Karlsruhe Institute of Technology

Vergleich von Algorithmen zur Erkennung von Clusterungen variabler Granularität anhand von Zufallsgraphen

Diplomarbeit
von

Geraud Oscar Fofie Lafou

An der Fakultät für Informatik
Institut für Theoretische Informatik

Gutachterin:	Prof. Dr. Dorothea Wagner
Betreuende Mitarbeiterin:	Dipl.-Inf. Andrea Schumm

Bearbeitungszeit: 30. Juni 2011 – 29. Dezember 2011

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine andere Hilfsmittel als die angegebenen Quellen verwendet habe.

Karlsruhe, den 29. Dezember 2011

Datum Geraud Oscar Fofie Lafou

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit behandelt das Clustern von Zufallsgraphen. Beim Clustern geht es darum, Graphen in mehrere Teilgraphen zu partitionieren, so dass es möglichst viele Kanten innerhalb der Teilgraphen und möglichst wenige Kanten zwischen den Teilgraphen gibt. Diese Problemstellung wird auch *Clustering* genannt. Eine Partition des Graphen heißt *Clusterung* und die dazugehörigen Teilgraphen sind die *Cluster*. In dem hier betrachteten Problem ist die Anzahl der entstehenden Teilgraphen nicht beschränkt und jeder Teilgraph kann beliebig viele Knoten enthalten. Nach diesen Kriterien kann ein Graph in mehrere unterschiedlichen Partitionen geclustert werden.

Die Güte einer Clusterung kann mit einer Qualitätsfunktion bewertet werden. Um eine gute Clusterung eines Graphen zu erzielen, nutzen manche Algorithmen eine Qualitätsfunktion als Zielfunktion, die sie versuchen, mit verschiedenen Heuristiken zu optimieren. Eine weit verbreitete Qualitätsfunktion ist *Modularity*. Andere Clusteringalgorithmen beruhen nicht explizit auf der Optimierung einer Zielfunktion, so zum Beispiel der auf Schnittbäumen basierende Algorithmus von Flake [FTT04]. Da es keine allgemein anerkannte Zielfunktion gibt, ist es oft schwierig, verschiedene Clusteringalgorithmen qualitativ zu vergleichen. Eine häufig benutzte Lösung dafür ist der Vergleich anhand von Zufallsgraphen mit einer bekannten Referenzclusterung. Dabei wird bewertet, wie gut ein Clusteringalgorithmus ist, indem es gemessen wird, wie gut er die Referenzclusterung wiedererkennt.

Die hier betrachteten Zufallsgraphen entstehen, indem in einem leeren Graphen mit vorgegebener Anzahl von Knoten jede mögliche Kante mit einer bestimmten Wahrscheinlichkeit eingefügt wird. Die einzelnen Kantenwahrscheinlichkeiten können dabei anhand von verschiedenen Modellen gewählt werden. Ein sehr bekanntes Modell ist das *Erdös-Renyi-Modell*, bei dem alle potentiellen Kanten mit einer konstanten Wahrscheinlichkeit gewählt werden. Um Zufallsgraphen zu erzeugen, die eine deutliche Referenzclusterung haben, werden mehrere Wahrscheinlichkeiten benötigt, da Kanten innerhalb der Cluster mit einer großen Wahrscheinlichkeit und Kanten zwischen den Clustern mit einer kleinen Wahrscheinlichkeit gewählt werden sollen. Dafür wird ein anderes Zufallsmodell benötigt.

Manche Graphen beinhalten nicht nur eine Referenzclusterung, sondern mehrere Referenzclusterungen, die in Beziehung zueinander stehen können. Die Referenzclusterungen eines Graphen können hierarchisch aufgebaut sein. Das heißt, die Cluster der verschiedenen Referenzclusterungen enthalten sich gegenseitig. Die Referenzclusterungen eines Graphen können sich aber auch überlagern. Das heißt, dass die Cluster in verschiedenen Referenzclusterungen sich überlappen. Um solche Zufallsgraphen zu erzeugen, wird ein Generator benötigt.

Es ist erwünscht, dass der Generator sowohl Zufallsgraphen mit einer hierarchischen Clusterstruktur als auch Zufallsgraphen mit einer sich überlagernden Clusterstruktur erzeugt. Ein guter Generator sollte Graphen in einer guten Laufzeit erzeugen, möglichst wenig Speicherplatz benötigen und sowohl dichte als auch dünne Graphen effizient erzeugen.

Das Ziel dieser Arbeit besteht darin, solch einen Generator zu entwickeln, und Clusteringalgorithmen aus der Literatur anhand von damit erzeugten Zufallsgraphen miteinander zu vergleichen. Es wird also anhand von Zufallsgraphen untersucht, wie gut die Algorithmen die Referenzclusterungen wiedererkennen. Dafür werden Clusteringalgorithmen betrachtet, die verschiedene grobe Clusterungen liefern.

Die Arbeit ist wie folgt gegliedert: Zunächst definieren wir einige Grundbegriffe. Dann beschreiben wir die Problemstellung und den dabei verwendeten Zufallsprozess. Danach stellen wir vier Algorithmen, die diesen Zufallsprozess simulieren, und zwei bekannte Zufallsgraphen aus der Literatur vor, die wir mit den Algorithmen erzeugt haben. Ein Generator hat eine lineare Laufzeit und der benötigte Speicherplatz ist linear in der Graphgröße. Anschließend beschreiben wir die Clusteringalgorithmen, die wir zum Vergleich betrachtet haben. Dabei stellen wir die Grundideen der Algorithmen und die Heuristiken vor, die diese Algorithmen nutzen, um die Grobheit der erzielten Clusterung zu beeinflussen. Wir stellen auch neue Kombinationen vor, die wir aus diesen Algorithmen und deren Heuristiken erzielt haben. Zum Schluss vergleichen wir die Algorithmen miteinander.

2. Grundlagen und Problemstellung

2.1. Notation

In dieser Arbeit bezeichnen wir mit $G = (V, E)$ einen Graphen, mit $V = \{0, \ldots, n-1\}$ die Knotenmenge und mit $m = |E|$ die Anzahl der Kanten von G. Eine *Clusterung* $\mathcal{C} = \{C_0, \ldots, C_k\}$ ist eine Partition von V. Die Knotenteilmengen C_i, $0 \leq i \leq k$ heißen *Cluster*. Wir speichern eine Clusterung $\mathcal{C}$ in einem Feld $\mathcal{C}[0, \ldots, n-1]$, wobei $\mathcal{C}[u] = i$ für alle im Cluster C_i liegende Knoten u. Eine Kante $\{w, v\}$ mit $\mathcal{C}[w] = \mathcal{C}[v]$ ist eine *Intraclusterkante*. Das heißt, die Endkonten w und v gehören zum gleichen Cluster. Falls $\mathcal{C}[w] \neq \mathcal{C}[v]$, liegen die Endknoten w und v in unterschiedlichen Clustern. In diesem Fall ist die Kante $\{w, v\}$ eine *Interclusterkante*. Die Menge der Intraclusterkanten im Cluster C_i ist $E(C_i)$ und $E(C_i, C_j)$ ist die Menge der Kanten, die Cluster C_i und C_j verbinden.

Falls der Graph gewichtet ist, bezeichnen wir mit W das Gesamtgewicht aller Kanten im Graphen, mit $\omega(C)$ das Gesamtgewicht der Intraclusterkanten eines Clusters C, mit $\deg_\omega(u)$ das Gesamtgewicht der zu einem Knoten u inzidenten Kanten, mit $\omega \deg(u)$ das Gesamtgewicht der Schleife an u und der zu u inzidenten Kanten und mit $e_{C,u}$ das Gesamtgewicht der Kanten, die u mit dem Cluster C verbinden.

2.2. Definitionen

Definition 1 (Clustering) *Es gibt keine allgemeine gültige Definition von* Clustering *aber die Intuition dahinter ist, einen Graphen zu partitionieren, so dass es möglichst viele Intraclusterkanten und möglichst wenige Interclusterkanten gibt. Ein Graph, der so partitioniert ist, heißt auch* geclustert*.*

Definition 2 (KnotenID) *Sei $G = (V, E)$ ein Graph mit n Knoten. Die Menge $V = \{0, \ldots, n-1\}$ heißt die Menge der* KnotenIDs*, so dass jeder Knoten einer eindeutigen Nummer dieser Menge entspricht und umgekehrt.*

Definition 3 (ClusterID) *Sei $\mathcal{C}[0, \ldots, n-1]$ eine Clusterung. Eine* ClusterID *ist ein Wert $i = \mathcal{C}[v], v \in \{0, \ldots, n-1\}$. In einer Clusterung ist eine ClusterID also die Nummer eines Clusters und alle Knoten, die im gleichen Cluster liegen, können mit dieser Nummer erkannt werden. Die ClusterIDs einer Clusterung sind entweder aufeinanderfolgende Nummer oder in jedem Cluster ist die ClusterID der kleinste Knoten in diesem Cluster. Das Bild 2.1 zeigt die ClusterIDs einer Clusterung.*

v	0	1	2	3	4	5	6	7	8	9
$\mathcal{C}[v]$	0	0	0	1	1	0	1	2	1	2

Die ClusterIDs sind: 0, 1 und 2.

v	0	1	2	3	4	5	6	7	8	9
$\mathcal{C}[v]$	0	0	0	3	3	0	3	7	3	7

Die ClusterIDs sind: 0, 3 und 7.

Abbildung 2.1.: Beispiel ClusterID. Im obigen Bild sind die ClusterIDs aufeinanderfolgende Nummer und im unteren Bild ist die ClusterID eines Clusters der kleinste Knoten in diesem Cluster.

Definition 4 (Intraclusterknotenpaar und Interclusterknotenpaar) *Seien $\mathcal{C}[0, \ldots, n-1]$ eine Clusterung und $\{w, v\}$ ein Knotenpaar. Das Knotenpaar $\{w, v\}$ heißt* Intraclusterknotenpaar *für die Clusterung $\mathcal{C}$, falls gilt $\mathcal{C}[w] = \mathcal{C}[v]$, sonst ist es ein* Interclusterknotenpaar*.*

Definition 5 (Hierarchische Clusterung) *Eine* hierarchische Clusterung *bezeichnet k Partitionen eines Graphen, die sich gegenseitig enthalten.*

Ein Graph hat eine *hierarchische Clusterstruktur*, falls es eine hierarchische Clusterung gibt, so dass jede dieser Partitionen eine sinnvolle Clusterung auf diesem Graphen ist.

Definition 6 (Überlagernde Clusterung) *Eine* überlagernde Clusterung *bezeichnet k Partitionen eines Graphen, die sich gegenseitig überlappen.*

Ein Graph hat eine sich *überlagernde Clusterstruktur*, falls es eine überlagernde Clusterung gibt, so dass jede dieser Partitionen eine sinnvolle Clusterung auf diesem Graphen ist. Bild 2.2 zeigt ein Beispiel eines Graphen mit einer sich überlagernden Clusterstruktur.

Für eine gegebene Clusterung eines ungewicheteten Graphen ist *Modularity* die Summe über alle Cluster der Differenz zwischen dem Anteil der Kanten in diesen Clustern und dem entsprechenden Erwartungswert in einem Zufallsgraphen mit ähnlichen Eigenschaften. Dies wird für gewichtete Graphen verallgemeinert. Dabei wird die Anzahl der Kanten innerhalb eines Clusters als das Gesamtgewicht der Intraclusterkanten in diesem Cluster und der Grad eines Knotens als das Gesamtgewicht der zu diesem Knoten inzidenten Kanten betrachtet.

Definition 7 (Modularity) *Für eine gegebene Clusterung $\mathcal{C}$ eines ungewichteten Graphen ist Modularity [NG04] gleich*

$$q(\mathcal{C}) = cov(\mathcal{C}) - \mathbb{E}(cov(\mathcal{C})) = \sum_{C \in \mathcal{C}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} \deg(v)}{2m} \right)^2 \right].$$

Dabei gilt $cov(\mathcal{C}) = \sum_{C \in \mathcal{C}} \frac{|E(C)|}{m}$.
Falls der Graph gewichtet ist, dann ist Modularity [New04] gleich

$$q(\mathcal{C}) = \sum_{C \in \mathcal{C}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{v \in C} \deg_\omega(v)}{2W} \right)^2 \right].$$

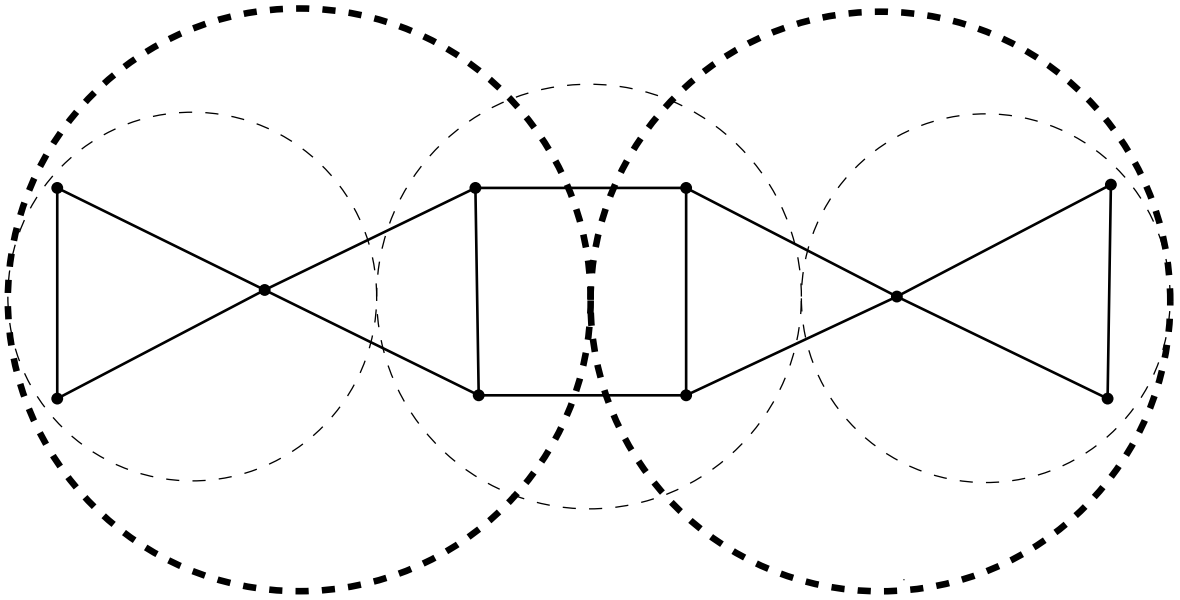


Abbildung 2.2.: Beispiel eines Graphen mit einer sich überlagernden Clusterstruktur. Der Graph kann in zwei unterschiedlichen Clusterungen geclustert werden. Eine Clusterung ist in dunklen und die andere in hellen gestrichelten Kreisen dargestellt.

2.3. Verwendetes Zufallsgraphmodell

Am meisten wird das Zufallsmodell von *Gilbert* [Gil59] oder *Erdös-Renyi-Modell* [ER59] verwendet. Es ist ein Zufallsmodell, bei dem für eine gegebene Anzahl n von Knoten und eine Wahrscheinlichkeit p ein einfacher ungerichteter Graph $G(n,p) \in \mathcal{G}(n,p)$ erzeugt wird, so dass alle potenziellen Kanten mit der Wahrscheinlichkeit p gewählt werden.

Um einen Graphen mit einer deutlichen sinnvollen Clusterung zu erzeugen, werden die Intraclusterknotenpaare mit einer großen Wahrscheinlichkeit und die Interclusterknotenpaare mit einer kleinen Wahrscheinlichkeit gewählt. Dafür wird nicht nur eine Wahrscheinlichkeit wie im vorigen Zufallsmodell sondern zwei Wahrscheinlichkeiten benötigt: eine *Intraclusterwahrscheinlichkeit* p^{in}, mit der Intraclusterknotenpaare gewählt werden und eine *Interclusterwahrscheinlichkeit* p^{out}, um Interclusterknotenpaare zu wählen. Ein Graph, der nach so einem Zufallmodell entsteht, heißt $G(n,p^{in},p^{out})$*-Graph*.

Der von uns verwendete Zufallsprozess orientiert sich an diesem Zufallsmodell und will es erweitern, um Graphen zu erzeugen, die mehrere sinnvollen Clusterungen enthalten. Dafür wird jedes Knotenpaar mit einer Wahrscheinlichkeit gewählt, die von seinen Endknoten und gegebenen parametern abhängt.

Die gegebenen parameter sind k Clusterungen und k dazugehörige Paare von Intra- und Interclusterwahrscheinlichkeiten $\mathcal{C}_0\left(p_0^{in},p_0^{out}\right),\ldots,\mathcal{C}_{k-1}\left(p_{k-1}^{in},p_{k-1}^{out}\right)$. Für zwei beliebige Knoten w und v berechnen wir die Wahrscheinlichkeit $p(w,v)$, mit der das Knotenpaar $\{w,v\}$ in diesem Zufallsprozess gewählt wird. Sei $p_i(w,v)$ die Wahrscheinlichkeit, mit der das Knotenpaar $\{w,v\}$ bei der Clusterung $\mathcal{C}_i$ gewählt wird.

$$p_i(w,v) = \begin{cases} p_i^{in}, & \text{falls} \quad \mathcal{C}_i[w] = \mathcal{C}_i[v] \\ p_i^{out}, & \text{falls} \quad \mathcal{C}_i[w] \neq \mathcal{C}_i[v] \end{cases}$$

Die Wahrscheinlichkeit $p(w,v)$ ist gleich der Wahrscheinlichkeit, das Knotenpaar $\{w,v\}$

mindestens bei einer Clusterung zu wählen.

$$\begin{aligned} p(w,v) &= p_0(w,v) + (1-p_0(w,v)) \times p_1(w,v) + (1-p_0(w,v)) \times (1-p_1(w,v)) \times p_2(w,v) + \\ &\quad \ldots + (1-p_0(w,v)) \times (1-p_1(w,v)) \times \ldots \times (1-p_{k-2}(w,v)) \times p_{k-1}(w,v) \\ &= p_0(w,v) + \sum_{i=1}^{k-1} \left[\left(\prod_{j=0}^{i-1} (1-p_j(w,v)) \right) p_i(w,v) \right] \end{aligned}$$

$$p(w,v) = \sum_{i=0}^{k-1} \left(p_i(w,v) \prod_{\substack{j=0, \\ j \neq i}}^{i-1} (1-p_j(w,v)) \right) \tag{2.1}$$

2.4. Problemstellung

Es soll einen Generator entwickelt werden, der Graphen erzeugt, die eine hierarchische oder sich überlagernde Clusterstruktur beinhalten können. Der Generator soll als Eingabe k Partitionen $\mathcal{C}_0, \ldots, \mathcal{C}_{k-1}$ einer Knotenmenge $V = \{0, \ldots, n-1\}$ bekommen. Eine Partition $\mathcal{C}_i$ kann auch als ein Feld $\mathcal{C}_i[0, \ldots, n-1]$ betrachtet werden, in dem gilt, dass $\mathcal{C}_i[v] = \mathcal{C}_i[w]$, falls die Knoten v und w zur gleichen Teilmenge gehören und $\mathcal{C}_i[v] \neq \mathcal{C}_i[w]$, falls die beiden Knoten zur unterschiedlichen Teilmengen gehören. Zu jeder Partition $\mathcal{C}_i$ gehören zwei Wahrscheinlichkeiten p_i^{in} und p_i^{out}. Mit der Wahrscheinlichkeit p_i^{in} werden die dieser Partition entsprechenden Intraclusterkanten eingefügt. Mit der Wahrscheinlichkeit p_i^{out} werden die Interclusterkanten dieser Partition eingefügt. Dies wird für alle Partitionen wiederholt. Der Generator soll ausgehend von diesen Eingabeparametern einen einfachen, ungerichteten Zufallsgraphen schrittweise konstruieren. Die Kanten sollen nacheinander in den Graphen eingefügt werden. Jede Kante soll mit einer nach Formel 2.1 berechneten Wahrscheinlichkeit gewählt werden. Falls eine Kante in den Graphen eingefügt ist, soll sie genau ein Mal eingefügt werden, damit sich keine Kante im Graphen wiederholt.

Die Problemstellung kann wie folgt formal zusammengefasst werden:

Gegeben: Eine Anzahl n von Knoten, k Clusterungen von $\{0, \ldots, n-1\}$ und die dazugehörigen Paare p^{in} und p^{out} von Wahrscheinlichkeiten $\mathcal{C}_0 \left(p_0^{in}, p_0^{out}\right)$, $\ldots, \mathcal{C}_{k-1} \left(p_{k-1}^{in}, p_{k-1}^{out}\right)$.

Problem: Erzeuge einen einfachen Zufallsgraphen, so dass jedes Knotenpaar $\{w, v\}$ nach Formel 2.1 mit der Wahrscheinlichkeit $p(w,v)$ gewählt wird.

2.5. Experimentelle Zielsetzung

Es soll mit dem zur Lösung des im Abschnitt 2.4 beschriebenen Problems entwickelten Generator Zufallsgraphen mit eingepflanzten Clusterungen erzeugt werden. Verschiedene Clusteringalgorithmen werden auf den erzeugten Zufallsgraphen ausgeführt. Die betrachteten Algorithmen werden miteinander verglichen. Dabei wird experimentell untersucht, wie gut jeder betrachtete Algorithmus die eingebauten Clusterungen wiedererkennt.

2.6. Verwendete Methode von *Batagelj* und *Brandes*

Um das oben beschriebene Problem 2.4 zu lösen, haben wir teilweise die Methode von *Batagelj* und *Brandes* [BB05] benutzt. Wir erläutern in diesem Abschnitt diese Methode.

Der Trick beruht auf der geometrische Methode [FMR62]. Die Anzahl m der Kanten in einem Graphen $G(n,p)$ nach Gilberts Modell folgt einer Binomialverteilung $\mathcal{B}(m \mid p, \binom{n}{2})$, wobei m die Anzahl der Erfolge ist. Ein Erfolg ist erzielt, wenn eine Kante in den Graphen

eingefügt ist. Ansonsten ist es ein Misserfolg. Beim Durchlaufen alle möglichen Kanten nacheinander gilt zu jedem Zeitpunkt: die Wahrscheinlichkeit, die nächste Kante erst nach s Versuchen zu wählen, ist $(1-p)^{s-1}p$. Zu jedem Zeitpunkt folgt die Anzahl der erfolglosen Versuche also einer geometrischen Verteilung. Seien $q = 1 - p$ und X eine Zufallsvariable gemäß dieser geometrischen Verteilung. Die Wahrscheinlichkeit, mindestens einen Erfolg nach s Versuchen zu erzielen, ist die Wahrscheinlichkeit, den ersten Erfolg nach $i = 1$ oder 2 oder $, \ldots, s$ Versuchen zu erzielen und ist gleich

$$\sum_{i=1}^{s} P(X = i) = \sum_{i=1}^{s} q^{i-1}p = p\sum_{i=0}^{s} q^i = p\frac{1-q^s}{1-q} = 1 - q^s.$$

Zu jedem Zeitpunkt wird eine Zahl r gleichverteilt aus $[0, 1)$ gezogen und die Anzahl der erfolglosen Versuche, um den nächsten Erfolg zu erzielen, ist der kleinste Wert s, für den gilt $r < 1 - q^s$. Zu jedem Zeitpunkt wird die Weite des Sprungs auf die nächste Kante also wie folgt berechnet:

$$r < 1 - q^s \iff q^s < 1 - r \iff s \log q < \log(1 - r) \iff s < \frac{\log(1 - r)}{\log q}.$$

Der kleinste Wert von s ist: $s = 1 + \lfloor \log(1 - r)/\log(1 - p) \rfloor$. Bild 2.3 illustriert anhand eines Beispiels, wie dieser Trick funktioniert. Dabei wird gezeigt, welche Kante nach jedem Sprung gewählt wird.

