazu zählen die anwendungsspezifischen Daten wie die Simulations-ID und der Name, die Adjazenzliste des Graphen, der Algorithmus und dessen Parameter. Die Verbindungsdaten (Endpunkt und Zugangsdaten) für MinIO werden direkt aus der ConfigMap und dem Secret extrahiert und anschließend auch über die Umgebungsvariablen konfiguriert. Die anwendungsspezifischen Daten werden genutzt, um die Klassen für Graph- und Algorithmusdaten zu initialisieren. Hierbei erfolgt die Erstellung eines Graphobjekts mit *NetworkX*, welches im Folgenden genutzt wird, um den Netzwerkzustand zu überwachen. Mit den aufbereiteten Daten wird die Runner-Klasse initialisiert, wobei jeder Algorithmusparameter einem Array hinzugefügt wird. Je nach eingesetztem Algorithmus kann dieses Array keine, einzelne oder mehrere Parameter beinhalten. Entsprechend Listing 6.6 erfolgt dann die Bildung aller Kombinationen für die verschiedenen Algorithmusparameter. Bei der Simulationsausführung werden für jede Kombination die Simulationen so oft wiederholt, wie dies in der Konfiguration festgelegt wurde. Dabei wird die Runner-Klasse genutzt, um die Parameter zu setzen, die Knotenwerthistorie zu initialisieren und das Gossiping auszuführen. Ebenfalls stößt sie die Speicherung von Ergebnissen an und kann genutzt werden, um Folgesimulationen einzuleiten. In Listing 6.7 ist die Ausführung der Runner-Instanz dargestellt.

Die Runner-Instanz verwaltet ein Verzeichnis von *gRPC-Stubs*. Die Stubs beinhalten Implementierungen entsprechend der Dienstmethoden und stellen die Kommunikationskanäle mit den Knoten dar. Dabei realisiert ein Stub die Parameterserialisierung sowie das Senden und Empfangen von Nachrichten. Die Initialisierung der Knotenwerthistorie führt gRPC-Aufrufe an alle Node-Services aus, um die initialen Knotenwerte abzufragen. Dabei erfolgen die Aufrufe parallel und asynchron über einen *ThreadPoolExecutor*. Wenn

```
algorithm_param_keys = list(algorithm_params.keys())
algorithm_param_values = list(algorithm_params.values())
param_combinations = [dict(zip(algorithm_param_keys, combo))
  for combo in itertools.product(*algorithm_param_values) if combo]
```

Listing 6.6: Erzeugung der Parameterkombinationen

```
# initialize the gossip runner
runner = GossipRunner(simulation_name, algorithmData, repeated,
    multi_params, graphData, minioAccess)

for i in range(0, num_executions):
  log.info(f'Execution #{i + 1}/{num_executions} starting...')
  if not no_params:
    chosen_params = param_combinations[i]
    log.info(f"Set of params: {chosen_params}")
    runner.set_algorithm_parameters(chosen_params)
  for j in range(0, repetitions):
    log.info(f'Run #{j + 1}/{repetitions} executing...')
    runner.init_node_value_history()
    # run the gossip simulation
    runner.run()
    runner.store_results()
    if j < repetitions - 1:
      runner.init_next_run()
  if repeated:
    runner.store_average_results()
  if i < num_executions - 1:
    runner.init_next_execution()

runner.stop_node_applications()
log.info("Stopping application.")
sys.exit(0)
```

Listing 6.7: Ausführung der Runner-Klasse

alle Aufrufe beendet sind, werden die Knotenwerte ausgelesen und entsprechend am NetworkX-Graphen als Label gesetzt. Für den Fall, dass eine Visualisierung des Graphzustands in den einzelnen Runden erfolgen soll, wird eine *Gephi-Datei* erzeugt. Der Graph kann dann mit dieser Datei in der Visualisierungssoftware nach Belieben dargestellt werden. Optional kann auch eine direkte Erzeugung von statischen und animierten Bilddateien mit *PyGraphViz* erfolgen. Die Visualisierungen werden in beiden Fällen (PyGraphViz und Gephi) in einen *ByteBuffer* geschrieben. Dieser wird dann in einem Dictionary mit der Rundennummer als Schlüssel hinterlegt, wodurch die Darstellungen später flexibel persistiert werden können.

Das Ausführen der Gossiping-Simulation wird auch durch gRPC-Aufrufe realisiert. Dabei erfolgen die Gossip-Calls synchron und nicht parallel, um ein deterministisches Verhalten zu gewährleisten. Die Knoten werden also nacheinander zum Gossiping aufgerufen. Insbesondere bei skalenfreien Graphen, wo Hub-Knoten sehr viele Nachbarn haben, ist der Ablauf einer Gossip-Runde sonst nicht mehr nachvollziehbar. Bei einer asynchronen Implementierung wählen alle Knoten zum gleichen Zeitpunkt einen Partner aus und kommunizieren mit diesem. Hub-Knoten erhalten dann eine große Anzahl von eingehenden Gossip-Anfragen, welche dann alle gleichzeitig verarbeitet werden müssen. Ändert sich bei einem Austausch der Knotenwert und laufen bereits parallele Prozesse, so wird weiterhin der alte Wert kommuniziert. Ein synchrones Gossiping stellt sicher,

dass Knotenwertänderungen in der korrekten Reihenfolge erfolgen. Nachdem alle Knoten zum Gossiping aufgerufen wurden, werden die Knotenwerte erneut abgefragt. Dies kann dann wiederum parallel und asynchron erfolgen. Auch hier erfolgt gegebenenfalls eine Visualisierung des Graphen mit den aktualisierten Knotenwerten.

Alle Ergebnisse werden im MinIO Object Storage persistiert. Zur Kommunikation mit der Speicherlösung wird ein Client initialisiert. Dieser stellt eine Verbindung her und überprüft, ob der konfigurierte Bucket existiert. Wenn der Bucket noch nicht vorhanden ist, wird er erstellt. Während der Simulationsausführung werden alle Daten in einer Datenklasse gesammelt. Diese beinhaltet zum Simulationsende beispielsweise die verschiedenen Graphmetriken, die Parameter des Algorithmus und die Anzahl an benötigten Runden. Die Datenklasse stellt Funktionalität zur Konvertierung in einen *BufferedReader* bereit. Dabei werden alle Daten im JSON-Format serialisiert, wodurch eine flexible Speicherung der Ergebnisse im Object Storage möglich wird. Der Aufbau einer solchen Ergebnisdatei ist in Listing 6.8 dargestellt. Um die Visualisierungen zu speichern, werden die Buffer-Objekte verwendet, welche direkt aus dem Dictionary entnommen werden können. Auch die Knotenwerthistorie wird als Datei im JSON-Format persistiert. Sie verfügt dazu über analoge Funktionalität zur Konvertierung wie die Datenklasse. Gleichermaßen wird bei einer wiederholten Ausführung die Speicherung der gemittelten Ergebnisse durchgeführt.

```
1  {
2    "timestamp": "2023-07-14-13-13-18",
3    "num_rounds": 25.45,
4    "algorithm": "default",
5    "adj_list": "16384 17456 17460,14337 6843 9643
         17341,...,40955,2044,14335",
6    "graph_metadata": {
7      "assortativity": "0.233",
8      "averageAuthorityScore": "0.001",
9      "averageBetweennessCentrality": "0.0115",
10     "averageClosenessCentrality": "0.0831",
11     "averageCommunitySize": "34.4828",
12     "averageDegreeCentrality": "0.0024",
13     "averageEccentricity": "29.26",
14     "averageNodeDegree": "2.356",
15     "averageEigenvectorCentrality": "0.0046",
16     "averageHubScore": "0.001",
17     "averageNearestNeighborsDegree": "2.6135",
18     "averagePageRank": "0.001",
19     "averagePathLength": "12.4291",
20     "averageRichClubCoefficient": "0.0199",
21     "density": "0.0024",
22     "diameter": "43",
23     "edgeConnectivity": "1",
24     "estimatedPowerLawExponent": "13.1555",
25     "lowerBoundPowerLawRegion": "4.0",
26     "modularity": "0.833",
27     "nodeConnectivity": "1",
28     "nodeCount": "1000",
29     "numCommunities": "29",
30     "numEdges": "1178",
31     "overallAverageCommunityClustering": "0.0083",
32     "overallStdevCommunityClustering": "0.0716",
33     "stdevAuthorityScore": "0.0069",
34     "stdevBetweennessCentrality": "0.0137",
35     "stdevClosenessCentrality": "0.0135",
36     "stdevCommunitySize": "12.2929",
37     "stdevDegreeCentrality": "0.0009",
```

```
38      "stdevEccentricity": "3.2346",
39      "stdevNodeDegree": "15964.8743",
40      "stdevEigenvectorCentrality": "0.0313",
41      "stdevHubScore": "0.0069",
42      "stdevNearestNeighborsDegree": "0.6686",
43      "stdevPageRank": "0.0003",
44      "stdevRichClubCoefficient": "0.0272",
45      "transitivity": "0.0121",
46      "graphType": "real-network"
47    }
48  }
```

Listing 6.8: Gemittelte Simulationsergebnisdatei

Der Runner verwaltet bei Ausführung einer Simulationsserie die verschiedenen Parameterkonfigurationen. Die korrekte Steuerung und insbesondere die Persistenz der richtigen Konfigurationsdaten muss gewährleistet werden. Der Runner kann zur jeweiligen Simulationsiteration eine genaue Zuordnung der Parameter durchführen. Bei Beginn einer Ausführung werden die aktuellen Parameter gesetzt. Gleichermaßen wird am Ende der Ausführung der Ursprungszustand wiederhergstellt. Dabei werden gesammelte Daten der vorigen Ausführung entfernt. So erfolgt die Löschung der Ergebnisse und Buffer-Objekte der abgeschlossenen Simulation aus dem Speicher.

Ist die Simulation beendet, so werden asynchron alle Node-Services terminiert. Dies wird realisiert, indem gRPC-Aufrufe an die Knoten versandt werden. Sobald alle Nachrichten versendet wurden, terminiert auch die Runner-Applikation. Durch die ordnungsgemäße Beendigung der Container-Anwendungen wechseln die Pods in den Status *Completed*. Sind weitere Simulationen auf anderen Netzwerkgraphen durchzuführen, so wird der Operator erneut aktiv. In diesem Fall entfernt er alle Simulationsressourcen und fährt mit dem Aufbau der nächsten Simulation fort. Andernfalls bleiben die Anwendungen in diesem Zustand bis ein Nutzer die Simulationsressource entfernt.

6.5.2 Node-Service

Der Node-Service bildet einen Netzwerkknoten ab und stellt die Gossip-Funktionalität bereit. Hierbei wird die gesamte Logik für das Gossiping sowie die entsprechenden Netzwerkfunktionen in der Service-Klasse gekapselt. Darüber hinaus beinhaltet die Applikation weitere Klassen für Daten und Algorithmen, welche bei Anwendungsstart initialisiert werden. Dazu werden auch hier zuerst die Umgebungsvariablen ausgelesen. Die Umgebungsvariablen umfassen applikationsspezifische Daten, die zur Konfiguration des Knotenverhaltens dienen. Beispiele beinhalten den Hostnamen, die Nachbarn im Netzwerk, Simulationsparameter sowie Daten zum ausgeführten Algorithmus. Im ersten Schritt wird mit den Daten die Service-Klasse initialisiert, wodurch auch der verwendete Algorithmus konfiguriert wird.

Die verschiedenen Algorithmen sind innerhalb des Node-Services implementiert. Diese sind, wie im entworfenen Konzept festgehalten, modular aufgebaut. Neben dem Baseline-Verfahren wurden die Algorithmen *WeightedFactor* und *CommunityProbabilities* realisiert. Die entworfenen Varianten, die auf verschieden komplexen Gedächtnissen basieren, wurden ebenfalls umgesetzt. Die Speicherlogik ist in den Klassen *Memory* und *ComplexMemory* zu finden. Hier werden verschiedene Methoden bereitgestellt, um frühere Kommunikationspartner durch Anpassung der Gewichte bei darauffolgenden Wahlen zu benachteiligen. Das heißt, die Algorithmen erinnern sich an zuvor ausgewählte Nachbarn und selektieren diese unwahrscheinlicher erneut. Exemplarisch ist in Listing 6.11 die Implementierung des WeightedFactor-Algorithmus und der Basisklassen dargestellt.

```python
class Algorithm:
  def __init__(self, name, neighbors):
    self.name = name
    self.neighbors = neighbors
    log.info(f"Running algorithm: {self.name}.")
    log.info(f"Received neighbors: {self.neighbors}.")
    self.weights = [1] * len(neighbors)
    self.modifiable_parameters = False
    log.info(f'Initialized Baseline Algorithm.')

  def select_neighbor(self):
    selected = choose_neighbor(self.neighbors, self.weights)
    return selected

class WeightedFactor(Algorithm):
  def __init__(self, name, neighbors, community_neighbors, factors):
    super().__init__(name, neighbors)
    self.community_neighbors = community_neighbors
    self.factors = factors
    self.factor_index = 0
    self.factor = self.factors[self.factor_index]
    self.compute_weights()
    self.modifiable_parameters = True
    log.info(f'Initialized WeightedFactor with factor {self.factor}.')

  def compute_weights(self):
    self.weights = {}
    for neighbor in self.neighbors:
      if neighbor in self.community_neighbors:
        weight = 1.0
      else:
        # prioritize non-community neighbors with a given factor
        weight = self.factor
      self.weights[neighbor] = weight

class ComplexMemory(Memory):
  def init_memory(self, prior_partner_factors):
    self.prior_partner_factors = prior_partner_factors
    self.prior_partner_factor_index = 0
    self.prior_partner_factor = self.prior_partner_factors[self.
        prior_partner_factor_index]
    self.start_weights = copy.deepcopy(self.weights)
    self.modifiable_parameters = True
    log.info(f'Initialized ComplexMemory with prior partner factor
        {self.prior_partner_factor}.')

  def remember(self, neighbor):
    self.weights[neighbor] *= self.prior_partner_factor

  def forget(self):
    self.weights = copy.deepcopy(self.start_weights)

  def reset_memory(self):
    log.info('Memory reset.')
    self.forget()

class WeightedFactorComplexMemory(WeightedFactor, ComplexMemory):
  def __init__(self, name, neighbors, community_neighbors, factors,
      prior_partner_factors):
    super().__init__(name, neighbors, community_neighbors, factors)
    self.init_memory(prior_partner_factors)
```

Listing 6.9: Node-Service Gossip-Funktionalität

Der Node-Service ist als gRPC-Server implementiert, der dem Runner Funktionalität bereitstellt. Er implementiert die *GossipServicer-Schnittstelle*, welche wie in der Konzeption beschrieben (siehe Kapitel 5.2.4) aus der proto-Datei erzeugt wird. Die finale Datei ist in Listing 6.10 dargestellt.

```
syntax = "proto3";
package gossip;

service Gossip {
  rpc Gossip(GossipRequest) returns (GossipResponse) {}
  rpc CurrentValue(CurrentValueRequest) returns (CurrentValueResponse)
      {}
  rpc History(HistoryRequest) returns (HistoryResponse) {}
  rpc Reset(ResetRequest) returns (ResetResponse) {}
  rpc StopApplication(StopApplicationRequest) returns (
      StopApplicationResponse) {}
}

message GossipRequest {}
message GossipResponse {}

message CurrentValueRequest {}
message CurrentValueResponse {
  int32 value = 1;
}

message HistoryRequest {}
message HistoryResponse {
  repeated ValueEntry value_entries = 1;
}
message ValueEntry {
  int32 participations = 1;
  int32 value = 2;
}

message ResetRequest {}
message ResetResponse {}

message StopApplicationRequest {}
message StopApplicationResponse {}
```

Listing 6.10: Finale gossip-service.proto

Während der generierte Code die Methoden-Signaturen für den gRPC-Server festlegt, muss der Service die Logik für die Methoden implementieren. Beim Aufruf der gRPC-Stubs aufseiten des Simulation Runners werden diese dann ausgeführt. Folgende Funktionen werden hierbei vom Server angeboten:

- **Gossip:** Durchführen eines Gossipings mit Auswahl des Nachbarn entsprechend dem Algorithmus, bidirektionalem Werteaustausch und gegebenenfalls Setzen des neuen Werts
- **CurrentValue:** Rückgabe des aktuellen Knotenwerts
- **History:** Rückgabe der Historie aller Werte, also Initialwert sowie Wert nach jedem erfolgtem Gossiping
- **Reset:** Zurücksetzen des Services auf Zustand zu Beginn des Gossipings für den Zweck eine neue Simulationswiederholung zu starten
- **StopApplication:** Beenden der Anwendung, wird durch den Simulation Runner nach Abschluss der Simulation aufgerufen

```
def Gossip(self, request, context):
  log.info('[GRPC Gossip invoked]')
  self.gossip()
  return gossip_pb2.GossipResponse()

def gossip(self):
  log.info('Starting to gossip.')
  try:
      self.gossip_lock.acquire()
      neighbor = self.algorithm.select_neighbor()
      log.info(f'Selecting neighbor {neighbor} to gossip using algorithm
          {self.algorithm.name}.')
      self.gossip_lock.release()
      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
          client_socket:
          client_socket.connect((neighbor, TCP_SERVICE_PORT))
          client_socket.sendall(bytes(f'{self.name}:{self.value}', 'utf
              -8'))
          log.info(f"Sent {self.value} over TCP socket to {neighbor}.")
          data = client_socket.recv(TCP_BUFSIZE)
          # process the received data
          self.process_gossip_data(data)
  except socket.error as e:
      log.error(f"Socket error occurred: {str(e)}")
  log.info(f'Finished gossiping with {neighbor}.')
```

Listing 6.11: Node-Service Gossip-Funktionalität

Zur Umsetzung der Gossip-Funktionalität wird das Threading-Modul von Python verwendet. Dies ist notwendig, da jeder Service im Hintergrund auf eingehende Gossip-Verbindungen warten muss. Dies erfolgt in einem separaten Listen-Thread, wodurch im Haupt-Thread weiterhin die gRPC-Aufrufe verarbeitet werden können. Im Fall des Gossip-Aufrufs erfolgt die Auswahl eines Nachbarns und der Verbindungsaufbau zu diesem. Die Verbindung dabei durch eine aktive Anfrage über den TCP-Socket realisiert. Im Anschluss senden sich beide Knoten gegenseitig den eigenen Knotenwert. Um sicherzustellen, dass bei simultanem Gossiping mehrerer Knoten keine gleichzeitigen Wertänderungen stattfinden, werden Threading-Locks eingesetzt. Dadurch kann nur ein Thread gleichzeitig den Knotenwert modifizieren. Listing 6.11 stellt die gRPC-Aufrufe für das Gossiping sowie die aktive Logik für den Verbindungsaufbau dar.

Um die Funktionen CurrentValue und History umzusetzen, muss der Service eine Historie seiner Werteinträge führen. Hier wird nach jedem erfolgten Gossiping ein Eintrag erzeugt. Die Historie kann im Gesamten über gRPC abgerufen oder aus dem Log ausgelesen werden. Sie dient der Analyse und der Verhaltensbeobachtung. Durch sie kann genau nachvollzogen werden, welche Interaktionen zu welchen Wertänderungen geführt haben.

Die oben beschriebenen Funktionen werden durch den Simulation Runner bei der Simulationssteuerung aufgerufen. Nach der Initialisierung führt ein Node-Service nur dann Logik aus, wenn er über gRPC dazu aufgefordert wird. Die Knoten sind somit vollkommen passiv und reagieren nur auf Anfragen. Bei einem dezentralen Gossiping hingegen müssten sich die Knoten selbst organisieren. Dadurch wäre die Implementierung viel komplexer und die Auswertung von Simulationen deutlich schwieriger. Durch die zentrale Steuerungseinheit in Form des Simulation Runners wurde die Implementierung einfacher.

6.6 Evaluationsanwendungen

Im Zuge der Evaluation mussten die Daten der einzelnen Simulationreihen und der Serien zusammengetragen werden. Anschließend war es notwendig, die Ergebnisse zu analysieren. Hierbei stand die Betrachtung des Konvergenzverhaltens im Mittelpunkt. Es wurde ein Skript zum Auslesen der Ergebnisdaten sowie Jupyter-Notebooks zur Datenanalyse erstellt.

Das Ausleseskript konzentriert sich auf die initiale Verarbeitung der Simulationsergebnisse. Die Ergebnisse sind hierbei im Object Storage als JSON-Dateien abgelegt. Im ersten Schritt lädt das Skript ausgewählte Dateien herunter. Anschließend werden diese mit *Pandas* eingelesen und verarbeitet. Pandas ist ein populäres Open-Source-Werkzeug zur Datenanalyse und -manipulation. Dabei werden zum Beispiel redundante Informationen entfernt, bestimmte Schlüssel umbenannt sowie relevante Metadaten extrahiert. Zudem erfolgt eine Kategorisierung der Daten entsprechend der Testreihen beziehungsweise Serien. Auch eine Zusammenfassung von Daten, um Vergleiche zu ziehen, ist möglich. Das Skript stellt außerdem sicher, dass die Datentypen innerhalb des DataFrame für die Speicherung in einer SQLite-Datenbank geeignet sind. Dies ist sinnvoll, da die Daten so zwischengespeichert werden können. Infolgedessen können die Erstellung der JupyterNotebooks zu einem späteren Zeitpunkt erfolgen. Sie können die Daten dann einfach erneut in Pandas-DataFrames einlesen und anschließend beliebige Analysen und Manipulationen durchführen. Dabei können wertvolle Erkenntnisse aus den Daten gezogen werden, ohne die ursprünglichen Ergebnisse zu verändern. Auch die Durchführung komplexer Berechnungen und die Erstellung aussagekräftiger Visualisierungen wird so erleichtert. SQLite wird zur temporären Speicherung der aufbereiteten Daten verwendet, da es eine leichtgewichtige Alternative zu einer dedizierten Datenbank ist. Zudem kann eine Einbettung der SQLite-Datenbank im Code erfolgen, wodurch kein externer Server benötigt wird, der eingerichtet und verwaltet werden muss. Stattdessen erfolgt die Speicherung der Datenbank als einzelne Datei, wodurch diese dann kopiert und wiederholt modifiziert werden kann. Auch für die Langzeitspeicherung der Ergebnisse bietet sich dies an, da die Datenbankdateien einfach im Code-Repository hinterlegt werden können.

Ist die Vorverarbeitung der Daten abgeschlossen, so wird die SQLite-Datenbank eingerichtet. Tabellen werden erstellt oder aktualisiert, um die verarbeiteten Daten zu speichern. Dann erfolgt das Einfügen der Daten, wobei die eindeutige Spalte *ObjectStoragePath* verwendet wird um sicherzustellen, dass keine doppelte Einträge entstehen. Zudem wird in der SQLite-Datenbank der letzte Ausführungszeitpunkt des Skripts hinterlegt. Es werden nur Ergebnisdateien heruntergeladen, die nach dem letzten Zeitpunkt erstellt wurden. Datenanalysen und -abfragen können so zu einem späteren Zeitpunkt und logisch getrennt erfolgen. Für jede Evaluation kann eine separate SQLite-Datenbank-Datei erzeugt werden, wodurch eine gemeinsame oder getrennte Betrachtung der Daten möglich wird.

Die Jupyter-Notebooks sind so aufgebaut, dass zuerst die Ergebnisse und deren Korrektheit verifiziert wird. So werden exemplarische Resultate betrachtet und Prüfungen wie zum Beispiel der Anzahl an Simulationen durchgeführt. Anschließend erfolgt die Visulisierung der Ergebnisse. Hierbei erfolgt die Darstellung je nach Simulationsreihe auch nach den verschiedenen Parameterkonfigurationen. Es werden Vergleiche zwischen den Netzwerken, Algorithmen und Konfigurationen gezogen, indem unter anderem Mittelwerte der benötigten Runden bis zur Konvergenz berechnet werden. Dabei erfolgt die Visualisierung der Verteilung der Rundenzahlen im Einzelnen. Hier wird stets der direkte Vergleich zum Baseline-Verfahren gezogen. Es werden darüber hinaus aber auch Plots erstellt, welche die Gesamtleistung zusammenfassen. Diese umfassen dann die Mittelwerte aller betrachteten Konfigurationen. Zusätzlich werden die untersuchten Graphen

analyisert und deren Metriken betrachtet. Beispielsweise wird die Verteilung der Modularitätswerte bei skalierfreien hochmodularen Graphen ausgewertet. Ebenfalls erfolgt die Evaluation der Korrelation zwischen den berechenten Metriken und der Konvergenzrate.

7 Evaluation

Das Evaluationskapitel befasst sich mit der Bewertung des Fertigstellungsgrads der Implementierung. Die zu Beginn aufgestellten funktionalen und nicht-funktionalen Anforderungen werden dazu mit der Umsetzung verglichen. Der Hauptfokus dieses Kapitels liegt auf der detaillierten Vorstellung der Simulationsreihen. In diesem Zusammenhang werden für jede Reihe zuerst die eingesetzten Graphen, Algorithmen und Parametrisierungen beschrieben. Anschließend erfolgt die Präsentation der gesammelten Messwerte. Hierbei wird ein Vergleich und eine Bewertung der entwickelten Verfahren auf den verschiedenen Netzwerken durchgeführt. Außerdem ergeben sich aus den Ergebnissen Schlussfolgerungen. Zum Schluss werden die Evaluationsergebnisse nochmals zusammengefasst, um die wichtigsten Erkenntnisse der Thesis zu verdeutlichen.

7.1 Anforderungserfüllung

Der Fertigstellungsgrad der Umsetzung wird basierend auf den Anforderungen beurteilt. Es wurden alle Must- und Should-Have-Anforderungen umgesetzt. Auch die Nice-To-Have-Anforderungen konnten abgedeckt werden. Folgt man der *0/100-Methode* zur Ermittlung eines Fortschrittsgrads für das Gesamtprojekt, erreicht man einen kumulierten Fertigstellungsgrad von hundert Prozent [116]. Die Umsetzung der festgelegten Anforderungen wurde im Einzelnen beurteilt. Die Bewertungen sind im Anhang C.1 in den Tabellen C.1, C.2 und C.3 zu finden.

7.2 Simulationsreihen

Zuerst wurden grobe Konzepte für die ersten Simulationsreihen festgelegt (siehe Kapitel 5.2.5). Hier erfolgt die Definition der zu untersuchenden Aspekte wie Graphtypen, Algorithmen oder deren Konfigurationen. Anschließend wurden die ersten Simulationen ausgeführt und ausgewertet. Das Ziel bestand darin, Graphtypen und Algorithmen zu ermitteln, die vertieften Untersuchungen bedürfen. Dabei wurden die Netzwerke mit dem größten Optimierungspotential ausgewählt. Die Performance der gewichteten Verfahren konnte dann für diese optimiert werden. Es wurden speziell die Algorithmen eingehender untersucht, welche die besten Leistungsresultate erzielten. Aufbauend auf den Erkenntnissen der vorangehenden Tests, erfolgten fortlaufend iterative Simulationsreihen. Dies war notwendig, um den zeitlichen Aufwand der Simulationsauswertung zu begrenzen. Das Erproben aller möglichen Algorithmen und deren verschiedenen Parametrisierungen hätte den zeitlichen Rahmen gesprengt. Schlussendlich wurde noch die Performance auf realen Netzwerken evaluiert und ein Vergleich mit einer verwandten Arbeit durchgeführt. Diese Arbeit beschäftigt sich ebenfalls mit der Optimierung von Gossiping-Verfahren durch Einsatz von Community-Strukturen und wurde während diesem Projekt veröffentlicht.

Zur Ausführung einer Simulationsreihe müssen mehrere Schritte durchgeführt werden. Dabei interagiert der Nutzer mit den verschiedenen Anwendungen der Simulationsumgebung. Der Gesamtprozess beinhaltet die folgenden Systeminteraktionen:

1. **Erzeugung von Graphen über das CLI-Tool:** In Vorbereitung auf jede Simulationsreihe werden Graphen erstellt. Diese bilden die Grundlage für die Simulation und dienen der Netzwerkmodellierung. Die Grapherzeugung kann hierbei mit verschiedenen Routinen erfolgen. Bei Verwendung des MxN-Erzeugungsverfahren werden mehrere Graphen mit verschiedenen Parametern generiert. Alternativ können auch Graphen generiert werden, bei denen eine Metrik (zum Beispiel die Modularität) einen bestimmten Wert approximiert. Hierbei erfolgt die Anwendung des C/MxN-Verfahrens, um die besten Graphen aus der Gesamtheit der erzeugten Graphen auszuwählen. Die verschiedenen Verfahren zur Mehrfacherzeugung von Graphen sind detailliert in den Kapiteln 5.2.1.2 und 6.3 dargestellt.

2. **Anlegen der generierten Graphressourcen:** Das CLI-Tool bietet die Option die erzeugten Graphen direkt als YAML-Dateien zu speichern. Infolgedessen können sie unmittelbar als Kubernetes-Objekte erstellt werden. Eine Markierung und Gruppierung von Graphen kann durch Labels erfolgen, wodurch eine effiziente Verwaltung ermöglicht wird. Anhand gesetzter Labels kann dann eine Selektion von den Simulationsressourcen erfolgen. Die Graphobjekte können, sobald sie einmal angelegt wurden, in beliebigen Simulationen wiederverwendet werden.

3. **Definieren der Simulationsressource:** Für jede Simulationsreihe wird eine Simulationsressource definiert, welche die Parameter, Graphen und Wiederholungen festlegt. Diese Ressource bildet die Grundlage für die automatisierte Durchführung der Simulation. Der Nutzer muss diese manuell definieren.

4. **Ausführen der Simulation im Cluster:** Die Simulation wird gestartet, indem die vordefinierte Simulationsressource im Kubernetes-Cluster angelegt wird. Der Simulation Operator überwacht die Ressource und führt die Simulation aus. Während der Simulation agieren die Node Pods als Hosts und tauschen untereinander Informationen aus. Der Runner sorgt für den Auf- und Abbau der Ressourcen, falls mehrere Simulationen auf verschiedenen Graphen oder mit unterschiedlichen Parametern ausgeführt werden. Ist die letzte Simulation beendet, so verbleiben alle Pods im Completed-Status bis die Simulationsressource entfernt wird. Die Ausführungsdauer kann dabei stark variieren. Sie ist abhängig von Faktoren wie der Performance des Algorithmus, der Anzahl an Graphen und Wiederholungen (statisch sowie durch verschiedene Parameterkombinationen). Besonders aufwendige Simulationen mit komplexen Berechnungen können mehrere Tage in Anspruch nehmen. Die erste Simulationsreihe hat zum Beispiel pro Algorithmus über fünf Tage benötigt.

Im Folgenden erfolgt nun die Vorstellung der verschiedenen Simulationsreihen. Jede dieser Reihen beginnt mit einer detaillierten Definition, in welcher ihr Aufbau und Zweck erläutert werden. Die Simulationen dienen der Untersuchung von verschiedenen Kombinationen von Netzwerken und Algorithmen. Hierbei werden stets gleich große Netzwerke verwendet, die aus tausend Knoten bestehen, wobei die Anzahl der Kanten je nach Testreihe variabel ist. Nach der Definition folgt die Auswertung der Ergebnisse. Auf Grundlage dieser Ergebnisse können verschiedene Erkenntnisse gewonnen, die gegebenenfalls mit den anfänglichen Annahmen verglichen werden. Abschließend wird ein umfassendes Fazit über sämtliche Simulationsreihen gezogen.

7.2.1 Modularität bei skalenfreien Netzwerken

Die erste Testreihe befasst sich mit der Korrelation zwischen Modularität und Konvergenzzeit bei skalenfreien Netzwerken. Diese Netze weisen eine Powerlaw-Gradverteilung auf.

Dabei handelt es sich um eine Eigenschaft, welche auch in vielen realen Netzwerken zu finden ist. Aus diesem Grund wurden zunächst skalenfreie Netzwerke untersucht. Die Modularität wiederum ist eine zentrale Metrik zur Bestimmung der Ausprägung von Community-Strukturen. Sie ist wichtig, um beurteilen zu können, wie stark Communities ausgeprägt sein müssen, um eine Optimierung erreichen zu können. Ziel dieser Simulationsreihe ist es nachzuweisen, dass durch topologiebasierte Gewichtung die Konvergenz bei modularen skalenfreien Netzwerken beschleunigt werden kann. Dazu wird der Baseline- mit dem WeightedFactor-Algorithmus (siehe Kapitel 5.2.4) verglichen. Bei dem WeightedFactor-Verfahren werden beim Gossiping Nachbarknoten aus anderen Communities bevorzugt. Dieser naive Ansatz soll dem Nachweis des Optimierungspotentials einer gänzlich zufälligen Auswahl dienen.

7.2.1.1 Definition der Simulationsreihe

Im Anschluss erfolgt die Definition der ersten Simulationsreihe:

- **Simulationsreihe:** Modularität bei skalenfreien Netzwerken (alternativ „Scale-Free Modularity“)
- **Nummer:** 1
- **Abhängigkeit:** /
- **Zweck:** Vergleich Baseline mit WeightedFactor (Faktor 1.5) in Abhängigkeit der Modularität
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen Baseline (A) und WeightedFactor (B) auf jeweils gleichen Graphen
  - Insgesamt 100 verschiedene Graphen: Ausführen auf je 20 Graphen mit gleichem Modularitätsniveau
  - Modularitätswerte aus $[0.1, 0.3, 0.5, 0.7, 0.9]$
  - Ausführen von 10 Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Pro Algorithmus: 1000 Ergebnisse, 200 pro verschiedenem Modularitätsniveau
- **Annahme:** Je höher die Modularität des Netzwerkes ist, desto besser wird die Leistung des WeightedFactor-Algorithmus im Vergleich zum Baseline-Verfahren. Er benötigt immer weniger Zeit zum Erreichen der Konvergenz. Eine Performancesteigerung sollte jedoch erst ab Modularitätswerten über 0.5 erkennbar sein.