Zeitpunkt	0		1			2									
Sprung s	2		3			1									
Zeiger	↓		↓			↓	↓								
Kantennummer		0	1	2	3	4	5	6	7	8	9	10	...	$\binom{n}{2} - 2$	$\binom{n}{2} - 1$

Abbildung 2.3.: Sprung auf Kanten mit dem Trick von *Batagelj* und *Brandes*. Zum Zeitpunkt 0 wird ein Sprung von $s = 2$ ausgeführt. Zum Zeitpunkt 1 wird ein Sprung von $s = 3$ ausgeführt und zum Zeitpunkt 2 wird ein Sprung von $s = 1$ ausgeführt.

3. Generatoren

Wir beschreiben in diesem Kapitel die von uns entwickelten Generatoren. Die Generatoren erzeugen einfache Zufallsgraphen, die eine gewünschte Clusterstruktur haben. Das Ziel der Generatoren ist, Graphen nach dem in Abschnitt 2.4 beschriebenen Zufallsmodell zu generieren. Mit jedem Generator können sowohl Zufallsgraphen mit einer hierarchischen als auch mit einer sich überlagernden Clusterstruktur erzeugt werden. Natürlich können auch einfache Graphen $G(n,p) \in \mathcal{G}(n,p)$ nach Gilberts Modell generiert werden. Zuerst stellen wir einen naiven Generator vor, der das Problem in quadratischer Zeit löst. Dann stellen wir einen Generator vor, der einfach zu implementieren, schnell und je nach Bedarf einfach zu modifizieren oder zu manipulieren ist. Außerdem braucht er sehr wenig Speicherplatz. Danach stellen wir einen anderen Generator vor. Die Laufzeit und der Speicherplatz dieses anderen Generators sind fast linear in der Größe des erzeugten Graphen. Anschließend stellen wir einen vierten Generator vor, dessen Laufzeit echt linear in der Graphgröße ist. Außer dem naiven Generator sind die Generatoren speziell für dünne Graphen gut geeignet.

3.1. Naiver Generator

Die Idee des naiven Generators ist, alle potenziellen Kanten zu durchlaufen und für jedes Knotenpaar die Clusterungen $\mathcal{C}_i$, $i = 0, \ldots, k-1$ nacheinander zu betrachten bis das Knotenpaar bei einer Clusterung in den Graphen eingefügt wird oder bis man bei allen Clusterungen versucht hat, das Knotenpaar erfolglos in den Graphen einzufügen. Wenn das Knotenpaar bei einer Clusterung in den Graphen eingefügt wird, dann werden die anderen für dieses Knotenpaar noch nicht betrachteten Clusterungen nicht mehr behandelt. Das Verfahren wird mit dem nächsten Knotenpaar weitergeführt. Dafür kann man für jedes Knotenpaar e die folgenden Operationen schrittweise ausführen:

1 Wähle eine beliebige Clusterung $\mathcal{C}_i$, die für das Knotenpaar e noch nicht betrachtet worden ist.

2 Falls e ein Intraclusterknotenpaar für die Partition $\mathcal{C}_i$ ist, dann füge e in den Graphen mit der Wahrscheinlichkeit p_i^{in} ein.

3 Falls e ein Interclusterknotenpaar für die Partition $\mathcal{C}_i$ ist, dann füge e in den Graphen mit der Wahrscheinlichkeit p_i^{out} ein.

4 Falls e nicht in den Graphen eingefügt worden ist, und es eine noch nicht betrachtete Clusterung gibt, dann wiederhole *1*, *2*, *3* und *4*.

Mit n Knoten gibt es insgesamt $\binom{n}{2} = n(n-1)/2$ potenzielle Kanten. Bei diesem Verfahren wird höchstens k Male versucht, ein Knotepaar in den Graphen einzufügen. Die Laufzeit ist in $O(kn^2)$. Dieses Verfahren ist geeignet, um dichte Graphen zu erzeugen. Da die Graphen, die wir zur Evaluation benutzen wollen, dünn sind, ist das obige Verfahren ineffizient. Deshalb stellen wir im nächsten Abschnitt einen besseren Generator vor, der sowohl dichte als auch dünne Graphen effizient erzeugen kann.

3.2. Generator Funktion + Heap

Um die Laufzeit des oben beschriebenen Generators zu verbessern, durchlaufen wir nicht mehr alle Knotenpaare nacheinander. Wir nutzen für jede Wahrscheinlichkeit p die Methode von *Batagelj* und *Brandes* [BB05] (siehe Abschnitt 2.6), um manche Knotenpaare zu überspringen. Die Idee dieses Generators ist, für jede Wahrscheinlichkeit p aus den Eingabeparametern alle möglichen Knotenpaare eines Graphen $G(n,p) \in \mathcal{G}(n,p)$ nach Gilberts Modell zu wählen. Dabei wird zwischen Intra- und Interclusterwahrscheinlichkeiten unterschieden. Dies wird für alle Wahrscheinlichkeiten wiederholt und mit einem Heap werden die wiederholten Knotenpaare entfernen.

Für jede gegebene Clusterung $\mathcal{C}_i$ gibt es zwei dazugehörige Wahrscheinlichkeiten p_i^{in} und p_i^{out}. Da es k Clusterungen gibt, gibt es also insgesamt $2k$ Wahrscheinlichkeiten. Wir nummerieren die Wahrscheinlichkeiten, so dass jede Wahrscheinlichkeit einer eindeutigen Zahl zwischen 0 und $2k-1$ entspricht. Wie die Wahrscheinlichkeiten den Nummern zugeordnet sind, kann unterschiedlich sein. Es ist aber wichtig, sie so zu nummerieren, dass man für jede Zahl $j \in \{0,\ldots,2k-1\}$ weißt, ob es sich um eine Intra- oder Interclusterwahrscheinlichkeit handelt, und die entsprechende Clusterung wiedererkennen kann. Wir haben die Wahrscheinlichkeiten wie folgt nummeriert: einer Zahl j ordnen wir die Wahrscheinlichkeit $p^{in}_{\frac{j}{2}}$ bzw. $p^{out}_{\lfloor\frac{j}{2}\rfloor}$ zu, falls j gerade bzw. ungerade ist. Das heißt, für eine beliebige Zahl $j \in \{0,\ldots,2k-1\}$ ist $\mathcal{C}_{\lfloor\frac{j}{2}\rfloor}$ die entsprechende Clusterung und es handelt sich um eine Intra- bzw. Interclusterwahrscheinlichkeit, falls j gerade bzw. ungerade ist. Die Tabelle 3.1 illustriert diese Nummerierung.

j	0	1	2	3	...	$2k-2$	$2k-1$
p_j	$p_0 = p_0^{in}$	$p_1 = p_0^{out}$	$p_2 = p_1^{in}$	$p_3 = p_1^{out}$	...	$p_{2k-2} = p_{k-1}^{in}$	$p_{2k-1} = p_{k-1}^{out}$

Tabelle 3.1.: Nummerierung der Wahrscheinlichkeiten

Für eine Nummer $j \in \{0,\ldots,2k-1\}$, die einer Wahrscheinlichkeit p zugeordnet ist, werden alle potenziellen Kanten mit der Wahrscheinlichkeit p betrachtet. Um in den Graphen Kanten mit einer konstanten Wahrscheinlichkeit p einzufügen, springen wir über die Knotenpaare anhand der Wahrscheinlichkeit p. Wenn ein Knotenpaar getroffen ist, wird geprüft, ob dieses Knotenpaar ein Intra- bzw. Interclusterknotenpaar ist, falls die Nummer j einer Intra- bzw. Interclusterwahrscheinlichkeit entspricht. Dann bleibt zu prüfen, ob die Kante bereits im Graphen existiert. Danach wird anhand der Wahrscheinlichkeit p auf das nächste Knotenpaar gesprungen. Dieser Prinzip wird für alle Nummern $j \in \{0,\ldots,2k-1\}$ angewendet.

Wir erklären in den beiden nächsten Abschnitten, wie wir folgende Operationen effizienter ausführen: Prüfen, ob ein Knotenpaar ein Intra- oder Interclusterknotenpaar ist und prüfen, ob ein Knotenpaar bereits im Graphen vorhanden ist.

3.2.1. Intra- oder Interclusterknotenpaar prüfen

Wenn für eine Nummer $j \in \{0,\ldots,2k-1\}$ ein Knotenpaar $\{w,v\}$ getroffen ist, dann wird wie folgt entschieden, ob dieses Knotenpaar berücksichtigt wird.

- Fall 1: Die Zahl j ist gerade
  In diesem Fall handelt es sich um eine Intraclusterwahrscheinlichkeit und die entsprechende Clusterung ist $\mathcal{C}_{\frac{j}{2}}$. Es werden also Knotenpaare berücksichtigt, die für diese Clusterung Intraclusterknotenpaare sind. Das bedeutet, das Knotenpaar $\{w, v\}$ wird angenommen, falls gilt $\mathcal{C}_{\frac{j}{2}}[w] = \mathcal{C}_{\frac{j}{2}}[v]$.
- Fall 2: Die Zahl j ist ungerade
  In diesem Fall handelt es sich um eine Interclusterwahrscheinlichkeit und die entsprechende Clusterung ist $\mathcal{C}_{\lfloor \frac{j}{2} \rfloor}$. Es werden also Knotenpaare berücksichtigt, die für diese Clusterung Interclusterknotenpaare sind. Das bedeutet, das Knotenpaar $\{w, v\}$ wird angenommen, falls gilt $\mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[w] \neq \mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[v]$.

Wir nutzen im Generator eine Funktion edgeset, die für jede Nummer j einer Wahrscheinlichkeit nach Fall 1 oder Fall 2 darüber entscheidet, ob ein Knotenpaar $\{w, v\}$ angenommen wird. Nachdem ein Knotenpaar von der Funktion edgeset angenommen wird, wird geprüft, ob dieses Knotenpaar bereits im Graphen vorhanden ist.

3.2.2. Prüfen auf das Vorhandensein der Knotenpaare im Graphen

Bevor wir ein von der Funktion edgeset angenommenes Knotenpaar in den Graphen einfügen, prüfen wir, ob dieses Knotenpaar im Graphen bereits vorhanden ist, damit sich keine Kante im Graphen wiederholt. Dafür werden alle möglichen Kanten nummeriert und anhand jeder Wahrscheinlichkeit aufsteigend besucht. Wir nutzen einen Min-Heap, in dem wir zu jedem Zeitpunkt für jede Warscheinlichkeit das zuletzt getroffene Knotenpaar speichern. Die Wahrscheinlichkeit mit dem kleinsten Knotenpaar darf auf das nächste Knotenpaar springen. Der Index der in den Graphen eingefügten Kanten wächst monoton und wird zu jedem Zeitpunkt gespeichert. Mit jeder Wahrscheinlichkeit kann zu jedem Zeitpunkt kein Knotenpaar mehr mit einer Nummer kleiner als der Index der zuletzt in den Graphen eingefügten Kante getroffen werden. Für jedes Knotenpaar wird also geprüft, ob seine Nummer größer als der Index der zuletzt in den Graphen eingefügten Kante ist. Damit wird bekannt, ob ein Knotenpaar bereits im Graphen vorhanden ist.

Wir erläutern nun, wie die Knotenpaare nummeriert werden. Dann beschreiben wir den Min-Heap. Danach erklären wir, wie man sich das Verfahren vorstellen kann. Anschließend gehen wir nochmal auf die Details ein.

Für die Durchnummerierung aller $\binom{n}{2}$ möglichen Kanten nutzen wir eine Bijektion zwischen der Menge $\{0, \cdots, \binom{n}{2} - 1\}$ und der Menge aller Knotenpaare $\{w, v\}$ mit $0 \leq w < v \leq n-1$ [BB05]. Diese Bijektion ist in Abbildung 3.1 illustriert und wird wie folgt definiert:

- Jedem Knotenpaar $w, v \in \{0, \cdots, n-1\}$ mit $0 \leq w < v \leq n-1$ wird eine eindeutige Kantennummer $i \in \{0, \cdots, \binom{n}{2} - 1\}$ zugeordnet:

  $$i = \tfrac{v(v-1)}{2} + w.$$

- Umgekehrt wird jeder Kantennummer $i \in \{0, \cdots, \binom{n}{2} - 1\}$ ein eindeutiges Knotenpaar $\{w, v\}$ mit $0 \leq w < v \leq n-1$ zugeordnet:

  $$v = \text{source}(i) := 1 + \lfloor -\tfrac{1}{2} + \sqrt{\tfrac{1}{4} + 2i} \rfloor$$

  $$w = \text{target}(i) := i - \tfrac{v(v-1)}{2}.$$

Der benutzte Min-Heap hat eine maximale Größe von $2k$. Jeder Heapeintrag besteht aus drei Attributen: die Kantennummer, die Wahrscheinlichkeit und die Nummer der Wahrscheinlichkeit. Dabei ist die Kantennummer die Priorität, mit der Einträge im Min-Heap verwaltet werden. Auf Abbildung 3.2 ist eine Illustration des Min-Heaps zu sehen.

$v \downarrow$ / $w \longrightarrow$	0	1	2	3	4	5	6	...	$n-2$	$n-1$
0										
1	$\underrightarrow{0}$									
2	$\underrightarrow{1}$	$\underrightarrow{2}$								
3	$\underrightarrow{3}$	$\underrightarrow{4}$	$\underrightarrow{5}$							
4	$\underrightarrow{6}$	$\underrightarrow{7}$	$\underrightarrow{8}$	$\underrightarrow{9}$						
5	$\underrightarrow{10}$	$\underrightarrow{11}$	$\underrightarrow{12}$	$\underrightarrow{13}$	$\underrightarrow{14}$					
6	$\underrightarrow{15}$	$\underrightarrow{16}$	$\underrightarrow{17}$	$\underrightarrow{18}$	$\underrightarrow{19}$	$\underrightarrow{20}$				
7	$\underrightarrow{21}$	$\underrightarrow{22}$	$\underrightarrow{23}$	$\underrightarrow{24}$	$\underrightarrow{25}$	$\underrightarrow{26}$	$\underrightarrow{27}$			
$\vdots$	...									
$n-1$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	...	$\rightarrow$	

Abbildung 3.1.: Adjazenzmatrix für die Illustration der Bijektion zwischen Knotenpaaren und Kantennummern. Auf den inneren Pfeilen sind die Kantennummern der Knotenpaare $\{w, v\}$.

Anhand jeder Wahrscheinlichkeit wird auf eine Kantennummer gesprungen, die in Min-Heap eingetragen wird. Bei jedem Schritt wird in Min-Heap den Eintrag mit der kleinsten Kantennummer extrahiert. Dann wird überprüft, ob wir die gleiche Kantennummer bereits im letzten Schritt hatten. Man kann sich das so vorstellen, dass man $2k$ Zeiger hat, die immer weiterspringen. Diese Zeiger sind so verwaltet, dass in jedem Schritt der Zeiger mit der kleinsten Kantennummer weiterspringen darf und alle anderen Zeiger fix bleiben. Der Min-Heap wird also benutzt, um den Zeiger mit der kleinsten Kantennummer zu erkennen. Eine Kante kann sich nur im Graphen wiederholen, wenn mehrere Zeiger die gleiche Kantennummer treffen. In diesem Fall werden diese Zeiger mithilfe des Heaps nacheinanderfolgend behandelt. Deshalb wird zu jedem Zeitpunkt der Index der zuletzt in den Graphen eingefügten Kante gespeichert. Dieser Index wird benutzt, um das Vorhandensein eines Knotenpaars im Graphen zu prüfen. Für jeden Zeiger wird also überprüft, ob seine Kantennummer ungleich also größer als dieser Index ist. Diese Vorstellung des Verfahrens wird in Abbildung 3.2 illustriert.

p_j		p_3		p_0		$p_1 p_2$							
Zeiger		$\downarrow$		$\downarrow$		$\downarrow\downarrow$							
Kantennummer	0	1	2	3	4	5	6	7	8	9	10	...	$\binom{n}{2}-2$	$\binom{n}{2}-1$

Min-Heap				
	1	3	5	5
	p_3	p_0	p_1	p_2
	3	0	1	2

Kantennummer
Wahrscheinlichkeit
Nummer der Wahrscheinlichkeit

Abbildung 3.2.: Darstellung des Min-Heaps und Sprung von Zeigern auf Knotenpaare: $k = 2$. Die Zeiger p_1 und p_2 haben das gleiche Knotenpaar getroffen. Der Zeiger p_3 hat die kleinste Kantennummer und darf beim nächsten Schritt weiterspringen.

3.2.3. Detaillierte Beschreibung des Generators

Wir erklären nun in allen Details die Vorgehensweise des Generators anhand des Pseudocodes in *Algorithm* 1. In Zeile 3 werden der Min-Heap, bezeichnet mit der Variable minheap, eine Variable $i_{\min}$ für den Index der zuletzt in den Graphen eingefügten Kante und die Kantenmenge E des zu konstruierenden Graphen initialisiert. Die Prozedur *initialization-heap* beschreibt die Initialisierung des Min-Heaps. In jedem Schritt des Algorithmus wird

in der Zeile 5 in einer Variable i_{top} die kleinste Kantennummer gespeichert, die sich im Min-Heap befindet.

Lemma 1 *Im Laufe des Verfahrens gelten zu jedem Zeitpunkt:*

(i) Die Variable $i_{\min}$ enthält die Kantennummer der zuletzt in den Graphen eingefügten Kante.

(ii) Alle bereits in den Graphen eingefügten Kanten haben eine Kantennummer kleiner oder gleich $i_{\min}$.

(iii) Jeder Zeiger p_j befindet sich auf einem Knotenpaar mit einer Nummer $i_j \geq i_{\min}$.

Nachdem ein Heapeintrag extrahiert wird, wird überprüft, ob seine Kantennummer und $i_{\min}$ gleich sind. Falls ja, dann ist die entsprechende Kante nach Lemma 1 (i) die zuletzt in den Graphen eingefügte Kante. Sie ist denn bereits im Graphen vorhanden und wird nicht mehr eingefügt. Falls i_{top} und $i_{\min}$ ungleich sind, das heißt i_{top} ist größer als $i_{\min}$, dann ist die entsprechende Kante nicht im Graphen vorhanden. In Zeile 7 wird diese Kante in den Graphen eingefügt. Danach wird die Variable $i_{\min}$ aktualisiert.

Für den extrahierten Heapeintrag wird in den Zeilen 9 bis 12 anhand der dazugehörigen Wahrscheinlichkeit p so lange auf eine immer größere Kantennummer gesprungen, bis die getroffene Kante von der Funktion edgeset angenommen wird oder bis der Zeiger außerhalb des Bereichs $0, \cdots, \binom{n}{2} - 1$ springt. In Zeile 13 wird überprüft, ob die in der vorigen Schleife getroffene Kante i_{top} von der Funktion edgeset angenommen wird. Falls die Kante außerhalb des gültigen Bereichs liegt, wird für diese Wahrscheinlichkeit kein Eintrag mehr in Heap eingetragen und der Heap hat einen Eintrag weniger. Falls diese Kante angenommen worden ist, dann wird sie in der Zeile 14 in den Min-Heap eingefügt. Das Verfahren wird in der Zeile 4 wiederholt, solange es sich im Min-Heap mindestens einen Eintrag befindet.

Algorithm 1: Generator function + heap

```
1  Input: k partitions C_0 (p_0^in, p_0^out), ..., C_{k-1} (p_{k-1}^in, p_{k-1}^out) of n nodes 0, ..., n-1 each with
   edge probability p^in, p^out
2  Output: a simple random graph G = (V, E) with V = {0, ..., n-1}
3  Initialization: initialization-heap(minheap); E ← ∅; i_min ← −1
4  while minheap not empty do
5      (i_top, p, ind_p) ← extract min from minheap
6      if i_top > i_min then
7          E ← E ∪ {i_top}
8          i_min ← i_top
9      repeat
10         draw r ∈ [0, 1) uniformly at random
11         i_top ← i_top + 1 + ⌊log(1−r) / log(1−p)⌋
12     until edgeset(i_top, ind_p) or i_top ≥ (n 2)
13     if edgeset(i_top, ind_p) then
14         insert (i_top, p, ind_p) to minheap with priority i_top
15
16
```

3.2.4. Laufzeitsanalyse und Speicherplatzbedarf des Generators

In diesem Abschnitt beschreiben wir, wie schnell das Verfahren Zufallsgraphen erzeugt und welcher Speicherplatz benötigt wird.

Procedure `initialization-heap`

Input: a heap minheap
for $j \leftarrow 0$ **to** $2k-1$ **do**
  if j *is even* **then**
    if $p^{in}_{\frac{j}{2}} > 0$ **then** insert $(-1, p^{in}_{\frac{j}{2}}, j)$ to minheap with priority -1
  if j *is odd* **then**
    if $p^{out}_{\lfloor \frac{j}{2} \rfloor} > 0$ **then** insert $(-1, p^{out}_{\lfloor \frac{j}{2} \rfloor}, j)$ to minheap with priority -1

Function `edgeset`

Input: index $i \in \{0, \ldots, \binom{n}{2} - 1\}$ for a edge, index j for a edge probability
Output: boolean
$v \leftarrow \text{source}(i)$
$w \leftarrow \text{target}(i)$
if j *is even* **then**
  return $\left(\mathcal{C}_{\frac{j}{2}}[v] = \mathcal{C}_{\frac{j}{2}}[w]\right)$
if j *is odd* **then**
  return $\left(\mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[v] \neq \mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[w]\right)$

Im Verfahren wird ein Min-Heap mit der maximalen Größe $2k$ benutzt. Die Kanten müssen nicht gespeichert werden. Man kann sie direkt ausgeben. Für den Generator wird also $O(k)$ Speicherplatz benötigt.

Es wird anhand jeder Wahrscheinlichkeit auf Knotenpaare gesprungen. Die erwartete Anzahl der mit einer Wahrscheinlichkeit p getroffenen Knotenpaare ist $\binom{n}{2}p$. Wenn $p_{\max}$ die größte Wahrscheinlichkeit ist, $p_{\max} \geq p_j$ für alle $j \in \{0, \cdots, 2k-1\}$, dann werden mit jeder Wahrscheinlichkeit p im Erwartungswert höchstens $\binom{n}{2}p_{\max}$ Knotenpaare getroffen. Es wird also im Erwartungswert auf höchstens $2k\binom{n}{2}p_{\max}$ Knotenpaare gesprungen. Im Worst Case werden alle getroffenen Knotenpaare von der Funktion edgeset angenommen. Jedes angenommene Knotenpaar wird ein Mal in den Heap eingefügt und extrahiert. Da der Heap eine maximale Größe von $2k$ hat, kostet die Operation Einfügen oder Extrahieren höchstens $O(\log 2k)$. Die erwarteten Gesamtkosten, um einen Graphen mit dem Generator zu erzeugen, betragen $O\left(2k\binom{n}{2}p_{\max} + 2k\binom{n}{2}p_{\max}2\log 2k\right)$. Die erwartete Laufzeit des Verfahrens ist also in $O\left(kn^2 p_{\max} \log k\right)$.

Dieser Generator ist einfach zu implementieren, braucht sehr wenig Speicherplatz und hat eine gute Laufzeit. Der Generator ist vor allem effizient, wenn für jede Wahrscheinlichkeit der Anteil der von der Funktion edgeset angenommenen Knotenpaare größer ist. Dies ist oft der Fall bei den Graphen, die wir zur Evaluation benutzen wollen. Der Generator bietet keine große Verbesserung zu dem naiven Generator im Abschnitt 3.1, falls nur wenige Knotenpaare von der Funktion edgeset angenommen werden. Zum Beispiel geschieht der Worst Case im Vergleich zur Graphgröße bei der Laufzeit, wenn alle möglichen Knotenpaare getroffen werden und keins von der Funktion edgeset angenommen wird. Dieser Fall kann zum Beispiel vorkommen, wenn man in den Eingabeparametern eine Singletoncluster-ung mit $p^{in} = 1$ hat. Solche Fälle kommen in der Praxis kaum vor, so dass der Generator generell eine gute Laufzeit im Vergleich zur Graphgröße hat.

Wir stellen im nächsten Abschnitt einen anderen Generator mit einer Laufzeit abhängig von der Graphgröße vor.

3.3. Generator Kluges Traversieren + Heap

In diesem Abschnitt stellen wir einen anderen Generator vor, der die Zufallsgraphen mit einer von der Graphgröße abhängigen Laufzeit erzeugt. Dieser Generator ist sehr ähnlich zu dem im vorigen Abschnitt. Sein Vorteil ist, durch geschicktes Traversieren der für jede Clusterung gültigen Paare die Funktion edgeset überflüssig zu machen. Es wird also für jede Wahrscheinlichkeit p nur auf Knotenpaare gesprungen, die für diese Wahrscheinlichkeit gültig sind.

Wir berücksichtigen für jede Wahrscheinlichkeit p alle gültigen Knotenpaare und es wird nur auf diese Knotenpaare aufsteigend gesprungen. Das heißt, es wird auf Intra- bzw. Interclusterknotenpaare gesprungen, falls p eine Intra- bzw. Interclusterwahrscheinlichkeit ist. Auf Abbildung 3.3 ist ein Beispiel von Intra- und Interclusterknotenpaaren zu sehen. Damit wir nur Intra- oder Interclusterknotenpaare durchlaufen, nummerieren wir die Knoten so um, dass man zu jedem Zeitpunkt mit einem Sprung s das s-nächste Intra- oder Interclusterknotenpaar berechnen kann. Diese Umnummerierung sorgt dafür, dass die Knoten in jedem Cluster aufsteigend und aufeinanderfolgend besucht werden können. Das heißt, anhand dieser Umnummerierung ist für jeden Knoten der nächstgrößere Knoten im gleichen Cluster bekannt. Die beiden unteren Graphiken in Abbildung 3.4 stellen eine Clusterung vor und nach der Umnummerierung dar. Wenn bei allen Clusterungen in den Eingabeparametern die Knoten in jedem Cluster aufeinanderfolgende Nummer sind, dann kann man sich die Umnummerierung ersparen. Wir erläutern nun, wie wir die Knoten umnummerieren.

0	1	2

3	4	5

Clusterung $\mathcal{C}$.

Kantennummer	0	1	2	3	4	5	6	7
Knotenpaare	$\{0,1\}$	$\{0,2\}$	$\{1,2\}$	$\{0,3\}$	$\{1,3\}$	$\{2,3\}$	$\{0,4\}$	$\{1,4\}$

Kantennummer	8	9	10	11	12	13	14
Knotenpaare	$\{2,4\}$	$\{3,4\}$	$\{0,5\}$	$\{1,5\}$	$\{2,5\}$	$\{3,5\}$	$\{4,5\}$

Alle möglichen Knotenpaare.

Kantennummer	0	1	2	9	13	14
Intraclusterknotenpaare	$\{0,1\}$	$\{0,2\}$	$\{1,2\}$	$\{3,4\}$	$\{3,5\}$	$\{4,5\}$

Alle möglichen Intraclusterknotenpaare.

Kantennummer	3	4	5	6	7	8	10
Interclusterknotenpaare	$\{0,3\}$	$\{1,3\}$	$\{2,3\}$	$\{0,4\}$	$\{1,4\}$	$\{2,4\}$	$\{0,5\}$

Kantennummer	11	12
Interclusterknotenpaare	$\{1,5\}$	$\{2,5\}$

Alle möglichen Interclusterknotenpaare.

Abbildung 3.3.: Beispiel Intraclusterknotenpaare und Interclusterknotenpaare für eine Clusterung $\mathcal{C}[0,\ldots,5]$ mit zwei Clustern $\{0,1,2\}$ und $\{3,4,5\}$.

3.3.1. Datenstrukturen für die Umnummerierung der Knoten

Die KnotenIDs aus der Eingabe werden *global* genannt. Jeder Knoten $v \in \{0, \ldots, n-1\}$ hat mehrere IDs: eine globale ID bezeichnet mit $global(v) \in \{0, \ldots, n-1\}$ und je eine lokale ID pro Clusterung bezeichnet mit $local(v) \in \{0, \ldots, n-1\}$. Für die Berechnung der lokalen IDs nutzen wir die Idee von Bucketsort. Dafür wandeln wir jede Clusterung $\mathcal{C}[0, \ldots, n-1]$ in ein Feld $id\mathcal{C}[0, \ldots, \mathrm{ID}_{\max}]$ um. Der Wert $\mathrm{ID}_{\max}$ ist die größte ClusterID der Clusterung. Wir nehmen an, dass $\mathrm{ID}_{\max} \leq n-1$, da jede Clusterung höchstens n Cluster hat. Das Feld $id\mathcal{C}$ besteht aus $\mathrm{ID}_{\max} + 1$ Listen. Jeder Knoten v wird in die Liste Nummer $\mathcal{C}[v]$ eingefügt. Für jede Clusterung $\mathcal{C}$ nutzen wir die Datenstruktur $id\mathcal{C}$, um ein Feld $localV[0, \ldots, n-1]$ zu erstellen, in dem die lokalen IDs gespeichert werden. In diesem Feld ist also $localV[v] = local(v) = localV[global(v)]$ die lokale ID des Knotens v. Danach nutzen wir das Feld $localV$, um ein Feld $globalV$ zu erstellen, in dem für jeden lokalen Knoten $local(v)$ der entsprechende globale Knoten $global(v) = globalV[local(v)]$ gespeichert wird.

Wir erklären nun, wie für jede Clusterung $\mathcal{C}$ die Datenstrukturen $id\mathcal{C}$, $localV$ und $globalV$ erzielt werden können. Dafür können folgende Operationen ausgeführt werden.

1 Fülle die Datenstruktur $id\mathcal{C}$ nach dem Prinzip von Bucketsort auf. Dafür durchlaufe alle Knoten $v = 0, \ldots, n-1$ in dieser Reihenfolge und füge jeden Knoten am Ende der Liste $id\mathcal{C}[\mathcal{C}[v]]$ ein.

2 Betrachte nacheinander die Listen $id\mathcal{C}[0], \ldots, id\mathcal{C}[\mathrm{ID}_{\max}]$ in dieser Reihenfolge. Durchlaufe jede Liste vom ersten bis zum letzten Element und fülle dabei das Feld $localV$ so auf, dass $localV[global(v)] = i - 1$, falls der Knoten v bei diesem Durchlauf der $i-$te getroffene Knoten ist.

3 Durchlaufe das Feld $localV$ und fülle das Feld $globalV$, so dass für alle $i = 0, \ldots, n-1$ $globalV[localV[i]] = i$.

Abbildung 3.4 zeigt ein Beispiel für die Berechnung von lokalen Knoten einer Clusterung. Dort werden die Datenstrukturen idC, $localV$ und $globalV$ wie eben beschrieben dargestellt.

Der obige Prozess wird für alle k Clusterungen aus den Eingabeparametern ausgeführt. Bei jeder Clusterung $\mathcal{C}_i$, $i = 0, \ldots, k-1$ wird ein Feld $localV_i$ und $globalV_i$ erstellt. Anschließend werden alle k Felder $localV_i$ bzw. $globalV_i$ für $i = 0, \ldots, k-1$ in einem Feld LocalID$[0, \ldots, k-1]$ bzw. GlobalID$[0, \ldots, k-1]$ gespeichert. Die Felder LocalID und GlobalID bestehen also aus k Vektoren, so dass LocalID$[i] = localV_i$ und GlobalID$[i] = globalV_i$ für $i = 0, \ldots, k-1$. Man kann sich auch die Datenstrukturen LocalID und GlobalID als $k \times n$-Matrizen vorstellen. Mit diesen Datenstrukturen wird für jedes globale Knotenpaar $\{global(w), global(v)\}$ das entprechende lokale Knotenpaar $\{local(w), local(v)\}$ berechnet und umgekehrt. Die Abbildungen 3.5 und 3.6 illustrieren diese Datenstrukturen.

3.3.2. Beschreibung des Generators

Wir beschreiben nun die Vorgehensweise des Generators. Dafür erklären wir den Pseudocode in *Algorithm* 4. Die Datenstrukturen LocalID und GlobalID werden wie in den obigen Absätzen beschrieben initialisiert. Zusätzlich zu den drei Attributen, Kantennummer, Wahrscheinlichkeit und Nummer der Wahrscheinlichkeit vom Min-Heap im vorigen Generator 3.2, hat jeder Eintrag im Heap ein Attribut first(v), das den Wert der kleinsten lokalen ID im Cluster eines Knoten $local(v)$ enthält. Die Prozedur initialization-heap beschreibt die Initialisierung des Min-Heaps.

Lemma 2 *Im Algorithmus werden lokale Knotenpaare $\{local(w), local(v)\}$ für jede Clusterung so durchlaufen, dass zu jedem Zeitpunkt folgende Bedingungen gelten:*

v	0	1	2	3	4	5	6	7	8	9
$\mathcal{C}[v]$	0	0	0	3	3	0	3	7	3	7

Clusterung $\mathcal{C}$, $\mathrm{ID}_{\max} = 7$.

0	$\to 0 \to 1 \to 2 \to 5$
1	
2	
3	$\to 3 \to 4 \to 6 \to 8$
4	
5	
6	
7	$\to 7 \to 9$

Datenstruktur $id\mathcal{C}$.

i	0	1	2	3	4	5	6	7	8	9
$localV[i]$	0	1	2	4	5	3	6	8	7	9

Feld $localV[0, \ldots, 9]$ für Clusterung $\mathcal{C}$.

i	0	1	2	3	4	5	6	7	8	9
$globalV[i]$	0	1	2	5	3	4	6	8	7	9

Feld $globalV[0, \ldots, 9]$ für Clusterung $\mathcal{C}$.

0	1	2	5		3	4	6	8		7	9

Darstellung der Clusterung $\mathcal{C}$ mit globalen Knoten.

0	1	2	3		4	5	6	7		8	9

Darstellung der Clusterung $\mathcal{C}$ mit lokalen Knoten.

Abbildung 3.4.: Beispiel Berechnung lokale, globale Knoten und Darstellung der Datenstruktur $id\mathcal{C}$ für eine Clusterung $\mathcal{C}[0, \ldots, 9]$ mit drei Clustern $\{0, 1, 2, 5\}$, $\{3, 4, 6, 8\}$ und $\{7, 9\}$.

i		0	1	2	$\ldots$	$n-1$
0	$\xrightarrow{localV_0}$	$local_0$	$local_1$	$local_2$	$\ldots$	$local_{n-1}$
1	$\xrightarrow{localV_1}$	$local_0$	$local_1$	$local_2$	$\ldots$	$local_{n-1}$
2	$\xrightarrow{localV_2}$	$local_0$	$local_1$	$local_2$	$\ldots$	$local_{n-1}$
$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$
$k-1$	$\xrightarrow{localV_{k-1}}$	$local_0$	$local_1$	$local_2$	$\ldots$	$local_{n-1}$

Abbildung 3.5.: Datenstruktur LocalID.

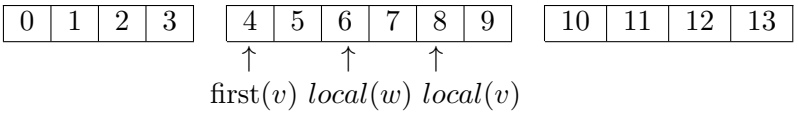
i		0	1	2	...	$n-1$
0	$globalV_0 \longrightarrow$	$global_0$	$global_1$	$global_2$	...	$global_{n-1}$
1	$globalV_1 \longrightarrow$	$global_0$	$global_1$	$global_2$	...	$global_{n-1}$
2	$globalV_2 \longrightarrow$	$global_0$	$global_1$	$global_2$	...	$global_{n-1}$
⋮		⋮	⋮	⋮	...	⋮
$k-1$	$globalV_{k-1} \longrightarrow$	$global_0$	$global_1$	$global_2$	...	$global_{n-1}$

Abbildung 3.6.: Datenstruktur GlobalID.

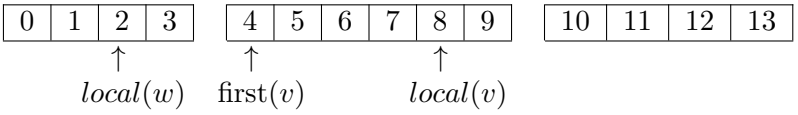
1 *Für eine Intraclusterwahrscheinlichkeit gelten* $\text{first}(v) \leq local(w) \lneq local(v)$.

2 *Für eine Interclusterwahrscheinlichkeit gelten* $local(w) \lneq \text{first}(v) \leq local(v)$.

Abbildung 3.7 zeigt einen Zustand der lokalen Knoten $local(v)$, $local(w)$ und $\text{first}(v)$ beim Durchlaufen der Intra- und Interclusterknotenpaare. Man kann sich dabei vorstellen, dass man drei Zeiger $local(v)$, $local(w)$ und $\text{first}(v)$ hat. Alle möglichen Intra- bzw. Interclusterknotenpaare $\{local(w), local(v)\}$ können durchlaufen werden, indem der Zeiger $local(v)$ nacheinander die Werte $0, \ldots, n-1$ bekommt, und für jeden Wert $local(v)$, der Zeiger $local(w)$ nacheinander die Werte $\text{first}(v), \ldots, local(v)-1$ bzw. $0, \ldots, \text{first}(v)-1$ bekommt.



Durchlaufen Intraclusterknotenpaare: $\text{first}(v) \leq local(w) \lneq local(v)$.



Durchlaufen Interclusterknotenpaare: $local(w) \lneq \text{first}(v) \leq local(v)$.

Abbildung 3.7.: Durchlaufen Intraclusterknotenpaare und Interclusterknotenpaare. Beispiel einer Clusterung mit drei Clustern $\{0,1,2,3\}$, $\{4,5,6,7,8,9\}$ und $\{10,11,12,13\}$

In Zeile 5 des Pseudocodes wird vom Min-Heap der Eintrag mit der kleinsten globalen Kantennummer extrahiert. Falls dieses globale Knotenpaar noch nicht im Graphen ist, dann wird es in den Graphen eingefügt und die zuletzt in den Graphen eingefügte Kantennummer $i_{\min}$ wird aktualisiert (Zeilen $8-10$). In den nächsten beiden Zeilen wird eine Zahl $r \in [0,1)$ gewürfelt und der entsprechende Sprung s berechnet. Die Prozedur next-pair berechnet das s-nächste lokale Intra- bzw. Interclusterknotenpaar $\{local(w), local(v)\}$, falls die betrachtete Wahrscheinlichkeit eine Intra- bzw. Interclusterwahrscheinlichkeit ist. Im Anhang befindet sich ein ausführlicher Pseudocode dieser Prozedur. In Zeile 14 wird geprüft, ob das berechnete lokale Knotenpaar noch im Bereich der potenziellen Kanten liegt. Falls ja, wird die entsprechende globale Kantennummer berechnet und in den Heap eingefügt. Das Verfahren wird in der Zeile 4 wiederholt, solange sich im Heap mindestens einen Eintrag befindet.

3.3.3. Laufzeitsanalyse und Speicherplatzbedarf des Generators

Wir berechnen zuerst den Speicherplatzbedarf des Generators. Danach beschreiben wir, wie die Laufzeit des Generators von der Größe der erzeugten Graphen abhängt.

Algorithm 4: Generator wise traverse + heap

Input: k partitions $\mathcal{C}_0\left(p_0^{in}, p_0^{out}\right), \ldots, \mathcal{C}_{k-1}\left(p_{k-1}^{in}, p_{k-1}^{out}\right)$ of n nodes $0, \ldots, n-1$ each with edge probability p^{in}, p^{out}
Output: a simple random graph $G = (V, E)$ with $V = \{0, \ldots, n-1\}$
Initialization: initialization-heap($minheap$); $E \leftarrow \emptyset$; $i_{\min} \leftarrow -1$; $local(v) \leftarrow 0$; $local(w) \leftarrow -1$; first$(v) \leftarrow 0$; initialize LocalID and GlobalID
while minheap *not empty* **do**
    $(i_{\text{top}}, p, ind_p, \text{first}(v)) \leftarrow$ extract min from minheap
    $global(w) \leftarrow \text{source}(i_{\text{top}})$
    $global(v) \leftarrow \text{target}(i_{\text{top}})$
    if $i_{\text{top}} > i_{\min}$ **then**
        $E \leftarrow E \cup \{global(w), global(v)\}$
        $i_{\min} \leftarrow i_{\text{top}}$
    draw $r \in [0, 1)$ uniformly at random
    $s \leftarrow 1 + \lfloor \frac{\log(1-r)}{\log(1-p)} \rfloor$
    next-pair$(local(v), local(w), \text{first}(v), ind_p, s)$
    if $local(v) < n$ **then**
        $i_{\text{top}} \leftarrow$ number of edge $\{global(w), global(v)\}$
        insert $(i_{\text{top}}, p, ind_p, \text{first}(v))$ to minheap with priority i_{top}

Procedure `initialization-heap`

Input: a heap minheap
for $j \leftarrow 0$ **to** $2k-1$ **do**
    if j *is even* **then**
        if $p_{\frac{j}{2}}^{in} > 0$ **then** insert $(-1, p_{\frac{j}{2}}^{in}, j, 0)$ to minheap with priority -1
    if j *is odd* **then**
        if $p_{\lfloor \frac{j}{2} \rfloor}^{out} > 0$ **then** insert $(-1, p_{\lfloor \frac{j}{2} \rfloor}^{out}, j, 0)$ to minheap with priority -1

Für den Generator werden die Datenstrukturen GlobalID und LocalID benutzt. Jede dieser beiden Datenstrukturen besteht aus k Feldern $localV_0, \ldots, localV_{k-1}$ oder $globalV_0, \ldots, globalV_{k-1}$. Um ein Feld $localV_i$ zu berechnen, wird eine Datenstruktur $id\mathcal{C}_i$ gebraucht, die $O(n)$ Speicherplatz benötigt. Für die Datenstrukturen GlobalID und LocalID werden also $O(kn)$ Speicherplatz benötigt. Im Generator wird ein Min-Heap mit der maximalen Größe $2k$ benutzt. Wir nehmen an, dass die erzeugten Kanten nicht gespeichert werden, sondern direkt ausgegeben. Der Generator benötigt also insgesamt $O(kn)$ Speicherplatz.

Für die Berechnung der lokalen Knoten $localV[0 \ldots, n-1]$ einer Clusterung $\mathcal{C}$ mithilfe der Datenstruktur $id\mathcal{C}$ werden $O(n)$ Schritte gebraucht. Die Laufzeit für die Berechnung aller lokalen Knoten in LocalID oder aller globalen Knoten in GlobalID ist also in $O(kn)$. Im Verfahren wird bei jedem Schritt nach einem Knotenpaar gesucht, das dem in diesem Schritt berechneten Sprung entspricht. Für jede Wahrscheinlichkeit liegt $local(w)$ insgesamt n-Male außerhalb des gültigen Bereichs. Dies verursacht einen Zusatzaufwand in $O(kn)$. Anhand jeder Wahrscheinlichkeit wird höchstens auf m Knotenpaare gesprungen. Im Worst Case werden für alle Intra- und Interclusterwahrscheinlichkeiten die gleichen Knotenpaare getroffen. In diesem Fall werden insgesamt km Knotenpaare getroffen. Jedes getroffene Knotenpaar wird in Min-Heap ein Mal eingefügt und ein Mal extrahiert in jeweils $O(\log 2k)$. Die gesamte Laufzeit des Generators ist also in $O(kn + km \log k)$.

3.4. Generator Kluges Traversieren + Vereinfachen

Wir stellen in diesem Abschnitt eine Modifizierung des vorigen Generators vor. Diese Modifizierung verbessert die Laufzeit um einen Faktor von $\log k$. Die Idee dieses Generators ist, wie im vorigen Generator die für jede Clusterung gültigen Knotenpaare zu traversieren. Hier wird kein Heap mehr benötigt. Jedes getroffene Knotenpaar wird ein Mal in eine Adjazenzliste eingetragen. In dieser Adjazenzliste werden wiederholte Knotenpaare in einem Nachbearbeitungsschritt entfernt. Der in der Adjazenzliste verbleibende Graph ist dann der einfache generierte Zufallsgraph.

Die Vorgehensweise des Generators kann mit folgenden Operationen beschrieben werden.

1 Traversiere für jede Wahrscheinlichkeit die gültigen Knotenpaare wie Algorithmus 3.5.

2 Trage jedes getroffene Knotenpaar $\{w, v\}$, $w < v$ in eine Adjazenzliste ein. Füge dafür den Knoten v in die Liste Nummer w ein.

3 Erstelle ein Feld edge-already-exist$[0, \ldots, n-1]$ und initialisiere edge-already-exist$[i] = 0$, $i = 0, \ldots, n-1$.

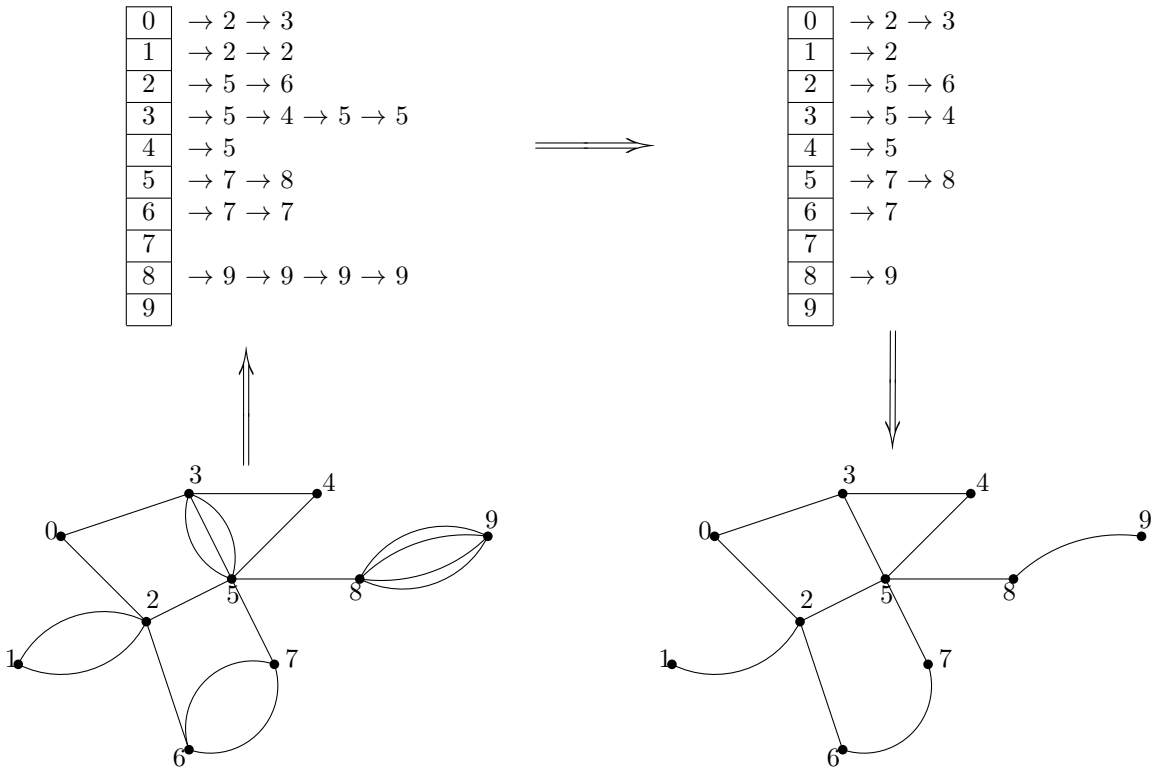
4 Durchlaufe jede Zeile der Adjazenzliste zwei Mal.