Die Simulation beinhaltet skalenfreie Graphen, die jeweils verschiedene Modularitätsniveaus ($[0.1, 0.3, 0.5, 0.7, 0.9]$) aufweisen. Graphen mit verschiedenen Werte der Modularität wurden dabei durch unterschiedliche Erzeugungsverfahren und Parametrisierungen generiert. Mit dem C/MxN-Verfahren des Grapherzeugungstools konnten dann aus vielen erzeugten Graphen, die mit Modularitätswerten möglichst nah am Zielwert selektiert werden. Das Verfahren ist detailliert in den Kapiteln 5.2.1.2 und 6.3 beschrieben. Dabei wurden die Graphen für jedes Modularitätsniveau einzeln erzeugt und pro Niveau erfolgte eine Auswahl von 20 aus 500 Graphen. Das heißt, für die 500 Graphen im Auswahlpool wurden jeweils 10 Graphen mit 50 verschiedenen Parametersätzen generiert (nach C/MxN somit *20/50x10*). Diese Erzeugungsroutine bietet den Vorteil, dass lediglich

die Angabe eines Start- und Endwerts erforderlich ist, anstatt jeweils 50 Werte für alle Parameter anzugeben. Zwischenwerte werden dann zufällig aus den gebildeten Intervallen ausgewählt. Auf diese Weise entstehen variable Graphen, die einen hohen Grad an Zufälligkeit aufweisen. Die Zielmodularität wird approximiert, indem die besten Graphen aus der Menge der 500 erzeugten Graphen ausgewählt werden. Exemplarische Darstellungen von Graphen der unterschiedlichen Niveaus sind in Abbildung 7.1 zu sehen. Hier können die Communities, Hubs und Verzweigungen erkannt werden. Es folgt nun die Darstellung der verwendeten Verfahren und Konfigurationen für die einzelnen Modularitätsniveaus:

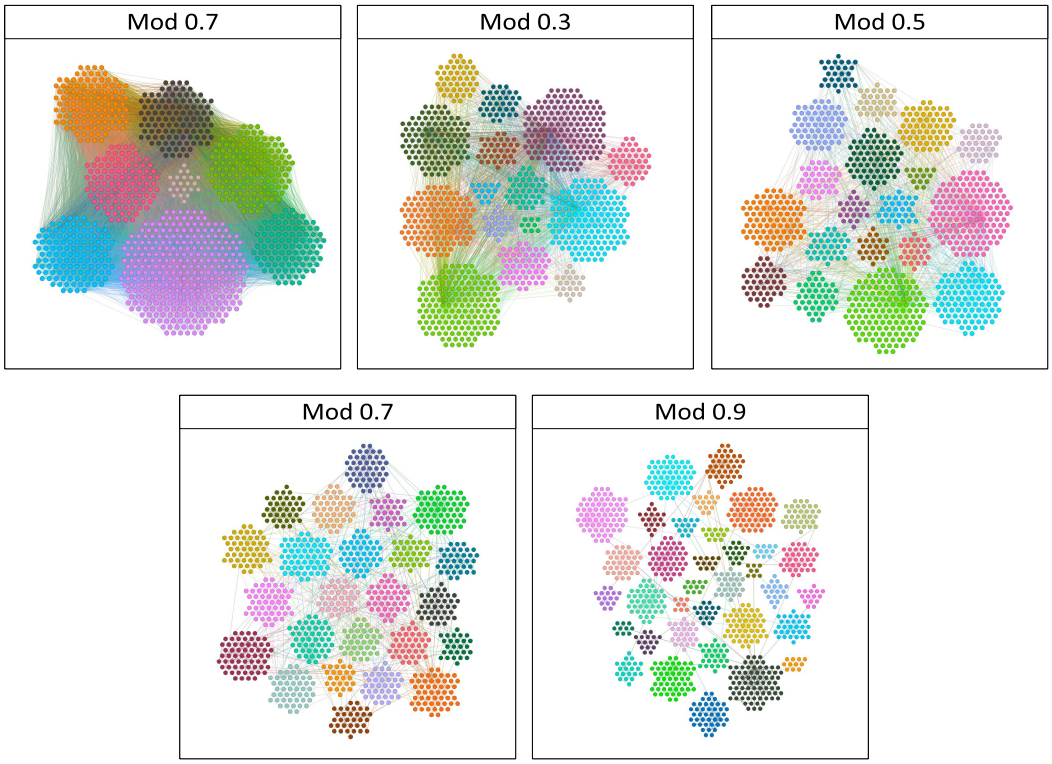


Abbildung 7.1: Beispielgraphen Simulationsreihe 1

- **0.1:** Die Erzeugung von stark unmodularen Graphen erfolgt mit der skalenfreien Erzeugungsroutine. Der Algorithmus wird mit hohen Werten für den Parameter β, sowie kleinen Werte für die Parameter α und γ ausgeführt. So entstehen Graphen ohne Community-Strukturen, die einen hohen Grad an Zufälligkeit aufweisen. Ebenso wird eine große Anzahl von Kanten (ca. 20.000) benötigt, um eine solch niedrige Modularität zu erreichen. Aufgrund der mathematischen Formel zur Berechnung der Modularität haben Graphen mit wenigen Kanten höhere Modularitätswerte, selbst bei Abwesenheit von Community-Strukturen. Eine hohe Anzahl von Kanten ermöglicht es erst, Modularitätswerte im Bereich von 0.1 zu erhalten.
- **0.3:** Hier wird ebenfalls die skalenfreie Erzeugungsroutine angewandt, jedoch mit reduzierter Anzahl von Kanten (ca. 3.000). Es werden weiterhin sehr hohe Werte für β verwendet, um eine Abwesenheit von Community-Strukturen zu gewährleisten. Der Algorithmus wird mit geringfügig höheren Werten für α und γ, sowie kleineren Werten für β ausgeführt. Infolgedessen erfolgt die Erzeugung von Graphen mit einer niedrigen Modularität von 0.3.

- **0.5:** Auch hier wird die skalenfreie Erzeugungsroutine verwendet. Dabei werden Graphen mit einer noch geringeren Anzahl von Kanten (ca. 1.800) erzeugt. Ebenso führen geringere Werte für den Parameter β und entsprechend höhere Werte für α dazu, dass schwache Community-Strukturen entstehen. So können Graphen mit mittlerer Modularität generiert werden.
- **0.7:** Die Graphen dieses Modularitätsniveaus werden mittels des Barabási-Albert-Modells erzeugt. Es zeichnet sich durch das Preferential Attachment aus, bei dem Knoten mit hoher Gradzahl eher neue Verbindungen erhalten. In jedem Schritt wird ein neuer Knoten sowie eine variable Anzahl an Kanten zu bestehenden Knoten (Werte zwischen 1 und 2) hinzugefügt. Auf diese Weise entstehen Graphen mit einer höheren Modularität.
- **0.9:** Schließlich wird die Grapherzeugung für die hochmodularen Graphen mit dem Holme-Kim-Modell durchgeführt. Mit dem Barabási-Albert-Modell können in der Regel nur schwer Modularitätswerte von über 0.9 erreicht werden. Das Holme-Kim-Modell hingegen wird zur Erzeugung von Graphen mit einer äußerst hohen Modularität angewandt. Es erweitert das Barabási-Albert-Modell, indem in jedem Schritt zusätzliche Dreieckskanten hinzufügt werden. Dies führt zu einer noch stärkeren Verbindung innerhalb der Communities, was wiederum zu einer erhöhten Modularität führt.

7.2.1.2 Ergebnisse

Im Anhang D.1.1 sind die gemittelten Graphmetriken zu finden. Diese sind nach Modularität aufgeteilt und ermöglichen so eine Einschätzung der Eigenschaften der verschiedenen Graphtypen. Des Weiteren erfolgt in Kapitel D.1.2 die Visualisierung der ermittelten Messwerte. Auch diese sind nach den jeweiligen Modularitätsniveaus gruppiert. Eine kompakte Darstellung ist in Abbildung 7.2 zu sehen. Das Säulendiagramm zeigt einen direkten Vergleich der Konvergenzzeit je nach Modularitätsniveau. Es ermöglicht, die Performance des Baseline- und des WeightedFactor-Algorithmus genauer nachzuvollziehen. Dabei ist zu erkennen, wie oft die Simulationen welche Anzahl an Runden zur Konvergenz benötigt haben. Die niedrig modularen Netzwerke mit Modularität 0.1 und 0.3 konvergieren bereits nach sehr wenigen Runden (meist unter sechs). Je höher die Modularität wird, desto größer wird auch die Anzahl an benötigten Runden. Für eine Modularität von 0.5 ist die durchschnittliche Rundenzahl im Bereich zwischen acht und neun. Bei den mittelmodularen Graphen (Modularität 0.7) werden bereits über zehn Runden im Durchschnitt benötigt. Die Rundenzahl steigt für die hochmodularen Graphen (Modularität 0.9) bis auf über vierzig. Hierbei erreicht der WeightedFactor-Algorithmus für die mittel- und hochmodularen Graphen eine erkennbare Verkürzung der Anzahl an benötigten Runden. Für die Graphen mit Modularitäten $\leq$ 0.5 können weniger als 4% der Runden eingespart werden. Erreicht die Modularität Werte von 0.7, so wird eine Optimierung von 4.7% erzielt. Die hochmodularen Graphen erreichen die beste Performance. Hier konvergiert das Gossiping durch die Gewichtung fast 16% schneller. Über alle Datenpunkte beschleunigt der naive WeightedFactor-Algorithmus das Gossiping um 11% gegenüber dem Baseline-Verfahren.

7.2.1.3 Schlussfolgerungen

Die Simulationsergebnisse bestätigen die anfänglichen Annahmen. Es kann eine klare Korrelation zwischen Modularität und Konvergenzzeit festgestellt werden. Das Gossiping konvergiert auf unmodularen Graphen in wenigen Runden. Dies ist auf die hohe

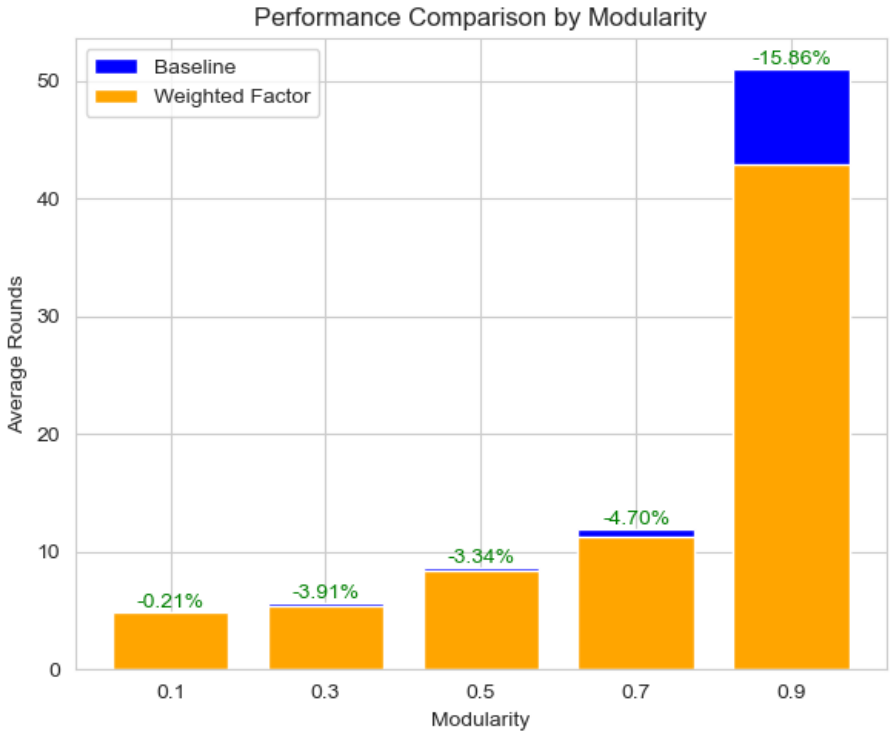


Abbildung 7.2: Auswertung Simulationsreihe 1

Kantenanzahl und die daraus resultierende starke Konnektivität zurückzuführen. Die niedrigen durchschnittlichen Pfadlängen und die geringe Standardabweichung zeigen, dass die meisten Knoten von jedem beliebigen Knoten aus schnell erreichbar sind. Diese Eigenschaften stellen für die Informationsausbreitung eine optimale Ausgangslage dar. Je höher die Modularität wird, desto schlechter sind die Konditionen für das Gossiping. Insbesondere bei den hochmodularen Graphen wird die Leistung sehr schlecht, da viele Knoten entlegen sind. Wenn es lange Pfade zwischen den Knoten gibt, wird eine erhebliche Zeit benötigt, bis sich ein globales Minimum über das gesamte Netzwerk verteilt hat. Auch von Bedeutung für die Informationsverbreitung sind die sogenannten Hubs, die in skalenfreien Netzwerken vorkommen und über viele Verbindungen verfügen. Diese können je nach Modularität entweder für eine schnelle Konvergenz sorgen oder diese blockieren. Grund dafür ist, dass periphere Knoten oft mit den angrenzenden Hub-Knoten interagieren. Miteinander verbundene Hubs tauschen sich jedoch verhältnismäßig selten aus. Beide haben viele Kanten, sodass bei einer zufälligen Auswahl viel Zeit vergehen kann, bis es zu einer Interaktion kommt. Aufgrund der Tatsache, dass Communities meist einzelne oder mehrere Hubs und deren periphere Knoten umfassen, kann selbst durch naive Bevorzugung eine bessere Kommunikation ermöglicht werden. Durch die Präferenz bei Auswahl der Knoten anderer Communities wird häufiger mit fremden Hub-Knoten interagiert und somit die Konvergenz beschleunigt.

Eine Verbesserung wird hierbei ab einer Modularität von über 0.5 erkennbar. Diese beläuft sich bei mittelmodularen Graphen auf fast 5% und bei hochmodularen auf nahezu 16%. Verglichen mit den anfänglichen Annahmen, fiel die Performancesteigerung geringer aus als erwartet. Insbesondere bei mittelmodularen Graphen wurde ein größeres Optimierungspotential angenommen. Der naive Ansatz kann hierbei als Grund für die geringe Ausnutzung des Potentials gesehen werden. Eine Gewichtung aller Knoten anderer Communities mit einem Faktor von 1.5 ist statisch und daher sehr unflexibel. Es konnte

jedoch anhand der Simulationsreihe nachgewiesen werden, dass eine Verbesserung des Gossipings auf modularen skalenfreien Graphen durch die Inkorporation von Wissen über Community-Strukturen möglich ist. In folgenden Simulationsreihen wurde daher geprüft, ob durch komplexere Verfahren oder andere Faktoren eine bessere Leistungssteigerung erzielt werden kann.

7.2.2 Konnektivität bei skalenfreien Netzwerken

Auch in dieser Simulationsreihe sollen skalenfreie Netzwerke untersucht werden. Diesmal wird jedoch nicht die Modularität betrachtet, sondern die Konnektivität. Bei der regulären Erzeugungsroutine nach Barabási-Albert ist der einzige anpassbare Parameter die Anzahl der hinzugefügten Kanten pro Schritt. Dieser hat jedoch einen direkten Einfluss auf die Konnektivität der generierten Graphen. Die vorangehende Simulationsreihe hat indirekt gezeigt, dass mit steigender Konnektivität die Konvergenz beschleunigt wird. Anhand dieser Simulationsreihe soll diese Korrelation nachgewiesen werden. Dazu wird die Konvergenzzeit auf skalenfreien Graphen mit unterschiedlichen Kantenzahlen untersucht. Zum Nachweis des Optimierungspotentials von spärlich verbundenen skalenfreien Graphen wertet auch diese Testreihe das Baseline- und das naive WeightedFactor-Verfahren aus.

7.2.2.1 Definition der Simulationsreihe

Im Anschluss wird die Definition der zweiten Simulationsreihe präsentiert:

- **Simulationsreihe:** Konnektivität bei skalenfreien Netzwerken (alternativ „Scale-Free New Edges“)
- **Nummer:** 2
- **Abhängigkeit:** /
- **Zweck:** Vergleich Baseline mit WeightedFactor (Faktor 1.5) in Abhängigkeit der Konnektivität durch den Parameter für die Anzahl neuer Kanten
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen Baseline (A) und WeightedFactor (B) auf jeweils gleichen Graphen
  - Insgesamt 140 Graphen: Ausführen auf je 20 Graphen mit unterschiedlicher Anzahl neuer Kanten
  - Werte für Anzahl neuer Kanten aus $[1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]$
  - Ausführen von 3 Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Pro Reihe: 420 Ergebnisse, 60 pro verschiedenem Kantengrad
- **Annahme:** Die Konvergenzzeit des Baseline- und des WeightedFactor-Algorithmus sinkt bei Zunahme der Kantenanzahl. Dabei sollte der WeightedFactor-Algorithmus nur bei niedriger Kantenanzahl eine signifikante Performancesteigerung erreichen. Nur hier verfügen die erzeugten Graphen über eine schwache Konnektivität und somit längere Pfade und entlegene Knoten.

Zur Erzeugung der skalenfreien Netzwerke mit variabler Konnektivität wird das Bárabasi-Albert-Verfahren verwendet. Konkret wird das MxN-Verfahren des CLI-Tools eingesetzt. In den Kapiteln 5.2.1.2 und 6.3 ist eine detaillierte Beschreibung der Routine zu finden. Mit dem Verfahren erfolgt die Erzeugung von 20 Graphen pro Parametersatz (nach MxN somit *7x20*). Für die verschiedenen Parameter werden die Start- und Endwerte abgerufen, welche zur Interpolation der Zwischenwerte verwendet werden. Hierbei ist die Anzahl neuer Kanten der einzige variable Parameter und nimmt Werte zwischen 1.0 und 4.0 an. Es werden also für die sieben verschiedenen Kantenzahlen je zwanzig unterschiedliche Graphen generiert.

7.2.2.2 Ergebnisse

Im Anhang unter D.2.1 sind die gemittelten Graphmetriken ausgelagert, welche eine Bewertung der Grapheigenschaften ermöglichen. Ebenfalls sind im Kapitel D.2.2 die genauen Messergebnisse der zweiten Simulationsreihe zu finden. Diese sind nach der Anzahl an hinzugefügten neuen Kanten unterteilt. In Abbildung 7.3 sind diese Ergebnisse zudem zusammengefasst. Das Säulendiagramm zeigt einen direkten Vergleich der Konvergenzzeit je nach Anzahl hinzugefügter Kanten. Die Leistung des Baseline- und WeightedFactor-Algorithmus werden in Abhängigkeit der Konnektivität der Graphen dargestellt. Bei einer hinzugefügten Kante werden für beide Verfahren im Durchschnitt über vierzig Runden benötigt. Dieser Wert sinkt bei 1.5 neuen Kanten auf etwa zehn und bei 2 liegt er zwischen sieben und acht. Die Rundenzahl fällt weiter, bis im Durchschnitt nur noch sechs Runden benötigt werden (bei 4 neuen Kanten). Der WeightedFactor-Algorithmus erreicht nur für die Graphen mit nur einer neuen Kante pro Schritt eine signifikante Verkürzung der Anzahl an benötigten Runden. Es ist erkennbar, dass der WeightedFactor-Algorithmus bei Graphen mit einer Anzahl von nur einer neuen Kante pro Schritt um 24% schneller konvergiert als das Standard-Verfahren. Die Beschleunigung ist in allen anderen Simulationen vernachlässigbar. Selbst bei einer Anzahl von 1.5 Kanten, die pro Schritt hinzugefügt werden, ist die Beschleunigung bereits auf 3% zurückgegangen. Als Ausnahme ist hier die Messung mit hinzugefügten 2.5 Kanten anzumerken, wo über 5% der Runden eingespart werden. Für höhere Kantenzahlen geht die Abweichung vom Standardverfahren auf Werte unter 1% zurück. Über alle Datenpunkte erreicht der WeightedFactor-Algorithmus eine Beschleunigung um 14% gegenüber dem Baseline-Verfahren.

7.2.2.3 Schlussfolgerungen

Die Tatsache, dass eine starke Beschleunigung bei geringer Konnektivität (genau eine neue Kante) erreicht wird, ist ein zufriedenstellendes Ergebnis und entspricht den Annahmen. Eine generelle Beschleunigung der Konvergenz und daraus folgende Reduktion des Optimierungspotentials mit zunehmender Anzahl an Kanten wurde ebenfalls vorhergesagt. Es wurde aber nicht angenommen, dass diese so schnell eintritt und bereits bei 1.5 Kanten das Optimierungspotential so stark reduziert ist. Die Beschleunigung ist in allen Simulationen außer bei genau einer hinzugefügter Kante vernachlässigbar. Das Ergebnis lässt sich jedoch durch die starke Reduktion der allgemeinen Konvergenzzeit in Folge der höheren Konnektivität erklären. Während bei einer Kante der durchschnittliche Knotengrad bei zwei liegt, ist dieser bei 1.5 Kanten bereits bei drei. Das Gossiping benötigt statt über vierzig Runden nur noch etwa zehn. Für durchschnittliche Knotengrade über vier verringert sich die Anzahl der erforderlichen Runden weiter, was gleichzeitig das Verbesserungspotential mindert. Man kann zusammenfassen, dass durch eine höhere Konnektivität die Konvergenz stark beschleunigt wird. In den meisten Fällen verfügen Knoten über zwei oder mehr Kanten und sind oft auch mit einem oder mehreren Hub-Knoten

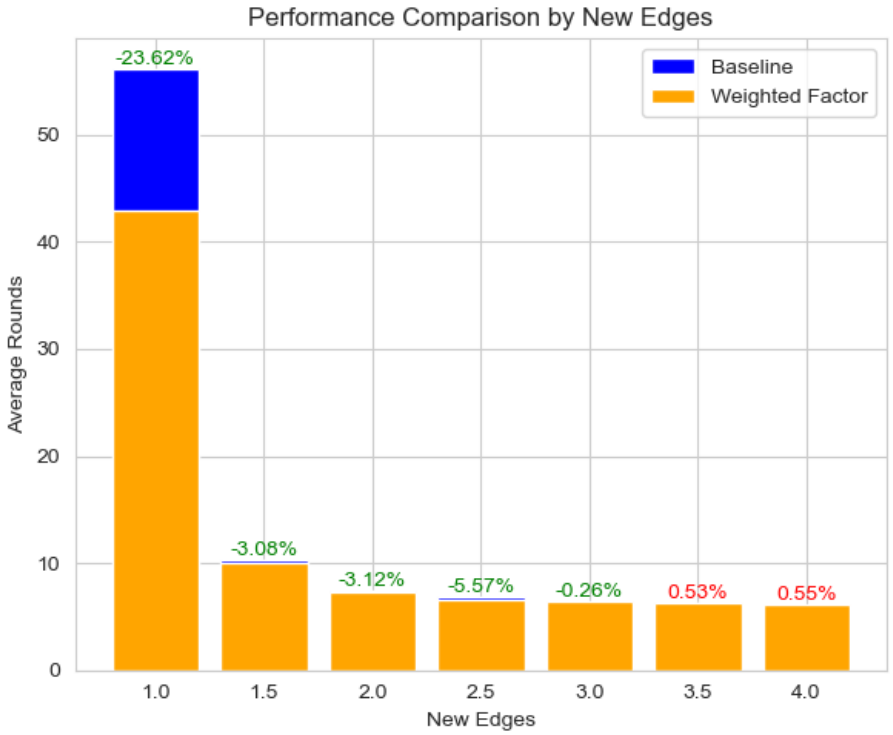


Abbildung 7.3: Auswertung Simulationsreihe 2

verbunden. Diese hochgradigen Knoten spielen eine zentrale Rolle in der Informationsverbreitung. Grund dafür ist, dass sie beim Gossiping oft als Kommunikationspartner ausgewählt werden. Die Zunahme der Kantenanzahl führt außerdem zur Entstehung von Hubs mit noch höheren Knotengraden, was ihre Bedeutung weiter steigert. Infolgedessen wird die Informationsverbreitung deutlich beschleunigt.

7.2.3 Verhältnis von Inter- und Intra-Community-Kanten bei Popularitätsnetzwerken

Popularitätsnetzwerke haben unterschiedliche Eigenschaften je nachdem, wie das Verhältnis von Intra- zu Inter-Community-Kanten ist. Sie können schwach und stark ausgeprägte Community-Strukturen aufweisen. Wenn die Knoten stark innerhalb und schwach außerhalb der Communities verbunden sind, so ist das Clustering hoch. Gleichermaßen nimmt es ab, je mehr sich das Verhältnis zwischen den Intra- zu Inter-Community-Kanten umkehrt. Im Gegensatz zu den zuvor untersuchten Netzwerken, haben die Popularitätsnetzwerke keine Powerlaw-Gradverteilung. Aus diesem Grund stellen sie eine sinnvolle Alternative zu den skalenfreien Netzwerken dar. Dementsprechend kann anhand dieser Simulationsreihe eine Untersuchung des Optimierungspotentials für andere Anwendungsfälle erfolgen. Ebenfalls sind die Popularitätsnetzwerke relevant, da sie in vorangehenden Arbeiten zur dezentralen Community Detection evaluiert wurden. Hier stellten sie sich zur Ausführung von Synchronisationsalgorithmen als sehr performant heraus [106] [105]. Ob und inwiefern ein gewichtetes Gossiping auf ihnen zur Leistungssteigerung eingesetzt werden kann, wird im Folgenden erörtert. Dabei wird erneut ein Vergleich zwischen dem Baseline- und dem WeightedFactor-Algorithmus gezogen.

7.2.3.1 Definition der Simulationsreihe

Die Simulationsreihe zu den Popularitätsnetzwerken ist nach folgender Definition durchgeführt worden:

- **Simulationsreihe:** Verhältnis von Inter- und Intra-Community-Kanten bei Popularitätsnetzwerken (alternativ „Popularity Inter-/Intra-Community Edges“)
- **Nummer:** 3
- **Abhängigkeit:** /
- **Zweck:** Vergleich Baseline mit WeightedFactor (Faktor 1.5) in Abhängigkeit des Inter-/Intra-Community-Kanten-Verhältnis
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen Baseline (A) und WeightedFactor (B) auf jeweils gleichen Graphen
  - Insgesamt 240 verschiedene Graphen: Ausführen auf je 40 Graphen mit gleichem Verhältnis von Inter- und Intra-Community-Kanten
  - Verhältnis aus $[1000/1000, 800/1200, 600/1400, 400/1600, 200/1800, 10/1990]$
  - Ausführen von 10 Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Pro Algorithmus: 2400 Ergebnisse, 240 pro verschiedenem Verhältnis
- **Annahme:** Je höher das Verhältnis zwischen Inter- und Intra-Community-Kanten wird, desto besser wird die Leistung des WeightedFactor-Algorithmus im Vergleich zum Baseline-Verfahren. Der WeightedFactor-Algorithmus sollte die Konvergenz immer schneller erreichen. Eine signifikante Performancesteigerung sollte jedoch erst erkennbar sein, wenn nur noch weniger als zehn Prozent der Kanten Inter-Community-Kanten sind (also bei 200/1800).

Die Grapherzeugung erfolgt mit dem im späteren Verlauf hinzugefügten Verfahren nach Singh [105]. Als Parameter werden hier die Knoten- und Community-Anzahl sowie der Inter- und Intra-Kanten abgefragt. Im Kapitel 6.3 ist eine detaillierte Beschreibung der Routine zu finden. Konkret werden die Graphen mit jeweils zehn Communities und den verschiedenen Inter-/Intra-Community-Kanten-Verhältnissen generiert. Alle Communities bekommen hierbei etwa gleich viele Knoten zugeteilt. Für jedes Verhältnis erfolgt die Erstellung von 40 Graphen mit dem MxN-Verfahren (siehe Kapitel 5.2.1.2). Somit werden für die sechs verschiedenen Verhältnisse je zwanzig unterschiedliche Graphen erzeugt.

7.2.3.2 Ergebnisse

Im Anhang im Kapitel D.3.1 sind die gemittelten Graphmetriken aufgelistet. Diese stellen die Grapheigenschaften in Abhängigkeit des Verhältnisses zwischen Inter- und Intra-Community-Kanten dar. Des Weiteren können detaillierte Messungen der dritten Simulationsreihe im Kapitel D.3.2 betrachtet werden. Hier werden die Ergebnisse je nach Verhältnis zwischen Inter- und Intra-Community-Kanten dargestellt. Die visualisierten Messwerte der Simulationen bilden die Leistung der Algorithmen für die einzelnen Simulationen in Abhängigkeit des Verhältnisses ab. In Abbildung 7.4 sind diese Ergebnisse zusammengefasst. Hier zeigt das Diagramm einen Vergleich zwischen dem Baseline- und dem

WeightedFactor-Algorithmus in Bezug auf die Konvergenzzeit. Wird die Anzahl an benötigten Runden betrachtet, so ist erkennbar, dass diese für alle Verhältnisse außer 10/1990 fast gleich ist. Die Anzahl der für die Konvergenz erforderlichen Runden liegt knapp über neun Runden für das Inter-/Intra-Community-Verhältnis von 1000/1000. Eine geringfügig langsamere Konvergenz wird für alle Verhältnisse bis zu 200/1800 festgestellt. Hier überschreitet die Anzahl an Runden erstmals den Wert zehn. Das Verhältnis von 10/1990 stellt einen Extremfall dar, bei dem mehr als 20 Runden für die Konvergenz erforderlich sind. Beim Vergleich der Leistung der Baseline- und WeightedFactor-Algorithmen fällt auf, dass diese auf allen betrachteten Graphen nahezu identische Leistungen aufweisen. Auch hier stellt das Verhältnis 10/1990 von Inter- zu Inter-Community-Kanten eine Ausnahme dar. Eine Verbesserung um 5% im Vergleich zum Baseline-Verfahren ist zu beobachten. Auf Graphen mit einem Verhältnis von 200/1800 sowie 400/1600 ist das WeightedFactor-Verfahren minimal schneller als das Standardverfahren. Mit steigender Zahl an Inter-Community-Kanten nimmt die Performancesteigerung weiter ab. Der Baseline-Algorithmus benötigt für die anderen drei Verhältnisse (600/1400, 800/1200 und 1000/1000) sogar weniger Runden bis zur Konvergenz. Die Leistungsunterschiede sind für alle Verhältnisse außer 10/1990 praktisch vernachlässigbar, da sie unter 2% liegen. Über alle Datenpunkte verringert der WeightedFactor-Algorithmus die benötigten Runden um 1% gegenüber dem Baseline-Verfahren.

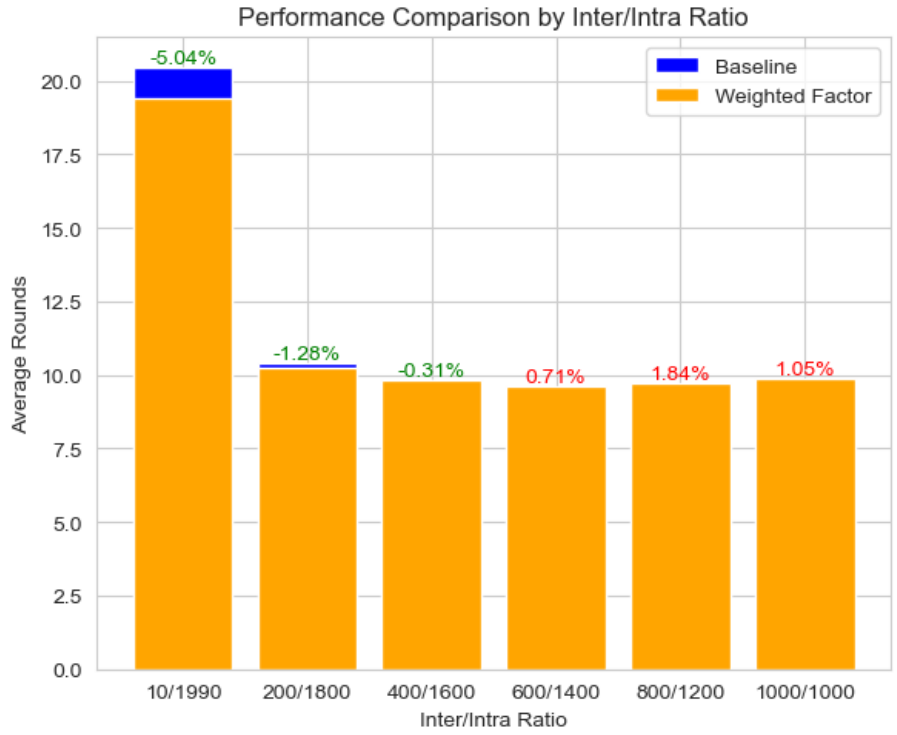


Abbildung 7.4: Auswertung Simulationsreihe 3

7.2.3.3 Schlussfolgerungen

Wie erwartet ist die Performancesteigerung für das kleinste Verhältnis von Inter- zu Intra-Community-Kanten am größten. Jedoch fällt selbst das Optimierungspotential dieser Graphen sehr gering aus. Der naive WeightedFactor-Algorithmus erreicht hier nur eine Steigerung von 5%. Für die anderen Verhältnisse war keine wesentliche Verbesserung erkennbar. Das Ergebnis ist somit wenig überzeugend im Vergleich zu den vorangehenden

Testreihen. Die geringe Performancesteigerung folgt aus der ohnehin schnellen Konvergenz. Diese wiederum lässt sich durch die Funktionsweise des Algorithmus und dem Aufbau der generierten Graphen erklären. Die Grapherzeugung verteilt die Knoten und Kanten gleichmäßig im Netzwerk. Dies führt dazu, dass es keine zentralen Hubs gibt. Das heißt, es existieren keine Knoten mit einer überdurchschnittlich hohen Anzahl von Kanten. Stattdessen haben alle Knoten ähnlich viele Verbindungen. Ebenfalls sind die durchschnittlichen Pfadlängen zwischen den Knoten im Netzwerk sehr klein. Dies bedeutet, dass die meisten Knoten direkt oder über eine kurze Anzahl von Zwischenschritten miteinander verbunden sind. Somit existieren keine entlegenen Knoten, die weit entfernt von anderen Knoten liegen. Als Folge benötigt selbst der Baseline-Algorithmus im Extremfall nicht außerordentlich lange, bis eine Konvergenz erreicht ist. Die Simulationsreihe verdeutlicht, dass ein Netzwerk über längere Pfade und Hub-Knoten verfügen muss, um ein ausreichendes Optimierungspotential zu bieten.

7.2.4 Gewichtete Algorithmen auf hochmodularen skalenfreien Netzwerken

Nach den ersten drei grundlegenden Simulationsreihen wurde eine Evaluation durchgeführt. Die Erkenntnisse zeigten, dass die hochmodularen skalenfreien Netzwerke das größte Optimierungspotential besitzen. Zudem weisen diese viele Eigenschaften realer Netzwerke auf, wodurch Erkenntnisse eine große Praxisrelevanz haben. Aus diesen Gründen wurde sich für die weiteren Untersuchungen auf hochmodulare skalenfreie Netzwerke beschränkt. Die nachfolgende Simulationsreihe war die erste einer Folge von Untersuchungen, welche spezifisch auf diese Netze ausgerichtet wurden. Ziel war es hierbei, einen möglichst optimalen gewichteten Algorithmus zu finden. Dabei werden verschiedene Faktoren sowie der alternative CommunityProbabilities-Algorithmus untersucht. Dieser wurde im Kapitel 5.2.4 vorgestellt und führt eine dynamische Verteilung von Gewichten durch. Hier erfolgt die Bestimmung der Gewichte der Nachbarknoten nach der invertierten Wahrscheinlichkeit, Teil der eigenen Community zu sein.

7.2.4.1 Definition der Simulationsreihe

Es folgt die Definition der fortgeschrittenen Simulationsreihe zu den hochmodularen skalenfreien Netzwerken:

- **Simulationsreihe:** Gewichtete Algorithmen auf hochmodularen skalenfreien Netzwerken (alternativ „High-Mod Scale-Free“)
- **Nummer:** 4
- **Abhängigkeit:** 1, 2, 3
- **Zweck:** Vergleich Baseline- mit WeightedFactor-Algorithmus (verschiedene Faktorwerte) und CommunityProbabilities auf hochmodularen Graphen
- **Aufbau:**
  - Drei Simulationsressourcen: Ausführen der verschiedenen Algorithmen Baseline (A), WeightedFactor (B) und CommunityProbabilities (C) auf jeweils gleichen Graphen
  - Insgesamt 160 verschiedene Graphen mit unterschiedlichen Modularitätswerten zwischen $[0.65, 0.95]$
  - Für B Faktoren aus $[1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3]$

  - Ausführen von 1 Wiederholung pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 1280 Ergebnisse, 160 für A und C sowie 960 für B (160 für jeden Faktor)

- **Annahme:** Der WeightedFactor-Algorithmus sollte für die hochmodularen Netzwerke eine deutliche Performancesteigerung gegenüber dem Baseline-Verfahren darstellen. Ebenfalls wird erwartet, dass das CommunityProbabilities-Verfahren die beste Leistung erzielt. Dabei wird von einer deutlichen Performancesteigerung gegenüber dem WeightedFactor-Algorithmus ausgegangen. Dieser führt eine naive und unflexible Verteilung der Gewichte bei Hub-Knoten mit unterschiedlichem Einflussgrad durch. Der CommunityProbabilities-Algorithmus hingegen weist dynamische Gewichte zu, wodurch diese Lücke geschlossen werden sollte.

Für die Simulationsreihe werden neue hochmodulare skalenfreie Netzwerke erzeugt. Dazu wird der Holme-Kim-Algorithmus verwendet, welcher im Kapitel 2.1.7 vorgestellt wird. Dieser nimmt als Parameter die Anzahl an Knoten, Communities und neuer Kanten sowie die Dreieckswahrscheinlichkeit entgegen. Konkret wird die MxN-Grapherzeugungsroutine (siehe Kapitel 5.2.1.2) eingesetzt. Dabei werden 80 verschiedene Parametersätze verwendet, um möglichst zufällige Graphen zu erstellen. Durch die Zufälligkeit wird sichergestellt, dass robuste Ergebnisse erzielt werden, die von spezifischen Muster oder Voraussetzungen in den Ausgangsdaten unbeeinflusst bleiben. Zufällige Graphen ermöglichen es, allgemeine Erkenntnisse und Trends abzuleiten, die auf eine Vielzahl von realen Szenarien anwendbar sind. Jeder der 80 Parametersätze wird zur Erzeugung von jeweils zwei Graphen verwendet (somit nach MxN *80x2*). Dabei werden alle Graphen mit zehn Communities generiert und sonstige Parameter zufällig aus den definierten Intervallen gewählt:

- Anzahl neue Kanten aus $[1.0, 2.0]$
- Wahrscheinlichkeit für Dreieckskanten aus $[0, 1.0]$

Die erzeugten Graphen weisen Modularitätswerte im Bereich von 0.65 bis 0.95 auf, während die durchschnittliche Modularität bei etwa 0.8 liegt. In Abbildung 7.5 ist die Verteilung der Modularitäten dargestellt. Durch die Wahl der Parameter wird bewusst die Erstellung von hochmodularen Graphen initiiert, wobei die Verteilung der Modularitätswerte trotzdem zufällig bleibt. Bei der Grapherzeugung werden mit 80 Parametersätzen insgesamt 160 Graphen erzeugt. Hierbei handelt es sich um eine kleine Referenzmenge, wodurch gelegentlich Ausreißer auftreten können. Ein Beispiel für einen solchen Ausreißer, ist die niedrige Anzahl von Graphen im Bereich $[0.785, 0.8]$.