(i) Erster Durchlauf. Beim Treffen eines Knotens v: Wenn edge-already-exist$[v] = 0$, dann setze edge-already-exist$[v] = 1$. Wenn edge-already-exist$[v] = 1$, dann lösche den Knoten v.

(ii) Zweiter Durchlauf. Beim Treffen eines Knotens v setze wieder edge-already-exist$[v] = 0$.

Schaubild 3.8 illustriert die Vorgehensweise dieses Generators. *Algorithm* 6 stellt den Pseudocode des Generators vor. Wir beschreiben nun den Speicherplatzbedarf und die Laufzeit des Verfahrens.

Die Laufzeit und der Speicherplatzbedarf des Generators lassen sich ähnlich wie im Generator Kluges Traversieren + Heap analysieren. Zusätzlich zu den Speichern GlobalID und LocalID, die für das Traversieren der gültigen Knotenpaare benötigt werden, werden die



i	0	1	2	3	4	5	6	7	8	9
edge-already-exist[i]	0	0	0	0	1	1	0	0	0	0

Abbildung 3.8.: Beschreibung der Vorgehensweise des Generators. Der Graph mit den wiederholten Kanten wird in einer Adjazenzliste gespeichert. In der Adjazenzliste werden die wiederholten Kanten gelöscht. Der verbleibende Graph ist der einfache generierte Graph. Die untere Tabelle zeigt den Zustand des Felds edge-already-exist beim ersten Durchlauf der Zeile Nummer 3 der Adjazenzliste.

Algorithm 6: Generator wise traverse + simplification

Input: k partitions $\mathcal{C}_0\left(p_0^{in}, p_0^{out}\right), \ldots, \mathcal{C}_{k-1}\left(p_{k-1}^{in}, p_{k-1}^{out}\right)$ of n nodes $0, \ldots, n-1$ each with edge probability p^{in}, p^{out}
Output: a simple random graph $G = (V, E)$ with $V = \{0, \ldots, n-1\}$
Initialization: adjacencylist $\leftarrow$ empty adjacency list; initialize LocalID and GlobalID
for *each edge probability* p **do**
    $local(v) \leftarrow 0$; $local(w) \leftarrow -1$; $\text{first}(v) \leftarrow 0$
    while $local(v) < n$ **do**
        draw $r \in [0, 1)$ uniformly at random
        $s \leftarrow 1 + \lfloor \frac{\log(1-r)}{\log(1-p)} \rfloor$
        next-pair$(local(v), local(w), \text{first}(v), ind_p, s)$
        if $local(v) < n$ **then**
            insert the edge $\{global(w), global(v)\}$ to adjacencylist
simplify adjacencylist
$G \leftarrow$ adjacencylist
return G

Adjazenzliste mit dem Speicherplatzbedarf in $O(n + km)$ und das Feld edge-already-exist benötigt. Die Datenstrukturen edge-already-exist bzw. GlobalID und LocalID benötigen $O(n)$ bzw. $O(kn)$ Speicherplatz. Der Speicherplatzbedarf des Generators ist also in $O(kn + km)$.

Was die Laufzeit angeht, werden für das Traversieren der gültigen Knotenpaare $O(kn+km)$ Schritte benötigt. Die Adjazenzliste wird zwei Mal in $O(n + 2km)$ Schritte durchlaufen. Die Laufzeit des Generators ist also in $O(kn + km)$.

Jeder Algorithmus, der das im Abschnitt 2.4 vorgestellte Problem löst, muss zumindest alle Eingabeparameter einlesen. Dies erfordert kn Schritte. Um einen Zufallsgraphen zu erzeugen, der m Kanten enthält, werden mindestens m Schritte benötigt. Wir nehmen an, für jede Kante e muss entweder zumindest die Wahrscheinlichkeit $p(e)$ berechnet werden, mit der die Kante gewählt wird oder es kann nicht bei einer einzigen Clusterung und nach einem einzigen Versuch entschieden werden, dass die Kante nicht gewählt ist. Die Berechnung einer Wahrscheinlichkeit $p(e)$ benötigt nach Formel 2.1 $\Theta(k)$ Schritte. Unter dieser Annahme benötigt jede Lösung des Problems mindestens $kn + km$ Schritte. Der in diesem Abschnitt vorgestellte Generator hätte also eine optimale Laufzeit.

In der Praxis ist die Höhe der Hierarchie in sozialen Netzwerken eine kleine Zahl. Die Anzahl k der Clusterungen in Eingabeparametern kann also mit einer Konstante beschränkt werden. In diesem Fall benötigen die Generatoren Kluges Traversieren + Heap und Kluges Traversieren + Vereinfachen jeweils $O(n)$ und $O(n + m)$ Speicherplatz und haben eine Laufzeit in $O(n + m)$. Die Laufzeit und der Speicherbedarf dieser Generatoren sind also linear in der Größe des erzeugten Graphen.

3.5. Fazit und Vergleich der vier Generatoren

Wir fassen in Tabelle 3.2 die Vorteile zusammen, die jeder Generator im Vergleich mit den anderen hat.

Der naiver Generator ist ein einfaches Verfahren, für das kein extra Speicher für die Implementierung benötigt wird. Wegen seiner quadratischen Laufzeit eignet sich dieser Generator dafür, dichte Graphen zu erzeugen.

Der Generator Funktion + Heap benötigt wenig Speicherplatz, ist einfach zu implementieren und erzeugt sowohl dichte Graphen als auch dünne Graphen mit einer guten Laufzeit.

Der Generator Kluges Traversieren + Heap hat eine sehr gute Laufzeit, die von der Größe des erzeugten Graphen abhängig ist.

Unter der oben genannten Annahme hat der Generator Kluges Traversieren + Vereinfachen eine optimale Laufzeit für das behandelte Problem. Für diesen Generator muss der erzeugte Graph für das Vereinfachen gespeichert werden. Deshalb wird $O(km)$ Speicherplatz zusätzlich benötigt.

Generator	Laufzeit	Speicherplatz	Vorteil
Naiver Generator	$O(kn^2)$	nichts	Kein Bedarf an Speicherplatz
Funktion + Heap	$O\left(kn^2 p_{\max} \log k\right)$	$O(k)$	Einfach zu implementieren
Kluges Traversieren + Heap	$O(kn + km \log k)$	$O(kn)$	Laufzeit hängt von der Graphgröße ab
Kluges Traversieren + Vereinfachen	$O(kn + km)$	$O(kn + km)$	Optimale Laufzeit

Tabelle 3.2.: Vergleich der vier Generatoren

3.6. Beispiele von generierten Graphen

In diesem Abschnitt zeigen wir anhand Adjazenzmatrizen zwei Beispiele von bekannten Graphen, die wir mit dem Generator erzeugt haben. Die beiden Graphen sind aus dem Papier von Reichardt und Bornholdt [RB06] und werden benutzt, um Algorithmen zur Erkennung von hierarchischen oder sich überlagernden Clusterstrukturen zu vergleichen. Wir beschreiben, wie die Kanten in den Graphen verteilt sind und wie wir den Generator benutzt haben, um ähnliche Graphen zu erzeugen.

Schaubild 3.11 zeigt die Adjazenzmatrix eines Graphen mit einer sich überlagernden Clusterstruktur. Dieses Netzwerk hat 1024 Knoten und besteht aus zwei großen Clustern A und B. Jedes dieser Cluster enthält 512 Knoten. Innerhalb der Cluster A bzw. B gibt es Untergruppen a bzw. b von 128 Knoten. In jedem der beiden Cluster A und B gibt es im Durchschnitt 12 interne Links pro Knoten. Unter diesen 12 internen Links hat jeder Knoten in a bzw. b 6 Links mit den anderen 127 Knoten der Untergruppe a bzw. b. Die Untergruppen a und b sind im Durchschnitt mit 3 Links pro Knoten miteinander verbunden. Zusätzlich ist jeder Knoten im Durchschnitt mit zwei beliebigen Knoten im Netzwerk verbunden. Die Adjazenzmatrix des Graphen sollte also ähnlich wie auf dem Bild 3.9 aussehen. Auf dem Bild ist die Wahrscheinlichkeit, mit der Kanten in jedem Bereich gewählt werden, zu sehen.

Wir beschreiben nun, wie wir den Generator benutzt haben, um den oben beschriebenen Graphen zu konstruieren. Wir benutzen vier Clusterungen $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$ und $\mathcal{C}_3$. Das Bild 3.10 illustriert diese Clusterungen. Die Clusterung $\mathcal{C}_0$ teilt den Graphen in zwei Knotenteilmengen mit jeweils 512 Knoten. Diese Knotenteilmengen stellen die beiden großen Cluster A und B dar. Mit dieser Clusterung werden Kanten innerhalb der Cluster A und B mit einer Wahrscheinlichkeit p_0^{in} eingefügt. Alle anderen Kanten werden mit einer Wahrscheinlichkeit p_0^{out} eingefügt. Die zweite Clusterung $\mathcal{C}_1$ besteht aus einem Cluster, das die beiden

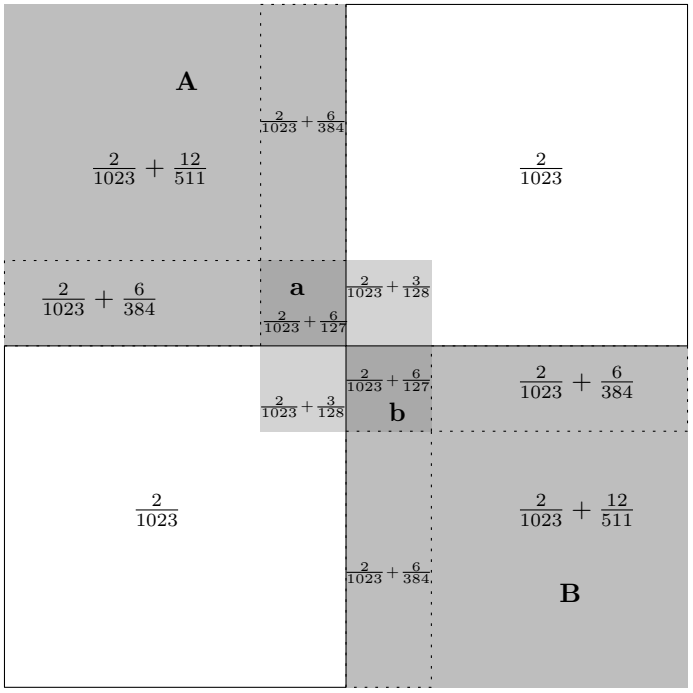


Abbildung 3.9.: Schema der Adjazenzmatrix eines Graphen mit einer sich überlagernden Clusterstruktur. Auf jedem Bereich werden Kanten mit der entsprechenden Wahrscheinlichkeit gewählt.

Cluster a und b enthält, und einem Singletoncluster für jeden Knoten, der nicht in a oder b liegt. Wir benutzen diese Clusterung, um Kanten zu erzeugen, die zwischen den Clustern a und b oder innerhalb dieser Cluster liegen. Die dritte Clusterung $\mathcal{C}_2$ besteht aus mehreren Clustern: a, b und ein Singletoncluster für jeden Knoten, der nicht in a oder b liegt. Wir benutzen diese Clusterung, um Kanten innerhalb der Cluster a und b zu erzeugen. Die vierte Clusterung $\mathcal{C}_3$ besteht aus den Clustern $A \setminus a$, $B \setminus b$ und einem Singletoncluster für jeden Knoten, der in a oder b liegt. Mit dieser Clusterung werden Kanten in $A \setminus a$ oder $B \setminus b$ erzeugt.

Für jede Clusterung sind die dazugehörigen Wahrscheinlichkeiten p^{in} und p^{out} wie folgt gewählt:

- Clusterung $\mathcal{C}_0$: Wir nutzen diese Clusterung so, dass alle Knotenpaare in A oder B mit einer gleichen Wahrscheinlichkeit p_0^{in} wie Knotenpaare, deren ein Endkonten in $A \setminus a$ bzw. $B \setminus b$ und der andere Endknoten in a bzw. b liegt, gewählt werden. Jedes beliebige Knotenpaar im Graphen wird mit einer Wahrscheinlichkeit p_0^{out} gewählt. Im Cluster A oder B ist jeder Knoten im Durchschnitt mit 12 inneren Knoten verbunden. Knoten in a bzw. b teilen 6 ihrer 12 internen Links mit den anderen 127 Knoten in a bzw. b. Im Netzwerk ist jeder Knoten im Allgemeinen zusätzlich mit 2 anderen beliebigen Knoten verbunden.

  $$p_0^{in} = \frac{2}{1023} + \frac{6}{384} = 0.01758$$

  $$p_0^{out} = \frac{2}{1023} = 0.00195$$

- Clusterung $\mathcal{C}_1$: Die zwischen a und b liegenden Knotenpaare sind bei der vorigen Clusterung mit einer Wahrscheinlichkeit p_0^{out} gewählt worden. Wir setzen für die Clusterung $\mathcal{C}_1$ die Wahrscheinlichkeit p_1^{in} so, dass diese Knotenpaare insgesamt mit einer höheren Wahrscheinlichkeit gewählt werden. Ein Knotenpaar, das in $a \cup b$ liegt

und nicht mit der Wahrscheinlichkeit p_0^{out} gewählt wird, wird nochmal mit der Wahrscheinlichkeit p_1^{in} gewählt. Die Untergruppen a und b sind im Allgemeinen mit 3 Links pro Knoten miteinander verbunden. Dazu wird noch hinzugefügt, dass jeder Knoten zusätzlich mit 2 beliebigen Knoten verbunden ist.

$$p_0^{out} + (1 - p_0^{out})p_1^{in} = \tfrac{2}{1023} + \tfrac{3}{128} \Rightarrow p_1^{in} = 0.02348$$
$$p_1^{out} = 0$$

- Clusterung $\mathcal{C}_2$: Für diese Clusterung setzen wir die Wahrscheinlichkeit p_2^{in} so, dass alle Knotenpaare innerhalb a oder b mit der korrekten gesamte Wahrscheinlichkeit gewählt werden. Dabei beachten wir, dass die Knotenpaare in diesen Untergruppen bereits mit niedrigeren Wahrscheinlichkeiten p_0^{in} und p_1^{in} gewählt werden. Im Cluster a oder b hat jeder Knoten im Durchschnitt 6 internen Links und 2 zusätzliche Links mit beliebigen Knoten im Graphen.

$$p_0^{in} + (1 - p_0^{in})p_1^{in} + (1 - p_0^{in})(1 - p_1^{in})p_2^{in} = \tfrac{2}{1023} + \tfrac{6}{127} \Rightarrow p_2^{in} = 0.00891$$
$$p_2^{out} = 0$$

- Clusterung $\mathcal{C}_3$: Intraclusterknotenpaare in $A \setminus a$ oder $B \setminus b$ werden bereits mit der Wahrscheinlichkeit p_0^{in} gewählt. Wir erhöhen diese Wahrscheinlichkeit mit p_3^{in} so, dass jeder Knoten in $A \setminus a$ bzw. $B \setminus b$ im Durchschnitt mit 12 Knoten in A bzw. B und zusätzlich mit 2 beliebigen Knoten verbunden ist.

$$p_0^{in} + (1 - p_0^{in})p_3^{in} = \tfrac{2}{1023} + \tfrac{12}{511} \Rightarrow p_3^{in} = 0.00799$$
$$p_3^{out} = 0$$

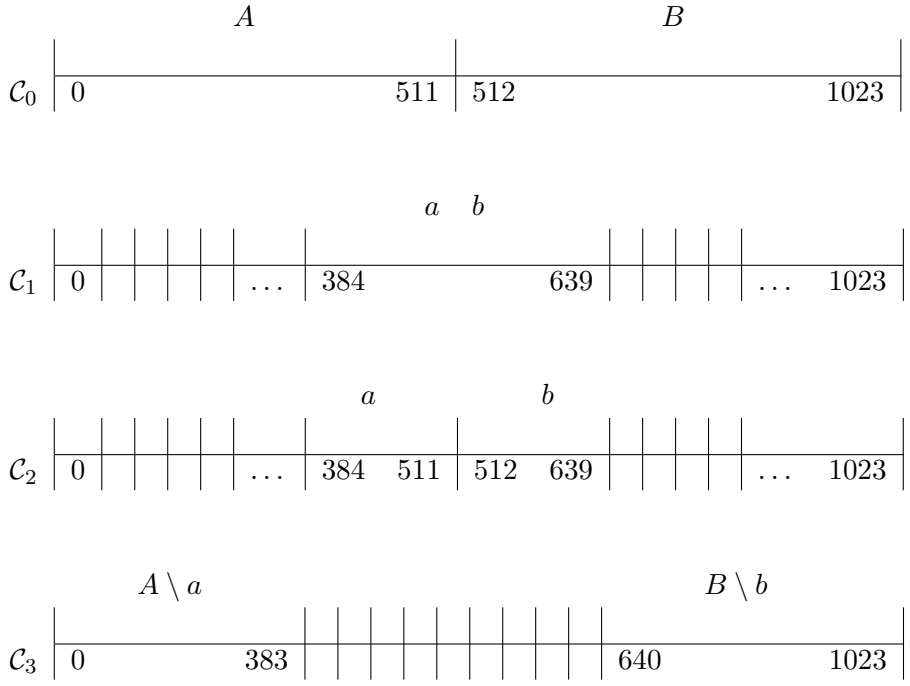


Abbildung 3.10.: Clusterungen $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$ und $\mathcal{C}_3$ für die Generierung des Graphen 3.11 mit einer sich überlagernden Clusterstruktur.

Schaubild 3.14 ist die Adjazenzmatrix eines Graphen mit einer hierarchischen Clusterstruktur. Das Netzwerk besteht aus vier Clustern mit jeweils 128 Knoten. In jedem Cluster hat jeder Knoten im Durchschnitt 7.5 Links mit den 127 internen Knoten und fünf Links mit den anderen 384 Knoten im Graphen. Jedes Cluster besteht wieder aus vier Clustern mit jeweils 32 Knoten, so dass innerhalb jedes Clusters jeder Knoten zusätzlich zehn Links mit

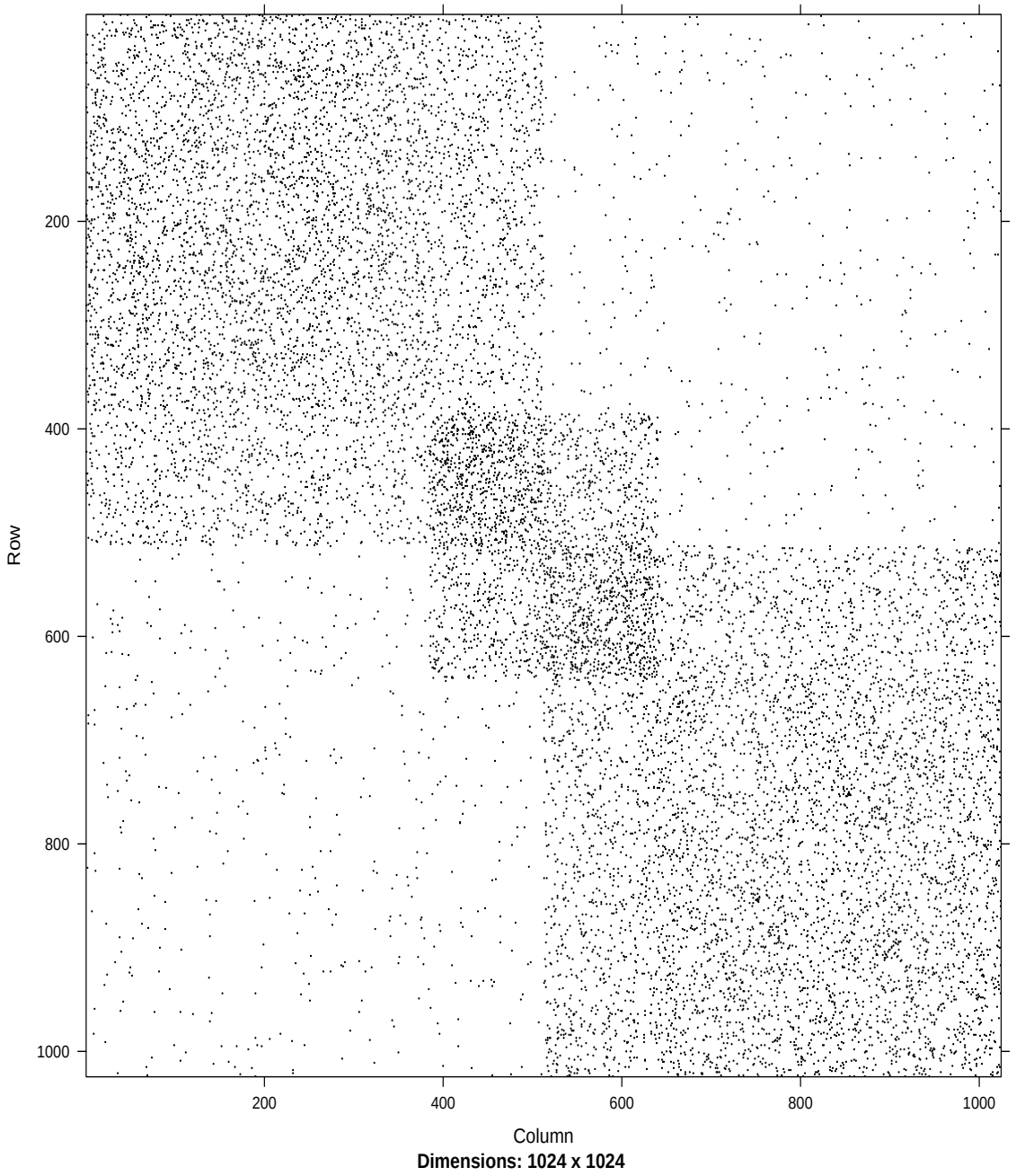


Abbildung 3.11.: Adjazenzmatrix eines Graphen mit einer sich überlagernden Clusterstruktur

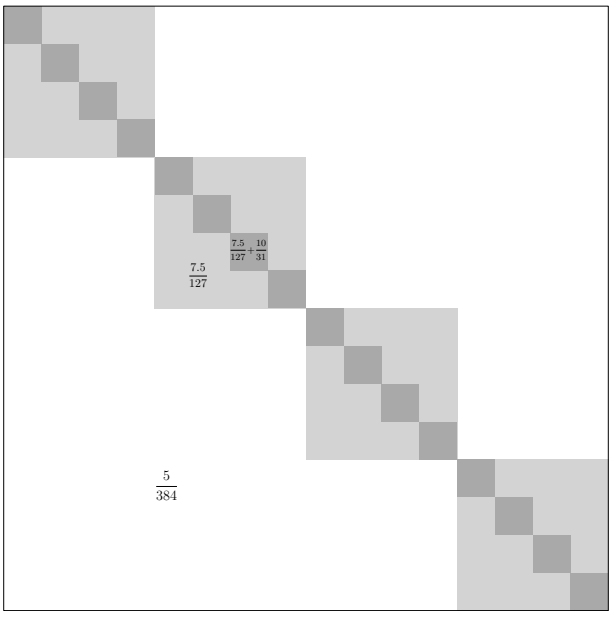


Abbildung 3.12.: Schema der Adjazenzmatrix eines Graphen mit einer hierarchischen Clusterstruktur. Auf jedem Bereich werden Kanten mit der entsprechenden Wahrscheinlichkeit gewählt.

den anderen 31 internen Knoten hat. Bild 3.12 zeigt das Schema der Adjazenzmatrix des Graphen.

Um diesen Graphen zu erzeugen, haben wir zwei Clusterungen $\mathcal{C}_0$ und $\mathcal{C}_1$ benutzt. Das Bild 3.13 zeigt, wie diese Clusterungen die Knotenmenge partitionieren. Die Clusterung $\mathcal{C}_0$ partitioniert die Knotenmenge in vier Knotenteilmengen. Mit dieser Clusterung werden Kanten auf *Level 1* der Hierarchie erzeugt. Die Clusterung $\mathcal{C}_1$ teilt die Knotenmenge in 16 Cluster und wird benutzt, um Kanten auf *Level 2* zu erzeugen. Wir haben die Wahrscheinlichkeiten p^{in} und p^{out} wie folgt berechnet:

- Clusterung $\mathcal{C}_0$: Für diese Clusterung lassen sich die Wahrscheinlichkeiten aus der Beschreibung des Graphen berechnen:

  $$p_0^{in} = \frac{7.5}{127} = 0.05905$$
  $$p_0^{out} = \frac{5}{384} = 0.01302$$

- Clusterung $\mathcal{C}_1$: Ein Intraclusterknotenpaar in dieser Clusterung ist auch ein Intraclusterknotenpaar in Clusterung $\mathcal{C}_0$. Wenn ein solches Knotenpaar nicht mit der Wahrscheinlichkeit p_0^{in} gewählt worden ist, wird es nochmal mit der Wahrscheinlichkeit p_1^{in} gewählt.

  $$p_0^{in} + (1 - p_0^{in})p_1^{in} = \frac{7.5}{127} + \frac{10}{31} \Rightarrow p_1^{in} = 0.34282$$
  $$p_1^{out} = 0$$

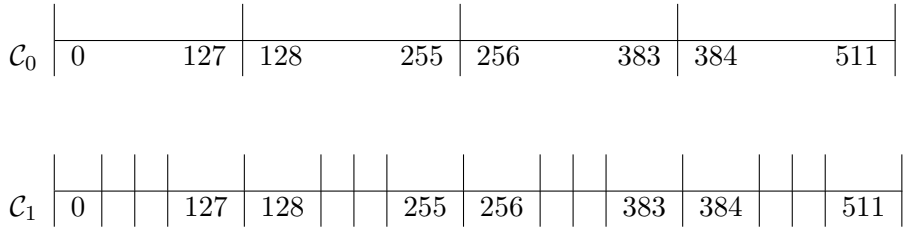


Abbildung 3.13.: Clusterungen $\mathcal{C}_0$ und $\mathcal{C}_1$ für die Generierung des Graphen 3.14 mit einer hierarchischen Clusterstruktur.

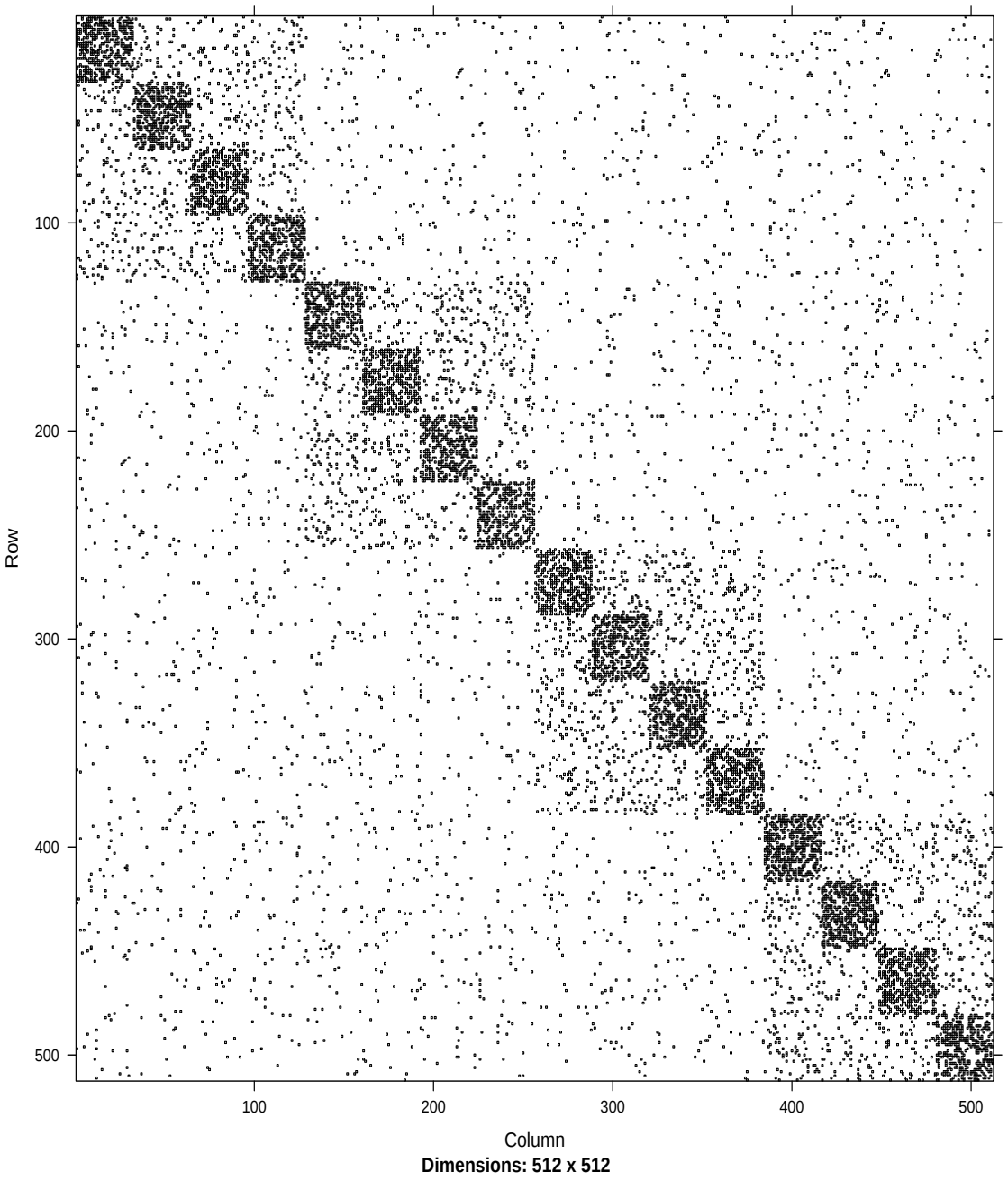


Abbildung 3.14.: Adjazenzmatrix eines Graphen mit einer hierarchischen Clusterstruktur

4. Betrachtete Algorithmen

Wir beschreiben in diesem Kapitel die Clusteringalgorithmen, die wir auf den von den Generatoren generierten Zufallsgraphen zur Auswertung ausgeführt haben. Wir beschreiben zuerst die Grundideen der Algorithmen, dann die von den Algorithmen genutzten Heuristiken, um die Grobheit der Clusterung zu beeinflussen. Danach beschreiben wir, welche dieser Kombinationen in der Literatur verwendet werden und anschließend die neuen Kombinationen, die wir ausgewertet haben.

4.1. Grundideen, um Modularity zu maximieren

In diesem Abschnitt beschreiben wir die auf Modularity basierenden Algorithmen. Ausgehend von der Idee, dass gute Clusterungen eines Graphen bessere Werte an Modularity haben, suchen diese Algorithmen mit verschiedenen Heuristiken eine Partition des Graphen mit einem möglichst großen Wert an Modularity. Die Algorithmen versuchen also die Zielfunktion Modularity zu maximieren. Die Optimierung von Modularity ist NP-Vollständig [BDH$^+$07] (Brandes et al.).

4.1.1. CNM Algorithmus

Der CNM Algorithmus [CNM04] betrachtet am Anfang jeden einzelnen Knoten des Graphen als Cluster und sucht bei jedem Schritt zwei Cluster, für die der größte Gewinn an Modularity bei ihrer Vereinigung erzielt wird. Die beiden Cluster werden zu einem Cluster vereinigt. Die Clusterpaare werden iterativ vereinigt, bis es kein Clusterpaar mehr gibt, dessen Vereinigung einen Zuwachs an Modularity bringt.

Seien $\mathcal{C} = \{C_1, \ldots, C_l\}$ eine betrachtete Clusterung, $\mathcal{C}_{i,j}$ die erzielte Clusterung, falls Cluster C_i und C_j vereinigt werden und $\Delta q_{i,j}$ die bei der Vereinigung verursachte Veränderung

an Modularity (Gewinn oder Verlust).

$$
\begin{aligned}
\Delta q_{i,j} &= q(\mathcal{C}_{i,j}) - q(\mathcal{C}) \\
&= \sum_{C \in \mathcal{C}_{i,j}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} \deg(v)}{2m} \right)^2 \right] - \sum_{C \in \mathcal{C}} \left[\frac{|E(C)|}{m} - \left(\frac{\sum_{v \in C} \deg(v)}{2m} \right)^2 \right] \\
&= \sum_{\substack{C_k \in \mathcal{C}, \\ k \neq i,j}} \left[\frac{|E(C_k)|}{m} - \left(\frac{\sum_{v \in C_k} \deg(v)}{2m} \right)^2 \right] + \frac{|E(C_i \cup C_j)|}{m} - \left(\frac{\sum_{v \in C_i \cup C_j} \deg(v)}{2m} \right)^2 \\
&\quad - \left[\sum_{\substack{C_k \in \mathcal{C}, \\ k \neq i,j}} \left[\frac{|E(C_k)|}{m} - \left(\frac{\sum_{v \in C_k} \deg(v)}{2m} \right)^2 \right] + \frac{|E(C_i)|}{m} - \left(\frac{\sum_{v \in C_i} \deg(v)}{2m} \right)^2 \right. \\
&\quad \left. + \frac{|E(C_j)|}{m} - \left(\frac{\sum_{v \in C_j} \deg(v)}{2m} \right)^2 \right] \\
&= \frac{|E(C_i \cup C_j)| - |E(C_i)| - |E(C_j)|}{m} - \left(\frac{\sum_{v \in C_i} \deg(v) + \sum_{v \in C_j} \deg(v)}{2m} \right)^2 \\
&\quad + \left(\frac{\sum_{v \in C_i} \deg(v)}{2m} \right)^2 + \left(\frac{\sum_{v \in C_j} \deg(v)}{2m} \right)^2 \\
\Delta q_{i,j} &= \frac{|E(C_i, C_j)|}{m} - 2 \left(\frac{\sum_{v \in C_i} \deg(v)}{2m} \right) \left(\frac{\sum_{v \in C_j} \deg(v)}{2m} \right)
\end{aligned}
$$

Bei der Implementierung wird für jedes Clusterpaar C_i, C_j die Anzahl der Kanten gespeichert, die beiden Cluster verbinden. Für jedes Cluster wird die Summe der Grade aller inneren Knoten gespeichert. Der Wert $\Delta q_{i,j}$ lässt sich also für jedes Clusterpaar schnell berechnen. Die Aktualisierung dieser Werte im Laufe des Algorithmus ist auch schnell.

Ein binärer Baum namens *Dendrogramm* lässt sich bei der Ausführung des CNM Algorithmus konstruieren. Es ist ein Baum, dessen Blätter die Knoten des Graphen sind. Ein Baumknoten bildet sich bei der Vereinigung eines Clusterpaares. Dabei sind die vereinigten Cluster die beiden Kinder. Die Laufzeit des CNM Algorithmus hängt von der Höhe dieses Dendrogramms ab.

Um das Clusterpaar mit dem größten Gewinn an Modularity, zu berechnen, werden für alle Clusterpaare C_i, C_j der Wert $\Delta q_{i,j}$ in einem Max-Heap gespeichert. Bei jeder Vereinigung $C_i \cup C_j$ werden für alle zu C_i oder C_j adjazenten Cluster C_k die Heapeinträge $\Delta q_{i,k}$ oder $\Delta q_{k,j}$ aktualisiert. Die gesamten Kosten im Algorithmus hängen von der Anzahl der gesamten Aktualisierungen ab. Sei d die Höhe des Dendrogramms. Für jeden Knoten u wird es insgesamt $O(\deg(u))$ Mal aktualisiert und es gibt höchstens d Vereinigungen, bei denen der Knoten inzident zu den vereinigten Clustern ist. Die Gesamtkosten im Algorithmus sind in $O(md \log n)$. Für das Optimierungsproblem Modularity hat der Algorithmus keine Gütegarantie.

4.1.2. CNM Algorithmus mit Prioritätsfunktion

Es ist beobachtet worden, dass das bei der Ausführung des CNM Algorithmus auf Graphen erzielte Dendrogramm unbalanciert ist, weil der CNM Algorithmus oft bei der Vereinigung Cluster bevorzugt, die viele Knoten enthalten. Die vom CNM Algorithmus erzielte Clusterung könnte wieder erzielt werden, wenn die Cluster in einer anderen Reihenfolge vereinigt wurden. Die *Prioritätsfunktion* ersetzt den Zuwachs an Modularity und entscheidet bei jedem Schritt des Algorithmus darüber, welches Clusterpaar vereinigt wird, um eine andere

Reihenfolge bei der Vereinigung von Clusterpaaren zu erzielen, so dass das Dendrogramm balanciert und der CNM Algorithmus folglich schneller wird. Eine Beschreibung verschiedener Prioritätsfunktionen ist im Kapitel 3 [Laf10] zu sehen.

Wir haben zur Auswertung die Prioritätsfunktion *Significance* abgekürzt *Sig* [NR09] betrachtet, weil diese Prioritätsfunktion im Allgemeinen recht gute Ergebnisse liefert. Mit dieser Prioritätsfunktion wird im CNM Algorithmus das Clusterpaar C_i, C_j mit dem größten Wert

$$\frac{\Delta q_{i,j}}{\sqrt{\deg(C_i)\deg(C_j)}}$$

vereinigt, wobei $\deg(C) = \sum_{v \in C} \deg(v)$ für ein Cluster C.

4.1.3. Multi-Level Algorithmus von Blondel

Der von Blondel et al. entwickelte Algorithmus [BGLL08] beginnt mit der Singletonclusterung und besteht aus zwei Phasen.

Bei der ersten Phase werden Knoten im Graphen nacheinander in einer beliebigen Reihenfolge besucht. Beim Besuch eines Knoten wird für jedes adjazente Cluster der Gewinn an Modularity berechnet, der erzielt würde, wenn man den Knoten von seinem Cluster in dieses Nachbarcluster schiebt. Der Knoten wird in das Cluster geschoben, bei dem diese Bewegung den größten Gewinn an Modularity bringt. Falls es mehrere solche Cluster gibt, wird der Knoten in ein beliebiges darunter geschoben. Falls es kein Nachbarcluster gibt, in das die Bewegung des Knoten einen Zuwachs an Modularity bringt, bleibt der Knoten in seinem Cluster. Falls ein besserer Gewinn an Modularity erzielt wird, wenn der Knoten sein eigenes Cluster bildet, dann wird der Knoten isoliert. Die Knoten werden nacheinander so oft bewegt, bis es keinen Knoten mehr gibt, dessen Bewegung einen Zuwachs an Modularity bringt. Die bei dieser ersten Phase erzielte Clusterung entspricht einem Level auf der Hierarchie.

Bei der zweiten Phase wird ausgehend von der aus der ersten Phase erzielten Clusterung ein Graph konstruiert. Jedes aus der ersten Phase erzielte Cluster wird zu einem Knoten kontrahiert. Jeder neue Knoten hat eine Schleife, dessen Gewicht die Summe aller Gewichte der Intraclusterkanten im kontrahierten Cluster ist. Eine Kante bildet sich zwischen zwei neuen Knoten, falls die entsprechenden Cluster adjazent sind, und das Gewicht der neuen Kante ist die Summe aller Gewichte der Interclusterkanten zwischen beiden Clustern. Nach der zweiten Phase wird der Algorithmus auf dem neuen Graphen wiederholt. Das Verfahren wird iterativ durchgeführt, bis zu einem Level, wo keine Bewegung eines Knoten Modularity verbessert oder der ganze Graph kontrahiert wird.

Seien u ein Knoten im Cluster C_1, C_2 ein zu C_1 adjazentes Cluster, $e_{C_1,u}$ bzw. $e_{C_2,u}$ das Gesamtgewicht der Kanten, die u mit dem Cluster C_1 bzw. C_2 verbinden, $\mathcal{C}$ die betrachtete Clusterung, $\mathcal{C}_{C_2 \cup \{u\}}$ die erzielte Clusterung, falls Knoten u sich ins Cluster C_2 bewegt und

Δq die bei dieser Bewegung verursachte Veränderung an Modularity.

$$
\begin{aligned}
\Delta q &= q(\mathcal{C}_{C_2\cup\{u\}}) - q(\mathcal{C}) \\
&= \sum_{C\in\mathcal{C}_{C_2\cup\{u\}}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] - \sum_{C\in\mathcal{C}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] \\
&= \sum_{\substack{C\in\mathcal{C}_{C_2\cup\{u\}},\\ C\neq C_1, C\neq C_2}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] + \frac{\omega(C_1\setminus\{u\})}{W} - \left(\frac{\sum_{v\in C_1\setminus\{u\}}\deg_\omega(v)}{2W}\right)^2 \\
&\quad + \frac{\omega(C_2\cup\{u\})}{W} - \left(\frac{\sum_{v\in C_2\cup\{u\}}\deg_\omega(v)}{2W}\right)^2 - \sum_{\substack{C\in\mathcal{C}_{C_2\cup\{u\}},\\ C\neq C_1, C\neq C_2}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] \\
&\quad - \frac{\omega(C_1)}{W} + \left(\frac{\sum_{v\in C_1}\deg_\omega(v)}{2W}\right)^2 - \frac{\omega(C_2)}{W} + \left(\frac{\sum_{v\in C_2}\deg_\omega(v)}{2W}\right)^2 \\
&= \frac{\omega(C_1\setminus\{u\}) - \omega(C_1)}{W} + \frac{\omega(C_2\cup\{u\}) - \omega(C_2)}{W} + \frac{1}{(2W)^2}\left[\left(\sum_{v\in C_1}\deg_\omega(v)\right)^2\right. \\
&\quad \left. - \left(\sum_{v\in C_1\setminus\{u\}}\deg_\omega(v)\right)^2 + \left(\sum_{v\in C_2}\deg_\omega(v)\right)^2 - \left(\sum_{v\in C_2\cup\{u\}}\deg_\omega(v)\right)^2\right] \\
\Delta q &= \frac{1}{W}\left(e_{C_2,u} - e_{C_1,u}\right) - \frac{1}{2W^2}\omega\deg(u)\left(\sum_{v\in C_1}\deg_\omega(v) - \sum_{v\in C_2}\deg_\omega(v) - \omega\deg(u)\right)
\end{aligned}
$$

Falls der größte Gewinn an Modularity erzielt wird, wenn der Knoten sich isoliert, dann wird die Veränderung an Modularity analog berechnet.

Die Autoren haben den Algorithmus auf mehreren Graphen ausgeführt und dabei große Werte an Modularity erzielt. Der Algorithmus ist auch schnell sowohl bei der Ausführung auf kleine als auch auf großen sozialen Netzwerken. Das liegt teilweise daran, dass der Wert Δq schnell zu berechnen ist. Die Aktualisierung der Komponenten in der Formel lassen sich bei jeder Bewegung eines Knoten auch schnell berechnen. Es gibt aber keine gute Abschätzung der Laufzeit.

4.1.4. Multi-Level Algorithmus von Blondel mit Verfeinerung

Die Idee bei der Verfeinerung des Algorithmus von Blondel ist, nach der Ausführung des Algorithmus die gebildeten Gruppen von Knoten in andere Cluster zu bewegen, um den Wert an Modularity zu verbessern.

Dafür werden die Cluster eines Levels auf den nächstniedrigeren Level projiziert. Eine Gruppe von Knoten wird analog zur ersten Phase des Algorithmus in ein anderes Cluster geschoben, falls diese Bewegung einen Zuwachs an Modularity bringt. Die betrachteten Gruppen von Knoten sind Cluster, die sich im Laufe des Algorithmus auf dem entsprechenden Level gebildet haben. Die Verfeinerung wird iterativ vom letzten Level auf den niedrigsten Level durchgeführt. Diese Verfeinerung wird zum ersten Mal von Noack und Rotta eingeführt [NR09].

4.2. Möglichkeiten, die Grobheit der Clusterung zu beeinflussen

In diesem Abschnitt beschreiben wir einige Heuristiken, die Clusteringalgorithmen nutzen, um mehrere Clusterungen eines Graphen zu erzielen. Diese Heuristiken sind vor allem

benutzt, um verschiedene sinnvolle Clusterungen eines Graphen mit einer hierarchischen bzw. sich überlagernden Clusterstruktur zu erzielen.

4.2.1. Hierarchie oder Zwischenergebnisse

Die Idee hinter der Heuristik Hierarchie oder Zwischenergebnisse, ist bei der Ausführung der Modularity-basierten Algorithmen jede Clusterung berechnen zu können, die bei einer Kontraktion der Knoten erzielt wird. Man will also jede Clusterung berechnen, die sich im Laufe des Algorithmus gebildet hat.

4.2.2. Einschränkung mit GID

GID bedeutet *global intracluster density*. Es ist für eine Clusterung $\mathcal{C}$ die Summe aller Intraclusterkanten geteilt durch die Summe aller Intraclusterpaare

$$\mathrm{GID}(\mathcal{C}) = \frac{\sum_{C \in \mathcal{C}} m_C}{\sum_{C \in \mathcal{C}} \binom{n_C}{2}},$$

wobei m_C die Anzahl der Intraclusterkanten und n_C die Anzahl der Knoten in einem Cluster C bezeichnen.

Die Grobheit der Clusterung wird mit dieser Heuristik beeinflusst, indem für einen Wert α unter allen Clusterungen mit $\mathrm{GID}(\mathcal{C}) > \alpha$ die Clusterung mit der besten Modularity gesucht wird. Mit mehreren Werten von $0 \leq \alpha \leq 1$ wird nach verschiedenen Clusterungen eines Graphen gesucht. Mit großen Werten von α wird nach Clusterungen mit feinen Clustern gesucht. Mit kleinen Werten von α werden grobe Clusterungen gebildet. Görke, Schumm und Wagner beschreiben diese Heuristik in [GSW11].