7.2.4.2 Ergebnisse

Die Grapheigenschaften der hochmodularen skalenfreien Graphen sind gemittelt im Anhang D.4.1 ausgelagert. Diese können zum besseren Verständnis der untersuchten Netzwerke hinzugezogen werden. Im Kapitel D.4.2 sind zudem die detaillierten Messwerte zu sehen. Anhand der Visualisierung der Messwerte kann die Performance des Baseline- und CommunityProbabilities-Algorithmus sowie der WeightedFactor-Algorithmen mit verschiedenen Faktorwerten nachvollzogen werden. In der Regel bewegen sich die Rundenzahlen im Bereich von zehn bis zwanzig, wobei jedoch auch Werte von über vierzig vorkommen können. Abbildung 7.6 fasst die Ergebnisse kompakt zusammen. Hier wird ein direkter Vergleich der Konvergenzzeiten der gewichteten Algorithmen mit dem

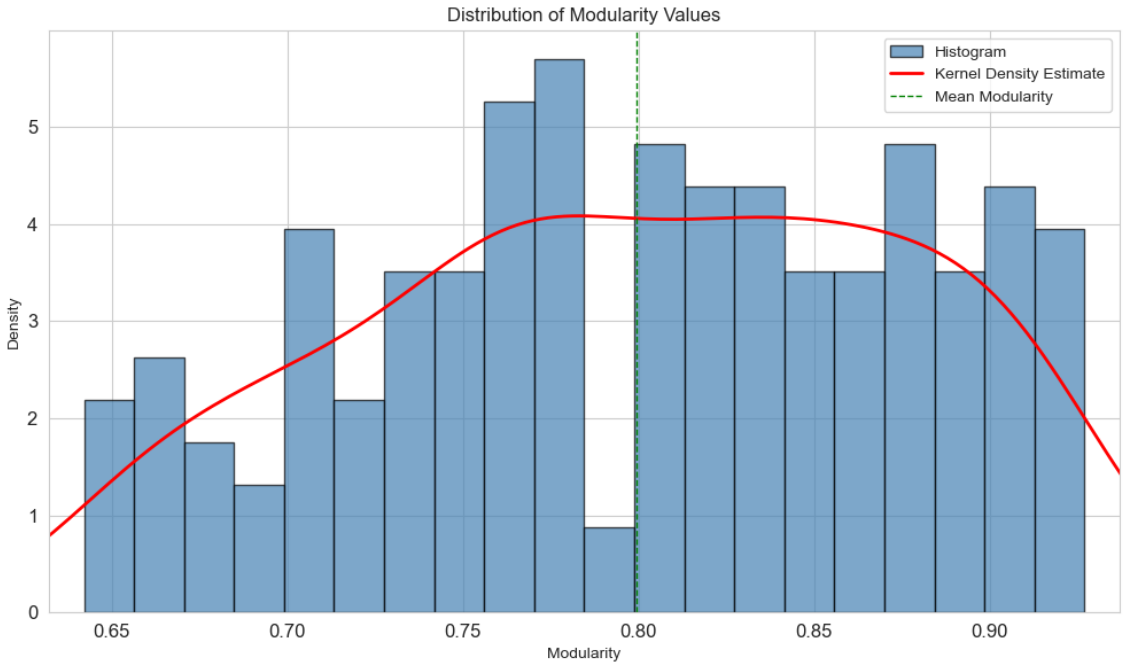


Abbildung 7.5: Verteilung der Modularitäten der hochmodularen skalenfreien Graphen

Baseline-Verfahren dargestellt. Die WeightedFactor-Algorithmen zeigen signifikante Verbesserungen in der Leistung. Es fällt auf, dass in der Regel eine höhere Gewichtung zu besseren Ergebnissen führt. Dieser Trend erstreckt sich von Anfang bis zu einem Gewichtungsfaktor von 2.75, bei dem der Algorithmus seine Spitzenleistung erreicht. Im Durchschnitt konvergiert er etwa 11% schneller als das Baseline-Verfahren. Bei einem Faktor von 3.0 gibt es jedoch einen Rückschritt, und die Beschleunigung beträgt nur noch 8%. Der CommunityProbabilities-Algorithmus erreicht mit Abstand das beste Ergebnis mit einer Verbesserung von über 23% gegenüber dem Baseline-Verfahren.

Anschließend wurde eine zusätzliche Analyse bezüglich der Modularität durchgeführt, um die Korrelation zwischen Leistung und Ausprägung von Community-Strukturen genauer zu untersuchen. Dazu wurden die Ergebnisse in die verschiedenen Modularitätsbereiche separiert. Die in Abbildung 7.5 dargestellte Distribution der Modularitätswerte zeigt, dass die Graphen nicht gleichmäßig verteilt sind. Für die Auswertung der einzelnen Modularitätsniveaus muss aber auch keine solche Gleichverteilung vorhanden sein, da die Niveaus individuell evaluiert werden und andere Ergebnisse unberücksichtigt bleiben. Die gruppierten Ergebnisse sind in Abbildung 7.7 visualisiert. Man kann erkennen, dass bei niedrigen Modularitäten die WeightedFactor-Algorithmen versagen. Insbesondere bei schwacher Modularität (Modularität 0.6 bis 0.7) wird keine Verbesserung erzielt. In den Bereichen 0.7 bis 0.8 und 0.8 bis 0.9 wird die Konvergenz moderat beschleunigt (bis zu neun Prozent je nach Faktor). Erst im Bereich 0.9 bis 1.0 erreichen die hohen Faktoren (2.75 und 3.0) sehr starke Optimierungen von über 20%. Hier erreicht der CommunityProbabilities-Algorithmus sogar eine Verbesserung von rund 30%. Er kann jedoch auch in weniger modularen Netzwerken signifikante Leistungssteigerungen erzielen. In den Bereichen 0.7 bis 0.8 und 0.8 bis 0.9 verringert sich die Konvergenzzeit um mehr als 20%. Sogar in den schwachmodularen Netzwerke im Bereich 0.6 bis 0.7 erreicht die Optimierung noch 19%.

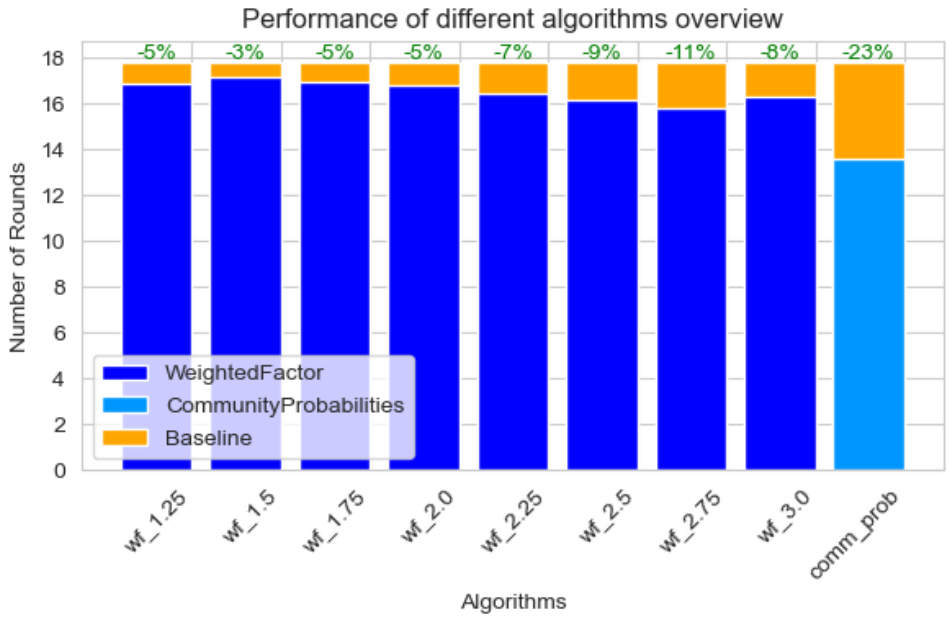


Abbildung 7.6: Auswertung Simulationsreihe 4

7.2.4.3 Schlussfolgerungen

Die Ergebnisse zeigen, dass der CommunityProbabilities-Algorithmus das Optimierungspotential der Netzwerke sehr gut ausschöpfen kann. Dem Algorithmus gelingt eine Optimierung der Informationsverbreitung selbst bei Netzwerken, wo der naive Ansatz versagt. Hierbei wurde eine Performancesteigerung gegenüber den WeightedFactor-Algorithmen erwartet. Die erzielten Ergebnisse waren jedoch besser als dies zu Beginn erwartet wurde. Eine Steigerung von bis zu 30% auf Netzwerken mit nur tausend Knoten stellt ein sehr zufriedenstellendes Ergebnis dar. Die starke Leistungssteigerung kann erklärt werden, indem die Gewichtung bei der Nachbarauswahl betrachtet wird. Der Algorithmus präferiert hierbei Knoten, die sehr unwahrscheinlich die gleiche Community haben. Dabei korreliert der Wahrscheinlichkeitswert mit der Anzahl an Nachbarn aus anderen Communities. Dadurch wählt der Algorithmus bevorzugt Nachbarn anderer Communities aus, die viele Kanten haben. Bei skalenfreien Netzwerken werden dementsprechend Hub-Knoten präferiert. Diese spielen unabhängig von der Modularität eine zentrale Rolle bei der Informationsverbreitung. Der Grund dafür ist, dass die Hubs viele Nachbarn besitzen, welche selbst wenig andere Verbindungen haben. Dementsprechend selektieren periphere Knoten beim Gossiping meist angrenzende Hub-Knoten. Hub-Knoten haben hingegen sehr viele mögliche Kommunikationspartner. Wird die Auswahl von Hubs bevorzugt, wie das bei dem CommunityProbabilities-Verfahren der Fall ist, so wird auch der Informationsaustausch zwischen Hubs beschleunigt. Infolgedessen wird ebenfalls bei wenig modularen Netzwerken eine Beschleunigung erreicht. Aufgrund der überzeugenden Performance des CommunityProbabilities-Algorithmus wurden weitere Simulationsreihen zu dessen Untersuchung durchgeführt. Dabei wurde der Algorithmus um eine Memory-Logik erweitert, um zu prüfen, ob eine weitere Leistungssteigerung erreicht werden kann. Zudem erfolgte die Entwicklung von fortgeschrittenen Algorithmen basierend auf den gewonnenen Erkenntnissen zur optimalen Ausschöpfung des Optimierungspotentials. Ziel war es hierbei, komplexere Alternativen zum CommunityProbabilities-Algorithmus zu schaffen.

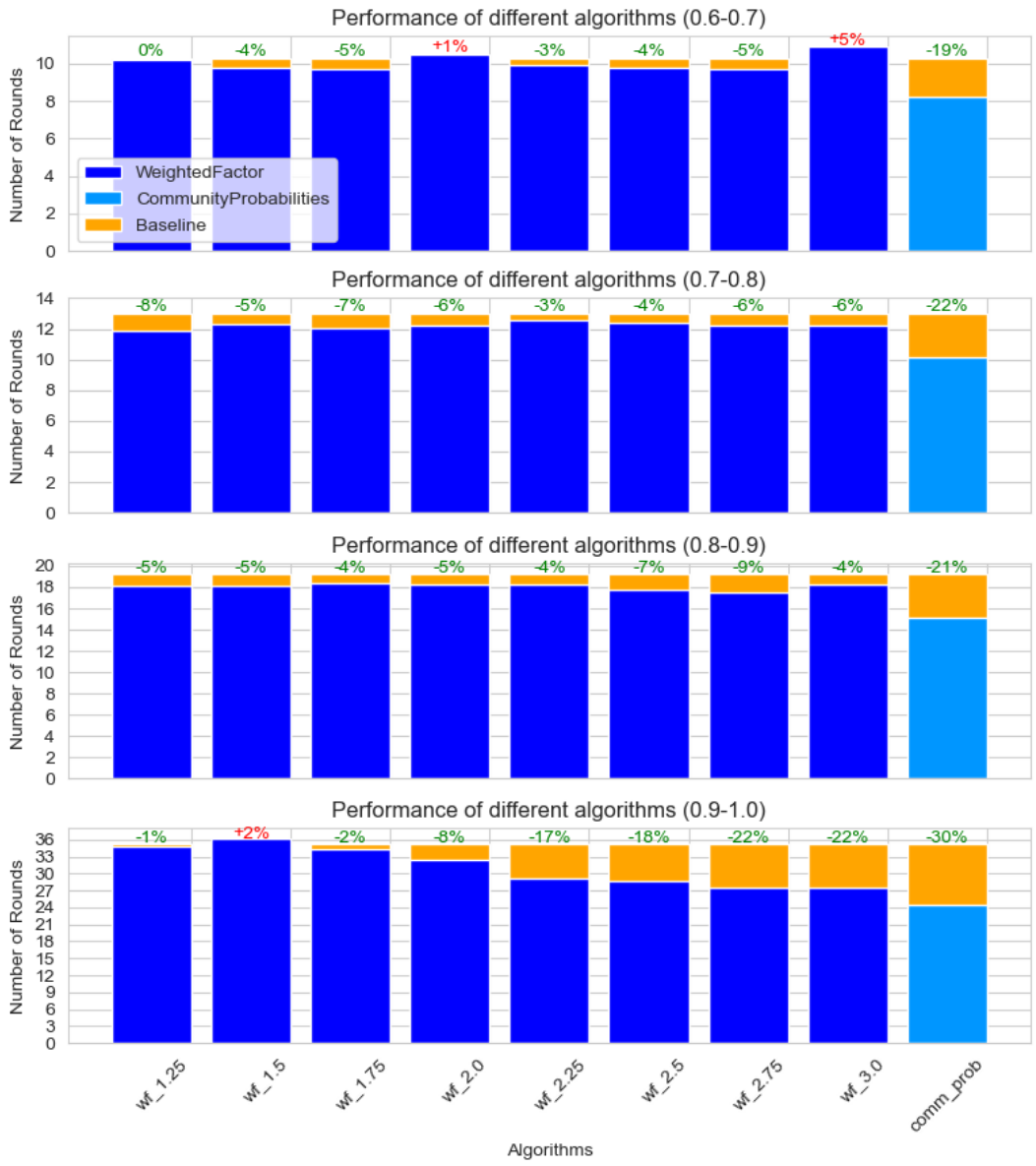


Abbildung 7.7: Auswertung nach Modularitätsbereichen Simulationsreihe 4

In den vorangehenden Testreihen wurde eine Schlussfolgerung über die Wichtigkeit von Pfadlängen und die Zentralität von Knoten gezogen. Es wurde erkannt, dass Metriken, welche diese Grapheigenschaften beschreiben, einen größeren Einfluss auf die Konvergenz haben, als die Community-Strukturen an sich. Diese These konnte durch die Aufstellung einer Korrelationsmatrix bestätigt werden. Eine solche Matrix zeigt, wie stark die Beziehung zwischen verschiedenen numerischen Variablen ist. Konkret werden die ermittelten Graphmetriken (siehe Kapitel 2.1.9) in Beziehung zur Anzahl der erforderlichen Runden gesetzt. Die Korrelationsmatrix stellt somit dar, wie sich Änderungen der Metriken auf die Konvergenzzeit auswirken. Die Werte liegen dabei zwischen -1 und 1, wobei ein Wert von 1 eine perfekte positive Korrelation anzeigt. Das bedeutet, dass die beiden Variablen zusammen in die gleiche Richtung variieren. Im Gegensatz dazu wird durch einen Wert von -1 eine perfekte negative Korrelation ausgedrückt, also erfolgen Wertänderungen in entgegengesetzte Richtungen. Ein Wert von 0 zeigt an, dass es keine Korrelation zwischen den Variablen gibt. Im Anhang D.4.2 in Abbildung D.6 ist die kom-

plette Korrelationsmatrix ausgelagert. Die wichtigsten Metriken sind in Abbildung 7.8 dargestellt. Bedeutend sind hierbei besonders folgende Korrelationen:

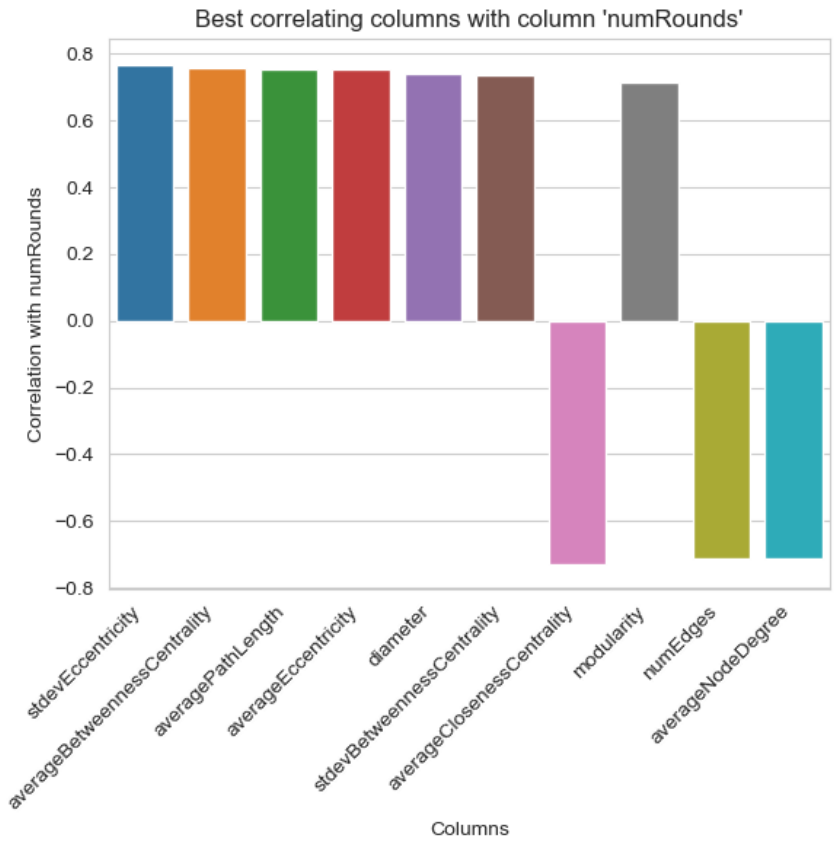


Abbildung 7.8: Einflussreichste Grapheigenschaften hochmodularer skalenfreier Graphen

- **Betweenness Centrality**: Als komplexeste Metrik für die Zentralität kapselt sie Informationen über kürzeste Pfadlängen. Zentrale Knoten, die auf vielen kürzesten Pfaden liegen, haben eine enorme Wichtigkeit für die Informationsausbreitung. Für die skalenfreien modularen Netze bedeutet ein höherer Wert der Metrik, dass mehr Hub-Knoten mit höheren Knotengraden existieren. Infolgedessen steigt die Anzahl an benötigten Runden, wodurch eine starke positive Korrelation zur Konvergenzzeit entsteht.
- **Path Length**: Pfadlängen bestimmen generell die Dauer des Gossipings in Graphen. Je länger die Pfade von Knoten sind, desto schwieriger wird eine Verbreitung von Daten im Netzwerk. Dementsprechend besteht eine starke positive Korrelation zwischen den durchschnittlichen Pfadlängen und der beobachteten Konvergenzzeit.
- **Diameter, Closeness und Eccentricity**: Diese Metriken bestätigen die oben genannten Zusammenhänge. Sie können alternativ für die Beschreibung von Nähe und Ferne von Knoten im Graphen verwendet werden. Auch diese Metriken korrelieren stark positiv mit der Konvergenzzeit.
- **Closeness Centrality**: Hierbei handelt es sich auch um eine Zentralitätsmetrik, welche die Nähe zwischen Knoten im Netzwerk ausdrückt. Sie korreliert stark negativ mit der Anzahl an benötigten Runden bis zur Konvergenz. Eine Informationsverbreitung wird durch entlegene Knoten verlangsamt. Graphen mit einer hohen Closeness Centrality haben wenige lange Pfade, wodurch die Informationsausbreitung

begünstigt wird. Dementsprechend führen hier niedrige Werte zu hohen Konvergenzzeiten.

Darüber hinaus bestätigt die Auswertung, dass die Modularität auch einen großen Einfluss auf das Konvergenzverhalten hat. Es kann eine starke positive Korrelation festgestellt werden, was den Beobachtungen der ersten Simulationsreihe entspricht (siehe Kapitel 7.2.1). Der Einfluss der Modularität fällt jedoch weniger stark aus als bei den zuvor beschriebenen Metriken. Trivialerweise wird auch eine starke Korrelation mit dem Knotengrad und der Kantenzahl beobachtet. Beide Metriken korrelieren stark negativ mit der Konvergenzzeit. Das heißt, je mehr Kanten die hochmodularen Graphen haben, desto weniger Runden werden benötigt. Diese negative Korrelation war ebenfalls bereits in den ersten beiden Simulationsreihe erkennbar (siehe Kapitel 7.2.1 und 7.2.2).

Im Rahmen dieser Testreihe wurden weitere Beobachtungen durchgeführt. Ziel war es, die Anwendbarkeit der Algorithmen zur Beschleunigung einer lokalen Konvergenz innerhalb der Communities zu evaluieren. Hierzu erfolgte die Durchführung von weiteren kleinen Simulationsreihen in der Testumgebung. Dabei wurden Gewichte zwischen 0 und 1 für den WeightedFactor-Algorithmus gewählt. Dadurch konnte bestätigt werden, dass so die Konvergenz in den Communities beschleunigt wird. Als logische Folge wird die globale Konvergenz gleichermaßen verzögert. Ist das primäre Ziel, dass Daten innerhalb der Communities verteilt werden, so ist diese Anwendung sinnvoll. Die stärkste lokale Konvergenz erreichte hierbei der modifizierte CommunityProbabilities-Algorithmus. Das Verfahren entspricht dem in Kapitel 5.2.4 dargestellten Vorgehen. Als einzige Modifikation wurde auf eine Invertierung der Gewichte nach Berechnungen der Wahrscheinlichkeiten verzichtet. Das heißt, die Gewichte wurden so verteilt, dass die Knoten priorisiert werden, die sich am wahrscheinlichsten in der gleichen Community befinden.

7.2.5 Verbesserung der gewichteten Algorithmen auf hochmodularen skalenfreien Netzwerken

Nach den Simulationsreihen zu den hochmodularen skalenfreien Netzwerken wurde eine Evaluation durchgeführt. Die Erkenntnisse zeigten, dass der CommunityProbabilities-Algorithmus die beste Performancesteigerung erreicht. In dieser Simulationsreihe wird er deshalb erneut ausgewertet. Dabei ist das Ziel zu prüfen, ob eine weitere Optimierung durch Einsatz einer Memory-Logik möglich ist. Weiterhin werden die initialen Gewichte mit dem CommunityProbabilities-Verfahren anhand der Netzwerktopologie gesetzt. Während der Ausführung führt die Memory-Logik eine Veränderung dieser Gewichte basierend auf den erfolgten Kommunikationen durch. Diese Adaptionen des Verfahrens kann den aktuellen Bedingungen besser gerecht werden.

7.2.5.1 Definition der Simulationsreihe

Im Anschluss folgt nun die Definition der fünften Simulationsreihe:

- **Simulationsreihe:** Verbesserung der gewichteten Algorithmen auf hochmodularen skalenfreien Netzwerken (alternativ „High-Mod Scale-Free Memory“)
- **Nummer:** 5
- **Abhängigkeit:** 4
- **Zweck:** Vergleich der vorangehenden Ergebnisse des CommunityProbabilities-Algorithmus mit den beiden Varianten CommunityProbabilitiesMemory und CommunityProbabilitiesComplexMemory jeweils mit verschiedenen Werten für den PriorPartnerFactor

- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen CommunityProbabilitiesMemory (A) und CommunityProbabilitiesComplexMemory (B) auf jeweils gleichen Graphen
  - Insgesamt 160 verschiedene Graphen mit unterschiedlichen Modularitätswerten zwischen $[0.65, 0.95]$
  - Für A und B Werte für PriorPartnerFactor in $[0.25, 0.5, 0.75]$
  - Ausführen von 1 Wiederholung pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 960 Ergebnisse: 160 Ergebnisse pro Faktor für A und B (somit in Summe je Testreihe 480)
- **Annahme:** Durch die Memory-Funktionalität wird eine zusätzliche Performancesteigerung erreicht. Dabei sollte die ComplexMemory-Logik durch die mehrfache Verringerung von Gewichten sowie das Vergessen bei Wertänderung, die beste Leistung erbringen.

7.2.5.2 Ergebnisse

Auch in dieser Simulationsreihe wurden die Algorithmen auf den hochmodularen skalenfreien Netzwerken ausgewertet. Dementsprechend sind die Graphmetriken analog zur vierten Messreihe (siehe Kapitel D.4.1). Im Anhang D.5.2 sind die detaillierten Messwerte ausgelagert. Anhand dieser kann die Performance des Baseline- und der unterschiedlichen Varianten des CommunityProbabilities-Algorithmus nachvollzogen werden. Dabei werden die benötigten Runden bis zum Erreichen der Konvergenz für die verschiedenen Verfahren dargestellt. Eine Zusammenfassung der Ergebnisse ist in Abbildung 7.9 zu finden. Hier werden die gemittelten Konvergenzzeiten der CommunityProbabilities-Algorithmen mit verschiedenen Gedächtnisvarianten zu dem Baseline-Verfahren verglichen. Dabei fällt auf, dass alle Algorithmen mit Speicherlogik eine bessere Performance als das normale CommunityProbabilities-Verfahren haben. Für das simple Gedächtnis erreichen hierbei die Simulationen mit PriorPartnerFactor 0.25 und 0.5 das beste Ergebnis. Während das Baseline-Verfahren im Schnitt fast 18 Runden bis zur Konvergenz benötigt, reichen hier weniger als 13 Runden aus. Dementsprechend wird 30% der Zeit eingespart und damit 7% mehr als bei dem normalen CommunityProbabilities-Verfahren. Allerdings erreicht die Variante mit einem Faktor von 0.75 sogar schlechtere Ergebnisse, mit einer Einsparung von etwa 24% im Vergleich zur Baseline. Ebenfalls kann man erkennen, dass das komplexe Gedächtnis die besten Ergebnisse erreicht. Auch hier führt eine stärkere Reduktion der Gewichte nach erfolgter Kommunikation zu besseren Resultaten. Mit Faktor 0.25 sinkt die Zahl der benötigten Runden auf etwa 7, was einer Optimierung von 59% entspricht. Währenddessen benötigen die Simulationen mit Faktor 0.5 im Schnitt über acht Runden. Dementsprechend wird dann nur noch 53% der Zeit eingespart. Das schlechteste Ergebnis liefert jedoch der Faktor 0.75. Hier nehmen Simulationen im Durchschnitt über neun Runden in Anspruch, was zu einer Einsparung von 46% führt.

7.2.5.3 Schlussfolgerungen

Mit dem CommunityProbabilities-Algorithmus konnte die Konvergenzzeit bereits stark reduziert werden. Konkret wurde durch den Einsatz der topologischen Informationen eine Beschleunigung von 23% erzielt. Im Vergleich dazu wird mit der Memory-Logik

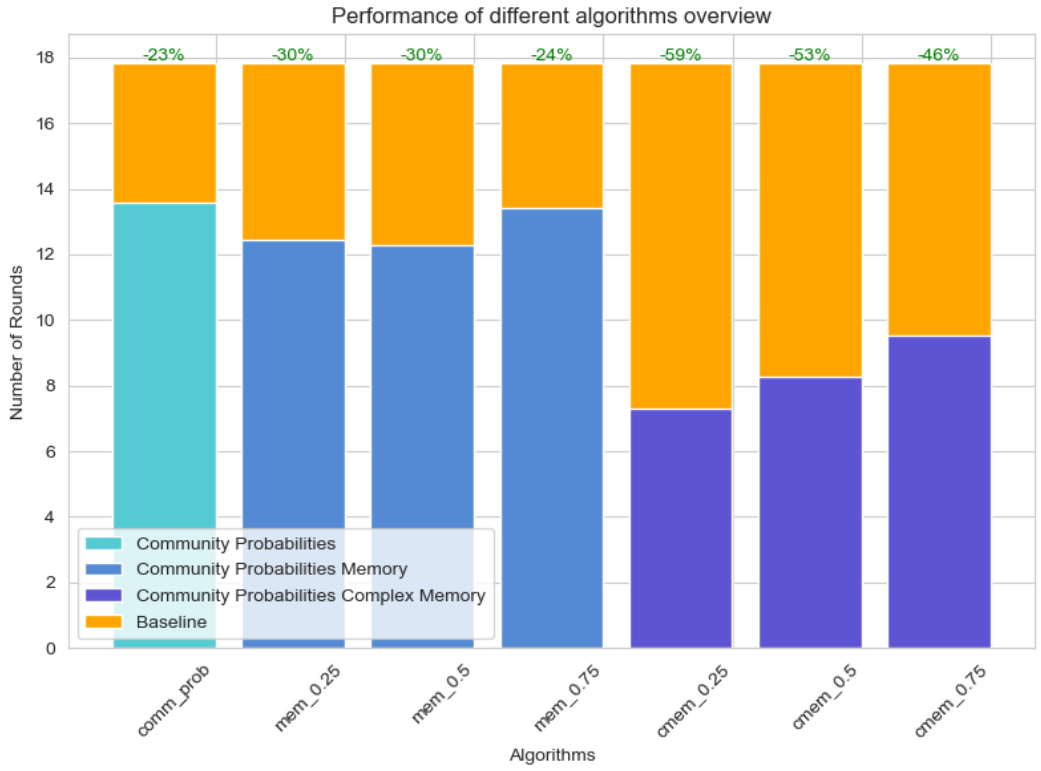


Abbildung 7.9: Auswertung Simulationsreihe 5

eine noch größere Optimierung erreicht. Das Hinzufügen der Speicherlogik führt dazu, dass zusätzliche 36% der Konvergenzzeit gespart werden. Die Tatsache, dass der Einsatz der Gedächtnis-Erweiterung eine signifikante Leistungssteigerung verursacht, entspricht den anfänglichen Annahmen. Der Grad der Optimierung fiel dabei jedoch wesentlich höher aus als erwartet. Dies kann auf die untereinander verknüpften Hub-Knoten zurückgeführt werden. Die betrachteten skalenfreien Netzwerke verfügen über viele Hubs mit unterschiedlichen Knotengraden, die auch untereinander verbunden sind. Durch eine starke Benachteiligung von vorigen Kommunikationspartnern wird gewährleistet, dass kleinere Hubs schneller ausgewählt werden. Zudem stellt das Wiederherstellen der anfänglichen Gewichte bei Erhalt von neuen Werten sicher, dass die wichtigsten Kommunikationspartner die neuen Daten zuerst erhalten. Als Folge wird eine Verringerung der benötigten Rundenzahl von aufgerundet 18 bis auf etwa 7 erreicht, was einer Einsparung von 11 Runden entspricht. Dies ist für kleinere Netzwerke mit tausend Knoten und durchschnittlichen Pfadlängen von etwas über 5 ein sehr zufriedenstellendes Ergebnis. In folgenden Simulationsreihen wird geprüft, ob komplexere Algorithmen ein besseres Ergebnis erzielen können. Ebenso erfolgt ein Vergleich mit der verwandten Arbeit [108]. Grund dafür ist, dass der hier vorgestellte Algorithmus eine hohe Ähnlichkeit zum CommunityProbabilities-Verfahren aufweist.

7.2.6 Fortgeschrittene Algorithmen auf hochmodularen skalenfreien Netzwerken

Die Evaluation der gesammelten Ergebnisse zeigte, dass die Konvergenz auf den hochmodularen skalenfreien Netzwerken stark durch die Zentralität, die Pfadlängen und Hubs beeinflusst werden. Der CommunityProbabilities-Algorithmus erzielte deshalb die bisher besten Ergebnisse. Es ergab sich jedoch die Vermutung, dass komplexere Algorithmen

das Optimierungspotential der Graphen noch besser ausschöpfen könnten. Aus diesem Grund wurde angedacht zur Gewichtung die Metriken nutzen, die stark mit der Konvergenzzeit korrelieren. Im Folgenden werden nun die ausgewählten Metriken beschrieben, für welche anschließend Algorithmen entwickelt wurden:

- **Betweenness Centrality:** Die Betweenness Centrality misst die Bedeutung eines Knotens innerhalb eines Netzwerks. Dabei quantifiziert sie, wie häufig ein Knoten auf einem kürzesten Pfad zwischen Paaren von anderen Knoten liegt. Sie berechnet sich über den Mittelwert der Summen der Verhältnisse zwischen den kürzesten Pfaden, die den Knoten v durchlaufen. Dieser wird ins Verhältnis gestellt zu allen kürzesten Pfaden im Graphen. Die Formel zur Berechnung ist im Grundlagenkapitel unter dem Index 2.27 zu finden.
- **Eigenvektorzentralität:** Auch die Eigenvektorzentralität misst die Bedeutung eines Knotens innerhalb eines Netzwerks. Sie berechnet die Einflusskraft eines Knotens anhand seiner Verbindungen zu anderen Knoten und dem Einfluss dieser anderen Knoten. Hierbei sind Knoten mit hoher Eigenvektorzentralität meist mit weiteren Knoten mit hoher Eigenvektorzentralität verbunden. Auch hier kann die Berechnung anhand der Formel 2.33 nachvollzogen werden.
- **Hub-Score:** Der Hub-Score misst den Einfluss eines Knotens und ist dementsprechend ein Autoritätsmaß. Er bewertet jeden Knoten anhand der Relevanz der Endknoten seiner Kanten. Dabei betrachtet der Hub-Score ausgehende Kanten und der artverwandte Authority-Score eingehende Kanten. Bei ungerichteten Graphen gibt es keine Kantenrichtungen, weshalb beide Metriken gleichbedeutend sind. Hat ein Knoten einen hohen Hub-Score, so verfügt er über viele Verbindungen zu weiteren Hubs. Die Formel zur Berechnung kann im Grundlagenkapitel unter 2.45 nachgeschlagen werden.

Die oben beschriebenen Metriken werden auf den Graphen global berechnet. Das heißt, sie werden am Graphen für jeden einzelnen Knoten kalkuliert. Dementsprechend ist es möglich, sie direkt zur Gewichtung zu verwenden. Es wird jedoch ein Ansatz bevorzugt, bei dem eine Verrechnung verschiedener Gewichte entsprechend einer MFWF durchgeführt wird. Hierbei wurde sich an der verwandten Arbeit aus Kapitel 3.4.2 orientiert. Dabei wird jedoch nur eine Kombination von bis zu zwei verschiedenen Gewichten ermöglicht. Durch die Optionalität können die neuen Verfahren alleinstehend ausgeführt werden. Ebenfalls wird eine Kombination mit den Gewichten des WeightedFactor- und CommunityProbabilities-Verfahrens möglich. Zur Berechnung der zusammengesetzten Gewichte werden beide Eingaben zuerst normalisiert. Anschließend wird folgende Formel für die Kombination verwendet:

$$updated\_weights = a \cdot w_1 + b \cdot w_2 \tag{7.1}$$

wobei:

w_1 = die ersten Gewichte (z.B. durch WeightedFactor oder CommunityProbabilities bestimmt)

w_2 = die zweiten Gewichte (z.B. Betweenness Centrality, Eigenvektorzentralität oder Hub-Scores)

a = Gewichtungsfaktor für die ersten Gewichte

b = Gewichtungsfaktor für die zweiten Gewichte

Dabei ist wichtig anzumerken, dass die verwendeten Metriken erneut zentral berechnet werden, um die Komplexität dieser Arbeit zu begrenzen. Vor der Auswahl der Metriken wurde geprüft, ob alternative dezentrale Verfahren bereitstehen. Solche dezentralen Algorithmen können als Ausgangslage verwendet werden, um die Daten auf den einzelnen Knoten zu gewinnen. Im Folgenden werden einige Verfahren zur dezentralen Bestimmung von Zentralitäts- und Autoritätsmaßen referenziert. Ein solches Verfahren wird beispielsweise in [125] vorgestellt. Das Paper beschreibt ein Framework, mit dem eine dezentrale Berechnung von Zentralitätsmetriken erfolgen kann. In diesem Zusammenhang wird auch die Betweenness Centrality berechnet, die als das komplexeste Zentralitätsmaß gilt. Im Vergleich zu anderen Zentralitätsmaßen ist sie am besten für die Analyse der Kommunikation auf Grundlage kürzester Pfade geeignet. In den Netzwerken, die im Rahmen dieser Thesis untersucht wurden, könnte das Verfahren angewandt werden, um jeden Knoten dazu zu veranlassen, die Zentralitätsmetriken seiner Nachbarn zu berechnen. Der wissenschaftliche Bericht [81] befasst sich ebenso mit der dezentralen Erkennung von Metriken. Hier wird unter anderem auch die Zentralität von Knoten bestimmt. Zudem erfolgt die Erprobung des Verfahrens auf verschiedenen skalenfreien Graphen. Daraus kann geschlussfolgert werden, dass auch hier die Anwendbarkeit der Methode gegeben ist. Die Dissertation [91] stellt einen verteilten Algorithmus für die Berechnung von Autoritätsmaßen in einem Peer-to-Peer Netzwerk vor. Der Algorithmus ermöglicht die dezentralte Berechnung der Authority- und Hub-Scores. Auch in diesem Fall könnten Knoten dieses Verfahren anwenden, um die Autoritätsmaße in unstrukturierten Netzwerken zu sammeln, ohne globales Wissen über die Topologie zu benötigen.