4.2.3. Idee von Bornholdt

Bornholdt [RB06] will die Grobheit der Clusterung beeinflussen, indem er den zweiten Term in der Formel von Modularity mit einem Paramater γ beeinflusst.

$$q'(\mathcal{C}) = cov(\mathcal{C}) - \gamma \mathbb{E}(cov(\mathcal{C}))$$

Die Idee ist, verschiedene Clusterungen eines Graphen mit unterschiedlichen Werten von γ zu finden. Mit dem Parameter $\gamma \in \mathbb{R}, \gamma \geq 0$ werden für großere Werte feine Cluster gebildet, und für kleine Werte Clusterungen mit groben Clustern erzielt.

4.3. Verwendete Kombinationen aus der Literatur

Wir beschreiben nun die bereits in der Literatur vorhandenen Kombinationen von Grundideen aus Abschnitt 4.1 und Möglichkeiten aus Abschnitt 4.2.

4.3.1. Multi-Level Algorithmus von Blondel mit Verfeinerung und Einschränkung GID

In dieser Kombination [GSW11] wird ein Knoten in der ersten Phase im Algorithmus von Blondel nicht mehr direkt in das Cluster geschoben, in das die Bewegung den größten Gewinn an Modularity bringt. Stattdessen wird der Knoten in das Nachbarcluster bewegt, das wie folgt bestimmt wird: Sei $\mathcal{C}$ die erzielte Clusterung, nachdem der Knoten sich in ein Nachbarcluster bewegt hat. Der Knoten wird in das Cluster geschoben, das unter allen Nachbarclustern mit $\mathrm{GID}(\mathcal{C}) > \alpha$ den größten Zuwachs an Modularity bringt. Bei der Verfeinerung wird analog darüber entschieden, in welches Cluster eine Gruppe von Knoten geschoben wird. Der Algorithmus wird mit mehreren Werten $0 \leq \alpha \leq 1$ ausgeführt. Mit dem Wert $\alpha = 0$ entspricht dieser Algorithmus dem Blondel Algorithmus.

4.3.2. CNM Algorithmus mit der Abwandlung von Bornholdt

Reichardt und Bornholdt [RB06] wandeln den CNM Algorithmus ab. Für jeden Wert γ wird bei jedem Schritt im CNM Algorithmus das Knotenpaar C_i, C_j vereinigt, dessen Vereinigung den größten positiven Wert $\Delta q'_{i,j}$ verursacht.

Seien $\mathcal{C} = \{C_1, \ldots, C_l\}$ eine betrachtete Clusterung, $\mathcal{C}_{i,j}$ die erzielte Clusterung, falls Cluster C_i und C_j vereinigt werden.

$$\begin{aligned}
\Delta q'_{i,j} &= q'(\mathcal{C}_{i,j}) - q'(\mathcal{C}) \\
&= cov(\mathcal{C}_{i,j}) - \gamma\mathbb{E}(cov(\mathcal{C}_{i,j})) - cov(\mathcal{C}) + \gamma\mathbb{E}(cov(\mathcal{C})) \\
&= \left[\sum_{C\in\mathcal{C}_{i,j}} \frac{|E(C)|}{m} - \sum_{C\in\mathcal{C}} \frac{|E(C)|}{m}\right] - \gamma\left[\sum_{C\in\mathcal{C}_{i,j}} \left(\frac{\sum_{v\in C}\deg(v)}{2m}\right)^2 - \sum_{C\in\mathcal{C}} \left(\frac{\sum_{v\in C}\deg(v)}{2m}\right)^2\right] \\
&= \frac{|E(C_i \cup C_j)| - |E(C_i)| - |E(C_j)|}{m} - \gamma\left[\left(\frac{\sum_{v\in C_i\cup C_j}\deg(v)}{2m}\right)^2 - \left(\frac{\sum_{v\in C_i}\deg(v)}{2m}\right)^2\right. \\
&\quad \left. - \left(\frac{\sum_{v\in C_j}\deg(v)}{2m}\right)^2\right] \\
\Delta q'_{i,j} &= \frac{|E(C_i, C_j)|}{m} - 2\gamma\left(\frac{\sum_{v\in C_i}\deg(v)}{2m}\right)\left(\frac{\sum_{v\in C_j}\deg(v)}{2m}\right)
\end{aligned}$$

Ähnlich wie beim CNM Algorithmus ist dieser Wert für jedes Clusterpaar schnell zu berechnen. Die Aktualisierung der Terme für die Formel bei jeder Vereinigung ist die gleiche wie im CNM Algorithmus. Diese Version von Bornholdt lässt sich also gut implementieren, in dem man im CNM den Wert $\Delta q_{i,j}$ durch $\Delta q'_{i,j}$ ersetzt und den Algorithmus mit mehreren Werten $\gamma \geq 0$ ausführt.

4.4. Neue Kombinationen

Wir stellen nun neue Kombinationen vor, die wir aus den Grundideen im Abschnitt 4.1 und Heuristiken im Abschnitt 4.2 zur Auswertung untersucht haben.

4.4.1. CNM Hierarchie

Bei CNM Hierarchie wird der CNM Algorithmus solange ausgeführt, bis kein Knotenpaar mehr im Graphen ist. Die Idee dabei ist, nicht nur die einzige Clusterung zu betrachten, die der CNM Algorithmus liefert, sondern auch alle Clusterungen zu berücksichtigen, die sich im Laufe des Algorithmus bilden. Diese Gedanke ist nicht ganz neu. Fortunato hat bereits vorgeschlagen, bei der Ausführung des CNM Algorithmus andere gebildeten Clusterungen zu berücksichtigen [For10]. Wir berücksichtigen noch mehr Clusterungen, da wir der CNM solange ausführen, bis nur ein Cluster im Graphen verbleibt. Wir beschreiben nun, wie wir alle gebildeten Clusterungen im Laufe des Algorithmus erzielt haben.

Bei jedem Schritt im CNM Algorithmus wird das Knotenpaar C_i, C_j vereinigt, dessen Vereinigung den größten Wert $\Delta q_{i,j}$ (positiv oder negativ) verursacht. Beginnend mit der Singletonclusterung werden Knoten iterativ vereinigt. Es wird insgesamt $n-1$ Mal vereinigt. Nach der Vereinigung von zwei Clustern $i < j$ heißt das neue Cluster i. Wir speichern dabei in einem Feld predecessor$[j] = i$. Am Anfang gilt predecessor$[i] = i, i = 0, \ldots, n-1$. Mit diesem Feld wissen wir zu jedem Zeitpunkt $t = 1, \ldots, n$ die gebildete Clusterung $\mathcal{C}_t[0, \ldots, n-1]$. Die Clusterung wird rekursiv konstruiert. Für $i = 0, \ldots, n-1$ gilt, $\mathcal{C}_t[i] = i$

falls predecessor$[i] = i$, sonst $\mathcal{C}_t[i] = \mathcal{C}_t[\text{predecessor}[i]]$. Falls ein Knoten der kleinste in seinem Cluster ist, wird seine ClusterID in einem Schritt berechnet, sonst werden zwei Schritte benötigt. Zu jedem Zeitpunkt im Algorithmus wird die gebildete Clusterung in $O(n)$ Schritten, also in linearer Zeit, berechnet.

4.4.2. CNM Hierarchie mit Prioritätsfunktion

Mit dieser Kombination wollen wir den CNM Algorithmus mit einer Prioritätsfunktion ausführen und dabei zu jedem Zeitpunkt die gebildete Clusterung berücksichtigen. Die Motivation dabei ist, bei der Vereinigung im CNM Algorithmus keine große Cluster zu bevorzugen, damit zu jedem Zeitpunkt die gebildete Clusterung anders wird. Zu jedem Zeitpunkt im CNM Algorithmus ist die Clusterung, die man mit einem balancierten Dendrogramm erzielen würde anders als die Clusterung, die sich mit einem unbalancierten Dendrogramm des CNM bildet. Dafür haben wir den CNM Hierarchie mit der Prioritätsfunktion *Sig* ausgeführt. Die Clusterungen werden wie im vorherigen Abschnitt beschrieben konstruiert.

4.4.3. Kombination Blondel Bornholdt

Wir haben den Multi-Level Algorithmus von Blondel und die Idee von Bornholdt kombiniert. Ähnlich wie Bornholdt den CNM Algorithmus abgewandelt hat, ersetzen wir im Algorithmus von Blondel die Zielfunktion Modularity $q(\mathcal{C}) = cov(\mathcal{C}) - \mathbb{E}(cov(\mathcal{C}))$ durch $q'(\mathcal{C}) = cov(\mathcal{C}) - \gamma\mathbb{E}(cov(\mathcal{C}))$. Wir wollen also für verschiedene Werte von γ unterschiedliche Clusterungen erzielen.

Um einen im Cluster C_1 liegenden Knoten u in ein anderes Cluster zu bewegen, suchen wir nach dem adjazenten Cluster C_2, in das diese Bewegung den größten Gewinn $\Delta q'$ bringt. Seien $e_{C_1,u}$ bzw. $e_{C_2,u}$ das Gesamtgewicht der Kanten, die u mit dem Cluster C_1 bzw. C_2 verbinden, $\mathcal{C}$ die betrachtete Clusterung und $\mathcal{C}_{C_2\cup\{u\}}$ die erzielte Clusterung, falls Knoten u sich ins Cluster C_2 bewegt.

$$
\begin{aligned}
\Delta q' &= q'(\mathcal{C}_{C_2\cup\{u\}}) - q'(\mathcal{C}) \\
&= \sum_{C\in\mathcal{C}_{C_2\cup\{u\}}} \left[\frac{\omega(C)}{W} - \gamma\left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] - \sum_{C\in\mathcal{C}} \left[\frac{\omega(C)}{W} - \gamma\left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] \\
&= \sum_{\substack{C\in\mathcal{C}_{C_2\cup\{u\}},\\ C\neq C_1, C\neq C_2}} \left[\frac{\omega(C)}{W} - \gamma\left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] + \frac{\omega(C_1\setminus\{u\})}{W} - \gamma\left(\frac{\sum_{v\in C_1\setminus\{u\}}\deg_\omega(v)}{2W}\right)^2 \\
&\quad + \frac{\omega(C_2\cup\{u\})}{W} - \gamma\left(\frac{\sum_{v\in C_2\cup\{u\}}\deg_\omega(v)}{2W}\right)^2 - \sum_{\substack{C\in\mathcal{C}_{C_2\cup\{u\}},\\ C\neq C_1, C\neq C_2}} \left[\frac{\omega(C)}{W} - \gamma\left(\frac{\sum_{v\in C}\deg_\omega(v)}{2W}\right)^2\right] \\
&\quad - \frac{\omega(C_1)}{W} + \gamma\left(\frac{\sum_{v\in C_1}\deg_\omega(v)}{2W}\right)^2 - \frac{\omega(C_2)}{W} + \gamma\left(\frac{\sum_{v\in C_2}\deg_\omega(v)}{2W}\right)^2 \\
&= \frac{\omega(C_1\setminus\{u\}) - \omega(C_1)}{W} + \frac{\omega(C_2\cup\{u\}) - \omega(C_2)}{W} + \frac{\gamma}{(2W)^2}\left[\left(\sum_{v\in C_1}\deg_\omega(v)\right)^2\right. \\
&\quad \left. - \left(\sum_{v\in C_1\setminus\{u\}}\deg_\omega(v)\right)^2 + \left(\sum_{v\in C_2}\deg_\omega(v)\right)^2 - \left(\sum_{v\in C_2\cup\{u\}}\deg_\omega(v)\right)^2\right]
\end{aligned}
$$

$$
\Delta q' = \frac{1}{W}\left(e_{C_2,u} - e_{C_1,u}\right) - \frac{\gamma}{2W^2}\omega\deg(u)\left(\sum_{v\in C_1}\deg_\omega(v) - \sum_{v\in C_2}\deg_\omega(v) - \omega\deg(u)\right)
$$

Diese Formel ist sich der Formel im Blondels Algorithmus ähnlich. Die Terme in der Formel werden wie im Originalalgorithmus aktualisiert. Das Einfügen des Parameters γ verursacht also keine zusätzliche Operation in der Implementierung. Diese Kombination verallgemeinert den Originalalgorithmus, der für $\gamma = 1$ erzielt wird. Der Algorithmus soll mit mehreren Werten $\gamma \in \mathbb{R}, \gamma \geq 0$ ausgeführt werden und mit kleinen Werten γ Clusterungen mit groben Clustern liefern, während es sich für große Werte γ Clusterungen mit feinen Clustern ergibt.

4.5. Schnitt Clustering Algorithmus von Flake

Wir haben zur Auswertung auch den Schnitt Clustering Algorithmus von Flake et al. [FTT04] betrachtet, der nicht auf Modularity sondern auf Schnittbäume basiert.

Der Algorithmus beruht auf den von Gomory und Hu vorgestellten minimalen Schnittbäumen [GH61]. Ein minimaler Schnittbaum eines Graphen $G(V, E)$ ist ein gewichteter Baum $T(G)$, der für jedes Paar $u, v \in V$ den minimalen u-v-Schnitt im G darstellt. Der minimale u-v-Schnitt im G ergibt sich, indem man in $T(G)$ den Weg betrachtet, der u und v verbindet. Die Kante mit dem kleinsten Gewicht auf diesem Weg entspricht dem minimalen u-v-Schnitt im G. Der Wert dieses Schnitts ist das Gewicht dieser Kante. Die Entfernung dieser Kante teilt den Baum $T(G)$ und den Graphen G in zwei Knotenteilmengen, die dem u-v-Schnitt entsprechen.

Bei der Durchführung des Algorithmus auf einem Graphen $G = (V, E)$ wird ein Dummy-Knoten t betrachtet, der mit jedem Knoten $v \in V$ verbunden wird. Jede neue Kante $\{t, v\}$ hat als Gewicht einen gegebenen Wert α. Sei $G' = (V', E')$ mit $V' = V \cup \{t\}$ und $E' = E \cup \{\{t, v\}, v \in V\}$ der entstehende Graph. Nach der Berechnung des minimalen Schnittbaums T' von G' wird der Dummy-Knoten t entfernt. Dabei entstehen Zusammenhangskomponenten von G, die den Clustern entsprechen. Das Verfahren wird mit mehreren Werten α durchgeführt. Für $\alpha = 0$ ist die vom Algorithmus gelieferte Clusterung der ganze Graph. Wenn α sehr groß ist, liefert der Algorithmus die Singletonclusterung.

Sei $\mathcal{T}$ die maximale Laufzeit für die Berechnung eines Flusses zwischen zwei Knoten im Graphen. Im Worst Case werden für die Berechnung der Cluster n Flüsse berechnet. Die Laufzeit des Verfahren ist in $O(n\mathcal{T})$. Flake et al. [FTT04] definieren für die Gütegarantie, mit der der Algorithmus Clusterungen liefert, zwei Begriffe: Die *Inter-Cluster-Qualität* eines Clusters C ist hoch, wenn die Verbindung $c(C, V \setminus C)$ mit allen anderen Knoten niedrig ist. Die *Intra-Cluster-Qualität* eins Clusters C ist hoch, wenn das Minimalgewicht $c(P, C \setminus P)$ über alle Untermengen $P \subseteq C$ hoch ist. Eine Clusterung $\mathcal{C}$ hat eine gute Qualität für einen Wert α, wenn für alle Cluster $C \in \mathcal{C}$ die beiden Bedingungen gelten:

$$c(C, V \setminus C) \leq \alpha \mid V \setminus C \mid$$

$$c(P, C \setminus P) \geq \alpha \min\{\mid P \mid, \mid C \setminus P \mid\}, \forall P \subset C.$$

5. Experimente

In diesem Kapitel vergleichen wir die im vorigen Kapitel 4 betrachteten Algorithmen miteinander. Dabei messen wir, wie gut jeder Algorithmus in einem Graphen eingepflanzte Clusterungen erkennt. Wir haben die Algorithmen auf den beiden Graphen von Bornholdt (Kapitel 3, Abschnitt 3.6) ausgewertet. Zusätzlich zu diesen beiden Graphen, haben wir zwei Zufallsgraphen konstruiert, die eine hierarchische Clusterstruktur haben und aus jeweils fünf und drei Leveln bestehen. Bevor wir die Algorithmen vergleichen, beschreiben wir zuerst diese Graphen im nächsten Abschnitt. Danach erklären wir, welche Distanzmaße wir verwendet haben, um die Ähnlichkeit zweier Clusterungen zu messen. Anschließend erläutern wir, wie wir jeden Algorithmus auf den Graphen ausgewertet haben.

5.1. Beispiele von konstruierten Zufallsgraphen

Wir stellen zwei Zufallsgraphen mit hierarchischen Clusterstrukturen der Höhe fünf und drei vor, die wir konstruiert haben. Wir beschreiben, wie die beiden Graphen aufgebaut sind und wie wir unser Generator benutzt haben, um diese Graphen zu erzeugen.

Der erste Graph hat eine hierarchische Clusterstruktur der Höhe fünf. Er besteht aus 1024 Knoten und beinhaltet 64 feine Cluster, die jeweils 16 Knoten enthalten. Dies ist *Level 5* der Hierarchie. Jedes Cluster auf *Level 4* besteht aus zwei Clustern auf *Level 5*. Auf *Level 3* besteht jedes Cluster aus zwei Clustern auf *Level 4*. Analog besteht jedes Cluster auf *Level 2* aus zwei Clustern auf *Level 3*. Jedes Cluster auf *Level 1* ist wieder eine Vereinigung von zwei Clustern auf *Level 2*.

Auf *Level 1* dieses Graphen ist im Durchschnitt ein Viertel der Knoten in jedem Cluster mit einem Knoten des selben Clusters verbunden. Die Wahrscheinlichkeit ist so gewählt, dass im Durchschnitt von 16 Knoten einer mit einem Knoten in einem anderen Cluster verbunden ist. Zusätzlich zu den Kanten auf *Level 1* ist die Hälfte der Knoten in jedem Cluster auf *Level 2* im Durchschnitt mit einem Knoten des selben Clusters verbunden. In jedem Cluster auf *Level 3* hat jeder Knoten zusätzlich zu den Links auf *Level 2* im Durchschnitt einen anderen Link mit einem inneren Knoten. Zusätzlich zu den Links auf *Level 3* ist jeder Knoten auf *Level 4* im Durchschnitt mit zwei Knoten vom gleichen Cluster verbunden. In jedem Cluster auf *Level 5* der Hierarchie zusätzlich zu den Links auf *Level 4* ist jeder Knoten im Durchschnitt mit vier inneren Knoten verbunden.

Um so einen Graphen zu erzielen, haben wir einen unserer Generatoren mit fünf Clusterungen $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ und $\mathcal{C}_4$ ausgeführt. Die Clusterungen haben alle 1024 Knoten, die

wie folgt geteilt sind: Die Clusterung $\mathcal{C}_0$ partitioniert die Knotenmenge in 64 Clustern. Mit dieser Clusterung werden Kanten auf *Level 5* erzeugt. Die Clusterungen $\mathcal{C}_1$ bzw. $\mathcal{C}_2$, $\mathcal{C}_3$ und $\mathcal{C}_4$ teilen die Knotenmenge in 32 bzw. 16, 8 und 4 Clustern, und werden benutzt, um Kanten auf *Level 4* bzw. *Level 3*, *Level 2* und *Level 1* zu erzeugen. Die Wahrscheinlichkeiten für jede Clusterung werden, wie folgt berechnet:

- Clusterung $\mathcal{C}_0$:

  $p_0^{in} = \frac{4}{15} = 0.26667$

  $p_0^{out} = 0$

- Clusterung $\mathcal{C}_1$:

  $p_1^{in} = \frac{2}{31} = 0.06451$

  $p_1^{out} = 0$

- Clusterung $\mathcal{C}_2$:

  $p_2^{in} = \frac{1}{63} = 0.01587$

  $p_2^{out} = 0$

- Clusterung $\mathcal{C}_3$:

  $p_3^{in} = \frac{1}{2}\frac{1}{127} = 0.00393$

  $p_3^{out} = 0$

- Clusterung $\mathcal{C}_4$:

  $p_4^{in} = \frac{1}{4}\frac{1}{255} = 0.00098$

  $p_4^{out} = \frac{1}{16}\frac{1}{1023} = 0.00006$

Der zweite Zufallsgraph, den wir konstruiert haben, hat eine hierarchische Clusterstruktur der Höhe drei. Dieser Graph ist ähnlich wie der erste Graph aufgebaut. Der Graph entsteht daraus, dass wir im ersten Graphen *Level 2* und *Level 4* entfernt haben. Das heißt der Graph hat 1024 Knoten. *Level 1* besteht aus 4 großen Clustern mit jeweils 256 Knoten. Die Kanten sind auf diesem Level genauso, wie auf *Level 1* des ersten Graphen verteilt. Jeder Knoten hat im Durchschnitt $\frac{1}{4}$ Link mit den anderen Knoten seines Clusters und $\frac{1}{16}$ Link mit den Knoten anderer Cluster. Auf diesem Graphen besteht *Level 2* aus 16 Clustern. Jeder Knoten auf diesem Level hat zusätzlich zu den Links aus vorigem Level einen Link mit Knoten seines Clusters. Der *Level 3* besteht aus 64 feinen Clustern. In jedem Cluster hat jeder Knoten zusätzlich zu den Links aus vorigen Leveln 4 Links mit inneren Knoten.

Um diesen Zufallsgraphen zu erzeugen, haben wir die Clusterungen $\mathcal{C}_0$, $\mathcal{C}_2$ und $\mathcal{C}_4$ vom ersten Graphen benutzt. Die Clusterungen $\mathcal{C}_0$, $\mathcal{C}_2$ und $\mathcal{C}_4$ teilen die Knotenmenge in jeweils 4, 16 und 64 Clustern und werden jeweils benutzt, um Kanten auf *Level 1*, *Level 2* und *Level 3* zu erzeugen. Die Wahrscheinlichkeiten werden wie im ersten Graphen berechnet:

- Clusterung $\mathcal{C}_0$:

  $p_0^{in} = \frac{4}{15} = 0.26667$

  $p_0^{out} = 0$

- Clusterung $\mathcal{C}_2$:

  $p_2^{in} = \frac{1}{63} = 0.01587$

  $p_2^{out} = 0$

- Clusterung $\mathcal{C}_4$:

$$p_4^{in} = \frac{1}{4}\frac{1}{255} = 0.00098$$

$$p_4^{out} = \frac{1}{16}\frac{1}{1023} = 0.00006$$

Schaubild 5.1 und Schaubild 5.2 stellen die Adjazenzmatrizen der beiden Zufallsgraphen dar.

5.2. Verwendete Distanzmaße zur Messung der Ähnlichkeit zweier Clusterungen

Wir haben zur Messung der Ähnlichkeit zweier Clusterungen den *Graph-theoretischen Rand-Index* benutzt [Gör10]. Seien $\mathcal{C}$, $\mathcal{C}'$ zwei Clusterungen, e_{00} die Anzahl aller Kanten, die in $\mathcal{C}$ und in $\mathcal{C}'$ Interclusterkanten sind, und e_{11} die Anzahl aller Kanten, die in $\mathcal{C}$ und in $\mathcal{C}'$ Intraclusterkanten sind. Der Graph-theoretische Rand-Index zwischen $\mathcal{C}$ und $\mathcal{C}'$ ist gleich

$$\mathcal{R}_g(\mathcal{C}, \mathcal{C}') = 1 - \frac{e_{11} + e_{00}}{m}.$$

Falls Clusterung $\mathcal{C}$ und Clusterung $\mathcal{C}'$ dieselbe Clusterung ist, dann gilt $e_{11} + e_{00} = m$ und folglich $\mathcal{R}_g(\mathcal{C}, \mathcal{C}') = 0$. Falls jede Intraclusterkante in $\mathcal{C}$ eine Interclusterkante in $\mathcal{C}'$ und jede Interclusterkante in $\mathcal{C}$ eine Intraclusterkante in $\mathcal{C}'$ ist, dann sind die beiden Clusterungen völlig unterschiedlich. In diesem Fall gilt $e_{11} + e_{00} = 0$ und $\mathcal{R}_g(\mathcal{C}, \mathcal{C}') = 1$. Diese Ähnlichkeitsmaße nennen wir auch *Distanz* zwischen zwei Clusterungen.

5.3. Experimentelle Auswertung

Wir beschreiben in diesem Abschnitt, wie wir jeden Algorithmus auf den Zufallsgraphen ausgeführt und ausgewertet haben. Die zur Auswertung betrachteten Zufallsgraphen beinhalten eingepflanzte Clusterungen, die entweder Leveln, falls die Graphen eine hierarchische Clusterstruktur haben, oder alle sinnvoll erscheinenden Clusterungen in Graphen mit einer sich überlagernden Clusterstruktur entsprechen. Wir betrachten jede eingepflanzte Clusterung als *Referenzclusterung*. Wir untersuchen also experimentell auf jedem Graphen, wie gut jeder Algorithmus jede Referenzclusterung erkennt. Wir haben die Algorithmen experimentell auf vier Zufallsgraphen ausgewertet: die beiden Zufallsgraphen von Bornholdt und die von uns konstruierten Zufallsgraphen. Tabelle 5.1 gibt einen Überblick über diese Zufallsgraphen.

Wir haben jeden der vier Zufallsgraphen zehn Mal generiert und jeden generierten Graphen ausgewertet. Für jeden der vier Zufallsgraphen haben wir jeden Algorithmus auf alle zehn generierten Graphen ausgeführt und bei jeder Ausführung die Distanz zwischen jeder Referenzclusterung des Zufallsgraphen und der vom Algorithmus gelieferten Clusterung berechnet. Die Distanz, die wir für jeden der vier Zufallsgraphen, bei jeder Referenzclusterung, für jeden Algorithmus gespeichert haben, ist die Summe der zehn berechneten Distanzen zwischen der Referenzclusterung und der bei jeder Ausführung des Algorithmus gelieferten Clusterung geteilt durch zehn. Wir erklären nun, wie wir jeden Algorithmus ausgeführt und die Distanz $\mathcal{R}_g$ berechnet haben. Seien $G_1, \ldots, G_{10}$ die für einen Zufallsgraphen G generierten Graphen.

Der CNM Algorithmus und der CNM Algorithmus mit der Prioritätsfunktion Significance liefern jeder eine einzige Clusterung bei der Ausführung auf einem Graphen. Bei der Ausfürung dieser Algorithmen auf jedem Graphen haben wir die Distanz zwischen der einzigen gelieferten Clusterung und jeder Referenzclusterung berechnet. Sei $\mathcal{C}_i$ die vom Algorithmus

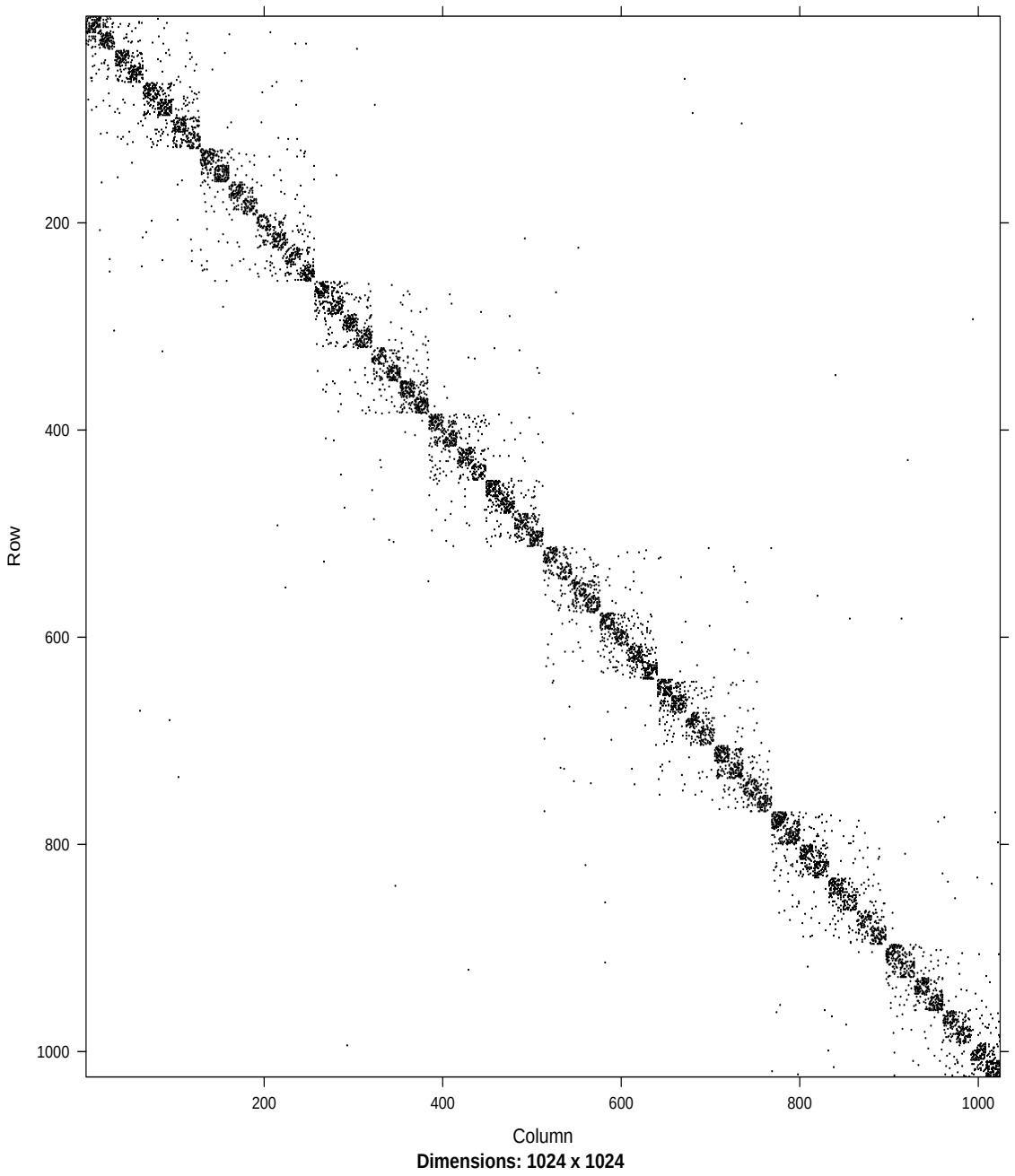


Abbildung 5.1.: Adjazenzmatrix eines Graphen mit einer hierarchischen Clusterstruktur. Der Graph besteht aus fünf Leveln.

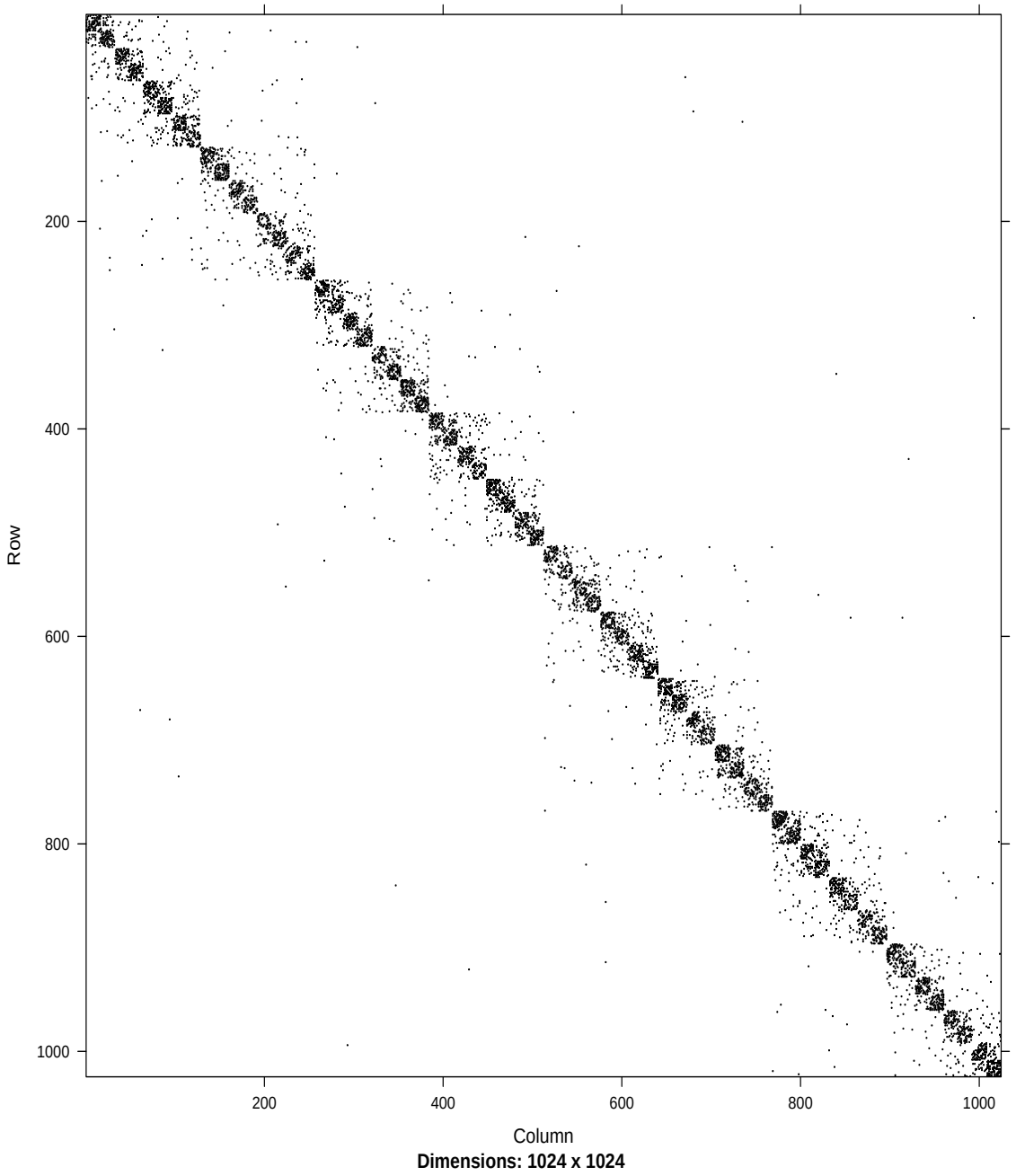


Abbildung 5.2.: Adjazenzmatrix eines Graphen mit einer hierarchischen Clusterstruktur der Höhe drei.

gelieferte Clusterung bei der Ausführung auf dem Graphen G_i. Für jede Referenzclusterung $\text{Ref}_{\mathcal{C}}$ speichern wir die Distanz

$$\mathcal{R}_g = \frac{\sum_{i=1}^{10} \mathcal{R}_g(\mathcal{C}_i, \text{Ref}_{\mathcal{C}})}{10}.$$

Bei der Ausführung der Algorithmen CNM Hierarchie und CNM Hierarchie mit Prioritätsfunktion Sig auf einem Graphen mit n Knoten ergibt sich n Zwischenergebnisse. Es werden also n Clusterungen betrachtet. Wir haben für jede Referenzclusterung des Zufallsgraphen die unter allen n Clusterungen ähnlichste Clusterung berücksichtigt. Wir haben also auf einem Zufallsgraphen die Distanz zwischen einer Referenzclusterung und jedem Zwischenergebnis berechnet und unter allen n erzielten Distanzen die kleinste für diese Konfiguration gespeichert. Seien $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,n}$ alle Zwischenergebnisse bei der Ausführung dieser Algorithmen auf einem Graphen G_i. Für jede Referenzclusterung $\text{Ref}_{\mathcal{C}}$ speichern wir die Distanz

$$\mathcal{R}_g = \frac{\sum_{i=1}^{10} \left(\min_{j=1}^{n}(\mathcal{R}_g(\mathcal{C}_{i,j}, \text{Ref}_{\mathcal{C}}))\right)}{10}.$$

Für den Multi-Level Algorithmus von Blondel haben wir bei der Ausfürung auf einem Graphen alle erzielten Clusterungen auf der Hierarchie untersucht. Wir haben jede auf einem Level erzielte Clusterung mit jeder Referenzclusterung verglichen und die kleinste Distanz gespeichert. Seien $\mathcal{C}_{i,1}, \ldots, \mathcal{C}_{i,h}$ alle erzielten Clusterungen auf der Hierarchie bei der Ausführung des Algorithmus auf einem Graphen G_i. Für jede Referenzclusterung $\text{Ref}_{\mathcal{C}}$ speichern wir die Distanz

$$\mathcal{R}_g = \frac{\sum_{i=1}^{10} \left(\min_{j=1}^{h}(\mathcal{R}_g(\mathcal{C}_{i,j}, \text{Ref}_{\mathcal{C}}))\right)}{10}.$$

Alle anderen betrachteten Algorithmen sind mit mehreren Werten des Parameters α oder γ ausgeführt worden. Bei der Ausführung eines Algorithmus mit einem Wert des Parameters ergibt sich eine Clusterung. Für jede Referenzclusterung haben wir unter allen Clusterungen, die mit verschiedenen Werten des Parameters erzielt sind, die am ähnlichste Clusterung berücksichtigt. Die Distanz ist also die beste Distanz unter allen erzielten Clusterungen. Sei $\mathcal{C}_{i,\alpha|\gamma}$ die mit einem Wert α oder γ erzielte Clusterung bei der Ausführung des Algorithmus auf einem Graphen G_i. Für jede Referenzclusterung $\text{Ref}_{\mathcal{C}}$ speichern wir die Distanz

$$\mathcal{R}_g = \frac{\sum_{i=1}^{10} \left(\min_{\alpha|\gamma}(\mathcal{R}_g(\mathcal{C}_{i,\alpha|\gamma}, \text{Ref}_{\mathcal{C}}))\right)}{10}.$$

In diesem experimentellen Teil haben wir beobachtet, dass der CNM Algorithmus nicht deterministisch ist. Bei der Vereinigung der Clusterpaare im Laufe des Algorithmus wird unter den Clusterpaaren, deren Vereinigung den maximalen Zuwachs an Modularity bringt, ein beliebiges gewählt und sofort kontrahiert. Diese Entscheidung hat zur Folge, dass der Algorithmus implementierungsabhängige Ergebnisse liefert. Bei einer mehrmaligen Ausführung des Algorithmus auf einem Graphen, könnten die erzielten Clusterungen leicht unterschiedlich sein. Der CNM Algorithmus und die von ihm abgeleiteten Algorithmen CNM Hierarchie, CNM mit Prioritätsfunktion, CNM Hierarchie mit Prioritätsfunktion und CNM mit der Abwandlung von Bornholdt, die wir betrachtet haben, haben bei einer mehrmaligen Ausführung auf einem Graphen leicht unterschiedliche Clusterungen geliefert. Dieser Unterschied ist aber sehr klein, so dass man ihn vernachlässigen kann.

Graphennummer	Knoten	Kanten	Clusterstruktur	Beschreibung	Schaubild
1	512	5726	hierarchisch	2 Levels	3.14
2	1024	7132	überlagernd	3 Clusterungen	3.11
3	1024	3810	hierarchisch	5 Levels	5.1
4	1024	2666	hierarchisch	3 Levels	5.2

Tabelle 5.1.: Überblick über die vier Zufallsgraphen. Jeder Zufallsgraph ist zehn Mal generiert worden. Der Wert der Spalte Kanten ist die Summe aller Kanten der zehn generierten Graphen geteilt durch zehn und abgerundet.

5.4. Vergleich von Algorithmen zur Erkennung von Clusterungen

In diesem Abschnitt vergleichen wir anhand vom Schaubild 5.3 die betrachteten Algorithmen miteinander. Das Schaubild besteht aus vier Bildern. Auf diesem Schaubild ist der Graph-theoretische Rand-Index gemittelt über allen zehn generierten Graphen. Falls für eine Referenzclusterung eine Farbe auf einem Bild nicht erscheint, bedeutet es, dass der entsprechende Algorithmus die Referenzclusterung optimal erkannt hat, $\mathcal{R}_g$ also 0 ist. Wir vergleichen zuerst die Algorithmen auf jedem einzelnen Bild. Danach fassen wir die Bemerkungen zusammen.

Für eine bessere Beschreibung der Ergebnisse haben wir für jeden betrachteten Algorithmus eine Abkürzung benutzt. Tabelle 5.2 zeigt eine Übersicht der Abkürzungen mit den entsprechenden Algorithmen.

Abkürzung	Algorithmus	Beschreibung
CNM	CNM Algorithmus	4.1.1
CNM+H	CNM Hierarchie	4.4.1
CNMS	CNM Algorithmus mit Prioritätsfunktion Significance	4.1.2
CNMS+H	CNM Hierarchie mit Prioritätsfunktion Significance	4.4.2
CNM+B	CNM Algorithmus mit der Abwandlung von Bornholdt	4.3.2
Flake	Schnitt Clustering Algorithmus von Flake	4.5
Blondel	Multi-Level Algorithmus von Blondel	4.1.3
Blondel+GID	Multi-Level Algorithmus von Blondel mit Verfeinerung und Einschränkung GID	4.3.1
Blondel+B	Kombination Blondel Bornholdt	4.4.3

Tabelle 5.2.: Übersicht der Abkürzungen mit den entsprechenden Algorithmen. Die Spalte Beschreibung verweist auf die Beschreibung des Algorithmus im vorigen Kapitel 4.

Das Bild links oben illustriert, wie gut jeder Algorithmus jeden Level des Graphen von Bornholdt mit einer hierarchischen Clusterstruktur erkannt hat. Der Graph besteht aus zwei Leveln. Unter Level 1 ist die grobe Clusterung gemeint, Level 2 entspricht der feinen Clusterung.

Auf diesem Bild ist zu bemerken, dass der Algorithmus Flake auf diesem Zufallsgraphen die beiden Level nicht gefunden hat. Die vom CNM Algorithmus gelieferte Clusterung ist sehr oft fast gleich Level 1. Der CNM Algorithmus hat mit der Abwandlung von Bornholdt die beiden Level deutlich besser erkannt. CNM+B hat auf den zehn generierten Graphen Level 1 sehr oft mit den Werten γ zwischen 0.7 und 1.7 erkannt. Der Algorithmus hat mit den Werten von γ zwischen 3 und 4.3 Clusterungen gefunden, die sehr ähnlich zu Level 2 sind.

Der aus der Kombination von Blondels Algorithmus mit der Idee von Bornholdt resultierende Algorithmus hat deutlich bei allen zehn generierten Graphen die beiden Level erkannt. Diese Kombination hat auf den zehn generierten Graphen Level 2 sehr oft mit Werten von γ zwischen 1.6 und 2.4 erkannt. Level 1 ist bei allen zehn generierten Graphen mit dem Wert $\gamma = 1$ erkannt worden. Für diesen Wert von γ ist diese Kombination gleich dem Blondels Algorithmus. Auf dem Schaubild ist dieses Ergebnis deutlich zu sehen, da der Blondels Algorithmus Level 1 auf alle generierten Graphen erkannt hat. Dies ist auch analog für den Algorithmus Blondel+GID, der Level 1 auf allen zehn generierten Graphen mit dem Wert $\alpha = 0$ erkannt hat. Der Algorithmus Blondel+GID hat Level 2 mit $\alpha = 0.3$ besser als den Algorithmus Blondel erkannt. Das heißt die Einschränkung mit GID hilft dem Originalalgorithmus die Grobheit der erzielten Clusterungen zu beeinflussen.

Das Bild rechts oben zeigt die erzielten Ergebnisse bei der Ausführung der Algorithmen auf einem Zufallsgraphen mit einer sich überlagernden Clusterstruktur. Der Graph besteht aus zwei großen Clustern A und B. Cluster A enthält eine Untergruppe von Knoten a und Cluster B enthält eine Untergruppe von Knoten b (für eine detaillierte Beschreibung des Graphen siehe Kapitel 3 Abschnitt 3.6). Der Graph kann in drei verschiedenen Clusterungen geclustert werden: eine Clusterung bestehend aus den beiden großen Clustern A und B, eine Clusterung mit drei Clustern $A \setminus a$, $a \cup b$ und $B \setminus b$, und eine Clusterung, deren Cluster $A \setminus a$, a, b und $B \setminus b$ sind. Wir bezeichnen die Menge $A \setminus a$ mit c, $a \cup b$ mit e und $B \setminus b$ mit d.

Auf diesem Bild stellen wir fest, dass Blondel und Blondel+GID oft eine gute Clusterung zur Referenzclusterung mit Clustern A und B liefern. Unsere Kombination Blondels Algorithmus mit der Idee von Bornholdt hat auf den zehn generierten Graphen sehr oft die großen Cluster A und B mit Werten von γ zwischen 0.3 und 0.9 fast erkannt. Die Algorithmen Blondel, Blondel+GID und unsere Kombination Blondel+B haben die anderen Clusterungen des Graphen kaum erkannt. Die Algorithmen CNM, CNM+H und CNM+B haben der Referenzclusterung A B oft nähere Clusterungen geliefert. Die anderen Algorithmen haben keine der Referenzclusterungen gefunden.

Das linke untere Bild stellt den Vergleich der betrachteten Algorithmen anhand des von uns konstruierten Zufallsgraphen dar. Der Graph hat eine hierarchische Clustersruktur und besteht aus fünf Leveln. Der Level 1 besteht aus 4 großen Clustern und Level 5 aus 64 dichten Clustern. Jedes Cluster auf Level 1 besteht aus zwei Clustern auf Level 2 und jedes Cluster auf Level 2 besteht aus zwei Clustern auf Level 3. Auf diesem Zufallsgraphen hat unsere Kombination Blondel+B deutlich Level 1, Level 2, Level 3 und knapp Level 4 erkannt. Auf den zehn generierten Graphen hat diese Kombination Level 1 oft mit $\gamma = 0.1$, Level 2 sehr oft mit $\gamma = 0.2$ und Level 3 oft mit $\gamma = 1, 1.1, 1.2$ erkannt. Mit den Werten von γ zwischen 3.7 und 5.1 hat diese Kombination dem Level 4 sehr ähnliche Clusterungen geliefert. Diese Kombination hat Level 5 besser als alle anderen Algorithmen erkannt.

Der Algorithmus Flake hat sehr oft Level 1 und Level 2 gut erkannt. Die Algorithmen Blondel und Blondel+GID haben auf den zehn generierten Graphen sehr oft Level 3 fast erkannt. Der Algorithmus CNM+B hat auf diesem Graphen Level 1 sehr oft mit $\gamma = 0.1$ erkannt. Dieser Algorithmus hat sehr ähnliche Clusterungen zum Level 2, Level 3 und gute Clusterungen zum Level 4 geliefert. Auf diesem Graphen bemerken wir, je grober eine Referenzclusterung ist, desto besser erkennen CNM+B und Blondel+B sie. Der CNM Algorithmus hat ähnliche Clusterungen zum Level 1 und Level 2 geliefert. Der Algorithmus CNM+H hat deutlich Level 1 und knapp Level 2 erkannt. Bemerkungswert ist, dass CNM+H sowohl Level 1 als auch Level 2 besser als CNM erkennt, obwohl die vom CNM gelieferte Clusterung den besseren Wert an Modularity als alle anderen Zwischenergebnisse hat. Mit dieser Bemerkung vermuten wir, gute Clusterungen haben zwar gute Modularity Werte aber Clusterungen mit besserer Modularity nicht unbedingt die besseren sinnvollen

Clusterungen sind.