In der folgenden Simulationsreihe wird geprüft, inwiefern die neuen gewichteten Algorithmen die Konvergenzzeit weiter verringern können. Dabei werden erneut die hochmodularen skalenfreien Netzwerke zur Ausführung der Simulationen verwendet. Dadurch wird ein Vergleich mit den beiden vorigen Testreihen möglich. Ebenfalls soll herausgefunden werden, ob die Algorithmen alleinstehend oder in Kombination mit den topologischen Daten die besten Resultate liefern.

7.2.6.1 Definition der Simulationsreihe

Im Nachgang folgt nun die Definition der sechsten Simulationsreihe:

- **Simulationsreihe:** Fortgeschrittene Algorithmen auf hochmodularen skalenfreien Netzwerken (alternativ „Advanced High-Mod Scale-Free“)
- **Nummer:** 6
- **Abhängigkeit:** 4
- **Zweck:** Vergleich Ergebnisse aus Simulationsreihe 4 mit fortgeschrittenen Algorithmen BetweennnessWeighted, BetweennnessCommunityProbabilities, EigenvectorWeighted, EigenvectorCommunityProbabilities, HubScoreWeighted, HubScoreCommunityProbabilities
- **Aufbau:**
  - Sechs Simulationsressourcen: Ausführen der verschiedenen Algorithmen BetweennnessWeighted (A), BetweennnessCommunityProbabilities (B), EigenvectorWeighted (C), EigenvectorCommunityProbabilities (D), HubScoreWeighted (E), HubScoreCommunityProbabilities (F) auf jeweils gleichen Graphen
  - Insgesamt 160 verschiedene Graphen mit unterschiedlichen Modularitätswerten zwischen $[0.65, 0.95]$

  - Ausführen von 1 Wiederholung pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 960 Ergebnisse: 160 Ergebnisse pro Algorithmus (A bis F)

- **Annahme:** Durch die vorangehenden Testreihen hat sich die Wichtigkeit der Zentralität sowie der Hubs bestätigt. Es wird erwartet, dass alle drei Algorithmen eine bessere Leistung als der CommunityProbabilities-Algorithmus erreichen. Die Frage ist, ob eine Kombination verschiedener Gewichtungen eine weitere Performancesteigerung ermöglicht.

7.2.6.2 Ergebnisse

Die Simulationen der fortgeschrittenen Algorithmen wurden ebenfalls auf den hochmodularen skalenfreien Graphen ausgeführt. Deshalb können auch hier die in Kapitel D.4.1 aufgelisteten Graphmetriken referenziert werden. Im Anhang D.6.2 sind die detaillierten Simulationsergebnisse ausgelagert. Anhand der Visualisierungen kann die Performance der verschiedenen neuen Algorithmen im Vergleich zum Standardverfahren nachvollzogen werden. Eine Zusammenfassung der Resultate ist in Abbildung 7.10 zu finden. Hier werden die Konvergenzzeiten der neuen Algorithmen mit den Ergebnissen der vorangehenden Simulationsreihen verglichen. Dabei wird für die Verfahren der Durchschnitt der benötigten Runden bis zum Eintreten der Konvergenz dargestellt. Man kann erkennen, dass alle Variationen der neuen Algorithmen eine deutlich bessere Leistung erreichen als die zuvor evaluierten Verfahren. Des Weiteren erzielen die Algorithmen ohne Kombination mit den Gewichten der Community-Strukturen die beste Performance. Dabei erreicht die Gewichtung anhand der Betweenness Centrality die stärkste Optimierung. Hier wird die durchschnittliche Rundenzahl von aufgerundet 18 bis auf knapp über 5 reduziert, was einer Beschleunigung von 69% gegenüber dem Baseline-Verfahren entspricht. Im Vergleich dazu benötigt der WeightedFactor-Algorithmus weniger als 16 Runden, während der CommunityProbabilities-Algorithus nach etwas über 13 Runden konvergiert. Damit wird also eine weitere Einsparung von 8 Runden gegenüber dem CommunityProbabilities-Verfahren erzielt. Zudem ist das Verfahren auch etwa 2 Runden schneller als die beste Variante des CommunityProbabilitiesComplexMemory-Verfahrens. Werden die Community-Strukturen zur Gewichtung verrechnet, steigt die Konvergenzzeit. Bei Kombination mit dem CommunityProbabilities-Verfahren wird eine Verlangsamung um 13% verursacht, damit ist der Algorithmus nur noch 56% schneller als das Baseline-Verfahren. Die gleichen Erkenntnisse lassen sich für die anderen Varianten (Eigenvektorzentralität und Hub-Score) feststellen. Dabei konvergiert das Gossiping bei Gewichtung mit der Eigenvektorzentralität langsamer. Hier sinkt die Optimierung auf 63% beziehungsweise auf 54% bei Kombination mit den CommunityProbabilities. Die Gewichtung nach Hub-Scores erzielt das schlechteste Ergebnis unter den fortgeschrittenen Algorithmen, mit einer Beschleunigung um 54% (beziehungsweise 50% bei Kombination).

Anschließend wurde eine weitere Auswertung für die unterschiedlichen Modularitätsbereiche durchgeführt. Hierbei konnte festgestellt werden, dass die neuen Verfahren stark abhängig von der Modularität sind. In Abbildung 7.11 sind die Ergebnisse dieser Untersuchung dargestellt. Bei Modularitäten zwischen 0.6 und 0.7 liegt die Leistungssteigerung bei 47% für den BetweennessCentrality-Algorithmus. Das sind 28% mehr als das CommunityProbabilities-Verfahren für diese Netzwerke einsparen kann. Je höher die Modularität des Graphen, desto weiter steigt die Optimierung. Das beste Ergebnis wird bei Modularitäten von über 0.9 erzielt, wo der BetweennessCentrality-Algorithmus die benötigten Runden um 84% reduziert. Für diese Netzwerke wird der Performanceunterschied zum CommunityProbabilities-Verfahren noch deutlicher. Das Verfahren

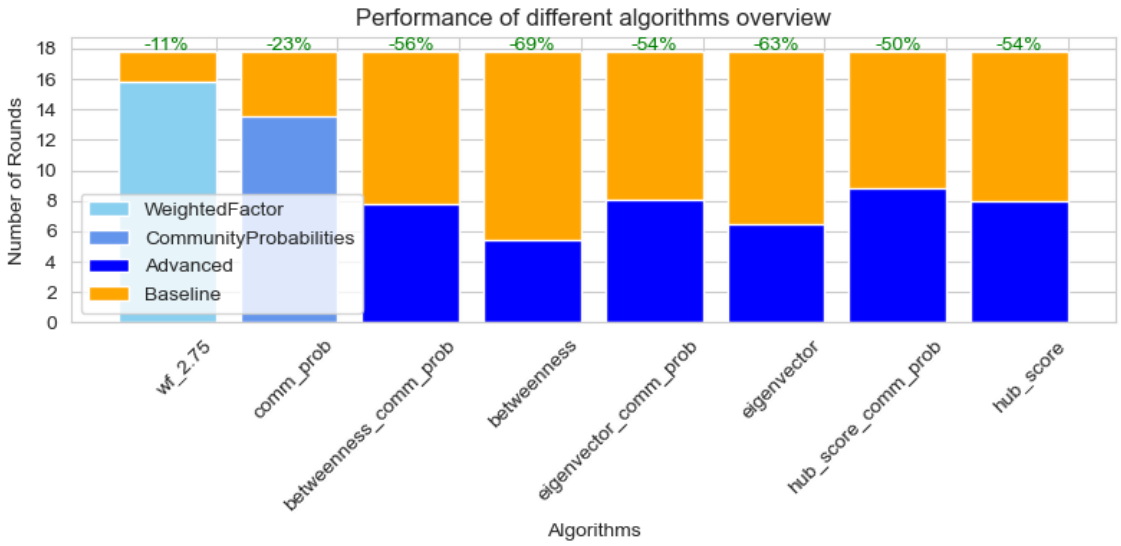


Abbildung 7.10: Auswertung Simulationsreihe 6

erreicht eine Zeiteinsparung von 30%, somit benötigt der fortgeschrittene Algorithmus 54% weniger Runden bis zur Konvergenz. Die gleichen Trends können bei den Algorithmen beobachtet werden, die Eigenvektorzentralitäten und Hub-Scores zur Gewichtung verwenden. Allerdings benötigen diese Algorithmen jeweils mehr Runden, um die Konvergenz zu erreichen. Dabei konvergieren die EigenvectorCentrality-Algorithmen im Schnitt 6% langsamer, während die Hub-Score-Algorithmen fast 15% länger laufen. Ebenfalls kann erkannt werden, dass die Inkorporation der CommunityProbabilities-Gewichte für alle Verfahren auf allen Modularitätsbereichen eine Verschlechterung verursacht. Die Verschlechterung verringert sich zwar mit Anstieg der Modularität, ist aber selbst bei Netzwerken mit Modularitäten von über 0.9 deutlich erkennbar.

7.2.6.3 Schlussfolgerungen

Die Ergebnisse bestätigen die Annahme, dass Zentralität und Hub-Strukturen einen größeren Einfluss auf die Konvergenzzeit haben, als die Modularität. Man muss jedoch anmerken, dass die Zentralitätsmaße inhärent auch die modularen Strukturen berücksichtigen. In Graphen mit ausgeprägten modularen Strukturen fungieren Knoten mit hoher Betweenness Centrality oft als Brücken zwischen Communities. Sie verbinden unterschiedliche Cluster und haben daher hohen Einfluss auf die Informationsausbreitung. In skalenfreien Netzwerken sind diese Knoten identisch mit den Hub-Knoten, welche auf vielen kürzesten Pfaden liegen. Folglich werden hier bei einer Gewichtung nach Zentralität auch die Hubs priorisiert. Des Weiteren wird sogar eine Abstufung zwischen den Hub-Knoten realisiert. Zentrale Hubs haben einen höheren Einfluss und werden dementsprechend bevorzugt, wodurch eine abgestufte Auswahl möglich wird.

Die Unterschiede der verschiedenen fortgeschrittenen Algorithmen können anhand der verwendeten Metriken erklärt werden. Hierbei gilt die Betweenness Centrality als die komplexeste der betrachteten Zentralitätsmetriken. Die hohe Komplexität erlaubt eine detaillierte Erfassung der Bedeutung von Knoten für die Informationsverbreitung. Entsprechend dieser Logik zeigt die Gewichtung nach Eigenvektorzentralität eine geringere Leistung, während die Gewichtung nach Hub-Scores die schwächste Performance aufweist. Die wesentlich schlechtere Performance der HubScores-Algorithmen lässt sich ebenfalls durch ihren Ansatz erklären. Der Hub-Score bezieht indirekte Verbindungen

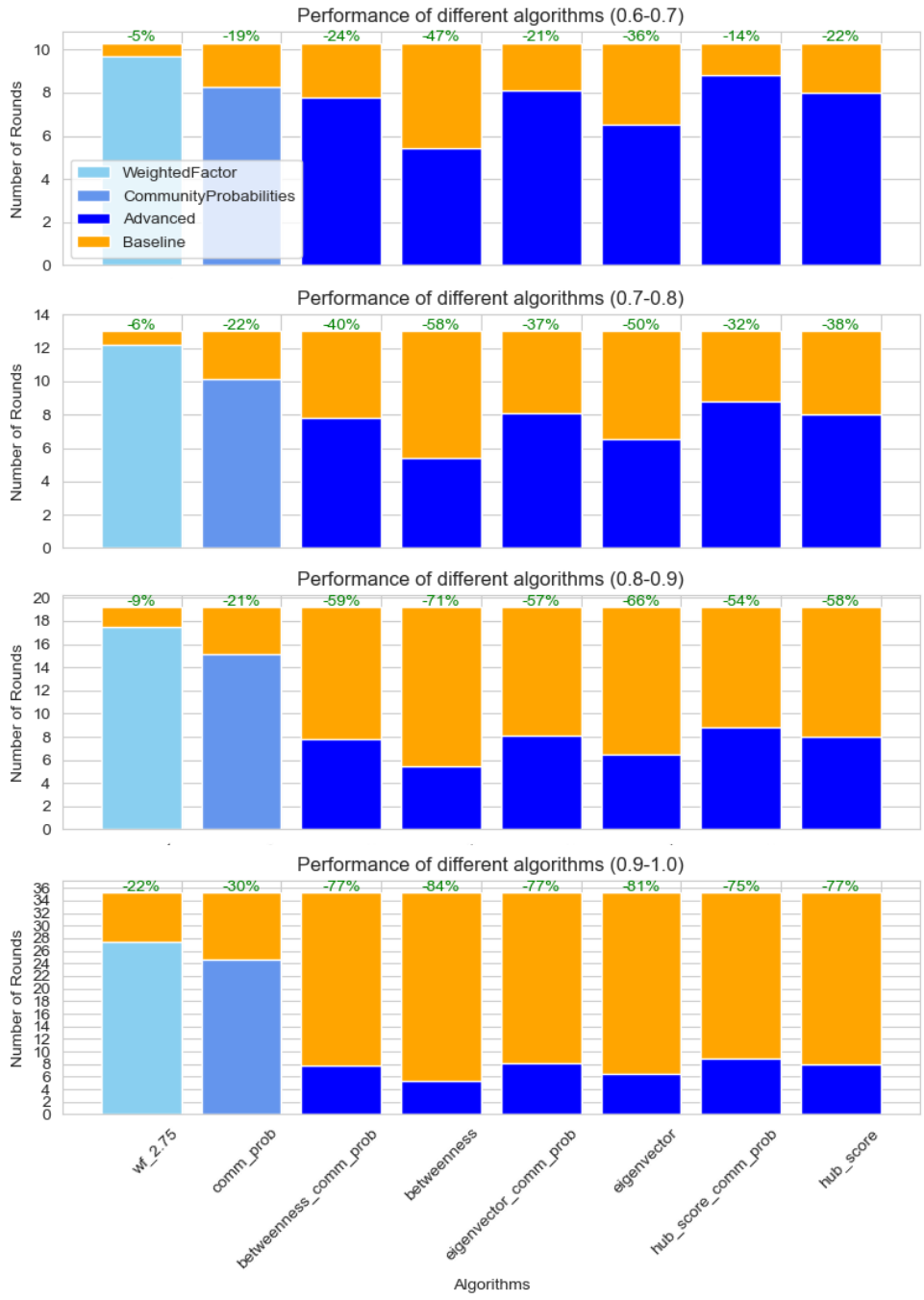


Abbildung 7.11: Auswertung nach Modularitätsbereichen Simulationsreihe 6

zwischen Knoten in die Berechnung ein, während direkte Verbindungen nur über eingehende und ausgehende Kanten berücksichtigt werden. Dadurch werden weniger Informationen über die Zentralität der Knoten eingefangen. Da dies jedoch maßgeblich für die Informationsausbreitung ist, wird eine schlechtere Performance gegenüber den evaluierten Zentralitätsmaßen erreicht.

Verglichen mit den zuvor evaluierten Algorithmen (WeightedFactor und CommunityProbabilities) verwenden alle fortgeschrittenen Verfahren wesentlich komplexere Metriken zur Gewichtung. In großen Netzwerken führt die Verwendung dieser zu einem erhöhten Berechnungsaufwand und zusätzlichen Ressourcenanforderungen. Ebenfalls

muss beachtet werden, dass die komplexen Metriken in allen Simulationen global berechnet wurden. Dadurch verfügen alle Knoten bei einer Simulation zu jedem Zeitpunkt über die optimalen Gewichte. Wird stattdessen eine dezentrale Berechnung durchgeführt, so müssen je nach Laufzeit unterschiedlich starke Ungenauigkeiten beziehungsweise Abweichungen in Kauf genommen werden. Hier kann das Optimierungspotential nicht vollständig ausgeschöpft werden, da gegebenenfalls suboptimale Gewichte verwendet werden müssen.

7.2.7 Verbesserung der fortgeschrittenen Algorithmen auf hochmodularen skalenfreien Netzwerken

Anschließend an die Simulationsreihen zu den fortgeschrittenen Algorithmen auf hochmodularen skalenfreien Netzwerkenden wurde eine Auswertung durchgeführt. Hierbei wurden die besten Varianten für weitere Untersuchungen ausgewählt. In den folgenden Simulationsreihen wird sich bei den fortgeschrittenen Verfahren auf die BetweennessWeighted- und EigenvectorWeighted-Algorithmen beschränkt. Diese Simulationsreihe evaluiert die Performance bei Erweiterung mit der komplexen Memory-Logik. Hierbei werden verschiedene Faktoren zur Benachteiligung voriger Kommunikationspartner verwendet. Vorige Simulationen zeigten eine starke Verbesserung der Konvergenzzeit bei Einsatz eines Gedächtnisses. Ziel ist es zu verifizieren, ob so auch die fortgeschrittenen Algorithmen weiter optimiert werden können. Dabei sind die Ergebnisse erneut mit den Resultaten der vorigen Testreihen zu vergleichen.

7.2.7.1 Definition der Simulationsreihe

Im Nachgang folgt die Definition der siebten Simulationsreihe:

- **Simulationsreihe:** Verbesserung der fortgeschrittenen Algorithmen auf den hochmodularen skalenfreien Netzwerken (alternativ „Advanced High-Mod Scale-Free Memory“)
- **Nummer:** 7
- **Abhängigkeit:** 5, 6
- **Zweck:** Vergleich voriger Ergebnisse mit BetweennessWeightedComplexMemory und EigenvectorWeightedComplexMemory ohne Gewichtung basierend auf Community-Strukturen
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen BetweennessWeightedComplexMemory (A) und EigenvectorWeightedComplexMemory (B) auf jeweils gleichen Graphen
  - Für A und B Werte von PriorPartnerFactor in $[0.1, 0.2, 0.3, 0.4, 0.5]$)
  - Insgesamt 160 verschiedene Graphen mit unterschiedlichen Modularitätswerten zwischen $[0.65, 0.95]$
  - Ausführen von 1 Wiederholung pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 1600 Ergebnisse: 160 Ergebnisse pro Faktor für A und B (somit in Summe je Testreihe 800)

- **Annahme:** Angesichts der extrem hohen Leistungsfähigkeit beider Algorithmen ist es schwer vorstellbar, eine signifikante weitere Optimierung durch Hinzufügen der Memory-Logik zu erreichen. Es wird dementsprechend von einer schwachen Beschleunigung ausgegangen.

7.2.7.2 Ergebnisse

Auch für die Simulationsreihe zur Verbesserung der fortgeschrittenen Algorithmen wurden die hochmodularen skalenfreien Graphen verwendet. Dementsprechend sind die gemittelten Graphmetriken analog zur vierten Simulationsreihe (siehe Kapitel D.4.1). Im Anhang D.7.2 ist die Verteilung aller Messwerte dargestellt. Anhand der Visualisierungen kann die Performance der verschiedenen Varianten genauer nachvollzogen werden. Alle Ergebnisse sind in Abbildung 7.12 kompakt zusammengefasst. Hier ist die Leistung der Algorithmen mit unterschiedlichem PriorPartnerFactor zu sehen. Die Verfahren können anhand der durchschnittlich benötigten Rundenanzahl bewertet werden. Dabei werden die Verfahren untereinander sowie mit dem Verfahren ohne Memory-Logik und dem Baseline-Verfahren verglichen. Es ist ersichtlich, dass die Algorithmen mit Speicherlogik eine höhere Leistungsfähigkeit aufweisen im Vergleich zu den Verfahren ohne diese. Während der Algorithmus mit Gewichtung nach Betweenness Centrality zwischen 5 und 6 Runden bis zur Konvergenz benötigt, wird durch das Hinzufügen der Speicherlogik im besten Fall (Faktor 0.1 und 0.2) bereits eine Konvergenz in unter 5 Runden erreicht. Es kann erkannt werden, dass die Anzahl an Runden geringfügig mit Erhöhung des PriorPartnerFactors steigt. Dementsprechend erreichen die Verfahren mit niedrigem PriorPartnerFactor bessere Ergebnisse. Dasselbe kann für den EigenvectorWeightedComplexMemory-Algorithmus beobachtet werden. Hier sinkt die Rundenzahl von etwa 6.5 auf knapp über 6 für die Faktoren 0.1, 0.3 und 0.4. Die Ergebnisse für den Faktor 0.2 stellen eine Ausnahme dar, da sie schlechter ausfallen als die Ergebnisse für die vorangehenden und nachfolgenden Faktoren. Für beide Verfahren führt die Anwendung der Memory-Logik zu einer Reduzierung um etwa eine halbe Runde. Somit gelingt es dem BetweennessWeightedComplexMemory-Algorithmus weitere 3% der Gesamtkonvergenzzeit einzusparen, was zu einer Reduktion um 72% gegenüber dem Baseline-Verfahren führt. Das Ergebnis fällt geringfügig schlechter aus bei Gewichtung mit der Eigenvektorzentralität. Hier wird zusätzlich 2% der Zeit gespart, wodurch die Optimierung von 63% auf 65% erhöht wird.

7.2.7.3 Schlussfolgerungen

Mit dem BetweennessWeighted-Algorithmus wurden in der vorangehenden Testreihe die besten Ergebnisse erzielt, während der EigenvectorWeighted-Algorithmus die zweitgrößte Zeiteinsparung erreichte. Zu Beginn dieser Simulationsreihe wurde angenommen, dass aufgrund der schnellen Konvergenz nur noch ein geringes Optimierungspotential durch die Speicherlogik ausgenutzt werden kann. Diese Annahme konnte durch die Testreihen bestätigt werden. Dabei muss man jedoch anmerken, dass die Betrachtungen im Vergleich zu den hohen Konvergenzzeiten des Standardverfahrens stehen. Die Beschleunigung der BetweennessWeighted- zur BetweennessWeightedComplexMemory-Variante beträgt etwa 10%. Gleichermaßen konvergiert der Algorithmus mit Gewichtung nach Eigenvektoren durch Hinzufügen der Speicherlogik 7% schneller. Diese Werte wirken dennoch verschwindend gering, wenn man sie mit der Beschleunigung der fünften Simulationsreihe vergleicht (siehe Kapitel 7.2.5.1). Hier wird die Rundenanzahl durch Hinzufügen der Speicherlogik im besten Fall fast halbiert. Obwohl die Verbesserung weniger signifikant ist, ist dennoch eine spürbare Leistungssteigerung zu verzeichnen. Die bereits kurz-

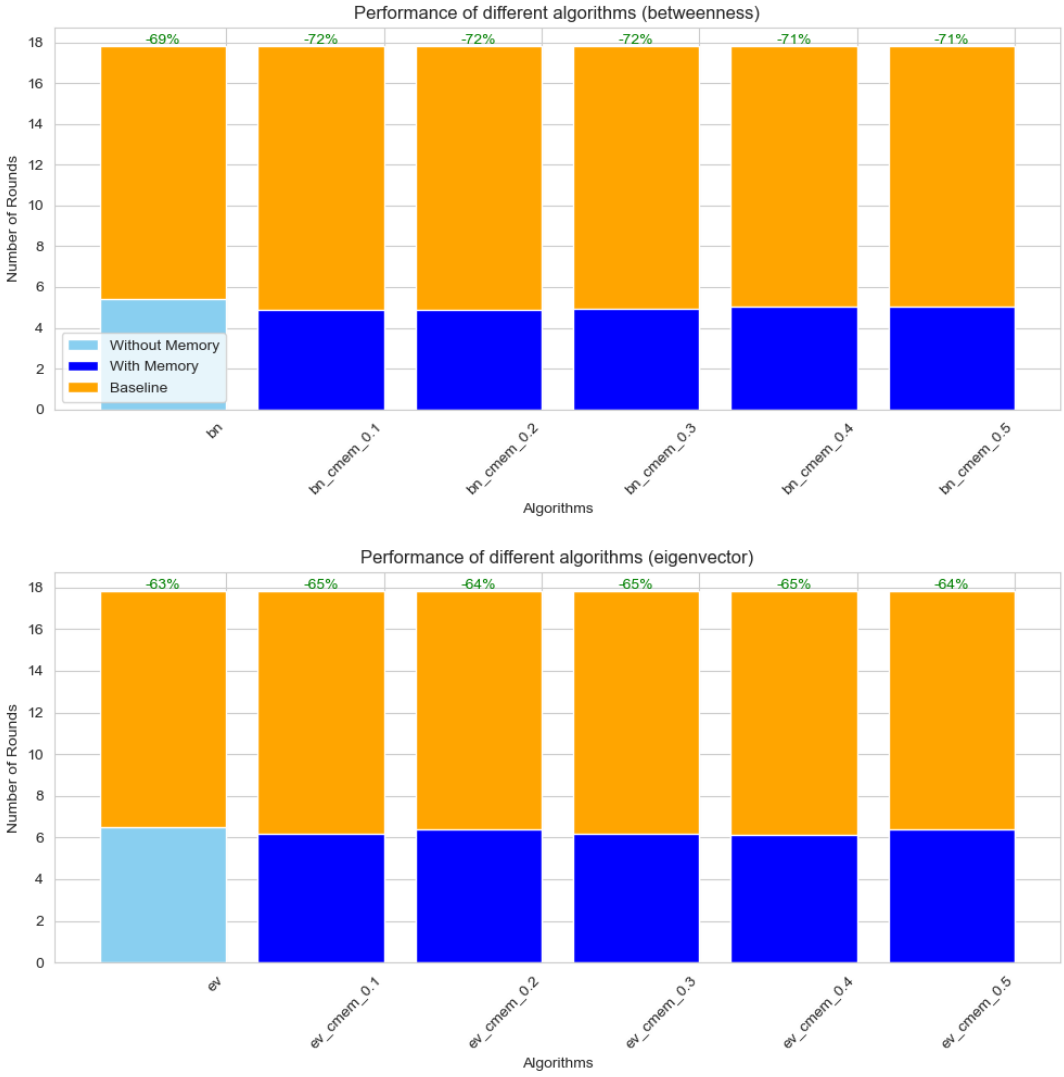


Abbildung 7.12: Auswertung Simulationsreihe 7

en Konvergenzzeiten erklären auch die schwächere Wirksamkeit der Speicherlogik. Der BetweennessWeightedComplexMemory-Algorithmus erreicht eine Verringerung der Konvergenzzeit um 72% gegenüber dem Standardverfahren. Damit wurde in dieser Testreihe ein neuer optimaler Algorithmus für das Gossiping auf den hochmodularen skalenfreien Graphen gefunden. Angesichts der anfänglichen Annahmen und ersten Testreihen ist eine Reduktion von aufgerundet 18 bis auf unter 5 Runden ein sehr zufriedenstellendes Ergebnis. Die simulierten Netzwerke haben durchschnittlichen Pfadlängen von gerundet 5.3 sowie einen Durchmesser von 12.45. Dementsprechend existieren in den Netzwerken oft lange Pfade mit entlegenen Knoten. Wie anhand des Baseline-Verfahrens gesehen werden kann, ist dies keine optimale Ausgangslage für eine zufällige Informationsausbreitung. Eine Reduktion der Rundenanzahl auf einen Wert, der geringer ist als die durchschnittliche Pfadlänge, kann als Annäherung an eine optimale Lösung betrachtet werden.

7.2.8 Community-Based Gossiping auf hochmodularen skalenfreien Netzwerken

Die verwandte Arbeit [108] stellt einen alternativen Gossip-Algorithmus vor. Das hier beschriebene Verfahren wird detailliert in Kapitel 3.5 diskutiert. Dabei werden die Ge-

wichte zur Auswahl der Kommunikationspartner so verteilt, dass zwischen allen angrenzenden Communities gleich wahrscheinlich ausgewählt wird. Ebenfalls werden alle Knoten innerhalb einer Community mit der gleichen Wahrscheinlichkeit selektiert. Die folgende Testreihe evaluiert diesen Algorithmus auf den hochmodularen skalenfreien Graphen. So kann ein Vergleich mit den Algorithmen, die im Rahmen dieser Arbeit entwickelt worden, gezogen werden. Hier ist vor allem die Performance im Vergleich zum CommunityProbabilities-Algorithmus bedeutend, da diese eine große Ähnlichkeit aufweisen.

7.2.8.1 Definition der Simulationsreihe

Im Anschluss erfolgt die Definition dieser Simulationsreihe:

- **Simulationsreihe:** Community-Based Gossiping auf hochmodularen skalenfreien Netzwerken (alternativ „Community-Based Gossiping“)
- **Nummer:** 8
- **Abhängigkeit:** 4
- **Zweck:** Vergleich des Algorithmus nach [108] mit Baseline sowie den entwickelten Algorithmen (WeightedFactor-/CommunityProbabilities)
- **Aufbau:**
  - Eine Simulationsressource: Ausführen des CommunityBased-Algorithmus (A) auf jeweils gleichen Graphen
  - Insgesamt 160 verschiedene Graphen mit unterschiedlichen Modularitätswerten zwischen $[0.65, 0.95]$
  - Ausführen von 1 Wiederholung pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 160 Ergebnisse
- **Annahme:** Die Performance des CommunityBased-Algorithmus sollte nicht allzu stark von dem CommunityProbabilities-Verfahren abweichen. Die Ergebnisse der verwandten Arbeit lassen vermuten, dass das CommunityBased-Verfahren schneller konvergiert. Der Logik zufolge sollten die Ergebnisse jedoch schlechter ausfallen, da das verwendete Verfahren weniger komplex ist und außerdem Hub-Knoten unberücksichtigt lässt.

7.2.8.2 Ergebnisse

Bei der achten Simulationsreihe handelt es sich um die letzte Reihe auf den hochmodularen skalenfreien Netzwerken. Diese Graphen wurden in den drei vorangehenden Simulationen ebenfalls ausgewertet. Bei Bedarf können die gemittelten Graphmetriken erneut im Anhang D.4.1 nachgeschlagen werden. Die detaillierten Messwerte sind im Kapitel D.8.2 zu finden. Hier werden die benötigten Runden bis zur Konvergenz der einzelnen Simulationen dargestellt. In Abbildung 7.13 sind die zusammengefassten Resultate visualisiert. Zu sehen sind die unterschiedlichen Konvergenzzeiten der gewichteten Algorithmen im Vergleich mit dem Baseline-Verfahren. Hierbei kann die Performance des CommunityBased-Algorithmus mit den CommunityProbabilities- und WeightedFactor-Algorithmen gegenübergestellt werden. Für den WeightedFactor-Algorithmus wurde als

Faktor der Wert 2.75 verwendet, womit bei der vierten Simulationsreihe die besten Ergebnisse erzielt wurden. Dieser konvergierte in etwas unter 16 Runden und war somit 2 Runden schneller als das Baseline-Verfahren. Man kann erkennen, dass der CommunityBased-Algorithmus geringfügig bessere Ergebnisse erzielte. Hier konnte eine Einsparung von 12% erreicht werden, womit er 1% weniger Runden als das WeightedFactor-Verfahren benötigte. Dennoch lag das Ergebnis weit hinter der Performance des CommunityProbabilities-Algorithmus. Dieser benötigte durchschnittlich nur etwa 13 Runden, was einer Optimierung von 23% entspricht.

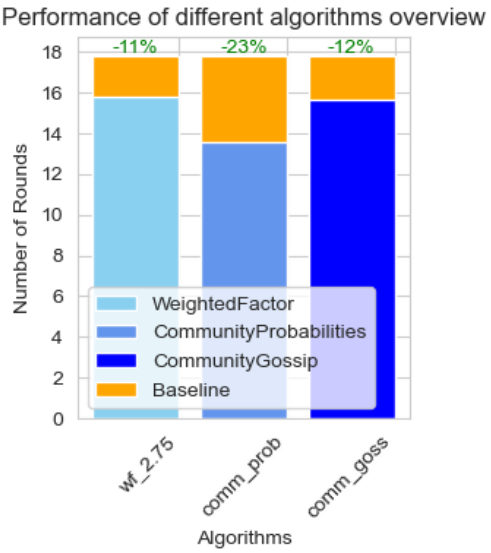


Abbildung 7.13: Auswertung Simulationsreihe 8

Im Anschluss erfolgte eine weitere Auswertung in Abhängigkeit der Modularität. Dabei wurden die Simulationsergebnisse nach Modularitätsniveaus unterteilt. Die Analyse zeigt, dass bei allen Algorithmen höhere Modularitätswerte zu höheren Rundenzahlen führen und somit eine positive Korrelation besteht. In Abbildung 7.14 ist die Performance der Algorithmen nach Modularitätsbereichen dargestellt. Hier können die exakten Rundenzahlen für den jeweiligen Algorithmus entnommen werden. Man kann erkennen, dass die Performance des CommunityBased-Verfahrens stark mit der Ausprägung von Community-Strukturen korreliert. Dementsprechend besteht das geringste Optimierungspotential für Netzwerke mit Modularitäten zwischen 0.6 und 0.7. Hier kann nur eine minimale Einsparung von 2% erzielt werden. Dieser Wert liegt knapp unterhalb des WeightedFactor-Verfahren mit 5% und deutlich unter dem CommunityProbabilitiesVerfahren mit 19%. Jedoch steigt die Performance mit zunehmender Modularität. So überholt der CommunityBased- den WeightedFactorAlgorithmus bereits im Bereich zwischen 0.7 und 0.8. Hier erreicht er eine Einsparung von 7% im Vergleich zu den 6% des WeightedFactor-Verfahrens. Dabei ist er immer noch weit von der Leistung des CommunityProbabilities-Verfahrens entfernt, welches hier 22% weniger Runden als Baseline benötigt. Ähnliche Ergebnisse liegen für den Modularitätsbereich 0.8 bis 0.9 vor, wo sowohl das CommunityBased- als auch das WeightedFactor-Verfahren eine Einsparung von 9% erreichen. Auch hier ist der CommunityProbabilities-Algorithmus mit 21% erneut am schnellsten. Bei Modularitäten zwischen 0.9 und 1.0 erreicht die Leistung des CommunityBased-Verfahrens ihren Höhepunkt. Mit einer Beschleunigung von 25% liegt die Optimierung nah an den 30% des CommunityProbabilities-Algorithmus und über den 22% des WeightedFactor-Algorithmus.