Das Bild rechts unten illustriert den Vergleich der Algorithmen auf einem Zufallsgraphen mit einer hierarchischen Clusterstruktur, den wir konstruiert haben. Der Graph hat drei Level. Level 1 ist die grobste Clusterung und Level 3 ist die feinste. Auf diesem Graphen bemerken wir, dass die Kombination Blondel+B Level 1 deutlich mit dem Wert $\gamma = 0.1$ erkennt. Der Algorithmus Blondel hat auf diesem Graphen Level 2 gut erkannt. Die Algorithmen Blondel+GID und Blondel+B, die den Algorithmus Blondel verallgemeinern, haben für Level 2 keine bessere Clusterung als den Originalalgorithmus geliefert. Der Algorithmus Flake liefert Clusterungen, die sich Level 1 sehr ähnlich sind. CNM+B erkennt deutlich Level 1 mit $\gamma = 0.1$ und liefert Clusterungen, die ähnlich zu Level 2. Blondel+B und CNM+B erkennen auf diesem Bild besser die groben als die feinen Clusterungen. CNM+H erkennt deutlich Level 1. Auf diesem Bild bemerken wir nochmal, dass trotz niedrigerer Werte von Modularity liefert CNM+H bessere Clusterungen zur Referenzclusterungen Level 1 und Level 2 als CNM. Diese Bemerkung haben wir bereits auf einem anderen Bild gemacht. Wir gehen auf diese Bemerkung nochmal im späteren Abschnitt, wo wir die Modularity-basierten Algorithmen nochmal durchgeführt haben, um die von Algorithmen erzielten Modularity Werte mit Modularity der Referenzclusterungen zu vergleichen.

5.5. Fazit und Bemerkungen vom Vergleich der Algorithmen

In diesem Abschnitt fassen wir die Bemerkungen vom Vergleich der Algorithmen aus dem vorigen Abschnitt zusammen.

Bemerkung 1 Mit kleinen Werten vom Parameter α oder γ erzielen die Algorithmen CNM+B, Blondel+GID und Blondel+B grobe Clusterungen. Mit großen Werten des Parameters erzielen diese Algorithmen feine Clusterungen. Die Heuristiken Hierarchie, Einschränkung mit GID und die Bornholdts Idee helfen den Originalalgorithmen die Grobheit der Clusterungen zu beeinflussen und bessere Clusterungen zu erzielen.

Bemerkung 2 Sehr feine Clusterungen werden von betrachteten Algorithmen schwer erkannt. Die Algorithmen CNM+B und Blondel+B erkennen grobe Clusterungen besser als feinere Clusterungen. Je grober eine Clusterung ist, desto besser erkennen diese beiden Algorithmen sie.

Bemerkung 3 Bei der Ausführung der Algorithmen CNM+B, Blondel+GID und Blondel+B auf mehreren generierten Graphen desselben Zufallsgraphen wird eine Referenzclusterung auf allen generierten Graphen mit keinem festen Wert des Parameters α oder γ für jeden Algorithmus erkannt. Ein Algorithmus erkennt eine Referenzclusterung eines Zufallsgraphen auf unterschiedlichen generierten Graphen mit unterschiedlichen Werten des Parameters α oder γ.

Bemerkung 4 Unter allen betrachteten Algorithmen hat unsere Kombination Blondel+B auf jedem Zufallsgraphen jede Referenzclusterung besser als alle anderen Algorithmen erkannt. Diese Kombination hat auf jedem Graphen mit einer hierarchischen Clusterstruktur mindestens eine Referenzclusterung optimal erkannt.

5.6. Zusätzliche Experimente

Wir haben bei den vorigen Experimenten bemerkt, dass gute Clusterungen bezüglich dem Vergleich mit einer Referenzclusterung gute Modularity Werte haben, aber Clusterungen mit besserer Modularity nicht unbedingt die Referenzclusterungen besser wiedererkennen. Um diese Vermutung zu belegen, haben wir die auf Modularity basierenden betrachteten Algorithmen nochmal auf den Zufallsgraphen ausgeführt und für jeden Algorithmus den

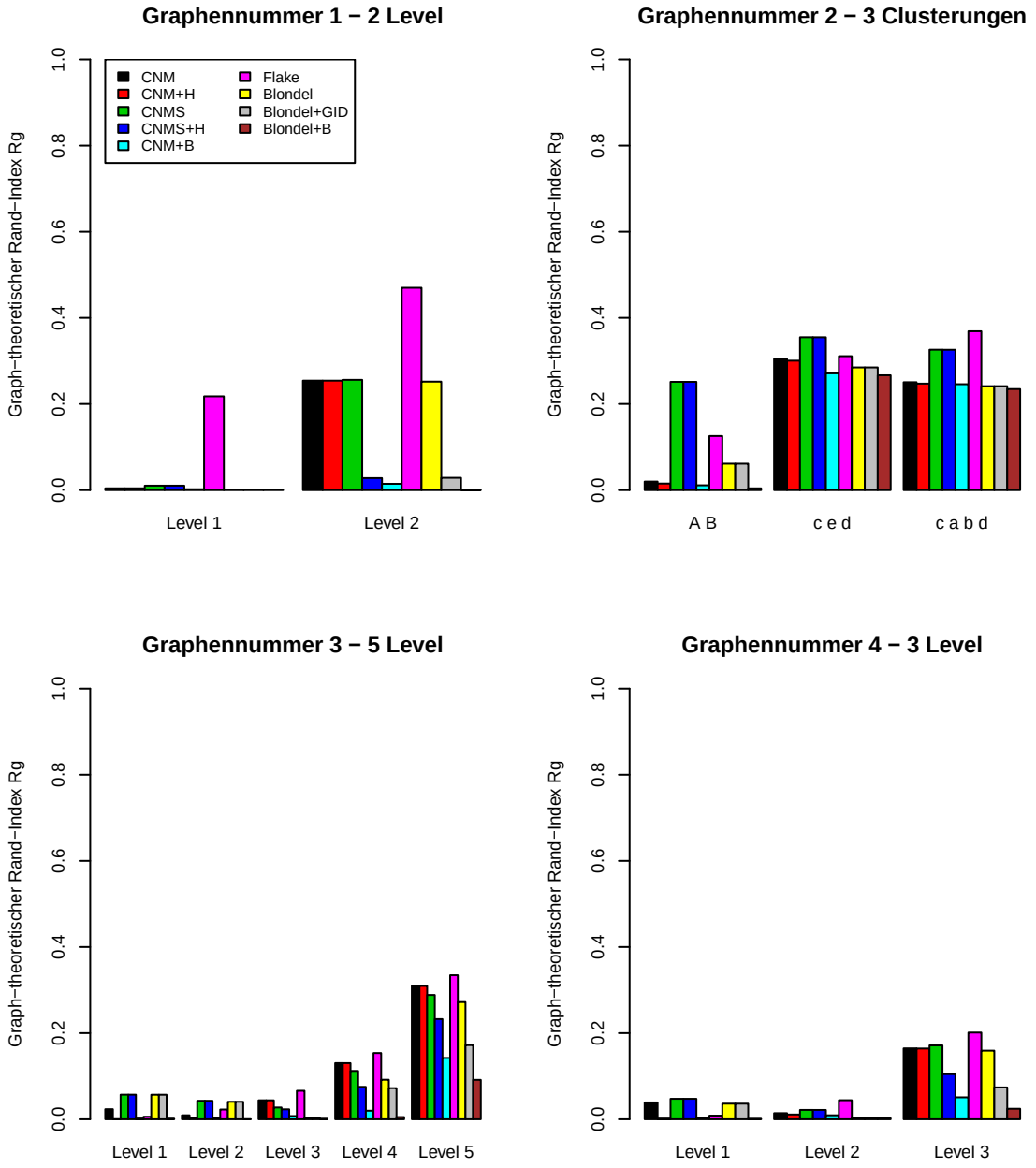


Abbildung 5.3.: Schaubild zum Vergleich von Algorithmen auf drei Graphen mit einer hierarchischen Clusterstruktur und auf einem Graphen mit einer sich überlagernden Clusterstruktur. Die Tabelle 5.2 zeigt einen Überblick über die Beschreibung der benutzten Abkürzungen in der Legende.

größten über allen Werten vom Parameter α, γ oder auf der Hierarchie erzielten Wert an Modularity berechnet. Für jeden Zufallsgraphen haben wir für jeden Algorithmus die Differenz vom größten erzielten Wert an Modularity und Modularity jeder Referenzclusterung gespeichert. Seien $G_1, \ldots, G_{10}$ die für einen Zufallsgraphen G generierten Graphen und $q(\mathcal{C}_i)$ den größten Wert an Modularity über alle auf der Hierarchie oder mit verschiedenen Werten α, γ erzielten Clusterungen von einem Algorithmus auf einem Graphen G_i. Für jede Referenzclusterung $\mathrm{Ref}_{\mathcal{C}}$ mit dem Modularity Wert $q(\mathrm{Ref}_{\mathcal{C}})$ speichern wir den Wert

$$\mathrm{Mod} = \frac{\sum_{i=1}^{10} \left(q(\mathcal{C}_i) - q(\mathrm{Ref}_{\mathcal{C}})\right)}{10}.$$

Das Schaubild 5.4 illustriert die Ergebnisse. Auf diesem Schaubild beobachten wir, dass die Algorithmen auf Graphen mit einer hierarchischen Clusterstruktur Clusterungen mit deutlich besseren Modularity Werten als die feinsten Referenzclusterungen liefern, obwohl diese Algorithmen diese feinsten Referenzclusterungen schlecht erkennen (siehe Schaubild 5.3). Diese Beobachtung belegt, dass die auf Modularity basierenden Algorithmen Clusterungen mit sehr großen Werten an Modularity liefern können, obwohl sie während ihrer Suche nach solchen Clusterungen bessere sinnvolle Clusterungen mit niedrigeren Modularity Werten nicht getroffen haben. Diese Algorithmen bauen nach und nach Clusterungen mit besseren Modularity Werten auf. Man sollte die feinsten Clusterungen mit kleineren Modularity Werten als Zwischenergebnisse haben. Dies ist aber nicht der Fall.

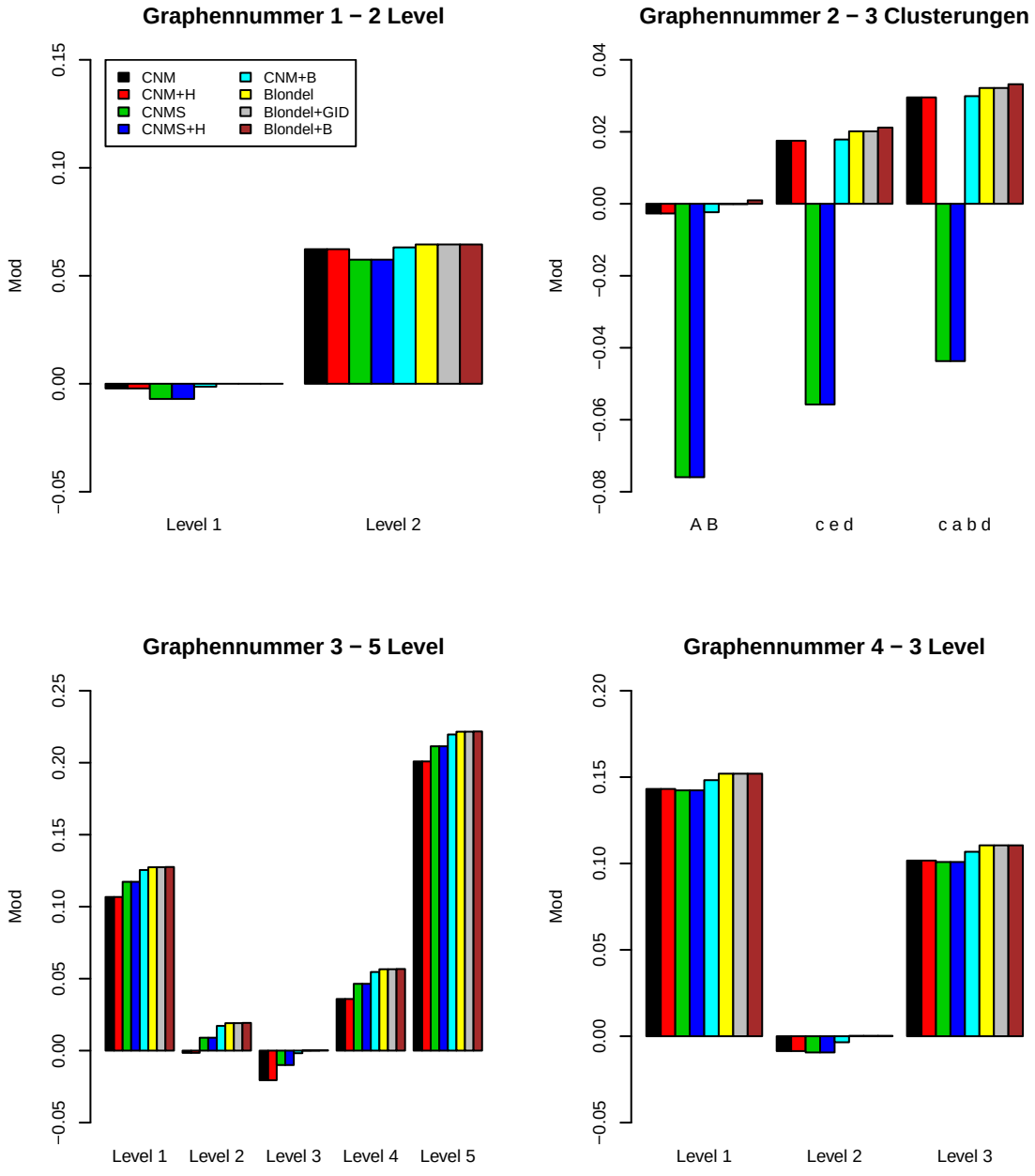


Abbildung 5.4.

6. Zusammenfassung

Wir haben in dieser Arbeit Generatoren entwickelt, um Zufallsgraphen zu erzeugen. Anhand von den damit erzeugten Zufallsgraphen haben wir Clusteringalgorithmen verglichen. Die Idee war, Graphen zu erzeugen, die eine bekannte Clusterstruktur haben, und zu untersuchen, wie gut jeder Clusteringalgorithmus die bekannte sinnvolle Clusterung wiedererkennt. Graphen können auch eine Clusterstruktur mit mehreren sinnvollen Clusterungen beinhalten. Die Clusterungen eines Graphen können zum Beispiel eine hierarchische Clusterstruktur darstellen, in der jedes Cluster einer Clusterung die Vereinigung von mehreren Clustern einer anderen Clusterung ist. Die Clusterungen eines Graphen können auch eine sich überlagernde Clusterstruktur darstellen, in der die verschiedenen Clusterungen sich überlappen. Das Ziel der Arbeit war also, mit einem Generator Graphen zu erzeugen, in denen Referenzclusterungen eingepflanzt werden, und zu untersuchen, wie gut die Clusteringalgorithmen die Referenzclusterungen erkennen.

Dafür haben wir vier Generatoren entwickelt. Die Generatoren erzeugen sowohl Zufallsgraphen mit einer hierarchischen als auch mit einer sich überlagernden Clusterstruktur. Der erste Generator ist ein einfaches Verfahren, das keinen zusätzlichen Speicherplatz benötigt und Zufallsgraphen in einer quadratischen Laufzeit erzeugt. Wegen seiner quadratischen Laufzeit eignet sich der Generator dafür, dichte Graphen zu erzeugen. Der zweite Generator hat eine bessere Laufzeit und ist einfach zu implementieren. Der dritte Generator erzeugt sowohl dichte Graphen als auch dünne Graphen mit guten Laufzeitgarantien. Der vierte Generator erzeugt Graphen in einer linearen Laufzeit und der benötigte Speicherplatz ist linear zur Graphgröße.

Mit den Generatoren haben wir zwei bekannte Graphen aus der Literatur und dazu Zufallsgraphen anhand von zwei verschiedenen Konfigurationen generiert. Anhand der erzeugten Zufallsgraphen haben wir Clusteringalgorithmen systematisch verglichen. Dabei haben wir auf Modularity basierende Algorithmen und einen auf Schnittbäumen basierenden Algorithmus betrachtet. Aus den Grundideen von Modularity-optimierenden Algorithmen und Ideen aus der Literatur, um die Grobheit der Clusterungen zu beeinflussen, haben wir neue Kombinationen evaluiert. Eine dieser neuen Kombinationen hat auf jedem Graphen jede Referenzclusterung besser als alle anderen betrachteten Algorithmen erkannt. Diese neue Kombination hat auf jedem Graphen mit einer hierarchischen Clusterstruktur mindestens eine Referenzclusterung optimal erkannt. Diese Kombination ist aus dem Multi-Level Algorithmus von Blondel [BGLL08] und der Idee von Bornholdt [RB06]. Darüber hinaus haben wir aus den Experimenten bemerkt, dass gute Clusterungen bezüglich dem Vergleich mit der Referenzclusterung gute Modularity Werte haben, aber Clusterungen mit besserer

Modularity die Referenzclusterung nicht unbedingt besser wiedererkennt. Die betrachteten Algorithmen haben sehr feine Referenzclusterungen schwer erkannt, obwohl die auf Modularity basierenden Algorithmen deutlich bessere Modularity Werte als Modularity der feinen Referenzclusterungen erzielt haben. Man sollte die feinen Referenzclusterungen als Zwischenergebnis haben. Dies ist aber nicht der Fall.

A. Anhang

Procedure `nextpair`

Input: locals nodes $local(v), local(w), \text{first}(v) \in \{0, \ldots, n-1\}$, index j for a edge probability and a jump s.
$local(w) \leftarrow local(w) + s$
if *j is even* **then**
    `/* find the` s`-next intra-cluster-pair */`
    while $local(w) \geq local(v)$ *and* $local(v) < n$ **do**
        $local(w) \leftarrow local(w) - local(v)$
        $local(v) \leftarrow local(v) + 1$
        while $local(v) < n$ *and* $\mathcal{C}_{\frac{j}{2}}[global(v)] \neq \mathcal{C}_{\frac{j}{2}}[global(v-1)]$ **do**
            `/*` $local(v)$ `and` $local(v) - 1$ `are not in the same cluster, one find the next not-singletoncluster */`
            $local(v) \leftarrow local(v) + 1$
        if $local(v) < n$ **then**
            if $\mathcal{C}_{\frac{j}{2}}[global(v)] \neq \mathcal{C}_{\frac{j}{2}}[global(\text{first}(v))]$ **then**
                $\text{first}(v) \leftarrow local(v) - 1$
            $local(w) \leftarrow local(w) + \text{first}(v)$
if *j is odd* **then**
    `/* find the` s`-next inter-cluster-pair */`
    while $local(w) \geq \text{first}(v)$ *and* $local(v) < n$ **do**
        $local(w) \leftarrow local(w) - \text{first}(v)$
        $local(v) \leftarrow local(v) + 1$
        if $local(v) < n$ **then**
            if $\mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[global(v)] \neq \mathcal{C}_{\lfloor \frac{j}{2} \rfloor}[global(v-1)]$ **then**
                `/*` $local(v)$ `and` $local(v) - 1$ `are not in the same cluster */`
                $\text{first}(v) \leftarrow local(v)$

Literaturverzeichnis

[BB05] V. Batagelj and U. Brandes, "Efficient Generation of Large Random Networks," *Physical Review E*, no. 036113, 2005. [Online]. Available: http://link.aps.org/abstract/PRE/v71/e036113

[BDH+07] U. Brandes, D. Delling, M. Höfer, M. Gaertler, R. Görke, Z. Nikoloski, and D. Wagner, "On Finding Graph Clusterings with Maximum Modularity," in *Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)*, ser. Lecture Notes in Computer Science, A. Brandstädt, D. Kratsch, and H. Müller, Eds., vol. 4769. Springer, October 2007, pp. 121–132. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74839-7_12

[BGLL08] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, 2008. [Online]. Available: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008

[CNM04] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 066111, 2004. [Online]. Available: http://link.aps.org/abstract/PRE/v70/e066111

[Cor10] T. H. Cormen, Ed., *Algorithmen - eine Einführung*, 3rd ed. München: Oldenbourg-Verl., 2010. [Online]. Available: http://deposit.d-nb.de/cgi-bin/dokserv?id=3492106&prov=M&dok_var=1&dok_ext=htm;http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=3889699&custom_att_2=simple_viewer

[ER59] P. Erdős and A. Rényi, "On Random Graphs I," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[FMR62] C. T. Fan, M. E. Muller, and I. Rezucha, "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital-Computers," *Journal of the American Statistical Association*, vol. 57, no. 298, pp. 387–402, 1962. [Online]. Available: http://www.jstor.org/stable/2281647

[For10] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3–5, pp. 75–174, 2010. [Online]. Available: http://www.sciencedirect.com/science/journal/03701573

[FTT04] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis, "Graph Clustering and Minimum Cut Trees," *Internet Mathematics*, vol. 1, no. 4, pp. 385–408, 2004. [Online]. Available: http://www.internetmathematics.org/volumes/1.htm

[GH61] R. E. Gomory and T. Hu, "Multi-terminal network flows," *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, no. 4, pp. 551–570, December 1961.

[Gil59] H. Gilbert, "Random Graphs," *The Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959.

[Gör10] R. Görke, "An Algorithmic Walk from Static to Dynamic Graph Clustering," Ph.D. dissertation, Fakultät für Informatik, February 2010. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000018288

[GSW11] R. Görke, A. Schumm, and D. Wagner, "Density-Constrained Graph Clustering," in *Algorithms and Data Structures, 12th International Symposium (WADS'11)*, ser. Lecture Notes in Computer Science, F. Dehne, J. Iacono, and J.-R. Sack, Eds., vol. 6844. Springer, August 2011.

[Hat11] K. . N. H. Hatzinger, Reinhold ; Hornik, *R - Einführung durch angewandte Statistik*, ser. Scientific tools. München [u.a.]: Pearson Studium, 2011. [Online]. Available: http://swbplus.bsz-bw.de/bsz332649733cov.htm;http://deposit.d-nb.de/cgi-bin/dokserv?id=3539820&prov=M&dok_var=1&dok_ext=htm;http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=4095795&custom_att_2=simple_viewer

[Laf10] O. F. Lafou, "Engineering von Modularity-basiertem Graphenclustern,," September 2010, student Project, Studienarbeit. [Online]. Available: http://i11www.iti.uni-karlsruhe.de/_media/teaching/theses/sa-fofielafou-10.pdf

[New04] M. E. J. Newman, "Analysis of Weighted Networks," *Physical Review E*, vol. 70, no. 056131, pp. 1–9, 2004. [Online]. Available: http://link.aps.org/abstract/PRE/v70/e056131

[NG04] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 026113, pp. 1–16, 2004. [Online]. Available: http://link.aps.org/abstract/PRE/v69/e026113

[NR09] A. Noack and R. Rotta, "Multi-level Algorithms for Modularity Clustering," in *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, ser. Lecture Notes in Computer Science, J. Vahrenhold, Ed., vol. 5526. Springer, June 2009, pp. 257–268. [Online]. Available: http://www.springerlink.com/content/qugv7708h3806230/

[RB06] J. Reichardt and S. Bornholdt, "Statistical Mechanics of Community Detection," *Physical Review E*, vol. 74, no. 016110, pp. 1–16, 2006.