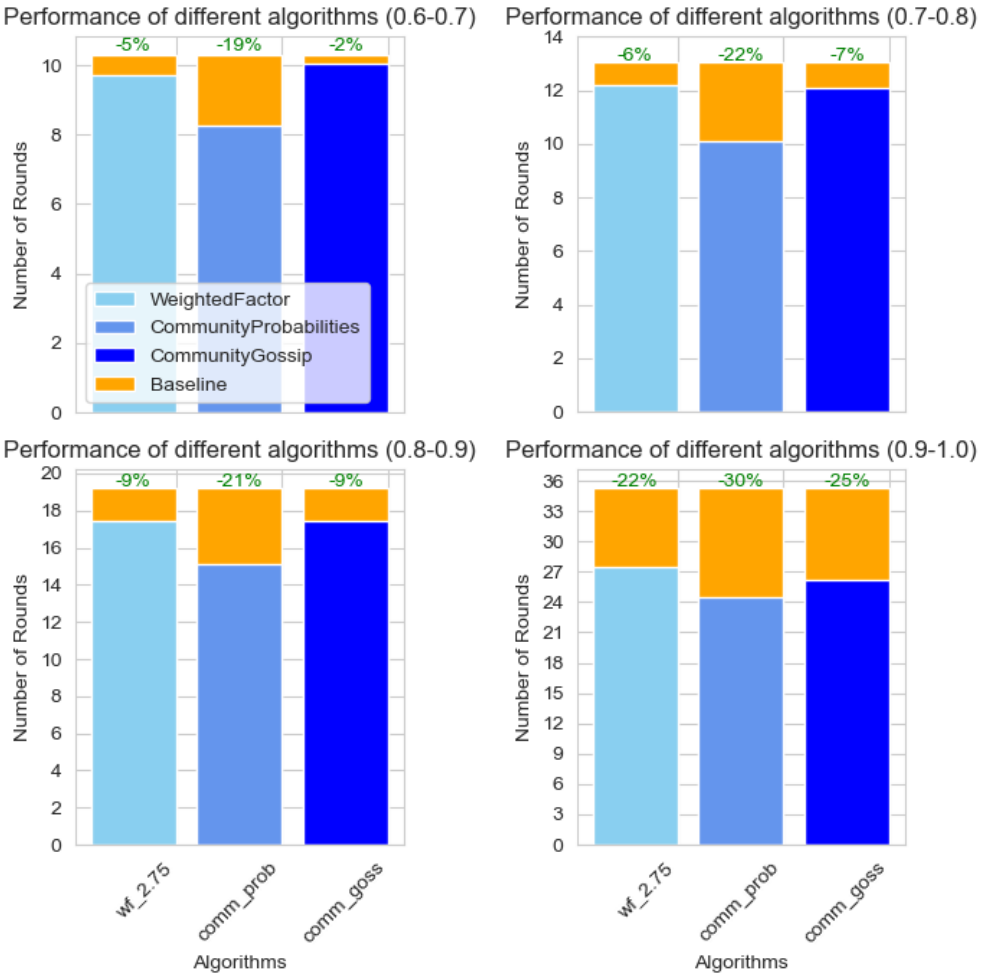


Abbildung 7.14: Auswertung nach Modularitätsbereichen Simulationsreihe 8

7.2.8.3 Schlussfolgerungen

Aus den Ergebnissen kann geschlussfolgert werden, dass der CommunityProbabilities-Algorithmus das Optimierungspotential besser ausnutzen kann als das CommunityBased-Verfahren. Dies kommt insbesondere bei weniger modularen Netzwerken zum Vorschein, wo die naiveren Ansätze versagen. Aber auch bei hochmodularen Graphen lieferte das Verfahren den Erwartungen entsprechend schlechtere Ergebnisse. Die starke Performancesteigerung, die auf den Graphen in der verwandten Arbeit beobachtet wurde, blieb für die hier betrachteten Graphen aus. Grund für die schlechtere Leistung des CommunityBased-Verfahrens ist die Tatsache, dass es sich ausschließlich auf Community-Strukturen konzentriert. Es wird weder eine direkte noch indirekte Berücksichtigung von Zentralitäts- und Autoritätsmaßen vorgenommen. Der CommunityBased-Algorithmus wählt immer zwischen allen Communities mit gleicher Wahrscheinlichkeit aus. Infolgedessen wird eine gezielte Optimierung für Grenzknoten realisiert, welche Bottlenecks beim Gossiping darstellen können. Grenzknoten verfügen über viele Nachbarn aus der eigenen Community und haben nur sehr wenige Nachbarn aus anderen Knoten. Der Algorithmus kann erfolgreich sicherstellen, dass Grenzknoten bevorzugt Knoten anderer Communities selektieren. Jedoch wird für die meisten anderen Knoten kaum eine merkbare Verbesserung erzielt. Der CommunityProbabilities-Algorithmus hingegen präferiert Knoten bei der Nachbarauswahl, die sehr unwahrscheinlich die gleiche Community haben. Dabei werden Knoten

mit möglichst vielen Nachbarn aus anderen Communities bevorzugt. Als Folge wählt der Algorithmus Hub-Knoten eher aus als normale Knoten, die wenig Nachbarn haben. Dadurch wird auch bei wenig modularen Netzwerken eine Beschleunigung erzielt. Damit bestätigen die Ergebnisse erneut, dass zur Optimierung von Gossiping-Protokollen in unstrukturierten Netzwerken die Zentralität und Hub-Strukturen berücksichtigt werden sollten.

7.2.9 Gnutella-P2P-Netzwerk

In der letzten Simulationsreihe werden verschiedene Algorithmen auf realen Netzwerken evaluiert. Hierbei erfolgte zuerst die Erzeugung von Partitionen des Gnutella-P2P-Netzwerkes, welche dann als Graphen angelegt wurden. Basierend auf den Auswertungen der zuvor durchgeführten Simulationen fand die Auswahl von Algorithmen statt. Dabei wurde festgelegt, dass sowohl der WeightedFactor- als auch der CommunityBased-Algorithmus untersucht werden sollen. Ebenfalls soll das CommunityProbabilities- sowie das BetweennessWeighted-Verfahren mit und ohne ComplexMemory-Erweiterung getestet werden. Dadurch können sowohl die naiven Verfahren als auch die leistungsstarken komplexeren Verfahren geprüft werden.

7.2.9.1 Definition der Simulationsreihe

Es erfolgte eine Aufteilung der Simulationsreihe in mehrere Teilreihen aufgrund ihres großen Umfangs. So konnten Auswertungen, wie zum Beispiel die Evaluation des WeightedFactor-Algorithmus, bereits in anfänglichen Projektphasen durchgeführt werden. Auch die anderen Teilreihen wurden in separaten Projektphasen durchgeführt, um die Gesamtdauer zu reduzieren. Zudem konnten so die Kubernetes-Clusterressourcen optimal genutzt werden. Die Simulationen wurden zwischen den anderen Simulationsreihen und den Entwicklungsphasen durchgeführt, wodurch kontinuierlich gearbeitet werden konnte und keine großen Leerlaufzeiten entstanden. Im Anschluss folgen nun die Definitionen der einzelnen Teilreihen:

Teilreihe Gnutella WeightedFactor:

- **Simulationsreihe:** WeightedFactor auf Gnutella-Partitionen (alternativ „Gnutella WeightedFactor“)
- **Nummer:** 9A
- **Abhängigkeit:** 1, 2, 3
- **Zweck:** Vergleich Baseline mit WeightedFactor mit jeweils verschiedenen Faktorwerten
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen Baseline (A) und WeightedFactor (B) auf jeweils gleichen Graphen
  - Insgesamt 50 verschiedene Graphen als unterschiedliche Netzwerkpartitionen
  - Für B Faktoren aus $[1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3]$
  - Ausführen von 20 (A) beziehungsweise 10 (B) Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 5000 Ergebnisse, 1000 für A und 4000 für B (500 für jeden Faktor)

- **Annahme:** Das naive Verfahren sollte nur einen sehr kleinen Mehrgewinn für die Gnutella-Partitionen darstellen. Die schwache Ausprägung der Community-Strukturen sollte kein großes Optimierungspotential ermöglichen.

Teilreihe Gnutella CommunityProbabilities:

- **Simulationsreihe:** CommunityProbabilities auf Gnutella-Netzwerkpartitionen (alternativ „Gnutella CommunityProbabilities“)
- **Nummer:** 9B
- **Abhängigkeit:** 5
- **Zweck:** Vergleich voriger Ergebnisse mit CommunityProbabilities und CommunityProbabilitiesComplexMemory mit verschiedenem PriorPartnerFactor
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen CommunityProbabilities (A) und CommunityProbabilitiesComplexMemory (B) auf jeweils gleichen Graphen
  - Insgesamt 50 verschiedene Graphen als unterschiedliche Netzwerkpartitionen
  - Für B PriorPartnerFactor aus $[0.1, 0.3, 0.5]$
  - Ausführen von 20 Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 4000 Ergebnisse, 1000 für A und 3000 für B (1000 für jeden Faktor)
- **Annahme:** Das Verfahren sollte ein besseres Ergebnis als das WeightedFactor--Verfahren erreichen. Da Clusterstrukturen wenig ausgeprägt sind, ist jedoch davon auszugehen, dass die Beschleunigung nicht sehr signifikant sein kann.

Teilreihe Gnutella BetweennessWeighted:

- **Simulationsreihe:** BetweennessWeighted auf Gnutella-Partitionen (alternativ „Gnutella BetweennessWeighted“)
- **Nummer:** 9C
- **Abhängigkeit:** 7
- **Zweck:** Vergleich voriger Ergebnisse mit BetweennessWeighted und BetweennessWeightedComplexMemory mit verschiedenem PriorPartnerFactor
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen der verschiedenen Algorithmen BetweennessWeighted (A) und BetweennessWeightedComplexMemory (B) auf jeweils gleichen Graphen
  - Insgesamt 50 verschiedene Graphen als unterschiedliche Netzwerkpartitionen
  - Für B PriorPartnerFactor aus $[0.1, 0.3, 0.5]$ und jeweils keine Gewichtung auf Community-Strukturen
  - Ausführen von 20 Wiederholungen pro Graph

  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 4000 Ergebnisse, 1000 für A und 3000 für B (1000 für jeden Faktor)

- **Annahme:** Hier ist entsprechend der vorigen Ergebnisse zu erwarten, dass die beste Leistung erreicht wird. Das BetweennessWeighted-Verfahren sollte selbst auf diesen Graphen, die keine klaren Gruppenstrukturen aufweisen, eine Leistungssteigerung erzielen. Durch Kombination mit der ComplexMemory-Logik sollte eine weitere Optimierung erfolgen.

Teilreihe Gnutella CommunityBased:

- **Simulationsreihe:** Community-Based Gossiping auf Gnutella-Partitionen (alternativ „Gnutella CommunityBased“)
- **Nummer:** 9D
- **Abhängigkeit:** 8
- **Zweck:** Vergleich voriger Ergebnisse mit CommunityBased
- **Aufbau:**
  - Zwei Simulationsressourcen: Ausführen des CommunityBased-Algorithmus nach [108]
  - Insgesamt 50 verschiedene Graphen als unterschiedliche Netzwerkpartitionen
  - Ausführen von 10 Wiederholungen pro Graph
  - Festgesetzte Knotenwerte zur besseren Vergleichbarkeit der Ergebnisse
  - Insgesamt 500 Ergebnisse
- **Annahme:** Die Performance sollte zwischen dem WeightedFactor- und CommunityProbabilities-Algorithmus liegen, also ebenfalls sehr nahe am Baseline-Verfahren.

7.2.9.2 Ergebnisse

Im Anhang D.9.1 sind die gemittelten Graphmetriken der Gnutella-Netzwerkpartitionen ausgelagert. Die zugehörigen Messwerte sind im Kapitel D.9.2 aufgeführt. Anhand der Darstellungen können Einblicke in die Performance der verschiedenen Algorithmen gewonnen werden. Die Ergebnisse sind in kompakter Form in Abbildung 7.15 dargestellt. Dabei wird ein Vergleich zwischen der Performance der verschiedenen Algorithmen gezogen. Für den WeightedFactor-Algorithmus wurde hierbei nur das Resultat des Faktors mit dem besten Ergebnis visualisiert. Man kann erkennen, dass die WeightedFactor-, CommunityBased- und CommunityProbabilities-Algorithmen nahezu identische Leistungen im Vergleich zum Standard-Verfahren aufweisen. Bei allen Verfahren kommt das Gossiping hier nach etwa 22 Runden zur Konvergenz. Dabei belaufen sich Abweichungen auf maximal 2%. Der BetweennessWeighted-Algorithmus erreicht ein wesentlich besseres Ergebnis. Hier werden zwischen 18 und 19 Runden benötigt bis eine Konvergenz erreicht wird, was einer Einsparung von 16% entspricht. Die Ergebnisse zeigen, dass das CommunityProbabilities-Verfahren durch die Integration der ComplexMemory-Logik erheblich verbessert werden kann. Durch das Hinzufügen der Speicherlogik sinkt die Anzahl an benötigten Runden auf knapp unter 14, somit steigt die Optimierung bis auf 37%. Das beste Ergebnis liefert jedoch das BetweennessWeightedComplexMemory-Verfahren. Hier wird die Anzahl an benötigten Runden bis auf etwa 13 reduziert, wobei eine Beschleunigung von 41% gegenüber dem Baseline-Verfahren erzielt wird.

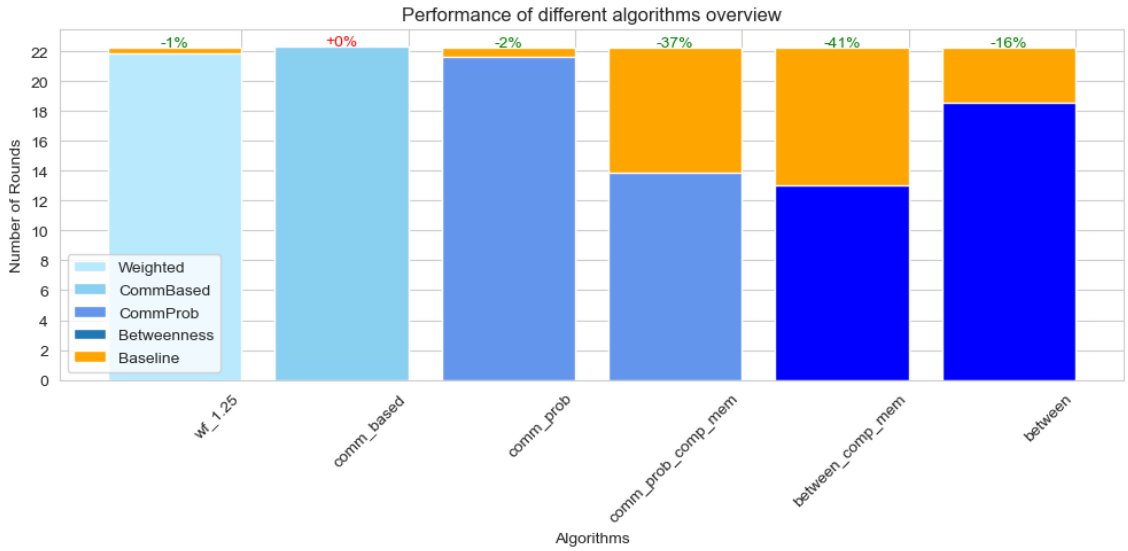


Abbildung 7.15: Auswertung Simulationsreihe 9

7.2.9.3 Schlussfolgerungen

Die Ergebnisse zeigen, dass die WeightedFactor-, CommunityBased- und CommunityProbabilities-Algorithmen keine wesentliche Beschleunigung gegenüber dem Baseline-Verfahren erreichen. Zu Beginn wurde von einer geringfügigen Verbesserung der Konvergenzgeschwindigkeit ausgegangen, jedoch fiel das tatsächliche Ergebnis schlechter aus als angenommen. Überraschend positiv war hingegen der Effekt der Memory-Logik. Durch Hinzufügen dieser konnte die Performance des CommunityProbabilities-Verfahren wesentlich beschleunigt werden. Konkret wurde eine Optimierung um zusätzliche 35% erreicht. Die anfänglichen Einschätzungen gingen nicht von einer solch signifikanten Steigerung aus. Den Annahmen entsprach die Performance des BetweennessWeighted-Algorithmus, welcher bereits bei den vorangehenden Testreihen die besten Ergebnisse erzielt. Er erreichte auch in dieser Simulationsreihe sowohl mit als auch ohne Speicherlogik die besten Resultate. Zudem fiel auch hier das Ergebnis mit Speicherlogik wiederum deutlich besser aus. Während das Verfahren ohne Speicherlogik eine Beschleunigung von 16% erzielte, konnte die Konvergenzzeit durch Hinzufügen dieser um insgesamt 41% reduziert werden. Die Performance der Algorithmen kann anhand der Komplexität der verwendeten Gewichte erklärt werden. Hierbei verwenden die WeightedFactor-, CommunityBased- und CommunityProbabilities-Verfahren lediglich Wissen über Community-Strukturen. Nur das CommunityProbabilities-Verfahren bevorzugt darüber hinaus auch indirekt Hub-Knoten. Jedoch ist auch dies für die Gnutella-Partitionen irrelevant, da es sich hierbei um keine skalenfreien Netzwerke handelt. Der BetweennessWeighted-Algorithmus hingegen berücksichtigt die Zentralität. Dadurch kann er auch in Netzwerken ohne modulare Strukturen zur Beschleunigung der Konvergenz eingesetzt werden. Des Weiteren konnte durch die Simulationsreihe der Nachweis der Begrenztheit der Aussagekraft der Modularitätsmetrik erfolgen. Aufgrund der Art der Berechnung weisen Graphen mit wenigen Kanten ein stets ein hohes Maß an Modularität auf. Insbesondere bei besonders spärlicher Konnektivität erreichen selbst Netzwerke mit sehr schwacher bis keiner Ausprägung von Clusterstrukturen hohe Modularitätswerte. Würde man nur die Modularitätswerte betrachten, so könnte man bei den Gnutella-Partitionen somit auch von hochmodularen Graphen sprechen. Ist man mit dem Verfahren des Aufbaus der P2P-Netzwerke bekannt, so widerspricht dies der Annahme. In diesen Extremfällen erweist sich die Modularität

als irreführend. Zudem kann diese Problematik durch das Hinzuziehen von weiteren Metriken, wie zum Beispiel dem Clustering-Koeffizienten, aufgezeigt werden. Dieser ist bei den Gnutella-Graphen mit einem gerundeten Durchschnittswert von 0.01 sehr niedrig. Im Vergleich dazu hat die gleiche Metrik bei den hochmodularen skalenfreien Netzwerken einen Wert von 0.14. Ein weiterer Grund für die langsame Konvergenzgeschwindigkeit sind die durchschnittlichen Pfadlängen. Diese sind im Vergleich zu den skalenfreien Graphen sehr hoch. Für die Gnutella-Partitionen beträgt die durchschnittliche Pfadlänge über 12, während sie bei den hochmodularen skalenfreien Graphen nur knapp über 5 liegt. Außerdem verlangsamt das Fehlen von Hub-Strukturen die Ausbreitung von Informationen. Der Grund dafür ist, dass diese durch die gewichteten Algorithmen als zentrale Umschlagpunkte fungieren und periphere Knoten so schnell an Informationen gelangen. Dies liegt daran, dass diese Hub-Strukturen in gewichteten Algorithmen als zentrale Vermittlungspunkte dienen und somit periphere Knoten schnell mit Informationen versorgen. Die besprochenen Eigenschaften der Gnutella-Partitionen können bei Betrachtung der Graphvisualisierung in Abbildung 7.16 nachvollzogen werden.

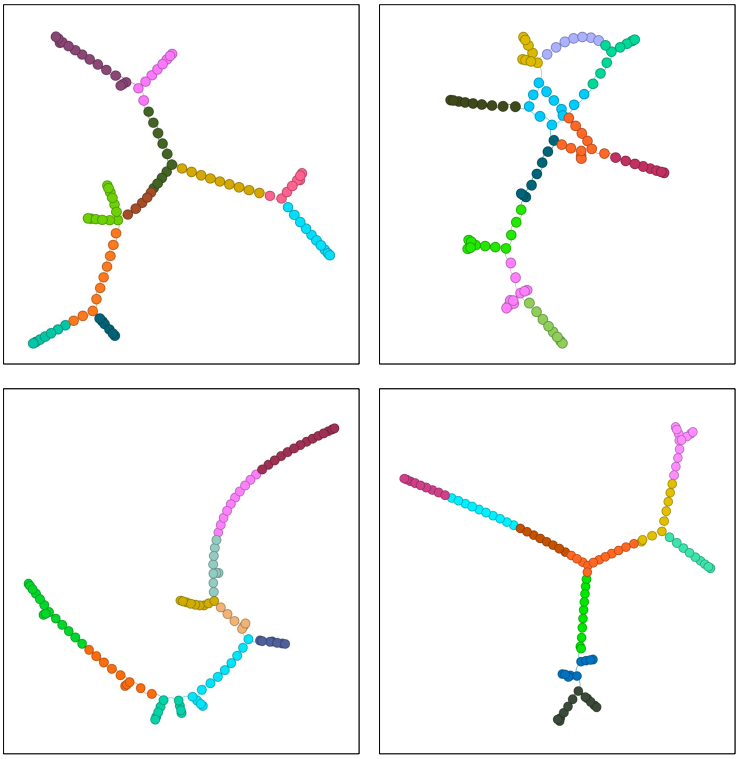


Abbildung 7.16: Beispiele Gnutella-Partitionen mit 100 Knoten

7.2.10 Zusammenfassung aller Ergebnisse

Ziel der Thesis bestand darin, Gossip-Algorithmen durch Gewichtung zu verbessern. In diesem Zusammenhang wurden verschiedene Typen von Netzwerken analysiert, um festzustellen, welche das größte Optimierungspotential bieten. Dabei wurden verschiedene Arten von Netzwerken untersucht, darunter modulare, skalenfreie und auch reale Netze. Die Ergebnisse zeigten, dass auf hochmodularen skalenfreien Netzwerken die größte Performancesteigerung erreicht werden kann.

Nachdem die Netzwerke mit dem höchsten Optimierungspotential identifiziert wurden, erfolgte die Auswertung von verschiedenen Algorithmen mit unterschiedlicher Komplexität. Hierbei zeigte sich, dass naive Ansätze meist weniger effizient sind. Zudem

erreichten Algorithmen, die eine Gewichtung nur anhand von Community-Strukturen durchführen, vergleichsweise schlechte Ergebnisse. Im Gegensatz dazu erzielten die Verfahren, die auch die Zentralität und die Hub-Strukturen der Graphen berücksichtigten, wesentlich bessere Resultate. Dabei erreichte der BetweennessWeighted-Algorithmus die schnellste Konvergenz. Dieser führt eine Gewichtung nach der Zentralität durch, also unter Berücksichtigung der Konnektivität, Nähe und Verflechtung der Knoten. Die Auswertung der Algorithmen anhand realer Netzwerkgraphen verdeutlichte, dass dieser Algorithmus auch in unmodularen Netzwerken effektiv angewandt werden kann. Dies unterscheidet ihn von den simpleren Ansätzen, die ausschließlich auf Community- und Hub-Strukturen zur Gewichtung basieren, da diese in derartigen Netzwerken wenig effizient sind.

Des Weiteren wurde festgestellt, dass die Integration von Memory-Logik in Gossip-Algorithmen die Konvergenzzeiten erheblich verbessert. Diese kann zusätzlich zu einer Gewichtung verwendet werden, um redundante Kommunikation zu minimieren. Der Einsatz der Speicherlogik kann unabhängig von der spezifischen Netzwerkstruktur erfolgen. Darüber hinaus kann sie mit beliebigen Verfahren kombiniert werden, um zusätzliche Beschleunigungseffekte zu erzielen.

Man kann zusammenfassen, dass die Wahl des optimalen Gossiping-Verfahrens von den individuellen Anforderungen abhängt. Hierbei sind die verfügbaren Informationen über die globale Netzwerkstruktur maßgeblich für die Entscheidung. In bestimmten Anwendungsszenarien kann es sinnvoll sein, auch weniger leistungsfähige Algorithmen einzusetzen, da sie weniger komplexe Gewichtungsberechnungen erfordern. Dies unterstreicht die Wichtigkeit von anpassbaren und flexiblen Gossiping-Verfahren, um den jeweiligen Bedürfnissen gerecht zu werden.

8 Zusammenfassung und Ausblick

Es folgt die Zusammenfassung der Thesis, wo die wichtigsten Punkte nochmals aufgeführt werden. Ebenfalls wird hier ein Ausblick gegeben, der sinnvolle Erweiterungen der Simulationsumgebung aufzeigt. Darüber hinaus erfolgt die Darstellung weiterer Evaluationsmöglichkeiten.

8.1 Zusammenfassung

Mit dem Wandel von zentralisierten Infrastrukturen zu hochgradig verteilten Systemen haben unstrukturierte Netzwerke stark an Bedeutung gewonnen. Unstrukturierte Netzwerke sind oft dezentral organisiert und verfügen über keine festen Hierarchien oder zentrale Kontrollinstanzen. In dezentralen Umgebungen entstehen darüber hinaus komplizierte und oft unvorhersehbare Netzwerktopologien. Beispiele sind dabei Peer-to-Peer-Netzwerke, verteilten Systeme und Blockchain-Technologien. In diesen Netzen wird häufig Gossiping zur Informationsverbreitung eingesetzt, da es effizient, skalierbar und robust ist. Hierbei werden Informationen verteilt beziehungsweise aggregiert, indem ein regelmäßiger Austausch zwischen zufällig ausgewählten Netzwerkteilnehmern durchgeführt wird. Durch die Zufälligkeit der Partnerwahl können jedoch bei bestimmten Netztopologien Performanceprobleme entstehen.

Der Schwerpunkt dieser Thesis liegt auf der Optimierung von Gossiping-Verfahren durch Gewichtung basierend auf topologischen Informationen. Dazu wurde eine Simulationsumgebung für ein Kubernetes-Cluster entwickelt, die auf einem Kubernetes Operator basiert. Der Operator realisiert die Orchestrierung komplexer Gossiping-Simulationen mit variablen Netzwerken und Algorithmen. Hier erfolgt die Verwaltung der Simulationen und Graphen über Kubernetes-Objekte, welche flexibel über Konfigurationsdateien definiert werden können. Zur Erzeugung von Netzwerkgraphen kann ein CLI-Tool verwendet werden. Infolgedessen kann die Erzeugung, Überwachung und Auswertung von einzelnen Simulationen sowie von Serien erfolgen. Dadurch wird die Evaluation verschiedener Netzwerke, Algorithmen und deren Konfigurationen möglich.

Es wurden verschiedene gewichtete Algorithmen mit unterschiedlicher Komplexität entworfen. Im ersten Schritt erfolgte die Entwicklung und Evaluation des einfachen WeightedFactor- sowie des dynamischen CommunityProbabilities-Algorithmus. Darüber hinaus wurden drei kompliziertere gewichtete Ansätze erprobt. Diese beruhen auf der Betweenness Centrality, der Eigenvektorzentralität sowie auf dem Authoritätsmaß der Hub-Scores. Jedes dieser Verfahren wurde mit einer rudimentären und einer komplexen Speicherlogik ergänzt. Ziel war es hierbei, eine weitere Optimierung der Algorithmen zu ermöglichen. Verschiedene Simulationen wurden durchgeführt, um die Verfahren untereinander und mit dem Baseline-Verfahren zu vergleichen.

Die Ergebnisse zeigen, dass je nach Graphtopologie starke Optimierungsmöglichkeiten bestehen. Der Höhepunkt der Performancesteigerung wurde bei hochmodularen skalenfreien Graphen erreicht. Hier ermöglicht der Einsatz von Wissen über Community-Zugehörigkeiten eine deutliche Verbesserung der Konvergenzzeit. Es konnte festgestellt werden, dass die Verwendung der Speicherlogik die Konvergenz ebenfalls stark beschleunigt. Außerdem wurde die Wichtigkeit der Zentralität sowie von Autoritätsmaßen deut-

lich. Die fortgeschrittenen Algorithmen erreichten die besten Leistungen, wobei das Verfahren basierend auf der Betweenness Centrality besonders auffiel. Selbst ohne Speicherlogik konnte hier eine Einsparung von 69% der benötigten Runden erreicht werden, während mit Speicherlogik sogar 72% weniger Runden benötigt wurden. Konkret sank die Gesamtrundenzahl im Mittel von etwa 18 bis auf unter 5. Dies ist für die betrachteten Netzwerke sehr nah an einer optimalen Informationsverbreitung.

Die Anwendbarkeit der Algorithmen wurde nicht nur auf synthetischen Netzwerken getestet, sondern es erfolgte auch die Evaluation anhand eines realen Netzwerks. Dabei wurden ausgewählte Verfahren auf Partitionen des Gnutella-Peer-to-Peer-Netzwerks simuliert. Trotz der zufälligen Natur des Netzwerks und der entsprechenden Abwesenheit von Community-Strukturen wurden Verbesserung der Konvergenzzeit erzielt. Der BetweennessWeighted-Algorithmus mit ComplexMemory-Erweiterung benötigte im Durchschnitt 41% weniger Runden.

Trotz der besseren Leistungen bestimmter Algorithmen muss der spezifische Anwendungsfall bei der Auswahl berücksichtigt werden. Je nach Algorithmus müssen verschiedene Daten zur Gewichtung gewonnen werden, wobei die Berechnung dieser Informationen unterschiedlich komplex ist. Zur Vereinfachung wurden in dieser Arbeit globale Verfahren zur Erhebung genutzt, in der Praxis müssen jedoch dezentrale Verfahren eingesetzt werden. Die Performance dieser kann die Leistung bei einer praktischen Umsetzung stark beeinflussen. Dezentrale Algorithmen können zum Beispiel in Bezug auf Laufzeit und Genauigkeit von den globalen Verfahren abweichen. Hier muss eine ergänzende Bewertung durchgeführt werden, um für den jeweiligen Anwendungsfall die beste Auswahl zu treffen.

Zusammenfassend kann festgestellt werden, dass Gossip-Algorithmen eine schnellere Konvergenz aufweisen, wenn sie auf die Struktur des untersuchten Netzwerks abgestimmt sind. Eine solche Abstimmung kann je nach dem konkreten Anwendungsfall variieren. Es wurde beobachtet, dass eine schnellere Konvergenz in hochmodularen Netzwerken durch die Gewichtung nach Community-Zugehörigkeit erreicht wird. Die Gewichtung nach Zentralität erzielt noch bessere Ergebnisse und kann für fast alle Netzwerktypen verwendet werden. Die erforderlichen Gewichtungsdaten können je nach Einsatzszenario flexibel erfasst werden, ohne die Anwendung unnötig zu verkomplizieren. Frühere Forschungsarbeiten haben gezeigt, dass beispielsweise eine Ermittlung von topologischen Informationen mit Hilfe von Synchronisationsprotokollen erfolgen kann. Zur Optimierung kann außerdem eine Gedächtnislogik verwendet werden, welche Gewichte basierend auf erfolgten Kommunikationen anpasst. Durch Kombination der Speicherlogik mit einer zum Anwendungsfall passenden Gewichtungsverteilung kann die kürzeste Konvergenzzeit erreicht werden.

8.2 Ausblick

Im Folgenden wird nun ein Ausblick über potentielle Verbesserungen und weitere relevante Evaluationen gegeben. Insbesondere wäre die Durchführung von Untersuchungen anderer Graphtypen sinnvoll, um Erkenntnisse für andere Anwendungsszenarien zu erlangen. Hierzu müssten zusätzliche Simulationen und dementsprechend auch weiterführende Evaluationen durchgeführt werden. Ebenfalls würden sich Analysen bezüglich der Korrelation von bestimmten Grapheigenschaften und der Konvergenzzeit anbieten. So wäre es möglich universelle Zusammenhänge festzustellen, welche auf beliebigen Netzwerken anwendbar sind. Zudem könnten weitere reale Netzwerke analysiert werden, um ein aussagekräftigeres Ergebnis zu erreichen. Eine fortführende Untersuchung wäre hierbei die Ausführung von größeren Simulationen mit wesentlich mehr Knoten. Dadurch

könnte die Skalierbarkeit der verbesserten Algorithmen geprüft werden. Außerdem wäre die Erforschung zusätzlicher Algorithmen von Bedeutung. Es ist denkbar, dass alternative Gewichtungsverfahren für andere Anwendungsfälle bessere Ergebnisse erzielen.

Eine Weiterentwicklungsmöglichkeit der Simulationsumgebung besteht in der Grapherzeugungsroutine. Hier könnte das CLI-Tool durch einen Web-Service ersetzt werden. Ein Service stellt eine komplexere Schnittstelle dar, durch eine weitere Automatisierung der Workflows erfolgen kann. Die Anlage von Graph- und Simulationsressourcen könnte hierbei in den Prozess eingebunden werden. Dazu müsste eine grafische Oberfläche entworfen werden, die eine Bedienung des Services zur Grapherzeugung ermöglicht. Der Nutzer würde über diese die Ressourcen direkt spezifieren und im Cluster anlegen können. Dadurch müsste er nur noch eine einzelne Interaktion mit dem System durchführen. Ein weiterer Vorteil ist, dass durch Verwendung eines Services die Grapherzeugung skalierbar wird.

Aufschlussreich wäre auch eine Evaluation der dezentralen Verfahren zur Gewinnung der Gewichte. Hier müsste eine praktische Integration der Verfahren in die Simulationsumgebung erfolgen. Beispiele beinhalten das Synchronisationsprotokoll für die Community-Strukturen beziehungsweise die Algorithmen für die dezentrale Bestimmung der Zentralitäts- und Autoritätsmaße. Ziel wäre es, herauszufinden, wie viel Performance bei welchen Verfahren verloren geht gegenüber den globalen Verfahren. Auf Basis dieser Erkenntnisse könnten die Algorithmen neu bewertet werden.

Literatur

[1] *About storage drivers | Docker Documentation*. https://docs.docker.com/storage/storagedriver/. (accessed on 2021-06-04).

[2] Avi Alkalay. *Object storage benefits, myths and options - Cloud computing news*. https://www.ibm.com/blogs/cloud-computing/2017/02/01/object-storage-benefits-myths-and-options/. (accessed on 2021-07-05). Feb. 2017.

[3] André Allavena, Alan Demers und John E. Hopcroft. „Correctness of a gossip based membership protocol“. In: *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005. DOI: 10.1145/1073814.1073871.

[4] Lina Altoaimy, Arwa Alromih, Shiroq Al-Megren, Ghada Al-Hudhud, Heba Kurdi und Kamal Youcef-Toumi. „Context-Aware Gossip-Based Protocol for Internet of Things Applications“. In: *Sensors* 18.7 (2018), S. 2233. DOI: 10.3390/s18072233.

[5] Owe Axelsson und Igor Kaporin. In: *Numerical Algorithms* 25.1/4 (2000), S. 1–22. DOI: 10.1023/a:1016694031362.

[6] S BOCCALETTI, V LATORA, Y MORENO, M CHAVEZ und D HWANG. „Complex networks: Structure and dynamics“. In: *Physics Reports* 424.4-5 (2006), S. 175–308. DOI: 10.1016/j.physrep.2005.10.009.

[7] Brenda Baker und Robert Shostak. „Gossips and telephones“. In: *Discrete Mathematics* 2.3 (1972), S. 191–193. DOI: 10.1016/0012-365x(72)90001-5.

[8] Luca Baldesi, Athina Markopoulou und Carter T. Buttsc. „Spectral Graph Forge: A Framework for Generating Synthetic Graphs With a Target Modularity“. In: *IEEE/ACM Transactions on Networking* 27.5 (2019), S. 2125–2136. DOI: 10.1109/tnet.2019.2940377.

[9] A.-L. Barabási und R. Albert. „Emergence of Scaling in Random Networks“. In: *The Structure and Dynamics of Networks*. Princeton University Press, 2011, S. 349–352. DOI: 10.1515/9781400841356.349.

[10] Mariano G Beiró, Sebastián P Grynberg und J Ignacio Alvarez-Hamelin. „Router-level community structure of the Internet Autonomous Systems“. In: *EPJ Data Science* 4.1 (2015). DOI: 10.1140/epjds/s13688-015-0048-y.

[11] F. Benezit, A. G. Dimakis, P. Thiran und M. Vetterli. „Order-Optimal Consensus through Randomized Path Averaging“. In: (19. Feb. 2008). arXiv: 0802.2587v1 [cs.IT].

[12] Florence Benezit, Vincent Blondel, Patrick Thiran, John Tsitsiklis und Martin Vetterli. „Weighted Gossip: Distributed Averaging using non-doubly stochastic matrices“. In: *2010 IEEE International Symposium on Information Theory*. IEEE, 2010. DOI: 10.1109/isit.2010.5513273.

[13] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu und Yaron Minsky. „Bimodal multicast“. In: *ACM Transactions on Computer Systems* 17.2 (1999), S. 41–88. DOI: 10.1145/312203.312207.

[14] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte und Etienne Lefebvre. „Fast unfolding of communities in large networks“. In: *J. Stat. Mech. (2008) P10008* (4. März 2008). DOI: 10.1088/1742-5468/2008/10/P10008. arXiv: 0803.0476v2 [physics.soc-ph].

[15] James Bowes. *Level Triggering and Reconciliation in Kubernetes | HackerNoon*. Online Article. (Accessed on 2023-06-06). Mai 2020. URL: https://hackernoon.com/level-triggering-and-reconciliation-in-kubernetes-1f17fe30333d.

[16] S. Boyd, A. Ghosh, B. Prabbakar und D. Shah. „Gossip algorithms: design, analysis and applications“. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* IEEE. DOI: 10.1109/infcom.2005.1498447.

[17] S. Boyd, A. Ghosh, B. Prabhakar und D. Shah. „Randomized gossip algorithms“. In: *IEEE Transactions on Information Theory* 52.6 (2006), S. 2508–2530. DOI: 10.1109/tit.2006.874516.

[18] M. Branco. „Topology-aware Gossip Dissemination for Large-scale Datacenters“. Magisterarb. Instituto Superior Técnico, Universidade Técnica de Lisboa, Okt. 2012.

[19] M. Branco, J. Leitã0 und L. Rodrigues. „Bounded Gossip: A Gossip Protocol for Large-Scale Datacenters“. In: *Proceedings of the 28st ACM Symposium on Applied Computing (SAC'13)*. ACM. Coimbra, Portugal, März 2013.

[20] Arnaud Browet, Pierre-Antoine Absil und Paul Van Dooren. „Community Detection for Hierarchical Image Segmentation“. In: Mai 2011, S. 358–371. ISBN: 978-3-642-21072-3. DOI: 10.1007/978-3-642-21073-0_32.

[21] G.E. Clarke. *CompTIA Network+ Certification Study Guide, Fourth Edition*. McGraw-Hill Education, 2009. ISBN: 9780071615396. URL: https://books.google.de/books?id=PRwzUDySqW0C.

[22] Aaron Clauset, M. E. J. Newman und Cristopher Moore. „Finding community structure in very large networks“. In: *Phys. Rev. E 70, 066111 (2004)* (9. Aug. 2004). DOI: 10.1103/PhysRevE.70.066111. arXiv: cond-mat/0408187v2 [cond-mat.stat-mech].

[23] *Cloud Controller Manager | Kubernetes*. https://kubernetes.io/docs/concepts/architecture/cloud-controller/. (Accessed on 2023-05-31).

[24] *Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)*. https://aws.amazon.com/s3/. (accessed on 2021-06-04).

[25] *Cluster Networking | Kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/networking/. (accessed on 2021-06-04).

[26] *ConfigMaps | Kubernetes*. https://kubernetes.io/docs/concepts/configuration/configmap/. (accessed on 2021-06-04).

[27] *Configure Service Accounts for Pods | Kubernetes*. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/. (Accessed on 2023-06-06).

[28] *Container Runtime Interface (CRI) | Kubernetes*. https://kubernetes.io/docs/concepts/architecture/cri/. (Accessed on 2023-07-14).

[29] *Container Runtimes | Kubernetes*. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. (Accessed on 2023-05-31).

[30] *Containers | Kubernetes*. https://kubernetes.io/docs/concepts/containers/. (accessed on 2021-06-04).

[31] *Controllers | Kubernetes*. https://kubernetes.io/docs/concepts/architecture/controller/. (Accessed on 2023-05-31).

[32] DataReportal. *Digital Around the World – Global Digital Insights*. (Accessed on 2023-05-25). 2023. URL: https://datareportal.com/global-digital-overview.

[33] Alan Demers, Dan Greene, Carl Hauser, Wes Irish und John Larson. „Epidemic algorithms for replicated database maintenance“. In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing - PODC '87*. ACM Press, 1987. DOI: 10.1145/41840.41841.

[34] *Deployments | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. (accessed on 2021-06-04).

[35] Alexandros G. Dimakis, Anand D. Sarwate und Martin J. Wainwright. „Geographic Gossip: Efficient Averaging for Sensor Networks“. In: (25. Sep. 2007). DOI: 10.1109/TSP.2007.908946. arXiv: 0709.3921v1 [cs.IT].

[36] *Docker overview | Docker Documentation*. https://docs.docker.com/get-started/overview/. (accessed on 2021-06-04).

[37] *Dockerfile reference | Docker Documentation*. https://docs.docker.com/engine/reference/builder/. (accessed on 2021-06-04).

[38] P. Erdös und A. Rényi. „On Random Graphs I“. In: *Publicationes Mathematicae Debrecen* 6 (1959), S. 290.

[39] P.T. Eugster, R. Guerraoui, S.B. Handurukande, A.-M. Kermarrec und P. Kouznetsov. „Lightweight probabilistic broadcast“. In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE Comput. Soc. DOI: 10.1109/dsn.2001.941428.

[40] Sonja Filiposka und Carlos Juiz. „Community-based complex cloud data center“. In: *Physica A: Statistical Mechanics and its Applications* 419 (2015), S. 356–372. DOI: 10.1016/j.physa.2014.10.017.

[41] George H. L. Fletcher, Hardik A. Sheth und Katy Börner. „Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance“. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, S. 14–27. DOI: 10.1007/11574781_2.

[42] Santo Fortunato. „Community detection in graphs“. In: *Physics Reports* 486.3-5 (2010), S. 75–174. DOI: 10.1016/j.physrep.2009.11.002.

[43] Cloud Native Computing Foundation. „CNCF Operator White Paper“. In: (Juli 2021). (Accessed on 2023-06-15). URL: https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf.

[44] Valerio Freschi, Emanuele Lattanzi und Alessandro Bogliolo. „Accelerating distributed averaging in sensor networks: Randomized gossip over virtual coordinates“. In: *2016 IEEE Sensors Applications Symposium (SAS)*. IEEE, 2016. DOI: 10.1109/sas.2016.7479874.

[45] Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides. *Design Patterns*. Prentice Hall, 1. Dez. 1995. ISBN: 0201633612. URL: https://www.ebook.de/de/product/3236753/erich_gamma_richard_helm_ralph_e_johnson_john_vlissides_design_patterns.html.

[46] *GanttProject - Free Project Management Application*. https://www.ganttproject.biz/. (accessed on 2021-06-04).

[47] E. N. Gilbert. „Random Graphs“. In: *The Annals of Mathematical Statistics* 30.4 (1959), S. 1141–1144. DOI: 10.1214/aoms/1177706098.

[48] M. Girvan und M. E. J. Newman. „Community structure in social and biological networks“. In: *Proceedings of the National Academy of Sciences* 99.12 (2002), S. 7821–7826. DOI: 10.1073/pnas.122653799.

[49] *GitHub - containerd/containerd: An open and reliable container runtime*. https://github.com/containerd/containerd. (Accessed on 2023-07-14).

[50] *GitHub - containernetworking/cni: Container Network Interface - networking for Linux containers*. https://github.com/containernetworking/cni. (Accessed on 2023-07-14).

[51] *GitHub - flannel-io/flannel: flannel is a network fabric for containers, designed for Kubernetes*. https://github.com/flannel-io/flannel. (Accessed on 2023-07-14).

[52] *GitHub - opencontainers/runc: CLI tool for spawning and running containers according to the OCI specification*. https://github.com/opencontainers/runc. (Accessed on 2023-07-14).

[53] *Günstiges Cloud Hosting - Hetzner Online GmbH*. https://www.hetzner.com/de/cloud. (Accessed on 2023-07-16).

[54] I. Gupta, A.M. Kermarrec und A.J. Ganesh. „Efficient epidemic-style protocols for reliable and scalable multicast“. In: *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* IEEE Comput. Soc. DOI: 10.1109/reldis.2002.1180187.

[55] Aric A. Hagberg, Daniel A. Schult und Pieter J. Swart. „Exploring Network Structure, Dynamics, and Function using NetworkX“. In: *Proceedings of the 7th Python in Science Conference*. Hrsg. von Gaël Varoquaux, Travis Vaught und Jarrod Millman. Pasadena, CA USA, 2008, S. 11 –15.

[56] Sandra M. Hedetniemi, Stephen T. Hedetniemi und Arthur L. Liestman. „A survey of gossiping and broadcasting in communication networks“. In: *Networks* 18.4 (1988), S. 319–349. DOI: 10.1002/net.3230180406.

[57] Petter Holme und Beom Jun Kim. „Growing scale-free networks with tunable clustering“. In: *Phys. Rev. E* 65 (2 2002), S. 026107. DOI: 10.1103/PhysRevE.65.026107. URL: https://link.aps.org/doi/10.1103/PhysRevE.65.026107.

[58] V. H. Hovnanyan, H. E. Nahapetyan, Su. S. Poghosyan und V. S. Poghosyan. „Tighter Upper Bounds for the Minimum Number of Calls and Rigorous Minimal Time in Fault-Tolerant Gossip Schemes“. In: (20. Apr. 2013). arXiv: 1304.5633v1 [cs.IT].

[59] *Introduction to gRPC | gRPC*. https://grpc.io/docs/what-is-grpc/introduction/. (Accessed on 2023-06-15).

[60] *JSON*. https://www.json.org/json-en.html. (accessed on 2021-07-05).

[61] Márk Jelasity, Alberto Montresor und Ozalp Babaoglu. „Gossip-based aggregation in large dynamic networks“. In: *ACM Transactions on Computer Systems* 23.3 (2005), S. 219–252. DOI: 10.1145/1082469.1082470.

[62] Michael Kaufmann und Katharina Zweig. „Modeling and Designing Real–World Networks“. In: *Algorithmics of Large and Complex Networks*. Springer Berlin Heidelberg, 2009, S. 359–379. DOI: 10.1007/978-3-642-02094-0_17.

[63] D. Kempe, A. Dobra und J. Gehrke. „Gossip-based computation of aggregate information“. In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE Computer. Soc. DOI: 10.1109/sfcs.2003.1238221.

[64] A.-M. Kermarrec, L. Massoulie und A.J. Ganesh. „Probabilistic reliable dissemination in large-scale systems“. In: *IEEE Transactions on Parallel and Distributed Systems* 14.3 (2003), S. 248–258. DOI: 10.1109/tpds.2003.1189583.

[65] Jon M. Kleinberg. „Authoritative sources in a hyperlinked environment“. In: *Journal of the ACM* 46.5 (1999), S. 604–632. DOI: 10.1145/324133.324140.

[66] *Kopf: Kubernetes Operators Framework — Kopf documentation*. https://kopf.readthedocs.io/en/stable/. (Accessed on 2023-06-15).

[67] *Kubernetes Components | Kubernetes*. https://kubernetes.io/docs/concepts/overview/components/. (Accessed on 2023-05-31).

[68] *Kubernetes: Production-Grade Container Scheduling and Management*. https://github.com/kubernetes/kubernetes. (accessed on 2021-06-04).

[69] Kasun Indrasiri & Danesh Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. 1. Aufl. O'Reilly Media, Inc., 2020. ISBN: 9781492058304.

[70] Roger Labahn. „Kernels of minimum size gossip schemes“. In: *Discrete Mathematics* 143.1-3 (1995), S. 99–139. DOI: 10.1016/0012-365x(94)00031-d.

[71] *Labels and Selectors | Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/. (accessed on 2021-06-04).

[72] Lun Li, David Alderson, John C. Doyle und Walter Willinger. „Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications“. In: *Internet Mathematics* 2.4 (2005), S. 431–523. DOI: 10.1080/15427951.2005.10129111.

[73] Malyszkz. *Diagram of different network topologies.* Wikipedia. (Accessed on 2023-06-01). URL: https://wwww.wikipedia.org/wiki/Network_topology#/media/File:NetworkTopologies.svg.

[74] Angélica Sousa da Mata. „Complex Networks: a Mini-review“. In: *Brazilian Journal of Physics* 50.5 (2020), S. 658–672. DOI: 10.1007/s13538-020-00772-9.

[75] Miguel Matos, António Sousa, José Pereira, Rui Oliveira, Eric Deliot und Paul Murray. „CLON: Overlay Networks and Gossip Protocols for Cloud Environments“. In: *On the Move to Meaningful Internet Systems: OTM 2009*. Springer Berlin Heidelberg, 2009, S. 549–566. DOI: 10.1007/978-3-642-05148-7_41.

[76] Stanley Milgram. *The small-world problem*. 1967. DOI: 10.1037/e400002009-005.

[77] *MinIO | High Performance, Kubernetes Native Object Storage*. https://min.io/. (accessed on 2021-06-04).

[78] *MinIO | The MinIO Quickstart Guide*. https://docs.min.io/docs/. (accessed on 2021-06-04).

[79] *MinIO Object Storage for Hybrid Cloud — MinIO Hybrid Cloud Documentation*. https://docs.min.io/minio/k8s/. (accessed on 2021-06-04).

[80] Yuchang Mo, Liudong Xing, Wenzhong Guo, Shaobin Cai, Zhao Zhang und Jianhui Jiang. „Reliability Analysis of IoT Networks with Community Structures“. In: *IEEE Transactions on Network Science and Engineering* 7.1 (2020), S. 304–315. DOI: 10.1109/tnse.2018.2869167.

[81] Decebal Constantin Mocanu, Georgios Exarchakos und Antonio Liotta. „Decentralized dynamic understanding of hidden relations in complex networks“. In: *Scientific Reports* 8.1 (2018). DOI: 10.1038/s41598-018-19356-4.

[82] *Namespaces | Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. (accessed on 2021-06-04).

[83] Ronak Nathani. *How a Kubernetes Pod Gets an IP Address*. Online Blog Post. (Accessed on 2023-07-14). Aug. 2020. URL: https://ronaknathani.com/blog/2020/08/how-a-kubernetes-pod-gets-an-ip-address/.

[84] *Network Data Repository | The First Interactive Network Data Repository*. https://networkrepository.com/. (Accessed on 2023-07-22).

[85] M. E. J. Newman. „Equivalence between modularity optimization and maximum likelihood methods for community detection“. In: *Physical Review E* 94.5 (2016). DOI: 10.1103/physreve.94.052315.

[86] M. E. J. Newman. „The Structure and Function of Complex Networks“. In: *SIAM Review* 45.2 (2003), S. 167–256. ISSN: 00361445. URL: http://www.jstor.org/stable/25054401 (besucht am 11. 06. 2023).

[87] *Nodes | Kubernetes*. https://kubernetes.io/docs/concepts/architecture/nodes/. (Accessed on 2023-05-31).

[88] *Nodes | Kubernetes*. https://kubernetes.io/docs/concepts/architecture/nodes/. (accessed on 2021-06-04).

[89] *Operator pattern | Kubernetes*. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. (Accessed on 2023-05-31).

[90] Lawrence Page, Sergey Brin, Rajeev Motwani und Terry Winograd. „The PageRank Citation Ranking : Bringing Order to the Web“. In: *The Web Conference*. 1999.

[91] Josiane Parreira. „Decentralized Link Analysis in Peer-to-Peer Web Search Networks“. In: (Jan. 2009).

[92] Ashish Patel. *Kubernetes — Architecture and Cluster Components Overview*. Devops Mojo. (Accessed on 2023-05-31). Aug. 2021. URL: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd.

[93] *Pods | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/. (accessed on 2021-06-04).

[94] Mason A. Porter, Jukka-Pekka Onnela und Peter J. Mucha. „Communities in Networks“. In: *Notices of the American Mathematical Society, Vol. 56, No. 9: 1082-1097, 1164-1166, 2009* (22. Feb. 2009). arXiv: 0902.3788v2 [physics.soc-ph].

[95] *Reference — NetworkX 3.1 documentation*. https://networkx.org/documentation/stable/reference/index.html. (Accessed on 2023-07-22).

[96] David Rensin. *Kubernetes*. City: O'Reilly Media, Inc, 2015. ISBN: 9781491931875.

[97] *ReplicaSet | Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. (Accessed on 2023-05-31).

[98] Mark Richards. *Software architecture patterns : understanding common architecture patterns and when to use them*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 9781491924242.

[99] Pratha Sah, Lisa O Singh, Aaron Clauset und Shweta Bansal. „Exploring community structure in biological networks with random graphs“. In: *BMC Bioinformatics* 15.1 (2014). DOI: 10.1186/1471-2105-15-220.

[100] Niklas Schütz. „Decentralized Community Detection in Unstructured Networks“. In: *Technical Reports of the System Technology Lab (STL) at htw saar* (Jan. 2023). ISSN: 2364-7167. URL: https://stl.htwsaar.de/tr/STL-TR-2023-01.pdf.

[101] *Secrets | Kubernetes*. https://kubernetes.io/docs/concepts/configuration/secret/. (accessed on 2021-06-04).

[102] *Service | Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/. (accessed on 2021-06-04).

[103] Devavrat Shah. „Gossip Algorithms“. In: *Foundations and Trends® in Networking* 3.1 (2007), S. 1–125. DOI: 10.1561/1300000014.

[104] Xuemin Shen, Heather Yu, John Buford und Mursalin Akon, Hrsg. *Handbook of Peer-to-Peer Networking*. Springer US, 2010. DOI: 10.1007/978-0-387-09751-0.

[105] Vikramjit Singh. „Decentralized community detection in unstructured networks by dynamic processes“. In: *Master Thesis at the University of Bonn* (Feb. 2016). URL: https://stl.htwsaar.de/tr/STL-TR-2023-01.pdf.

[106] Vikramjit Singh, Markus Esch und Ingo Scholtes. „Decentralized Cluster Detection in Distributed Systems Based on Self-Organized Synchronization“. In: *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2016. DOI: 10.1109/saso.2016.23.

[107] Satyajit Sinha. *State of IoT 2023: Number of connected IoT devices*. (Accessed on 2023-05-25). 2023. URL: https://iot-analytics.com/number-connected-iot-devices/#:~:text=The%20latest%20IoT%20Analytics%20%E2%80%9CState,to%2016%20billion%20active%20endpoints..

[108] Christel Sirocchi und Alessandro Bogliolo. „Community-Based Gossip Algorithm for Distributed Averaging“. In: *Distributed Applications and Interoperable Systems*. Hrsg. von Marta Patiño-Martínez und João Paulo. Cham: Springer Nature Switzerland, 2023, S. 37–53. ISBN: 978-3-031-35260-7.

[109] Michael Small, Yingying Li, Thomas Stemler und Kevin Judd. „Super-star networks: Growing optimal scale-free networks via likelihood“. In: (28. Mai 2013). arXiv: 1305.6429v3 [nlin.AO].

[110] Rajagopal Subramaniyan, Pirabhu Raman, Alan D. George und Matthew Radlinski. „GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems“. In: *Cluster Computing* 9.1 (2006), S. 101–120. DOI: 10.1007/s10586-006-4900-5.

[111] *The Official YAML Web Site*. https://yaml.org/. (accessed on 2021-06-04).

[112] S. Trajanovski, F.A. Kuipers, J. Martin-Hernández und P. Van Mieghem. „Generating graphs that approach a prescribed modularity“. In: *Computer Communications* 36.4 (2013), S. 363–372. DOI: 10.1016/j.comcom.2012.10.004.

[113] *Understanding Kubernetes Objects | Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. (accessed on 2021-06-04).

[114] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune und John Wilkes. „Large-scale cluster management at Google with Borg“. In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM, Apr. 2015. DOI: 10.1145/2741948.2741964.

[115] Xiao Fan Wang und Guanrong Chen. „Complex networks: Small-world, scale-free and beyond“. In: *IEEE Circuits and Systems Magazine* 3.1 (2003), S. 6–20. DOI: 10.1109/mcas.2003.1228503.

[116] *Was ist die 0/100-Methode? - Wissen kompakt - t2informatik*. https://t2informatik.de/wissen-kompakt/0-100-methode/. (accessed on 2021-08-18).

[117] Duncan J. Watts und Steven H. Strogatz. „Collective dynamics of 'small-world' networks“. In: *Nature* 393.6684 (1998), S. 440–442. DOI: 10.1038/30918.

[118] *Welcome!* | *minikube*. https://minikube.sigs.k8s.io/docs/. (accessed on 2021-07-05).

[119] *Welcome to powerlaw's documentation! — powerlaw 1.4.3 documentation*. https://pythonhosted.org/powerlaw/. (Accessed on 2023-07-22).

[120] *What is a Container?* | *App Containerization* | *Docker*. https://www.docker.com/resources/what-container. (accessed on 2021-06-04).

[121] Bobby Woolf. *How Operators work in Kubernetes* | *Red Hat Developer*. Online Article. (Accessed on 2023-06-06). 2021. URL: https://developers.redhat.com/articles/2021/06/22/kubernetes-operators-101-part-2-how-operators-work#the_structure_of_kubernetes_operators.

[122] Fetahi Wuhib, Rolf Stadler und Mike Spreitzer. „A Gossip Protocol for Dynamic Resource Management in Large Cloud Environments“. In: *IEEE Transactions on Network and Service Management* 9.2 (2012), S. 213–225. DOI: 10.1109/tnsm.2012.031512.110176.

[123] Ruiqi Yang. „Analysing the efficiency and robustness of gossip in different propagation processes with simulations“. In: *Journal of Physics: Conference Series* 1486.3 (2020), S. 032001. DOI: 10.1088/1742-6596/1486/3/032001.

[124] Lei Zhang und Wanqing Tu. „Six Degrees of Separation in Online Society“. In: (Jan. 2009).

[125] Katharina Zweig und Michael Kaufmann. „Decentralized Algorithms for Evaluating Centrality in Complex Network“. In: (Jan. 2003).

[126] *etcd*. https://etcd.io/. (Accessed on 2023-05-31).

[127] *kube-apiserver* | *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/. (Accessed on 2023-05-31).

[128] *kube-proxy* | *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. (Accessed on 2023-05-31).

[129] *kube-scheduler* | *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. (Accessed on 2023-05-31).

[130] *kubelet* | *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. (Accessed on 2023-05-31).

[131] *pygraphviz · PyPI*. https://pypi.org/project/pygraphviz/. (Accessed on 2023-06-15).

[132] *rabbanyk/FARZ: Benchmark Generator for Community Detection Validation*. https://github.com/rabbanyk/FARZ. (Accessed on 2023-06-15).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API	Application Programming Interface
AUC	Area under the Curve
AWS	Amazon Web Services
Amazon S3	Amazon Simple Storage Service
CLI	Command Line Interface
CNI	Container Networking Interface
CRD	Custom Resource Definition
CRI	Container Runtime Interface
DNS	Domain Name System
DSL	Distributed Systems Lab
GUI	Graphical User Interface
gRPC	Google Remote Procedure Call
HITS	Hyperlink-Induced Topic Search
HTTP	Hypertext Transfer Protocol
htw saar	Hochschule für Technik und Wirtschaft des Saarlandes
IDL	Interface Definition Language
IoT	Internet of Things
IP	Internet Protocol
ISP	Internet Service Provider
JSON	JavaScript Object Notation
k8s	Kubernetes
LAN	Local Area Network
LPA	Label Propagation Algorithm
NMI	Normalized Mutual Information
MFWF	Multi-factor weighting function
OCI	Open Container Initiative
PVC	Persistent Volume Claim
PV	Persistent Volume
RBAC	Role-Based Access Control
RGVC	Randomized Gossip over Virtual Coordinates
RPC	Remote Procedure Call
SDK	Software Development Kit
SGF	Spectral Graph Forge

SoC	Separation of Concerns
TCP	Transmission Control Protocol
TMGG	Tunable Modularity Graph Generator
UX	User Experience
WAN	Wide Area Network
YAML	YAML Ain't Markup Language

Anhang

A Weiterführende Informationen zur Analyse

A.1 Use Cases

Aktor - Administrator

- Netzwerke als Graphen generieren
- Graphressourcen erzeugen
- Graphen auswerten (Metriken, Visualisierungen)
- Simulationsressourcen anlegen
- Testreihen definieren
- Simulationen ausführen
- Simulationen überwachen
- Simulationen auswerten
- Vergleich verschiedener Algorithmen

Aktor - Entwickler

- Deployment und Skalierung über Kubernetes
- Neue Grapherzeugungsroutinen flexibel hinzufügen
- Neue Gossip-Algorithmen flexibel hinzufügen
- Dynamische Auswertungen erstellen (Berechnungen, Diagramme)

B Anhang Konzeption

B.1 Generierung von Graphen

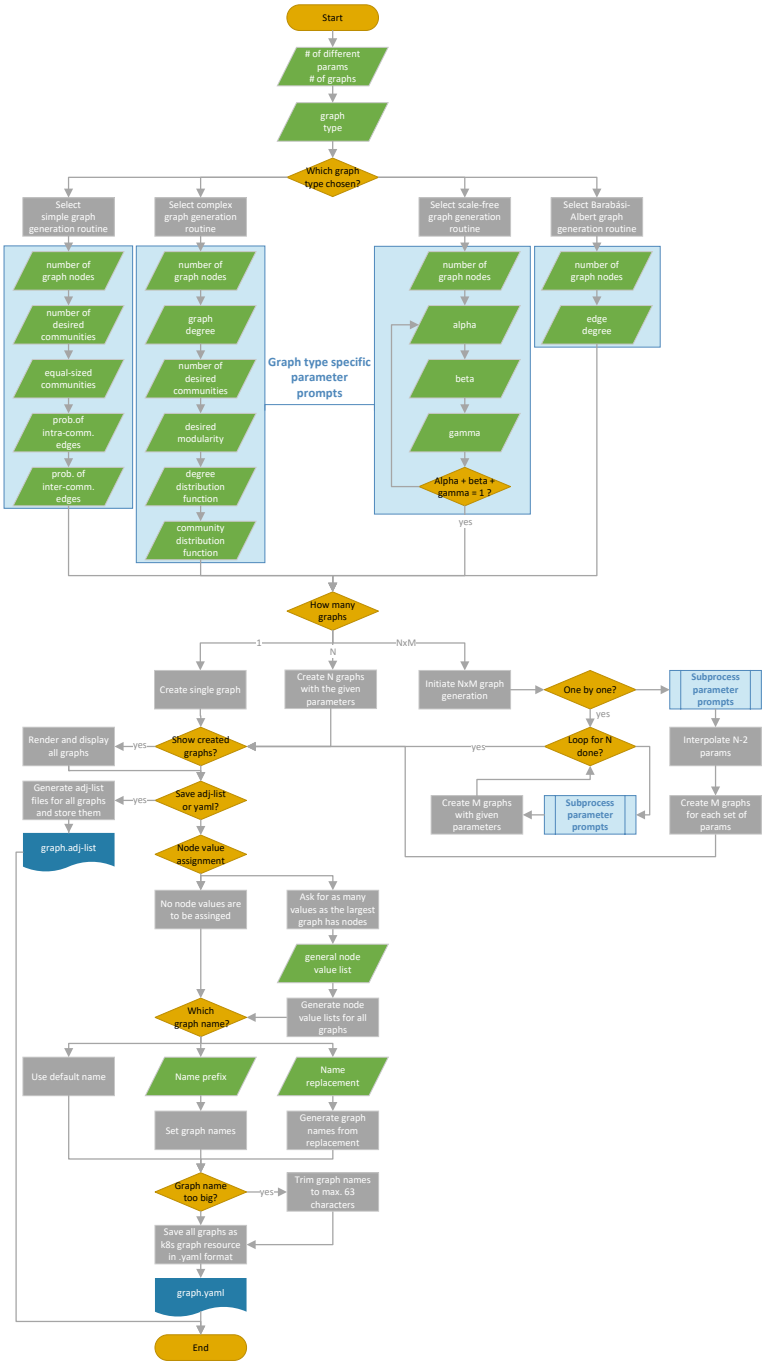


Abbildung B.1: Ablauf der Erzeugung von Netzwerkgraphen

B.2 Simulationsablauf

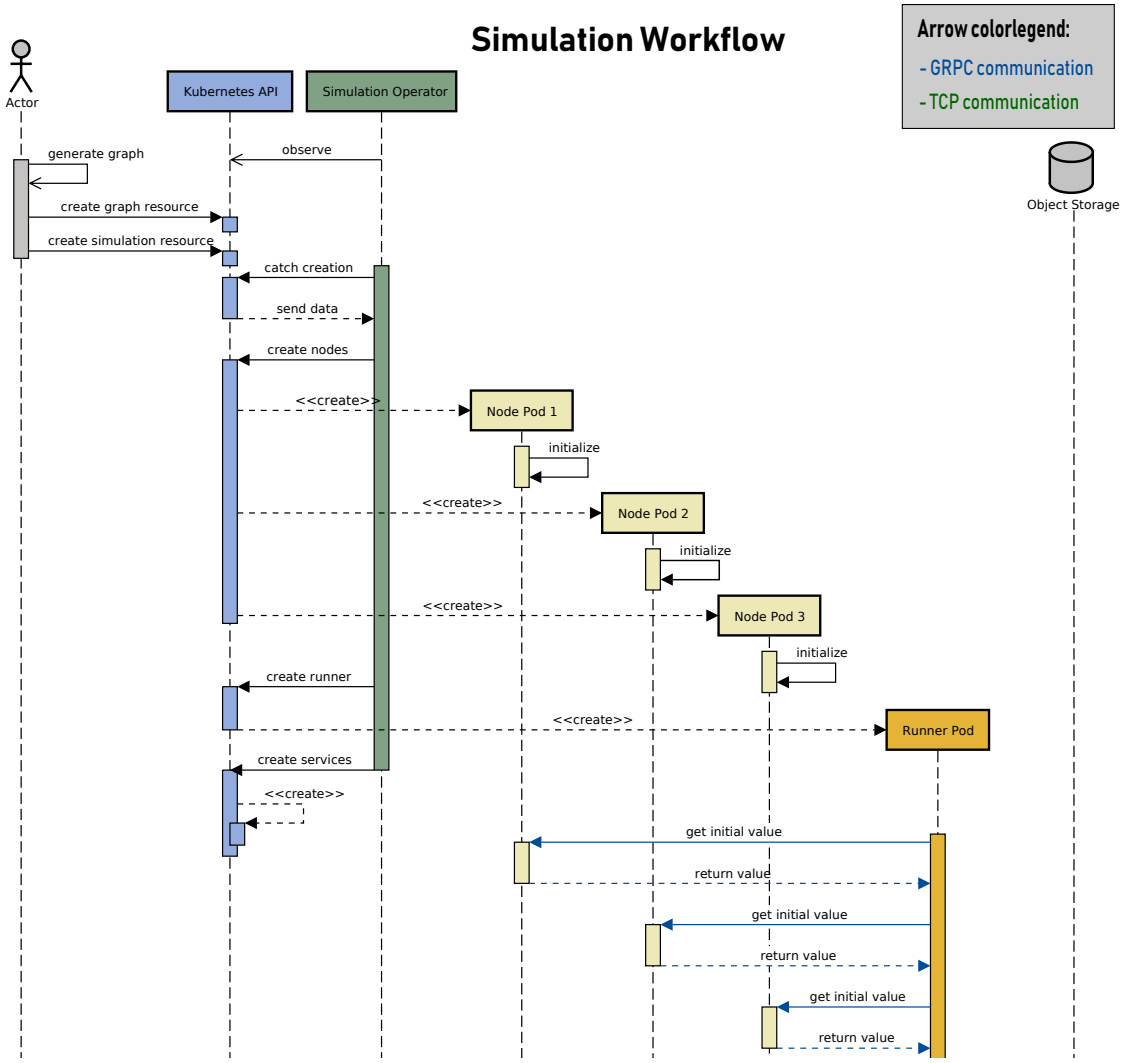


Abbildung B.2: Sequenzdiagramm Simulationsablauf Teil 1

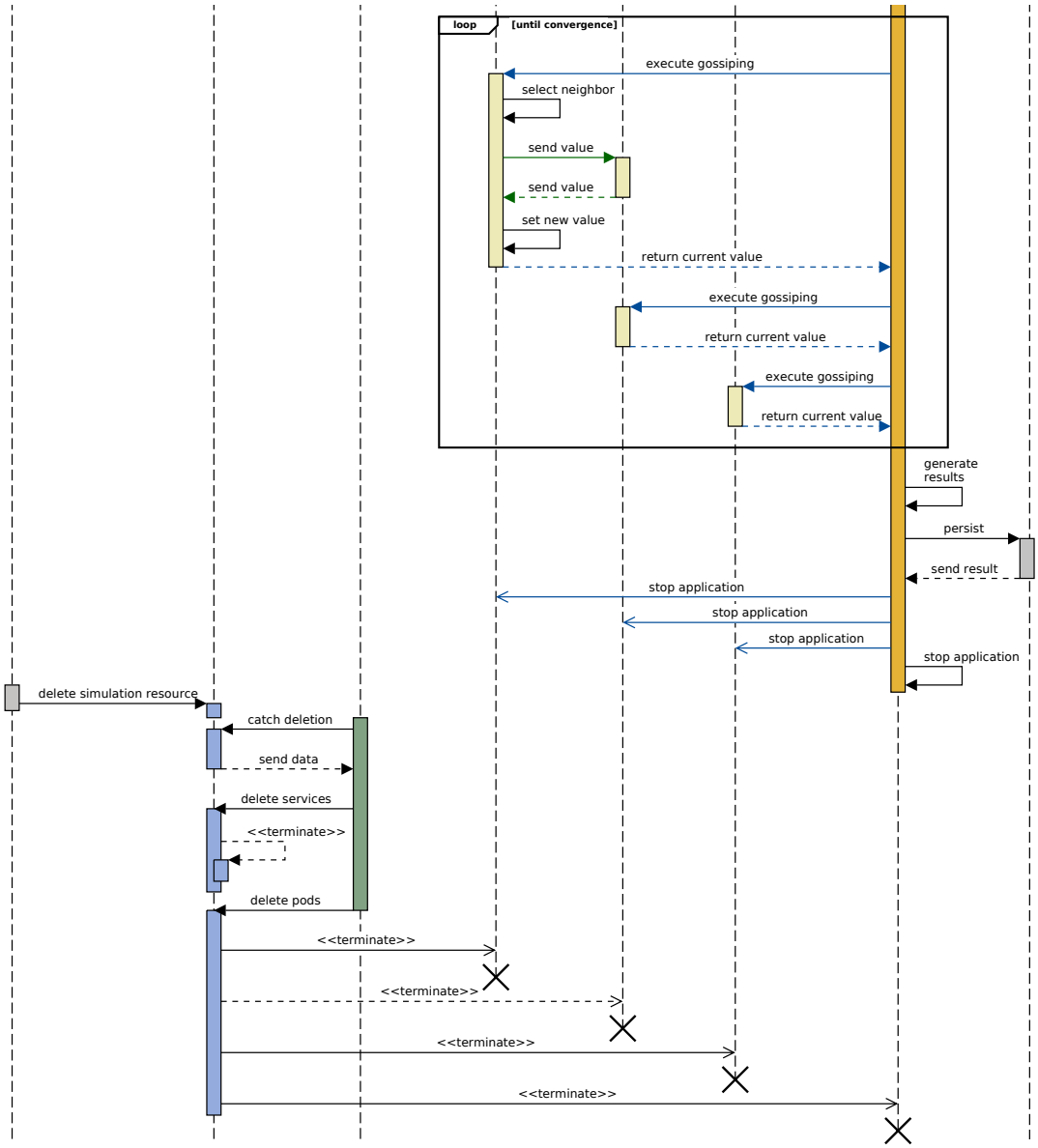


Abbildung B.3: Sequenzdiagramm Simulationsablauf Teil 2

C Anhang Evaluation - Simulationsumgebung

C.1 Anforderungserfüllung

Tabelle C.1: Erfüllung funktionaler Anforderungen Teil 1

Anforderung	**Umsetzung**
Must-Have	
Simulation von Gossiping	Paarweises rundebasiertes Gossiping zur Bestimmung eines netzwerkweiten Minimums wird simuliert.
Gewichtete Algorithmen basierend auf Netzwerktopologie	Vollständig erfüllt durch Implementierung von WeightedFactor-, CommunityProbabilities- sowie Betweenness-, Eigenvector- und HubScores-Algorithmen.
Simulation von Netzwerken basierend auf Graphen	Graphen können über Adjazenzlisten in Ressourcen angelegt werden. Netzwerk wird anhand der Graphstruktur erzeugt.
Abbildung verschiedener topologischer Eigenschaften	Über Grapherzeugungsroutine können verschiedene Graphen generiert werden. Beliebige Graphen können dann als Ressourcen angelegt werden.
Evaluierung verschiedener gewichteter Algorithmen	Testreihen zu den verschiedenen Algorithmen wurden durchgeführt und ausgewertet.
Vergleich mit Baseline	Bei jeder Auswertung wurden auch stets Vergleiche mit dem Baseline-Verfahren gezogen.
Dynamische Generierung von Graphen	Die Grapherzeugungsroutine unterstützt eine Vielzahl von Graphen und Parameter können dynamisch gesetzt werden.
Rundenbasierte Simulation	Der Simulation Runner steuert die Simulationsausführung und startet die Gossiping-Runden.
Dokumentation der Ergebnisse	Der Simulation Runner speichert alle Messergebnisse in roher und aggregierter Form in einem Object Storage.
Schlussfolgerungen und Vergleich mit aufgestellten Annahmen	Die Evaluation beinhaltet eine Auswertung der Ergebnisse und versucht Abweichungen von den anfänglichen Annahmen logisch zu begründen.

Tabelle C.2: Erfüllung funktionaler Anforderungen Teil 2

Anforderung	**Umsetzung**
Should-Have	
Gegenüberstellung der Anwendbarkeit der Algorithmen und ihrer Konfigurationen	Auf Grundlage der Erkenntnisse werden die Algorithmen final bewertet. Vor- und Nachteile werden diskutiert.
Nice-To-Have	
Erfassung und Auswertung genauer Messwerte aus umfangreichen Simulationen	Die meisten durchgeführten Simulationen hatten bereits einen sehr großen Umfang (tausende Messungen) und eine lange Ausführungsdauer.

Tabelle C.3: Erfüllung nichtfunktionaler Anforderungen

Anforderung	**Umsetzung**
Must-Have	
Simulation mit Kubernetes	Kubernetes wurde als Zielplattform für die Simulationsumgebung verwendet.
Nutzung eines Kubernetes Operators	Der Simulation Operator wurde entwickelt zur Verwaltung der Graph- und Simulationsressoucen.
Effizienter Einsatz der Cluster-Ressourcen	Cluster-Ressourcen werden durch genaue Steuerung optimal genutzt. Die Lastverteilung wird über die Knoten reguliert.
Trennung der Generierung von Graphen und der Simulationsdurchführung	Eine logische Trennung erfolgt durch die Separierung der Grapherzeugungsroutine und der Simulationssteuerung über den Operator im Kubernetes-Cluster.
Object Storage zur Datenspeicherung	Es wurde ein MinIO Object Storage zur Persistenz der Ergebnisdateien sowie Visualisierungen verwendet.
Should-Have	
Minimierung des administrativen Aufwands bei der Simulationsdurchführung	Der administrative Aufwand ist auf die Grapherzeugung und Simulationsdefinition begrenzt. Bis zum Abschluss der Simulation und der darauffolgenden Evaluation sind keine Eingriffe notwendig.
Nice-To-Have	
Einsatz einer Schnittstelle zur Generierung von Graphen	Es wurde eine simple Schnittstelle in Form eines CLI-Tools entwickelt.

D Anhang Evaluation - Simulationsreihen

D.1 Auswertung Simmulationsreihe 1

D.1.1 Gemittelte Graphmetriken

alpha	0.076820
assortativity	-0.269280
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.001000
averageClosenessCentrality	0.513100
averageDegreeCentrality	0.045720
averageEccentricity	1.998950
averageEigenvectorCentrality	0.024010
averageHubScore	0.001000
averageNeighborsDegree	282.984525
averagePageRank	0.001000
averagePathLength	1.954280
averageRichClubCoefficient	0.924475
averagenodeDegree	45.669600
beta	0.805010
density	0.045720
diameter	2.000000
edgeConnectivity	8.450000
edgeMultiplier	24.704465
estimatedPowerLawExponent	2.418025
gamma	0.118165
lowerBoundPowerLawRegion	22.750000
modularity	0.100015
nodeConnectivity	8.450000
nodeCount	1000.000000
numCommunities	7.900000
numEdges	22834.800000
overallAverageCommunityClustering	0.532690
overallStdevCommunityClustering	0.149140
stdevAuthorityScore	0.000870
stdevBetweennessCentrality	0.009140
stdevClosenessCentrality	0.034315
stdevCommunitySize	2.214000
stdevDegreeCentrality	0.081980
stdevEccentricity	0.027630
stdevEigenvectorCentrality	0.020580
stdevHubScore	0.000870
stdevNeighborsDegree	86.764485
stdevPageRank	0.001640
stdevRichClubCoefficient	0.209505
stdevNodeDegree	288.819400
transitivity	0.139765

Tabelle D.1: Graphmetriken aggregiert für Mod 0.1

alpha	0.077190
assortativity	-0.357010
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.001570
averageClosenessCentrality	0.396705
averageDegreeCentrality	0.006295
averageEccentricity	4.189350
averageEigenvectorCentrality	0.019115
averageHubScore	0.001000
averageNeighborsDegree	209.154300
averagePageRank	0.001000
averagePathLength	2.561695
averageRichClubCoefficient	0.923020
averagenodeDegree	6.281100
beta	0.841110
density	0.006295
diameter	5.600000
edgeConnectivity	1.000000
edgeMultiplier	1.444000
estimatedPowerLawExponent	2.415895
gamma	0.081695
lowerBoundPowerLawRegion	6.250000
modularity	0.308185
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	13.950000
numEdges	3140.550000
overallAverageCommunityClustering	0.442280
overallStdevCommunityClustering	0.404515
stdevAuthorityScore	0.001315
stdevBetweennessCentrality	0.017440
stdevClosenessCentrality	0.047810
stdevCommunitySize	3.846700
stdevDegreeCentrality	0.025625
stdevEccentricity	0.393350
stdevEigenvectorCentrality	0.025200
stdevHubScore	0.001315
stdevNeighborsDegree	139.403490
stdevPageRank	0.003850
stdevRichClubCoefficient	0.193100
stdevNodeDegree	288.819400
transitivity	0.040135

Tabelle D.2: Graphmetriken aggregiert für Mod 0.3

alpha	0.178985
assortativity	-0.271540
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.002160
averageClosenessCentrality	0.324975
averageDegreeCentrality	0.003795
averageEccentricity	5.569850
averageEigenvectorCentrality	0.017250
averageHubScore	0.001000
averageNeighborsDegree	107.291720
averagePageRank	0.001000
averagePathLength	3.152025
averageRichClubCoefficient	7.584635
averagenodeDegree	3.779600
beta	0.663280
density	0.003795
diameter	7.700000
edgeConnectivity	1.000000
estimatedPowerLawExponent	2.527195
gamma	0.157735
lowerBoundPowerLawRegion	5.950000
modularity	0.496620
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	18.750000
numEdges	1889.800000
overallAverageCommunityClustering	0.125205
overallStdevCommunityClustering	0.269135
stdevAuthorityScore	0.001520
stdevBetweennessCentrality	0.019615
stdevClosenessCentrality	0.048135
stdevCommunitySize	5.335175
stdevDegreeCentrality	0.015135
stdevEccentricity	0.591630
stdevEigenvectorCentrality	0.026510
stdevHubScore	0.001520
stdevNeighborsDegree	110.658285
stdevPageRank	0.003615
stdevRichClubCoefficient	5.845820
stdevNodeDegree	288.819400
transitivity	0.026110

Tabelle D.3: Graphmetriken aggregiert für Mod 0.5

assortativity	-0.106470
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.003955
averageClosenessCentrality	0.206570
averageDegreeCentrality	0.002760
averageEccentricity	8.293400
averageEigenvectorCentrality	0.013720
averageHubScore	0.001000
averageNeighborsDegree	10.813520
averagePageRank	0.001000
averagePathLength	4.944025
averageRichClubCoefficient	0.387200
averagenodeDegree	2.765700
density	0.002760
diameter	11.100000
edgeConnectivity	1.000000
estimatedPowerLawExponent	3.244930
lowerBoundPowerLawRegion	6.600000
modularity	0.699405
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	24.250000
numEdges	1382.850000
overallAverageCommunityClustering	0.009135
overallStdevCommunityClustering	0.080880
stdevAuthorityScore	0.002100
stdevBetweennessCentrality	0.016845
stdevClosenessCentrality	0.029765
stdevCommunitySize	7.054560
stdevDegreeCentrality	0.004095
stdevEccentricity	0.823085
stdevEigenvectorCentrality	0.028475
stdevHubScore	0.002100
stdevNeighborsDegree	12.177855
stdevPageRank	0.001260
stdevRichClubCoefficient	0.293190
stdevNodeDegree	288.819400
transitivity	0.006065
newEdges	1.390775

Tabelle D.4: Graphmetriken aggregiert für Mod 0.7

assortativity	-0.106985
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.005315
averageClosenessCentrality	0.166075
averageDegreeCentrality	0.002000
averageEccentricity	11.952950
averageEigenvectorCentrality	0.009920
averageHubScore	0.001000
averageNeighborsDegree	13.924335
averagePageRank	0.001000
averagePathLength	6.303900
averageRichClubCoefficient	0.259400
averagenodeDegree	1.998000
density	0.002000
diameter	16.450000
edgeConnectivity	1.000000
estimatedPowerLawExponent	3.286035
lowerBoundPowerLawRegion	6.550000
modularity	0.911445
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	41.550000
numEdges	999.000000
overallAverageCommunityClustering	0.000000
overallStdevCommunityClustering	0.000000
stdevAuthorityScore	0.007715
stdevBetweennessCentrality	0.036080
stdevClosenessCentrality	0.033985
stdevCommunitySize	12.070525
stdevDegreeCentrality	0.004145
stdevEccentricity	1.416800
stdevEigenvectorCentrality	0.030045
stdevHubScore	0.012000
stdevNeighborsDegree	24.863040
stdevPageRank	0.001740
stdevRichClubCoefficient	0.230330
stdevNodeDegree	288.819400
transitivity	0.000000
newEdges	1.000000
triangleProbability	0.745995

Tabelle D.5: Graphmetriken aggregiert für Mod 0.9

D.1.2 Ergebnisse

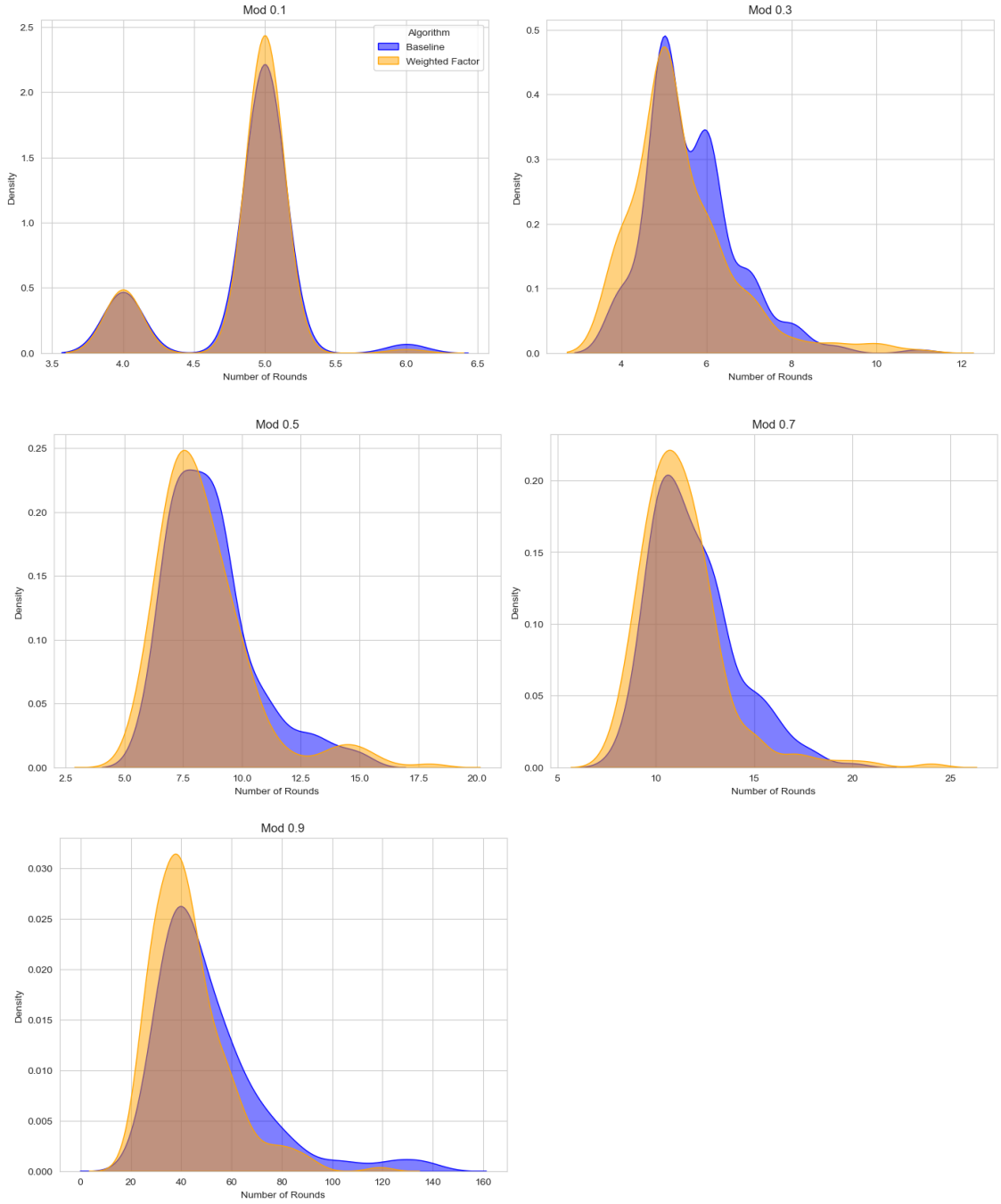


Abbildung D.1: Detaillierte Auswertung nach Modularität für Simulationsreihe 1

D.2 Auswertung Simmulationsreihe 2

D.2.1 Gemittelte Graphmetriken

assortativity	-0.131395
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.006135
averageClosenessCentrality	0.145950
averageCommunitySize	29.277205
averageDegreeCentrality	0.002000
averageEccentricity	12.844650
averageEigenvectorCentrality	0.009440
averageHubScore	0.001000
averageNeighborsDegree	9.011880
averageNodeDegree	1.998000
averagePageRank	0.001000
averagePathLength	7.122825
averageRichClubCoefficient	0.221760
density	0.002000
diameter	17.350000
edgeConnectivity	1.000000
estimatedPowerLawExponent	3.216950
lowerBoundPowerLawRegion	4.950000
modularity	0.925065
newEdges	1.000000
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	34.350000
numEdges	999.000000
overallAverageCommunityClustering	0.000000
overallStdevCommunityClustering	0.000000
stdevAuthorityScore	0.014390
stdevBetweennessCentrality	0.038365
stdevClosenessCentrality	0.027860
stdevCommunitySize	18.114270
stdevDegreeCentrality	0.003170
stdevEccentricity	1.548580
stdevEigenvectorCentrality	0.030180
stdevHubScore	0.005080
stdevNeighborsDegree	11.591205
stdevNodeDegree	288.819400
stdevPageRank	0.001340
stdevRichClubCoefficient	0.209620
transitivity	0.000000

Tabelle D.6: Graphmetriken aggregiert für NewEdges 1.0

assortativity	-0.101860
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.003715
averageClosenessCentrality	0.216635
averageCommunitySize	43.622040
averageDegreeCentrality	0.003005
averageEccentricity	7.736050
averageEigenvectorCentrality	0.013975
averageHubScore	0.001000
averageNeighborsDegree	11.465770
averageNodeDegree	3.003200
averagePageRank	0.001000
averagePathLength	4.706200
averageRichClubCoefficient	0.361320
density	0.003005
diameter	10.100000
edgeConnectivity	1.000000
estimatedPowerLawExponent	3.282220
lowerBoundPowerLawRegion	6.300000
modularity	0.654630
newEdges	1.500000
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	23.000000
numEdges	1501.600000
overallAverageCommunityClustering	0.016385
overallStdevCommunityClustering	0.111135
stdevAuthorityScore	0.002060
stdevBetweennessCentrality	0.015645
stdevClosenessCentrality	0.030030
stdevCommunitySize	15.885005
stdevDegreeCentrality	0.004420
stdevEccentricity	0.778600
stdevEigenvectorCentrality	0.028355
stdevHubScore	0.002060
stdevNeighborsDegree	12.528385
stdevNodeDegree	288.819400
stdevPageRank	0.001245
stdevRichClubCoefficient	0.303110
transitivity	0.008370

Tabelle D.7: Graphmetriken aggregiert für NewEdges 1.5

assortativity	-0.100830
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.003080
averageClosenessCentrality	0.248355
averageCommunitySize	51.271605
averageDegreeCentrality	0.004000
averageEccentricity	6.012300
averageEigenvectorCentrality	0.016890
averageHubScore	0.001000
averageNeighborsDegree	12.545030
averageNodeDegree	3.992000
averagePageRank	0.001000
averagePathLength	4.075000
averageRichClubCoefficient	0.350300
density	0.004000
diameter	7.200000
edgeConnectivity	1.950000
estimatedPowerLawExponent	3.095270
lowerBoundPowerLawRegion	6.450000
modularity	0.523565
newEdges	2.000000
nodeConnectivity	1.950000
nodeCount	1000.000000
numCommunities	19.650000
numEdges	1996.000000
overallAverageCommunityClustering	0.026905
overallStdevCommunityClustering	0.140830
stdevAuthorityScore	0.001590
stdevBetweennessCentrality	0.012495
stdevClosenessCentrality	0.027620
stdevCommunitySize	20.542675
stdevDegreeCentrality	0.005235
stdevEccentricity	0.551425
stdevEigenvectorCentrality	0.026745
stdevHubScore	0.001590
stdevNeighborsDegree	10.118595
stdevNodeDegree	288.819400
stdevPageRank	0.001120
stdevRichClubCoefficient	0.261700
transitivity	0.009865

Tabelle D.8: Graphmetriken aggregiert für NewEdges 2.0

assortativity	-0.086140
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.002740
averageClosenessCentrality	0.271495
averageCommunitySize	58.066315
averageDegreeCentrality	0.004985
averageEccentricity	5.444650
averageEigenvectorCentrality	0.018225
averageHubScore	0.001000
averageNeighborsDegree	14.463765
averageNodeDegree	4.977900
averagePageRank	0.001000
averagePathLength	3.723145
averageRichClubCoefficient	0.357605
density	0.004985
diameter	6.800000
edgeConnectivity	2.000000
estimatedPowerLawExponent	3.125345
lowerBoundPowerLawRegion	7.350000
modularity	0.444990
newEdges	2.500000
nodeConnectivity	2.000000
nodeCount	1000.000000
numCommunities	17.300000
numEdges	2488.950000
overallAverageCommunityClustering	0.030710
overallStdevCommunityClustering	0.122800
stdevAuthorityScore	0.001415
stdevBetweennessCentrality	0.010735
stdevClosenessCentrality	0.028625
stdevCommunitySize	26.209225
stdevDegreeCentrality	0.006260
stdevEccentricity	0.522070
stdevEigenvectorCentrality	0.025840
stdevHubScore	0.001415
stdevNeighborsDegree	10.704790
stdevNodeDegree	288.819400
stdevPageRank	0.001065
stdevRichClubCoefficient	0.264710
transitivity	0.014645

Tabelle D.9: Graphmetriken aggregiert für NewEdges 2.5

assortativity	-0.079060
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.002480
averageClosenessCentrality	0.289430
averageCommunitySize	63.396955
averageDegreeCentrality	0.006000
averageEccentricity	4.952850
averageEigenvectorCentrality	0.019360
averageHubScore	0.001000
averageNeighborsDegree	15.720330
averageNodeDegree	5.982000
averagePageRank	0.001000
averagePathLength	3.484780
averageRichClubCoefficient	0.441215
density	0.006000
diameter	6.000000
edgeConnectivity	3.000000
estimatedPowerLawExponent	3.101595
lowerBoundPowerLawRegion	7.100000
modularity	0.388250
newEdges	3.000000
nodeConnectivity	3.000000
nodeCount	1000.000000
numCommunities	15.850000
numEdges	2991.000000
overallAverageCommunityClustering	0.031840
overallStdevCommunityClustering	0.087290
stdevAuthorityScore	0.001290
stdevBetweennessCentrality	0.009370
stdevClosenessCentrality	0.027360
stdevCommunitySize	27.006960
stdevDegreeCentrality	0.007030
stdevEccentricity	0.370740
stdevEigenvectorCentrality	0.025015
stdevHubScore	0.001290
stdevNeighborsDegree	9.663260
stdevNodeDegree	288.819400
stdevPageRank	0.001005
stdevRichClubCoefficient	0.321795
transitivity	0.017595

Tabelle D.10: Graphmetriken aggregiert für NewEdges 3.0

assortativity	-0.070690
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.002330
averageClosenessCentrality	0.303290
averageCommunitySize	66.775990
averageDegreeCentrality	0.006995
averageEccentricity	4.738700
averageEigenvectorCentrality	0.020195
averageHubScore	0.001000
averageNeighborsDegree	17.078430
averageNodeDegree	6.981400
averagePageRank	0.001000
averagePathLength	3.323170
averageRichClubCoefficient	0.390025
density	0.006995
diameter	5.650000
edgeConnectivity	2.900000
estimatedPowerLawExponent	3.098160
lowerBoundPowerLawRegion	8.250000
modularity	0.349255
newEdges	3.500000
nodeConnectivity	2.900000
nodeCount	1000.000000
numCommunities	15.100000
numEdges	3490.700000
overallAverageCommunityClustering	0.033385
overallStdevCommunityClustering	0.077115
stdevAuthorityScore	0.001195
stdevBetweennessCentrality	0.008195
stdevClosenessCentrality	0.027610
stdevCommunitySize	27.417970
stdevDegreeCentrality	0.007805
stdevEccentricity	0.443920
stdevEigenvectorCentrality	0.024345
stdevHubScore	0.001195
stdevNeighborsDegree	9.255180
stdevNodeDegree	288.819400
stdevPageRank	0.000950
stdevRichClubCoefficient	0.291285
transitivity	0.021415

Tabelle D.11: Graphmetriken aggregiert für NewEdges 3.5

assortativity	-0.066905
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.002180
averageClosenessCentrality	0.316965
averageCommunitySize	72.489615
averageDegreeCentrality	0.008000
averageEccentricity	4.412500
averageEigenvectorCentrality	0.020765
averageHubScore	0.001000
averageNeighborsDegree	18.866190
averageNodeDegree	7.968000
averagePageRank	0.001000
averagePathLength	3.177895
averageRichClubCoefficient	0.472895
density	0.008000
diameter	5.000000
edgeConnectivity	3.700000
estimatedPowerLawExponent	3.112860
lowerBoundPowerLawRegion	8.050000
modularity	0.319950
newEdges	4.000000
nodeConnectivity	3.700000
nodeCount	1000.000000
numCommunities	13.900000
numEdges	3984.000000
overallAverageCommunityClustering	0.035430
overallStdevCommunityClustering	0.065875
stdevAuthorityScore	0.001150
stdevBetweennessCentrality	0.007835
stdevClosenessCentrality	0.027725
stdevCommunitySize	27.068620
stdevDegreeCentrality	0.008740
stdevEccentricity	0.497795
stdevEigenvectorCentrality	0.023845
stdevHubScore	0.001150
stdevNeighborsDegree	9.556020
stdevNodeDegree	288.819400
stdevPageRank	0.000930
stdevRichClubCoefficient	0.325100
transitivity	0.023805

Tabelle D.12: Graphmetriken aggregiert für NewEdges 4.0

D.2.2 Ergebnisse

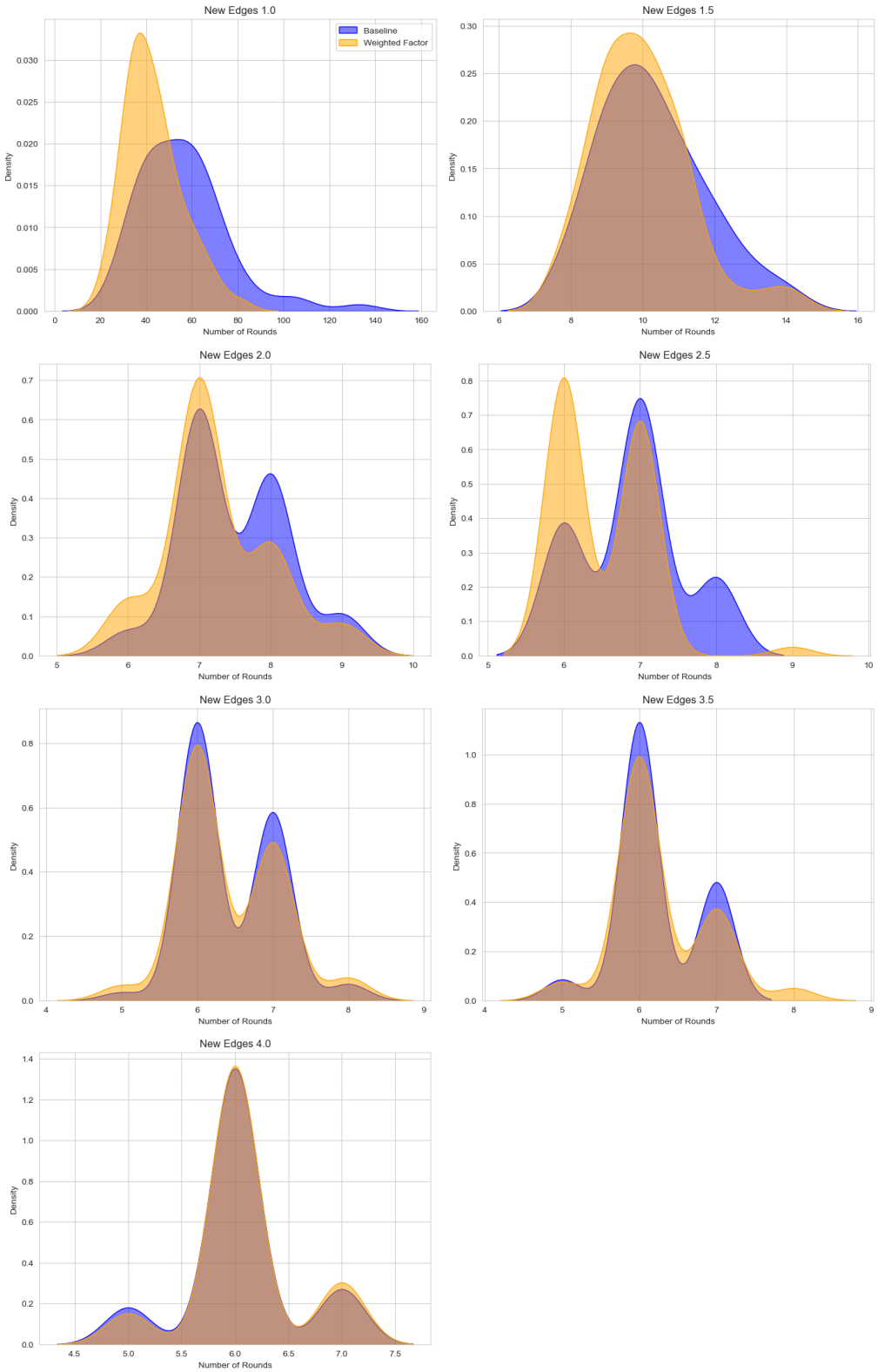


Abbildung D.2: Detaillierte Auswertung nach Konnektivität für Simulationsreihe 2

D.3 Auswertung Simmulationsreihe 3

D.3.1 Gemittelte Graphmetriken

assortativity	-0.042985
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.009475
averageClosenessCentrality	0.097970
averageCommunitySize	100.000000
averageDegreeCentrality	0.004000
averageEccentricity	15.531925
averageEigenvectorCentrality	0.008997
averageHubScore	0.001000
averageNeighborsDegree	4.987488
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	10.455435
averageRichClubCoefficient	0.014478
communityCount	10.000000
density	0.004000
diameter	19.100000
edgeConnectivity	1.000000
estimatedPowerLawExponent	13.294158
lowerBoundPowerLawRegion	7.525000
modularity	0.893310
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	10.000000
numEdges	2000.000000
numInterCommunityEdges	10.000000
numIntraCommunityEdges	1990.000000
overallAverageCommunityClustering	0.035083
overallStdevCommunityClustering	0.105423
stdevAuthorityScore	0.003445
stdevBetweennessCentrality	0.029875
stdevClosenessCentrality	0.016125
stdevCommunitySize	0.986282
stdevDegreeCentrality	0.001938
stdevEccentricity	1.772690
stdevEigenvectorCentrality	0.030297
stdevHubScore	0.003445
stdevNeighborsDegree	1.153895
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.017430
transitivity	0.035145
intra_inter_ratio	199.000000
inter_intra_ratio	0.005025

Tabelle D.13: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1990/10

assortativity	-0.037645
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004980
averageClosenessCentrality	0.168663
averageCommunitySize	100.000000
averageDegreeCentrality	0.004000
averageEccentricity	9.478800
averageEigenvectorCentrality	0.022063
averageHubScore	0.001000
averageNeighborsDegree	4.980627
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	5.971770
averageRichClubCoefficient	0.013065
communityCount	10.000000
density	0.004000
diameter	11.800000
edgeConnectivity	1.000000
estimatedPowerLawExponent	18.157020
lowerBoundPowerLawRegion	8.175000
modularity	0.770417
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	10.000000
numEdges	2000.000000
numInterCommunityEdges	200.000000
numIntraCommunityEdges	1800.000000
overallAverageCommunityClustering	0.024415
overallStdevCommunityClustering	0.089350
stdevAuthorityScore	0.001045
stdevBetweennessCentrality	0.005632
stdevClosenessCentrality	0.014017
stdevCommunitySize	5.191542
stdevDegreeCentrality	0.001922
stdevEccentricity	0.725757
stdevEigenvectorCentrality	0.022395
stdevHubScore	0.001045
stdevNeighborsDegree	1.151693
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.014977
transitivity	0.024477
intra_inter_ratio	9.000000
inter_intra_ratio	0.111111

Tabelle D.14: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1800/200

assortativity	-0.037315
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004487
averageClosenessCentrality	0.183980
averageCommunitySize	93.854897
averageDegreeCentrality	0.004000
averageEccentricity	8.706050
averageEigenvectorCentrality	0.024412
averageHubScore	0.001000
averageNeighborsDegree	4.999190
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	5.474948
averageRichClubCoefficient	0.021490
communityCount	10.000000
density	0.004000
diameter	11.075000
edgeConnectivity	1.000000
estimatedPowerLawExponent	15.984350
lowerBoundPowerLawRegion	7.675000
modularity	0.650973
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	10.700000
numEdges	2000.000000
numInterCommunityEdges	400.000000
numIntraCommunityEdges	1600.000000
overallAverageCommunityClustering	0.016215
overallStdevCommunityClustering	0.074540
stdevAuthorityScore	0.000823
stdevBetweennessCentrality	0.004612
stdevClosenessCentrality	0.015335
stdevCommunitySize	21.150210
stdevDegreeCentrality	0.001942
stdevEccentricity	0.677815
stdevEigenvectorCentrality	0.020040
stdevHubScore	0.000823
stdevNeighborsDegree	1.173655
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.034270
transitivity	0.015738
intra_inter_ratio	4.000000
inter_intra_ratio	0.250000

Tabelle D.15: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1600/400

assortativity	-0.030377
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004295
averageClosenessCentrality	0.191138
averageCommunitySize	58.398522
averageDegreeCentrality	0.004000
averageEccentricity	8.393600
averageEigenvectorCentrality	0.025130
averageHubScore	0.001000
averageNeighborsDegree	5.009465
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	5.271002
averageRichClubCoefficient	0.012065
communityCount	10.000000
density	0.004000
diameter	10.575000
edgeConnectivity	1.000000
estimatedPowerLawExponent	14.186293
lowerBoundPowerLawRegion	8.025000
modularity	0.567128
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	17.225000
numEdges	2000.000000
numInterCommunityEdges	600.000000
numIntraCommunityEdges	1400.000000
overallAverageCommunityClustering	0.010198
overallStdevCommunityClustering	0.060030
stdevAuthorityScore	0.000768
stdevBetweennessCentrality	0.004337
stdevClosenessCentrality	0.016088
stdevCommunitySize	22.878623
stdevDegreeCentrality	0.001960
stdevEccentricity	0.653353
stdevEigenvectorCentrality	0.019183
stdevHubScore	0.000768
stdevNeighborsDegree	1.183490
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.012048
transitivity	0.009903
intra_inter_ratio	2.333333
intra_inter_ratio	0.428571

Tabelle D.16: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1400/600

assortativity	-0.026430
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004198
averageClosenessCentrality	0.194685
averageCommunitySize	52.042090
averageDegreeCentrality	0.004000
averageEccentricity	8.252975
averageEigenvectorCentrality	0.025050
averageHubScore	0.001000
averageNeighborsDegree	5.020252
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	5.176170
averageRichClubCoefficient	0.015648
communityCount	10.000000
density	0.004000
diameter	10.425000
edgeConnectivity	1.000000
estimatedPowerLawExponent	15.292400
lowerBoundPowerLawRegion	7.525000
modularity	0.538018
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	19.350000
numEdges	2000.000000
numInterCommunityEdges	800.000000
numIntraCommunityEdges	1200.000000
overallAverageCommunityClustering	0.005697
overallStdevCommunityClustering	0.044733
stdevAuthorityScore	0.000765
stdevBetweennessCentrality	0.004290
stdevClosenessCentrality	0.016665
stdevCommunitySize	17.125395
stdevDegreeCentrality	0.001983
stdevEccentricity	0.645387
stdevEigenvectorCentrality	0.019287
stdevHubScore	0.000765
stdevNeighborsDegree	1.207822
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.020010
transitivity	0.005467
intra_inter_ratio	1.500000
inter_intra_ratio	0.666667

Tabelle D.17: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1200/800

assortativity	-0.019832
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004135
averageClosenessCentrality	0.196275
averageCommunitySize	51.781980
averageDegreeCentrality	0.004000
averageEccentricity	8.254275
averageEigenvectorCentrality	0.025223
averageHubScore	0.001000
averageNeighborsDegree	5.013095
averageNodeDegree	4.000000
averagePageRank	0.001000
averagePathLength	5.135350
averageRichClubCoefficient	0.021543
communityCount	10.000000
density	0.004000
diameter	10.650000
edgeConnectivity	1.000000
estimatedPowerLawExponent	15.948133
lowerBoundPowerLawRegion	7.375000
modularity	0.529303
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	19.375000
numEdges	2000.000000
numInterCommunityEdges	1000.000000
numIntraCommunityEdges	1000.000000
overallAverageCommunityClustering	0.002670
overallStdevCommunityClustering	0.026945
stdevAuthorityScore	0.000753
stdevBetweennessCentrality	0.004275
stdevClosenessCentrality	0.017025
stdevCommunitySize	16.105728
stdevDegreeCentrality	0.001980
stdevEccentricity	0.652572
stdevEigenvectorCentrality	0.019052
stdevHubScore	0.000753
stdevNeighborsDegree	1.201375
stdevNodeDegree	288.819400
stdevPageRank	0.000400
stdevRichClubCoefficient	0.037802
transitivity	0.002720
intra_inter_ratio	1.000000
inter_intra_ratio	1.000000

Tabelle D.18: Graphmetriken aggregiert für Intra/Inter-Verhältnis 1000/1000

D.3.2 Ergebnisse

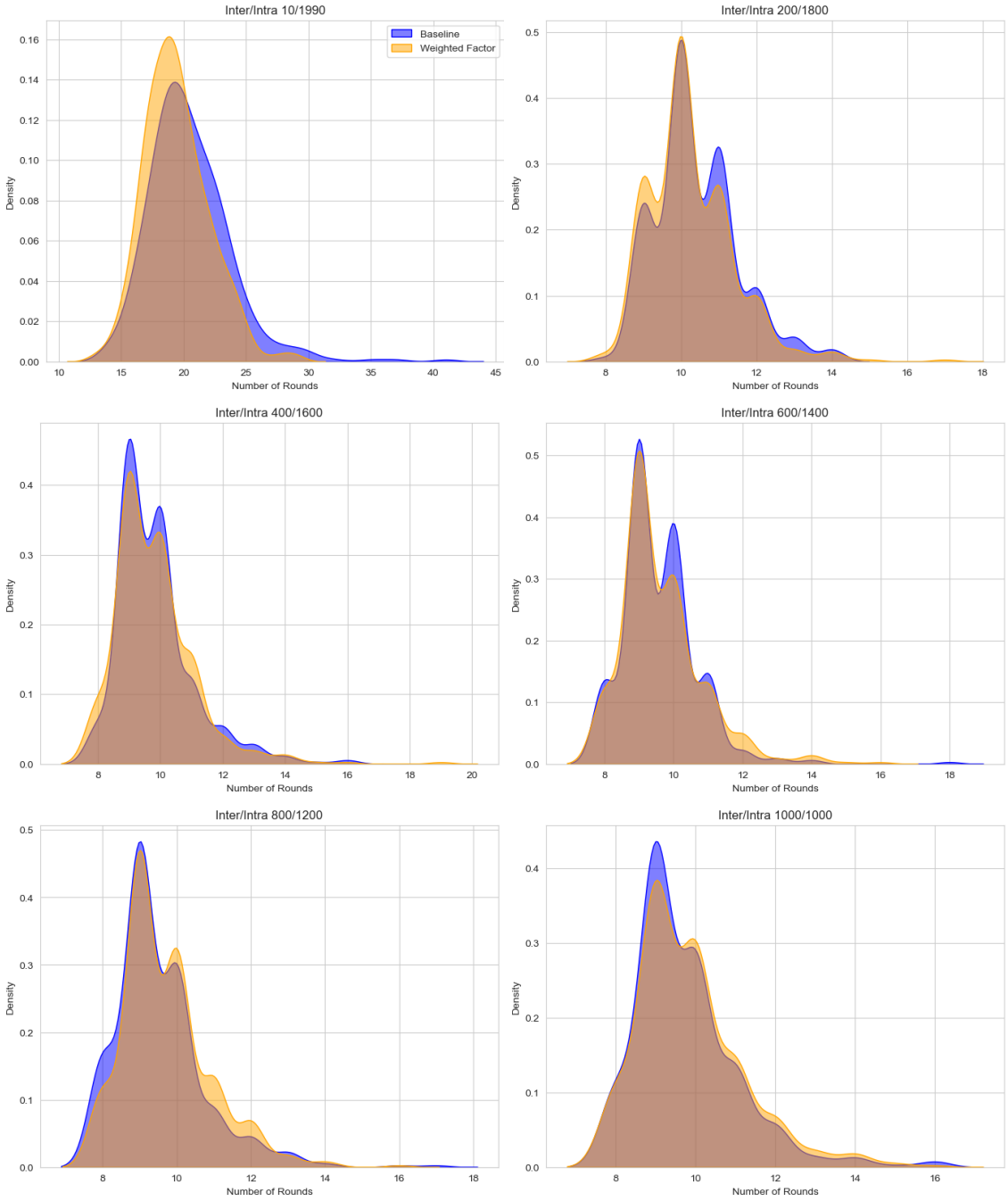


Abbildung D.3: Detaillierte Auswertung der Popularitätsgraphen für Simulationsreihe 3

D.4 Auswertung Simmulationsreihe 4

D.4.1 Gemittelte Graphmetriken

assortativity	-0.106767
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.004304
averageClosenessCentrality	0.196738
averageCommunitySize	37.345215
averageDegreeCentrality	0.002673
averageEccentricity	9.213006
averageEigenvectorCentrality	0.012366
averageHubScore	0.001000
averageNeighborsDegree	12.381417
averageNodeDegree	2.671088
averagePageRank	0.001000
averagePathLength	5.298866
averageRichClubCoefficient	0.377631
density	0.002673
diameter	12.450000
edgeConnectivity	1.000000
estimatedPowerLawExponent	3.203487
lowerBoundPowerLawRegion	6.150000
modularity	0.799543
newEdges	1.337813
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	27.393750
numEdges	1335.543750
overallAverageCommunityClustering	0.139184
overallStdevCommunityClustering	0.263553
stdevAuthorityScore	0.002396
stdevBetweennessCentrality	0.023527
stdevClosenessCentrality	0.032009
stdevCommunitySize	17.989169
stdevDegreeCentrality	0.004413
stdevEccentricity	1.013451
stdevEigenvectorCentrality	0.029071
stdevHubScore	0.002396
stdevNeighborsDegree	15.917541
stdevNodeDegree	288.819400
stdevPageRank	0.001406
stdevRichClubCoefficient	0.290660
transitivity	0.036307
triangleProbability	0.450501

Tabelle D.19: Graphmetriken aggregiert für hochmodulare skalenfreie Graphen

D.4.2 Ergebnisse

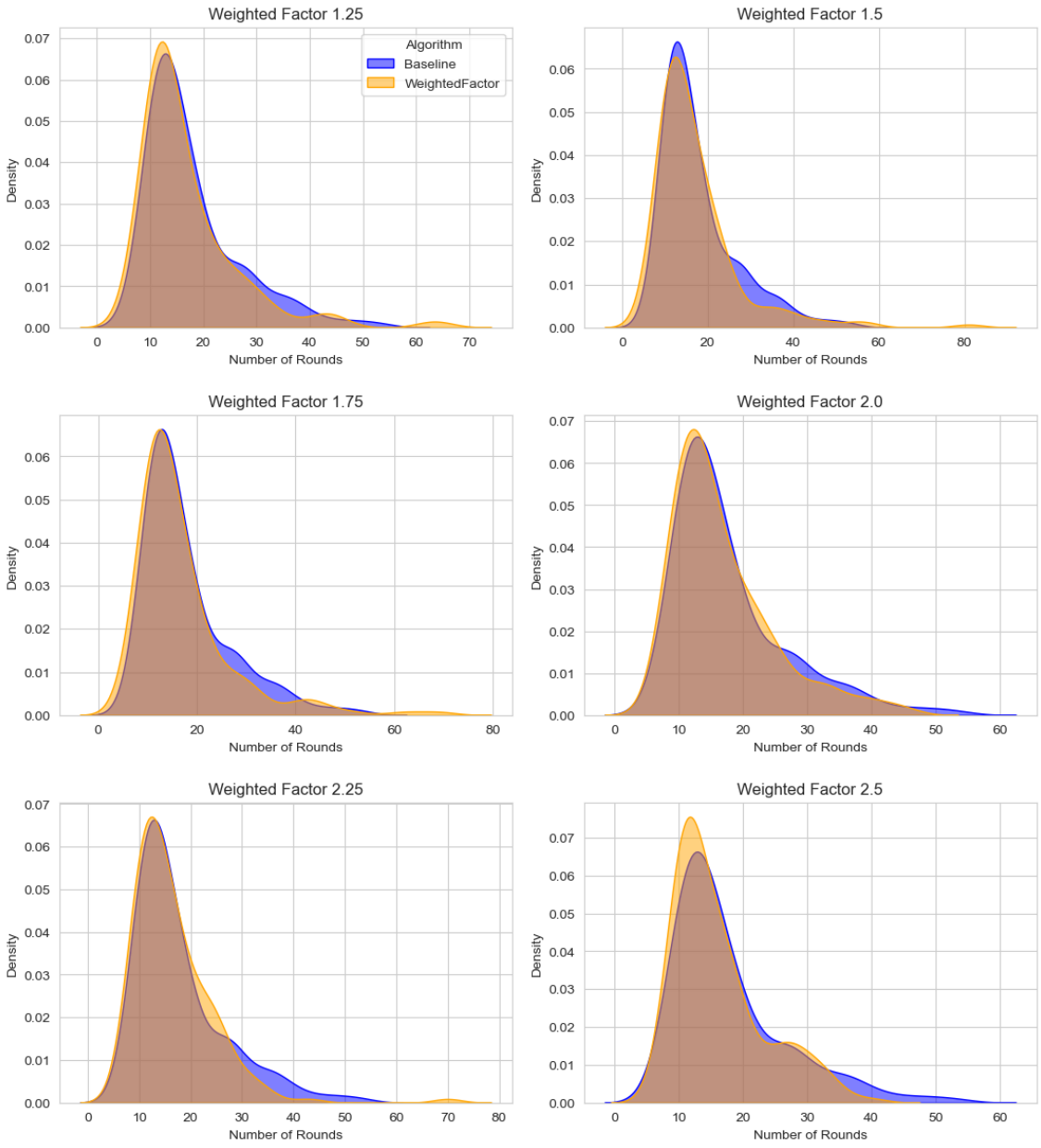


Abbildung D.4: Detaillierte Auswertung auf hochmodularen skalenfreien Graphen für Simulationsreihe 4 Teil 1

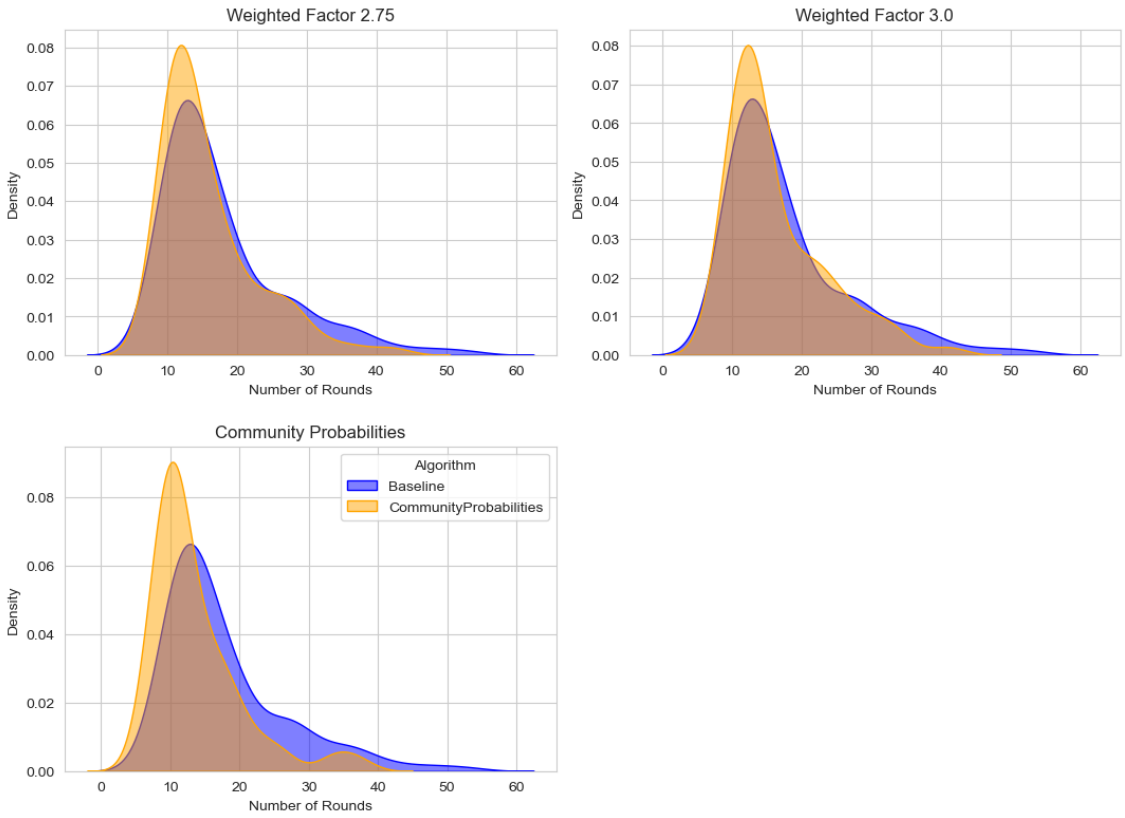


Abbildung D.5: Detaillierte Auswertung auf hochmodularen skalenfreien Graphen für Simulationsreihe 4 Teil 2

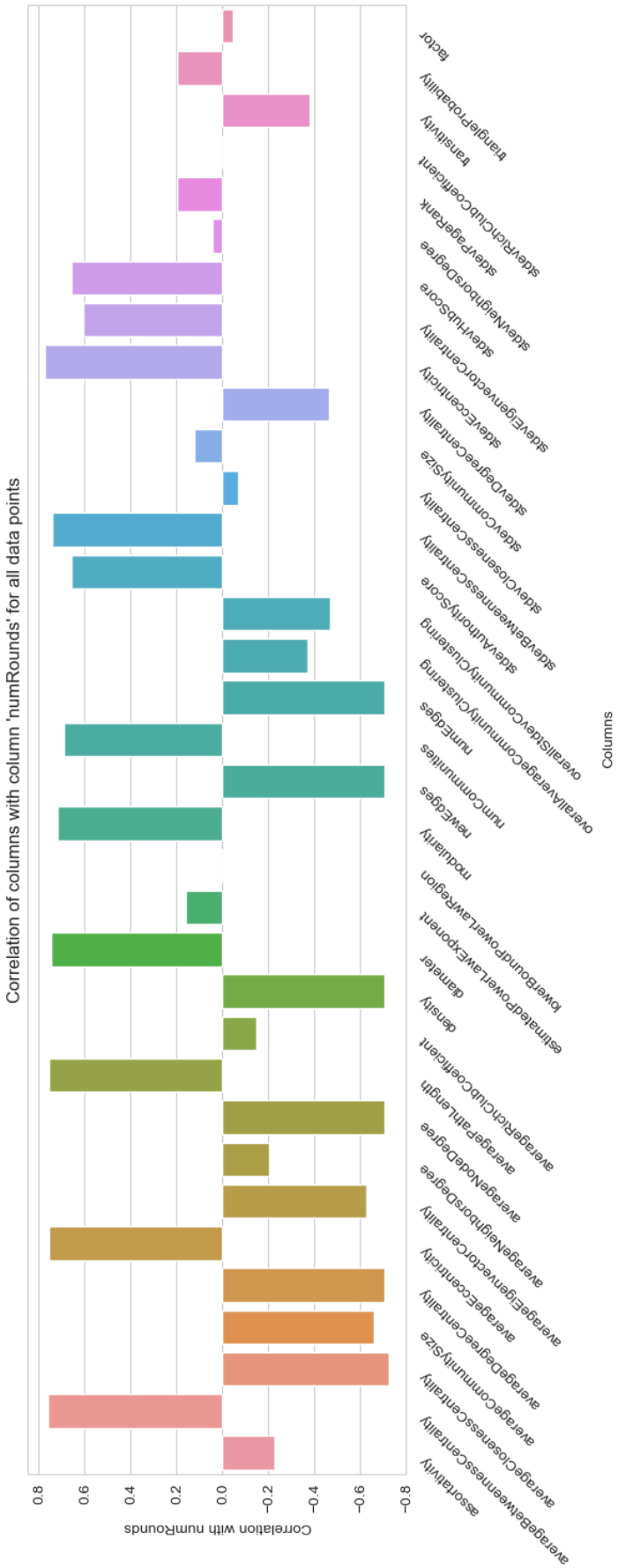


Abbildung D.6: Korrelationsmatrix der Grapheigenschaften hochmodularer skalenfreier Graphen

D.5 Auswertung Simmulationsreihe 5

D.5.1 Gemittelte Graphmetriken

Analog Simulationsreihe 4

D.5.2 Ergebnisse

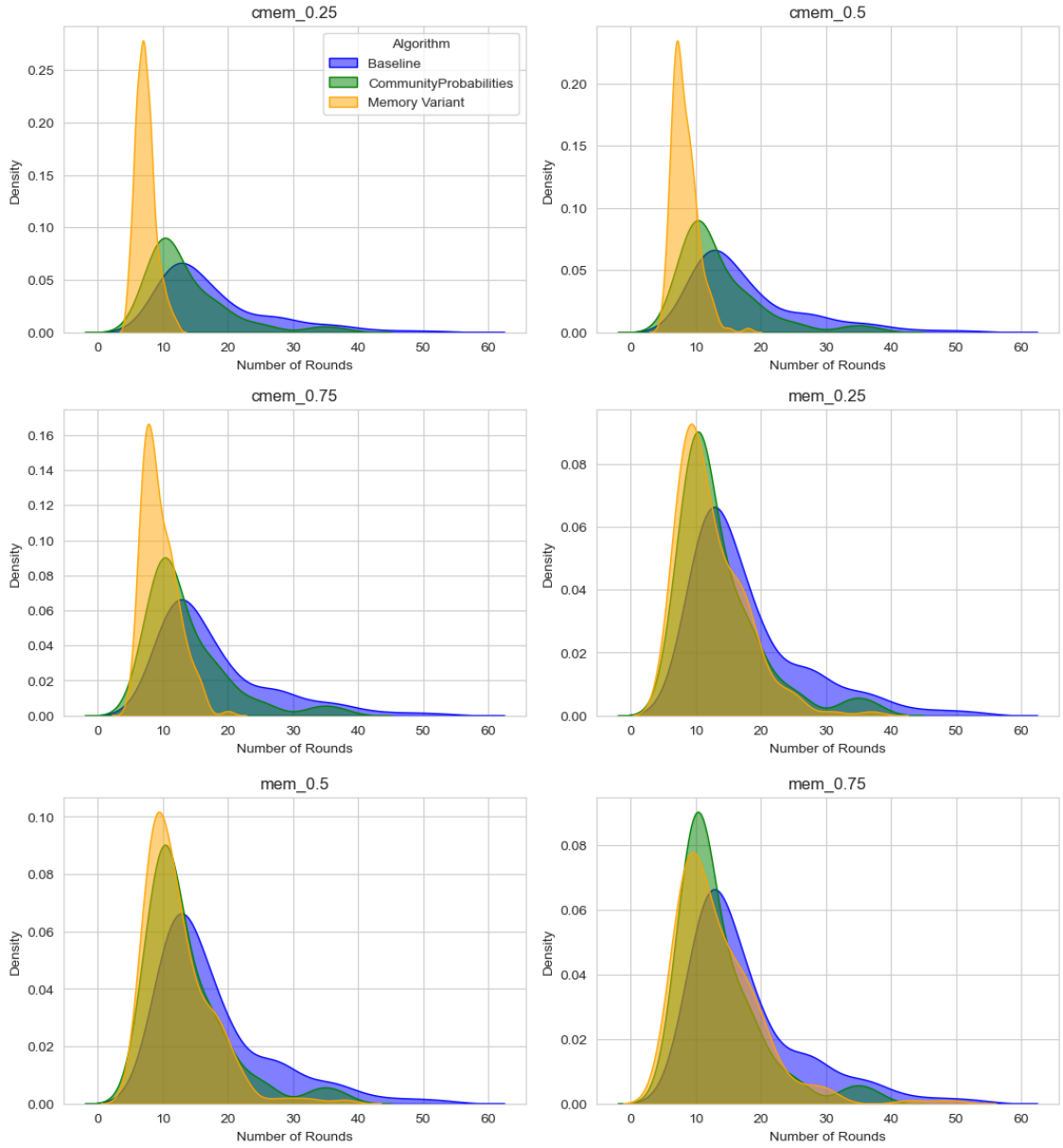


Abbildung D.7: Detaillierte Auswertung der Speicherlogik für Simulationsreihe 5

D.6 Auswertung Simmulationsreihe 6

D.6.1 Gemittelte Graphmetriken

Analog Simulationsreihe 4

D.6.2 Ergebnisse

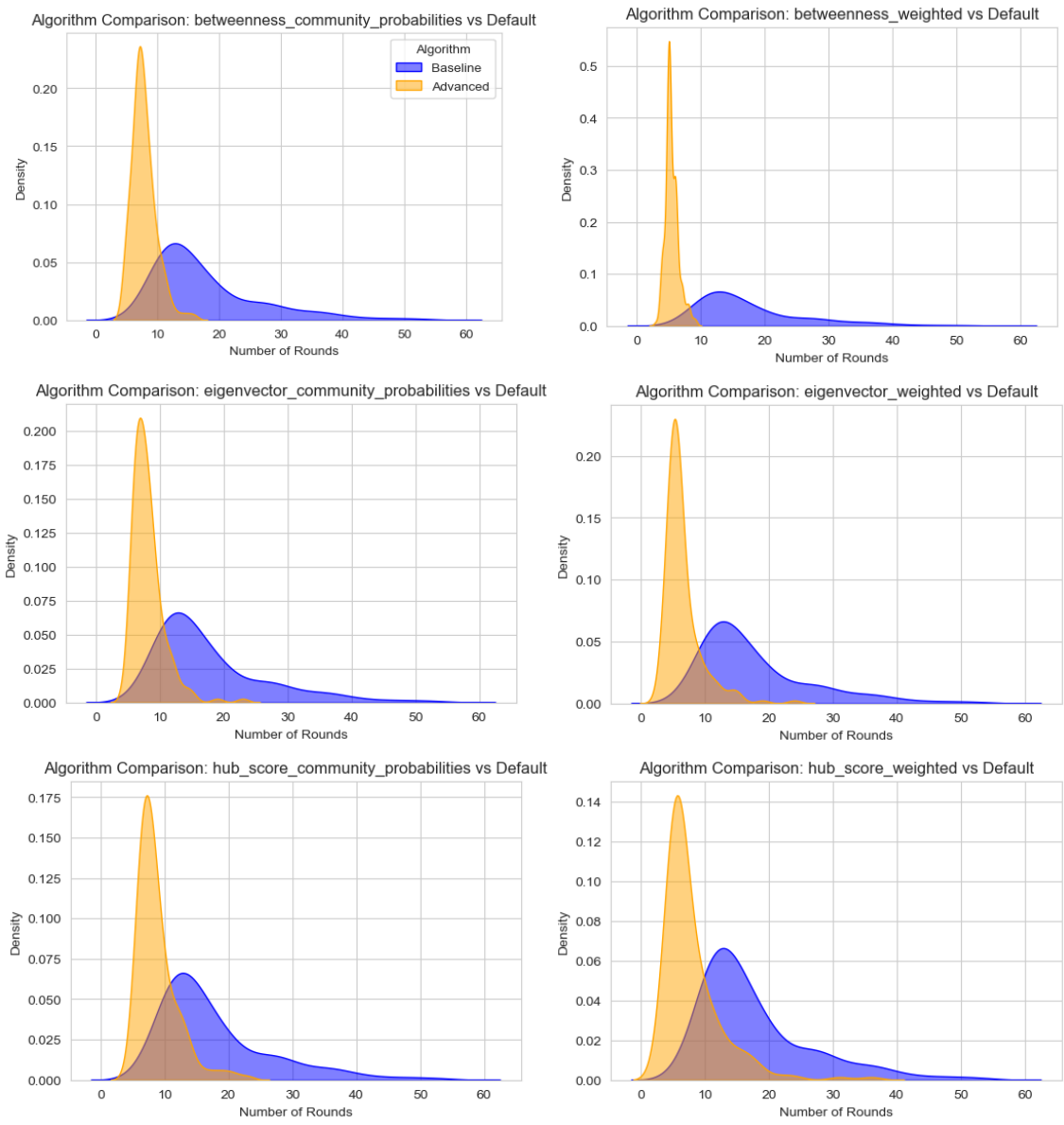


Abbildung D.8: Detaillierte Auswertung der fortgeschrittenen Verfahren für Simulationsreihe 6

D.7 Auswertung Simmulationsreihe 7

D.7.1 Gemittelte Graphmetriken

Analog Simulationsreihe 4

D.7.2 Ergebnisse

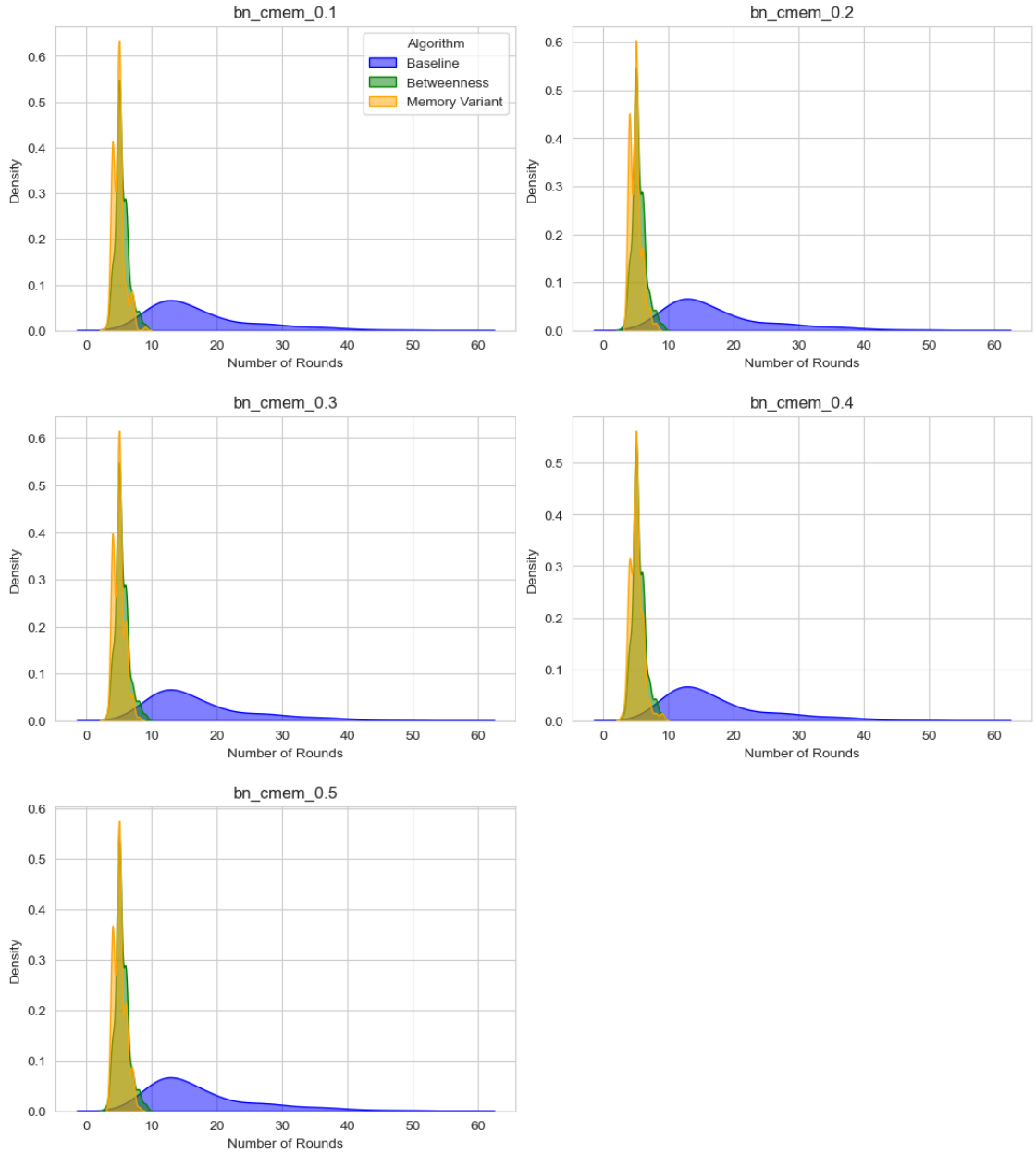


Abbildung D.9: Detaillierte Auswertung der Betweenness-Varianten für Simulationsreihe 7

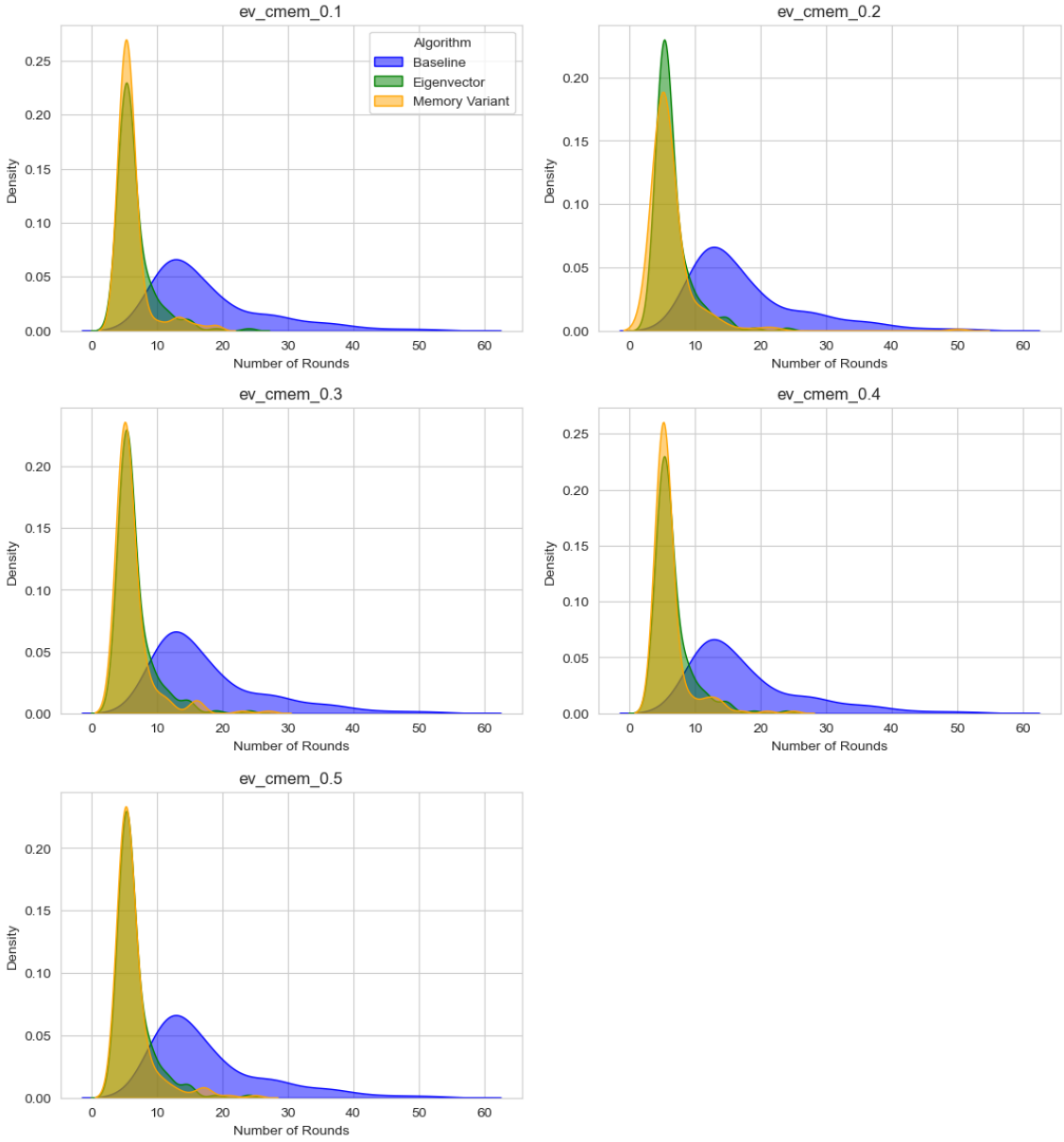


Abbildung D.10: Detaillierte Auswertung der Eigenvector-Varianten für Simulationsreihe 7

D.8 Auswertung Simmulationsreihe 8

D.8.1 Gemittelte Graphmetriken

Analog Simulationsreihe 4

D.8.2 Ergebnisse

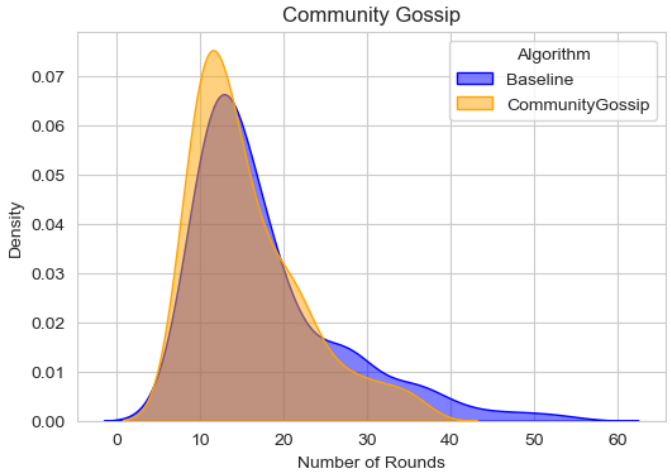


Abbildung D.11: Detaillierte Auswertung CommunityBased-Variante für Simulationsreihe 8

D.9 Auswertung Simmulationsreihe 9

D.9.1 Gemittelte Graphmetriken

assortativity	0.236592
averageAuthorityScore	0.001000
averageBetweennessCentrality	0.011202
averageClosenessCentrality	0.084738
averageCommunitySize	34.978035
averageDegreeCentrality	0.002339
averageEccentricity	24.927953
averageNodeDegree	2.349656
averageEigenvectorCentrality	0.006650
averageHubScore	0.001000
averageNearestNeighborsDegree	2.607717
averagePageRank	0.001000
averagePathLength	12.168179
averageRichClubCoefficient	0.032883
density	0.002339
diameter	35.380702
edgeConnectivity	1.000000
estimatedPowerLawExponent	24.197714
lowerBoundPowerLawRegion	4.712281
modularity	0.830476
nodeConnectivity	1.000000
nodeCount	1000.000000
numCommunities	28.696491
numEdges	1174.828070
overallAverageCommunityClustering	0.009680
overallStdevCommunityClustering	0.075085
stdevAuthorityScore	0.004963
stdevBetweennessCentrality	0.013452
stdevClosenessCentrality	0.013158
stdevCommunitySize	10.772073
stdevDegreeCentrality	0.000875
stdevEccentricity	2.651218
stdevNodeDegree	16187.431169
stdevEigenvectorCentrality	0.030877
stdevHubScore	0.004963
stdevNearestNeighborsDegree	0.664395
stdevPageRank	0.000300
stdevRichClubCoefficient	0.050502
transitivity	0.013514

Tabelle D.20: Graphmetriken aggregiert Gnutella-P2P-Netzwerkpartitionen

D.9.2 Ergebnisse

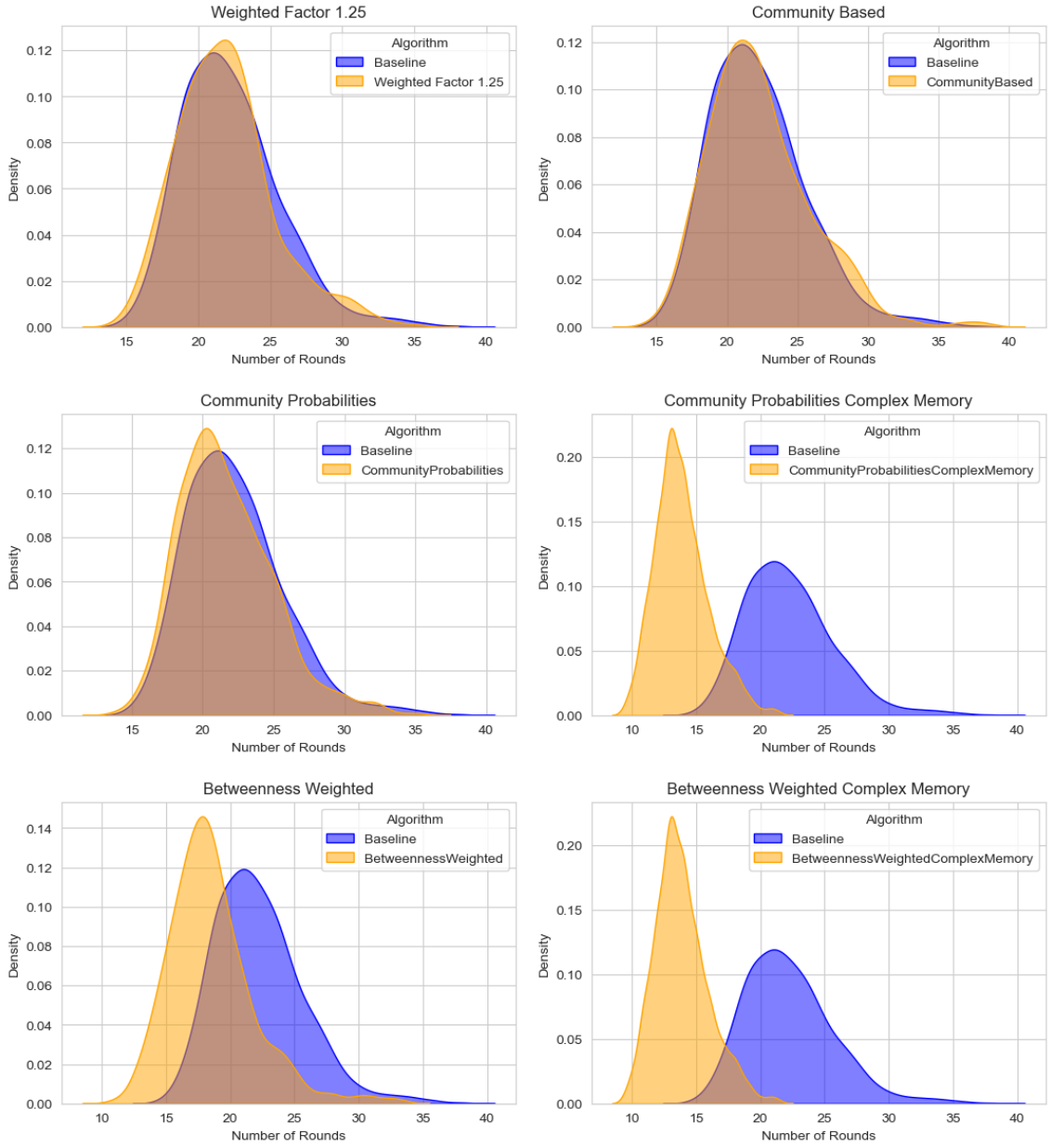


Abbildung D.12: Detaillierte Auswertung auf Gnutella-Graphen für Simulationsreihe 9

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 1.0). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt