Fernfachhochschule Schweiz
Studiengang INF

Vergleich von Graph-Kompressionsalgorithmen k^2 Tree und Huffman Encoding zur Komprimierung von Barabási-Albert Graphen

Seminararbeit

Disposition zur Seminararbeit im Studiengang INF

von

Matthias Heimberg

Eingereicht bei:	**Referentin:**
Usrula Deriu	Usrula Deriu
Dozentin	Dozentin
SemA INF	SemA INF

Melchnau, 16. Oktober 2023

Zusammenfassung

In der Arbeit wurden die beiden Kompressionsalgorithmen K2-Tree und Huffman Encoding zur verlustfreien Kompression auf Barabási-Albert Graphen angewandt. Die Kompressionsalgorithmen wurden bezüglich Kompressionsfaktor und Kompressionszeit verglichen. Beide Algorithmen wurden auf Basis der Literaturdaten eigenständig in Python implementiert.

Die erzielten Ergebnisse zeigen deutlich, dass der Huffman Algorithmus den Algorithmus k^2-Tree hinsichtlich der Kompressionszeit und des Kompressionsfaktors übertrifft. Als Faktoren, welche die Leistung der Algorithmen beeinflussen wurden primär die Graphengrösse und die Graphendichte identifiziert. Diese Erkenntnisse können zur Optimierung der Kompressionsstrategie genutzt werden und bieten einen Ausgangspunkt für zukünftige Forschungen zur Graphenkompression.

Inhaltsverzeichnis

1. Einleitung

Die digitale Revolution hat zu einer exponentiellen Zunahme von Daten geführt, die in Form von Graphen repräsentiert werden können. Graphen stellen komplexe Beziehungen und Muster dar und finden heute Anwendung in den unterschiedlichsten Disziplinen, wie in den Sozialwissenschaften, in der Netzwerkanalyse und in der Bioinformatik. Angesichts der wachsenden Grösse und Komplexität von Graphen sind effiziente Speicher- und Verarbeitungsstrategien von grosser Bedeutung. Eine der Strategien ist die Datenkompression, die den Speicherplatz und die Verarbeitungszeit für die Handhabung von Graphen reduziert.

In dieser Arbeit wird die Kompression von Graphen untersucht. Insbesondere wird die Effizienz und Wirksamkeit zweier Kompressionsalgorithmen, K2-Tree und Huffman Encoding im Kontext von Barabási-Albert Graphen, einem gängigen Modell für komplexe Netzwerke, evaluiert und verglichen.

1.1. Motivation

In einer Zeit in welcher das Volumen, die Vielfalt und die Geschwindigkeit der Datenerzeugung exponentiell zunimmt [9], ist die effiziente Speicherung und Verarbeitung von Daten zu einer grossen Herausforderung geworden. Graphen repräsentieren komplexe Strukturen und Beziehungen in solchen Datensätzen und haben dadurch in unterschiedlichen Disziplinen eine zentrale Rolle eingenommen. Um mit Graphen von grossen Datensätzen umgehen zu können, wurden unterschiedliche Kompressionsalgorithmen mit dem Ziel der Speicherreduktion und gleichzeitiger Wahrung der Datenintegrität entwickelt [3]. Die Arbeit untersucht die Anwendung der beiden Algorithmen K2-Tree und Huffman Encoding auf ungerichteten Barabási-Albert Graphen.

1.2. Problemstellung

Die Arbeit untersucht die folgenden Fragestellungen:

1. Wie effektiv sind K2-Tree und Huffman Encoding in Bezug auf die Kompressionsrate und die Kompressionszeit bei der Kompression von Barabási-Albert Graphen?
2. Wie unterscheiden sich die beiden Algorithmen in ihrer Leistung unter verschiedenen Bedingungen, z.B. in Bezug auf die Grösse und Komplexität des Graphen?
3. Welche Faktoren beeinflussen die Leistung jedes Algorithmus und wie können diese Informationen genutzt werden, um die Kompressionsstrategie zu optimieren?

Um diese Fragen zu beantworten, wurden die beiden Algorithmen auf Grundlage der Literatur durch den Autor in Python implementiert und untersucht.

1.3. Vergleichbare Arbeiten

Die Arbeit [10] untersucht Huffman Encoding zur Komprimierung unterschiedlicher Real World Graphen auf Basis der Adjazenzmatrix und erreicht Kompressionsfaktoren bis zu 0.8. Den Algorithmus k^2-Tree wird in [5] zur Kompression von Web-Graphen vorgeschlagen. Die Kompression skalenfreier Graphen wird in [8] mathematisch untersucht.

2. Theoretischer Hintergrund

Im folgenden werden die für das Verständnis der Arbeit erforderlichen theoretischen Grundlagen erläutert. Insbesondere werden die benutzten Graphen und die beiden Algorithmen genauer vorgestellt. Weiter wird auf den aktuellen Forschungsstand im Gebiet der Graphenkompression eingegangen.

2.1. Barabási-Albert Graphen

Barabási-Albert Graphen sind ein Modell für skalierbare Netzwerke, das von Albert-László Barabási und Réka Albert entwickelt wurde [1]. Barabási-Albert Graphen werden auf Grund zweier Grundregeln aufgebaut:

- Wachstum: Pro Zeitschritt wird dem Graphen ein Knoten angesetzt, welcher mit einer definierten Anzahl m der bereits bestehenden Knoten verbunden wird.
- Verbindung: Mit welchen Knoten der neue Knoten verbunden wird, ist abhängig vom Grad k der Knoten, je grösser der Grad, desto grösser die Wahrscheinlichkeit der Verbindung.

Auf Grund dieser Eigenschaften werden Graphen generiert, wessen Gräder einer Potenzgesetz-Verteilung folgen [2]:

$$P(k) \sim m^{-3} \tag{2.1}$$

Das Barabási-Albert Modell erlaubt die Generierung von zufälligen Graphen beliebiger Grösse mit wählbarer Charakteristiken und eignet sich somit gut für die geplanten Untersuchungen.

2.1.1. Erzeugung der Graphen

Barábasi-Albert Graphen werden gemäss des folgenden Algorithmus mit den Parametern n und m erzeugt [1]:

1. Beginn mit einem Stern-Graph von $m + 1$ Knoten.

2. Hinzufügen eines neuen Knotens, welcher mit $m \leq m_0$ vorhandenen Knoten verbunden wird

3. Wiederholung von Schritt 2. bis der Graph aus n Knoten besteht.

2.2. Repräsentation von Graphen

Graphen lassen sich auf unterschiedliche Arten darstellen. Die für die vorliegende Arbeit wichtigen Darstellungsarten sind die Edge List, die Adjazenzliste und die Adjazenzmatrix (vgl. Abbildung 2.2).

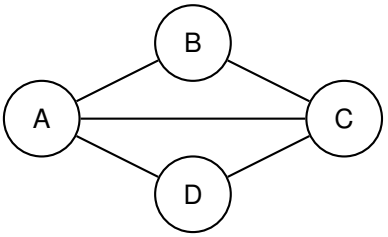

Abbildung 2.1: Visuelle Repräsentation des Graphen G mit $n = 4$ (Knoten) und $m = 5$ (Kanten)

Der Graph G aus Abbildung 2.1 wird in Abbildung 2.2 in drei unterschiedlichen Repräsentationsformen dargestellt. Diese werden in den nächsten Unterkapitel kurz beschrieben.

a) Edge List

(A, B)
(A, D)
(A, C)
(B, C)
(C, D)

b) Adjacency List

A: B, D, C
B: A, C
C: A, B, D
D: A, C

c) Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

Abbildung 2.2: Der Graph G wird in Form einer Edge List a), Adjacency List b) und Adjacency Matrix c) dargestellt.

2.2.1. Edge List

Die Edge List (Abbildung 2.2 a)) oder Kantenliste ist eine einfache Darstellung eines Graphen, die aus einer Liste aller Kanten (Edges) besteht. Jede Kante wird als ein Paar von Knoten dargestellt. Die Darstellung als Edge List ist besonders speichereffizient für Graphen mit wenigen Kanten, da für jede Kante nur ein Paar von Knoten gespeichert werden muss.

2.2.2. Adjiazenzliste

Eine Adjazenzliste (Adjacency List) ist eine effizientere Darstellung, besonders für Graphen mit vielen Kanten. In einer Adjazenzliste wird jeder Knoten des Graphen zusammen mit einer Liste der Knoten dargestellt, zu denen er eine direkte Verbindung hat. Diese Darstellung in Form eines zusammenhängenden Arrays ermöglicht einen schnellen Zugriff auf alle Nachbarn eines bestimmten Knotens, benötigt jedoch im Vergleich zur Kantenliste mehr Speicherplatz, da für jeden Knoten eine separate Liste von Nachbarknoten gespeichert werden muss. Für die Repräsentation als Adjazenzliste werden $\mathcal{O}(n\log(m) + m\log(n))$ Bit benötigt ([3]).

2.2.3. Adjiazenzmatrix

Die Darstellung in Form der Adjazenzmatrix (Adjacency Matrix) ist schliesslich eine quadratische Matrix, bei der jede Zelle (i, j) angibt, ob eine Kante zwischen den Knoten i und j existiert oder nicht. Diese Darstellung erlaubt das schnelle Überprüfen, ob eine direkte Verbindung zwischen zwei Knoten besteht. Allerdings ist sie weniger speichereffizient als die anderen beiden Darstellungsformen, insbesondere für Graphen mit wenigen Kanten, was in Abbildung 2.2 deutlich wird. Für einen Graphen mit n Knoten werden n^2 Einträge ($\mathcal{O}(n^2)$ Bit) benötigt.

2.3. Verlustfreie Graphenkompression

Unter verlustfreier Graphenkompression verstehen wir eine Technik, welche die für die Repräsentation eines Graphen benötigte Datenmenge zu reduzieren ohne die strukturellen und semantischen Informationen des ursprünglichen Graphen zu beeinträchtigen. Der Speicherplatz soll also reduziert werden, ohne dass Informationen verlorengehen. Ein weiterer Aspekt, welcher jedoch in der vorliegenden Arbeit nicht behandelt wurde, be-

trifft die Verbesserung der Geschwindigkeit von Graphanalysemethoden durch die Kompression.

2.3.1. Überblick

Einen ausführlichen Überblick zu den bekannten verlustfreien Kompressionsalgorithmen liefert [3]. Aus diesem Übersichtspaper wurden die beiden folgenden Kompressionsalgorithmen ausgesucht:

2.3.2. Huffman Encoding

Die Huffman Kodierung (Huffman Encoding) ist eine Form der Entropiekodierung welche 1952 von David A. Huffman entwickelt wurde ([7]). Sie basiert auf unterschiedlichen Anteilen von Mustern in einem Bitstrom oder Texts, also auf der unterschiedlichen Frequenz von Bitmustern oder Charaktern. Für Bitmuster mit hoher Frequenz werden kurze Huffman Codes verwendet, für seltene Bitmuster längere Codes. Damit werden der zur Repräsentation des Text eine reduzierte Anzahl Bits verwendet, solange eine unterschiedliche Frequenz in den Bitmustern auftaucht. Die Huffman Kompression lässt sich mittels eines Baums visualisieren. Die Huffman Kodierung komprimiert einen Datensatz (hier einen Graphen in Kantenlistendarstellung) umso stärker, je häufiger sich bestimmte Muster im Datensatz wiederholen.

2.3.3. K2-Tree

Der Algorithmus k^2-Tree wurde zur Kompression von Webgraphen vorgeschlagen ([6]). Der Algorithmus teilt die Adjazenzmatrix eines Graphen in $k \times k$ Teilmatritzen auf. Die Anwesenheit einer Kante in einer Teilmatrix wird durch den Wert 1 des Elternknotens dargestellt. Besteht eine Teilmatrix nur aus Nullen, wird diese als Knoten mit dem Wert 0 im Baum eingefügt. Dieser Kompressionsalgorithmus eignet sich für sparse Matrizen, da er auf der Komprimierung der leeren Bereichen in der Adjazenzmatrix aufbaut.

2.3.4. Theoretisches Maximum für den Kompressionsfaktor

Die theoretische Obergrenze zur verlustfreien Datenkompression C_{max} wird durch die Entropie des Datensatzes $H(x)$ bestimmt:

$$C_{max} \sim H^{-1}(x) \tag{2.2}$$

Der tatsächliche Kompressionsfaktor C hängt von den vorliegenden Daten ab, eine generelle Aussage zu einem spezifischen Kompressionsfaktor eines Kompressionsalgorithmus kann nicht gemacht werden, wie sich einfach durch das sogenannte “pigeon-hole principle” beweisen lässt: Existiert ein Algorithmus $\mathcal{A}$ welcher Daten mit einem garantierten Kompressionsfaktor p komprimieren könnte, so bildet dieser jeden Datensatz der Länge n auf einen komprimierten Datensatz der Länge pn ab. Es existieren maximal 2^n Datensätze mit der Länge n und 2^{pn} komprimierte Datensätze der Länge pn. Da $2^n > 2^{pn}$ für $0 < p < 1$ wird der Algorithmus $\mathcal{A}$ auf jeden Fall mehrere unterschiedliche Datensätze auf den gleichen komprimierten Datensatz abbilden. Somit ist der Algorithmus nicht umkehrbar und daher können nicht alle komprimierten Datensätze wieder entkomprimiert werden.

3. Methoden

3.1. Wahl der Kompressionsalgorithmen

Die Kompressionsalgorithmen wurden auf Grund der folgenden Kriterien ausgewählt:

- Komplexität (in der für die Arbeit zur Verfügung stehenden Zeit mussten die Algorithmen in Python implementiert werden können, es wurde darauf geachtet, Algorithmen mit einer verständlichen Komplexität zu wählen).
- Unterschiedliche Ansätze: Die verwendeten Algorithmen sollten auf unterschiedlichen Ansätzen beruhen.
- Anwendungsgebiet: Bislang wurden keine spezifischen Untersuchungen zu Komprimierung von Barabási-Albert Graphen durchgeführt. Die verwendeten Algorithmen sollten daher nicht zu domänenspezifisch sein.

Schlussendlich wurden die Algorithmen Huffman Encoding und k^2-Tree ausgewählt, da diese die drei Kriterien gut erfüllen. Beide Algorithmen sind etabliert und in der Literatur gut beschrieben. In der Literatur finden sich zu fast keinen der gesichteten Algorithmen verfügbare Implementationen in Python, weshalb die beiden Algorithmen durch den Autor implementiert wurden.

3.2. Implementierung der Algorithmen in Python

Die Algorithmen wurden in Python 3.11.4 implementiert. Dazu wurden die folgenden Bibliotheken verwendet:

- `collections`[1] (Huffman Encoding)
- `heapq`[2] (Huffman Encoding)
- `Numpy`[3], Version 1.24.2 (k^2-Tree)

[1] https://docs.python.org/3/library/collections.html
[2] https://docs.python.org/3/library/heapq.html
[3] https://numpy.org/

3.2.1. Huffman Encoding

Der für die vorliegende Arbeit entwickelte Algorithmus ist im Anhang unter A.3 zu finden und basiert auf den folgenden Schritten:

1. **Berechnung der Häufigkeit** (`calculate_frequency`): Die Kantenliste des Graphen wird in eine Zeile geschrieben und in 8-Bit-Blöcke unterteilt. Die Häufigkeit jedes 8-Bit-Musters wird gezählt und in einem `defaultdict` gespeichert

2. **Aufbau des Huffman-Baums** (`build_huffman_tree`): Es wird ein Heap erstellt, in welchem jeder Knoten die Attribute `symbol` (das Muster), `frequency` (die Häufigkeit des Musters) und die Zeiger auf die linken und rechten Kinder besitzt. Die Knoten werden aufgrund ihrer Häufigkeit priorisiert, wobei Knoten mit geringerer Häufigkeit höhere Priorität haben. Der Heap wird solange modifiziert, bis nur noch ein Knoten übrig ist, der die Wurzel des Huffman-Baums darstellt. Je häufiger ein Muster vorkommt, desto näher an der Wurzel ist dieses.

3. **Zuordnung der Codes** (`assign_codes`): Jedem Muster im Huffman-Baum wird ein eindeutiger binärer Code zugewiesen. Der Code hängt von der Position des Musters im Baum ab, dazu wird ein rekursiver Durchlauf durch den Huffman-Baum durchgeführt, wobei jedem linken Kind ein `0` und jedem rechten Kind ein `1` hinzugefügt wird. Dadurch wird sichergestellt, dass kein Muster einen Code (Huffman-Code) erhält, welcher selbst Präfix eines anderen Musters ist. Dieser Baum stellt das Codebuch für den Graphen dar.

4. **Datenkodierung** (`encode_data`): Die ursprüngliche Abfolge der Kantenlisteneinträge wird nun durch die entsprechenden Huffman-Codes ersetzt. Wird ein Muster nicht im Codebuch gefunden, wird dieses nicht ersetzt.

3.2.2. K2-Tree

Für die vorliegende Arbeit wurde eine eigene Implementation in Python erstellt. Diese ist im Anhang unter A.4 zu finden. Der Algorithmus führt rekursiv die folgenden Schritte durch:

1. **Vorbereitung**: Der Algorithmus nimmt den Graphen in Form einer Adjazenzmatrix auf, welche in ein Numpy-Array umgewandelt wird. Ausserdem wird der Parameter k benötigt, welcher die Anzahl der Kinder pro Knoten des k^2-Tree angibt.

2. **Fallunterscheidung - Nur Nullen**: Besteht die Matrix aus lauter Nullen, (keine Kanten zwischen den Knoten im Graphen), wird ein Knoten mit dem Wert 0 erstellt und zurückgegeben.

3. **Fallunterscheidung - Einzelelement-Matrix**: Wenn die Matrix nur aus einem Element besteht (der Graph besitzt nur einen Knoten), wird ein Knoten mit dem Wert dieses Elements erstellt und zurückgegeben.

4. **Aufteilung der Matrix**: Besitzt die Matrix mehr als ein Element, wird sie in $k \times k$ Teilmatritzen aufgeteilt. Dabei wird die Grösse der Matrix durch k geteilt um die Grösse jeder Teilmatritzen zu bestimmen.

5. **Rekursion**: Nun wird der Algorithmus rekursiv auf die Teilmatritzen angewandt.

Durch die rekursive Anwendung des Algorithmus wird ein k^2-Baum erstellt, welcher die Beziehungen der gegebenen Adjazenzmatrix darstellt und als Blätter die nichtleeren Teilmatritzen enthält.

3.3. Datensatz

In der Arbeit wurden ausschliesslich Barábasi-Albert Graphen verwendet. Grund für diese Wahl ist die Möglichkeit, Parameter wie die Grösse (hier die Anzahl der Knoten n) und die Komplexität (hier die Anzahl Kanten pro Knoten m) kontrollieren zu können. So können die Einflüsse dieser Parameter direkt untersucht werden. Für die Messungen der Kompressionszeit und des Kompressionsfaktors wurden deshalb 50 Varianten mit unterschiedlichen Werten für n (Knotenanzahl) und m (Anzahl mit einem Knoten verbundene Kanten) erstellt (vgl. Tabelle 3.1). Von jeder Kombination (n, m) wurden 10 unterschiedliche Graphen erzeugt, insgesamt also 500 Graphen. Für die Graphen $G_i(n, m)$ gilt jeweils $m < n$. Der Code zur Erzeugung der Graphen ist im Anhang unter A.1 zu finden.

3.4. Messgrössen

Um die Effektivität der beiden Algorithmen k^2-Tree und Huffman Encoding beurteilen zu können, wurden die folgenden Messkritereien festgelegt:

Werte für n	Werte für m
100	10, 50
200	10, 50, 100
300	10, 50, 100, 200
400	10, 50, 100, 200
500	10, 50, 100, 200, 400
600	10, 50, 100, 200, 400
700	10, 50, 100, 200, 400, 600
800	10, 50, 100, 200, 400, 600
900	10, 50, 100, 200, 400, 600, 800
1000	10, 50, 100, 200, 400, 600, 800, 900

Tabelle 3.1: Die in der Arbeit verwendeten Werte für die Anzahl Knoten n und der Kanten pro Knoten m zur Erzeugung der Barabási-Albert Graphen

3.4.1. Kompressionsfaktor

Unter dem Kompressionsfaktor C eines Algorithmus verstehen wir das Verhältnis des Speicherbedarfs des komprimierten Datensatzes $d_{\text{komprimiert}}$ zum Speicherbedarf des Originaldatensatzes d_{original}:

$$C = \frac{d_{\text{komprimiert}}}{d_{\text{original}}} \tag{3.1}$$

Der Kompressionsfaktor C ist ein Mass für die Effizienz der Datenreduktion, die der Algorithmus erreichen kann. Für C gilt $0 \leq C \leq 1$, wobei kleinere Werte eine effizientere Kompression darstellen. Der Kompressionsfaktor ist für jeden Graphen eindeutig bestimmbar. Da die Graphen zufällig erzeugt werden und dadurch mehrere Graphen $G_i(n, m)$ bei konstanten Parametern n und m unterschiedliche Kompressionsfaktoren C_i erhalten, werden die Kompressionsfaktoren $C(n, m)$ für eine spezifische Kombination der Parameter n und m gemittelt:

$$C(n, m) = \frac{1}{h} \sum_{i=0}^{h-1} C_i(n, m) \tag{3.2}$$

3.4.2. Kompressionszeit

Die Kompressionszeit t_C ist die Zeit, die der Algorithmus benötigt, um den Originaldatensatz d_{original} in den komprimierten Datensatz $d_{\text{komprimiert}}$ überzuführen:

$$t_C = t_{\text{komprimiert}} - t_{\text{original}} \tag{3.3}$$

Wobei $t_{\text{komprimiert}}$ den Zeitpunkt der erfolgreich abgeschlossenen Kompression und t_{original} den Zeitpunkt des Starts des Kompressionsvorgangs bezeichnen. Diese Zeiten werden mit einer Auflösung von mindestens 1×10^{-9} s gemessen. Um die Varianz durch externe Faktoren zu minimieren, wird jeder Kompressionsprozess mehrmals durchgeführt und der Durchschnitt der Zeiten als endgültige Kompressionszeit genommen. Der zur Messung der Kompressionszeiten verwendete Code ist im Anhang unter A.2 zu finden.

3.4.3. Dichte

Die Barábasi-Albert-Graphen werden mittels der beiden Parameter n und m erzeugt (vgl. Kapitel 2.1.1). Für gewisse Auswertungen ist jedoch die Dichte d der erzeugten Graphen wichtig. Die Dichte d eines Graphen G mit n Knoten und E Kanten ist definiert als das Verhältnis der tatsächlich vorhandenen Kanten zu den potenziell möglichen Kanten in G ([4]):

$$d = \frac{2|E|}{n(n-1)} \tag{3.4}$$

Für die Dichte d eines Graphen gilt $0 \leq d \leq 1$. Die Dichte der generierten Graphen wurde mittels der Funktion `density(G)` der Python Bibliothek `networkx`[4] bestimmt (vgl. Tabelle 4.1).

3.4.4. Kanten

Die Anzahl der Kanten in einem Barábasi-Albert Graphen hängt direkt von den beiden Parametern n und m ab. Sie wird in dieser Arbeit jedoch aus den erzeugten Graphen mittels der Funktion `number_of_edges()` aus der Bibliothek `networkx` bestimmt.

[4] https://networkx.org/

4. Experimente und Ergebnisse

4.1. Systemumgebung

Sowohl die Erzeugung der Graphen wie auch sämtliche Messungen wurden auf einer dedizierten Google Cloud Instanz vom Typ `e2-standard-2`[1] (3.5 GHz, 8 GB RAM) durchgeführt. Auf der Instanz wurde das Betriebssystem Ubuntu Server 22.04 installiert. Der zur Erzeugung der Graphen und für die Messungen benötigte Code wurde in Python erstellt. Dazu wurde Python in der Version 3.11 benutzt. Sämtlicher Code ist im Anhang zu finden.

4.1.1. Python Bibliotheken

Die Tabelle 4.1 zeigt die in der Arbeit verwendeten Python Bibliotheken mit der jeweils benutzten Version. Weiter wurden ausschliesslich die in der benutzten Version von Python mitgelieferten Standardbilbiotheken verwendet.

Python Bibliothek	Version
`matplotlib`	`3.7.1`
`networkx`	`3.1`
`numpy`	`1.24.3`
`pandas`	`2.0.1`
`scipy`	`1.10.1`
`seaborn`	`0.12.2`

Tabelle 4.1: Sämtliche in der Arbeit verwendeten Python Bibliotheken inklusive der verwendeten Versionen

[1] https://gcloud-compute.com/e2-standard-2.html

4.2. Kompression mittels Huffman Encoding

Im Folgenden werden die Ergebnisse der Kompression mittels des Huffman Encoding Algorithmus (vgl. 3.2.1) vorgestellt.

4.2.1. Kompressionszeit

Die Messungen zur Bestimmung der Kompressionszeit t_C mittels Huffman Encoding wurden auf dem gesamten Datensatz durchgeführt (vgl. Kapitel 3.3)., Die Kompressionszeiten wurden für konstante Werte (n, m) gemittelt. Eine Analyse der Korrelationskoeffizienten zwischen Kompressionszeit und den Parametern n und m zeigt eine deutliche Abhängigkeit der Kompressionszeit von der Anzahl Knoten n (vgl. Tabelle 4.2). Die Kompressionszeit steigt, wie aus Abbildung 4.1 zu entnehmen ist, über den gesamten Messbereich quadratisch mit der Anzahl Knoten an. Die Abhängigkeit von der Variable m wird in der Abbildung farblich unterschieden, da deren Korrelation mit der Kompressionszeit t_C gemäss Tabelle 4.2 gering ist, wird diese Beziehung nicht weiter verfolgt.

Variable	Korrelationskoeffizient
n	0.977
m	0.436

Tabelle 4.2: Korrelationskoeffizienten zwischen der Kompressionszeit (Huffman Encoding) und den Graph-Parametern (n, m)

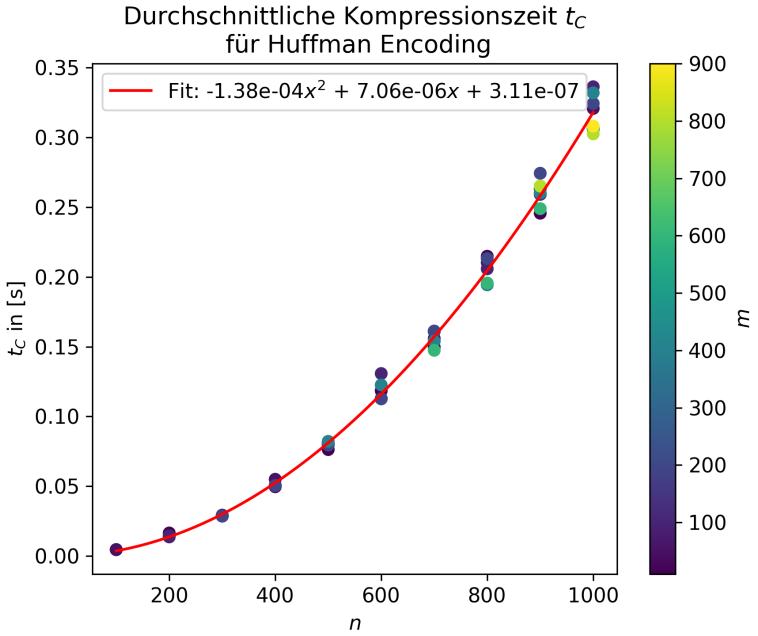

Abbildung 4.1: Die gemessenen durchschnittlichen Kompressionszeiten für Huffman Encoding der Barábasi-Albert Graphen mit gegebenen Parametern n (Anzahl Knoten und m (Anzahl mit einem Knoten verbundene Kanten)

4.2.2. Kompressionsfaktor

Zur Bestimmung des Kompressionsfaktors C wurden sämtliche Graphen aus dem Datensatz (vgl. Kapitel 3.3 mittels Huffman Encoding komprimiert.

Variable	Korrelationskoeffizient
n	-0.240
m	-0.193
d	-0.675
E	0.261

Tabelle 4.3: Korrelationskoeffizienten zwischen dem Kompressionsfaktor (Huffman Encoding) C_{Huffman} und den Graph-Parametern (n, m), der Graphdichte d und der Anzahl Kanten E

Die Korrelation mit den beiden Graph-Parametern n und m ist klein, deshalb wurden zusätzlich die aus den Graphen ermittelten Grössen E und d untersucht. Die grösste (absolute) Korrelation mit dem Kompressionsfaktor C_{Huffman} weist gemäss Tabelle 4.3 die

Graph-Dichte d auf, diese Korrelation ist negativ. Deshalb wird in Abbildung 4.2 diese Abhängigkeit dargestellt. Der Kompressionsfaktor C ist für eine Dichte von ca. 0.4 und Werte von $m < 200$ maximal, nimmt mit Werten von $m > 200$ stark ab.

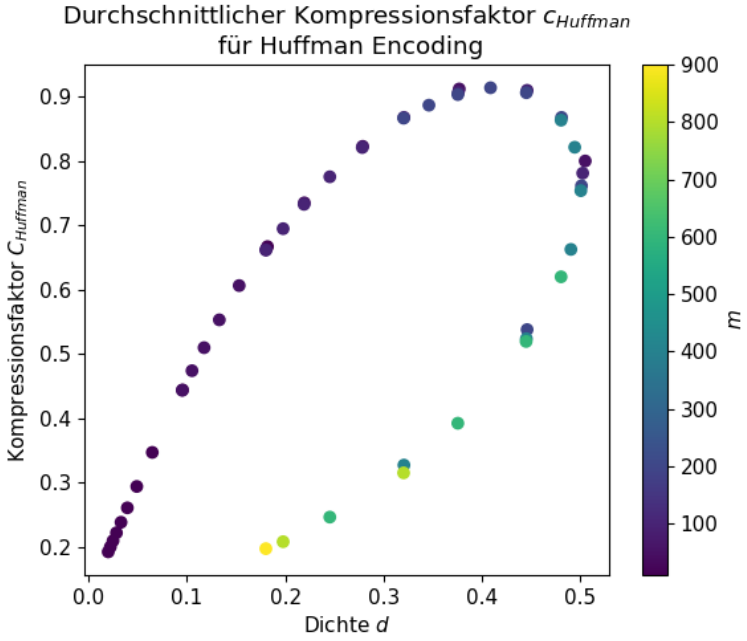

Abbildung 4.2: Die gemessenen durchschnittlichen Kompressionsfaktoren für Huffman Encoding der Barábasi-Albert Graphen in Abhängigkeit der Dichte d und des Graph-Parameters m

4.3. Kompression mittels k^2-Tree

Die Kompression mittels k^2-Tree (vgl. 3.2.2) wurde analog zur Kompression mittels Huffman Encoding durchgeführt. Für sämtliche Messungen wurde der Parameter $k = 2$ verwendet.

4.3.1. Kompressionszeit

Auch die Messungen zur Bestimmung der Kompressionszeit mittels k^2-Tree wurden auf dem gesamten Datensatz durchgeführt. Ebenfalls wurden die jeweils 10 Messungen pro Parameterpaar (n, m) gemittelt, um den Einfluss von Varianzen abzuschwächen. Die Tabelle 4.4 zeigt, dass die Kompressionszeit t_C primär von der Anzahl Kanten n im Graphen

abhängt.

Variable	Korrelationskoeffizient
n	0.759
m	0.187

Tabelle 4.4: Korrelationskoeffizienten zwischen der Kompressionszeit (k^2-Tree) und den Graph-Parametern (n, m)

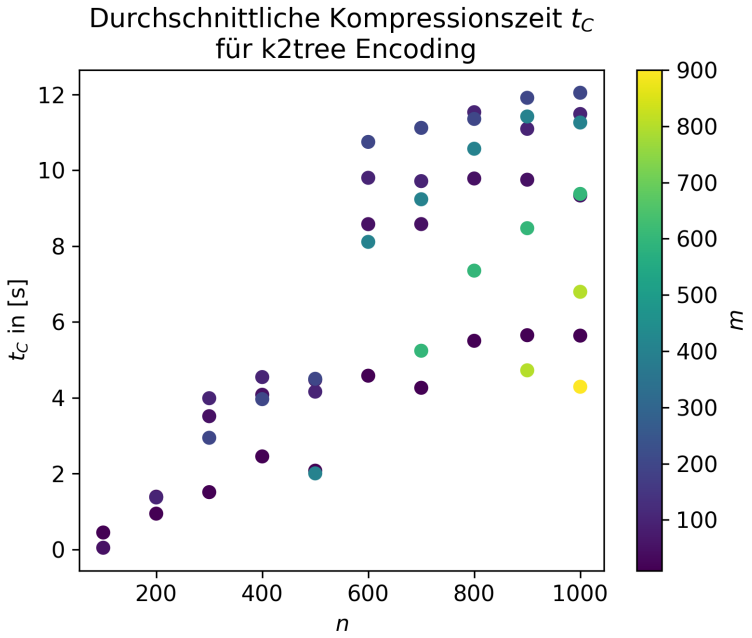

Abbildung 4.3: Die bei der Kompression mittels k^2-Tree gemessenen Kompressionszeiten t_C in Abhängigkeit der Anzahl Knoten d

4.3.2. Kompressionsfaktor

Der Kompressionsfaktor $C_{k2-Tree}$ für k^2-Tree wurde auf dem gesamten Datensatz gemessen, die jeweils 10 Messungen mit identischen Graph-Parametern (n, m) wurden gemittelt. Auch hier wurden die Korrelationskoeffizienten der Graph-Parameter und der Graph-Eigenschaften d und E mit der Kompressionszeit bestimmt (vgl. Tabelle 4.5).

Variable	Korrelationskoeffizient
n	-0.455
m	-0.303

Tabelle 4.5: Korrelationskoeffizienten zwischen dem Kompressionsfaktor $C_{k2-Tree}$ und den Graph-Parametern (n, m)

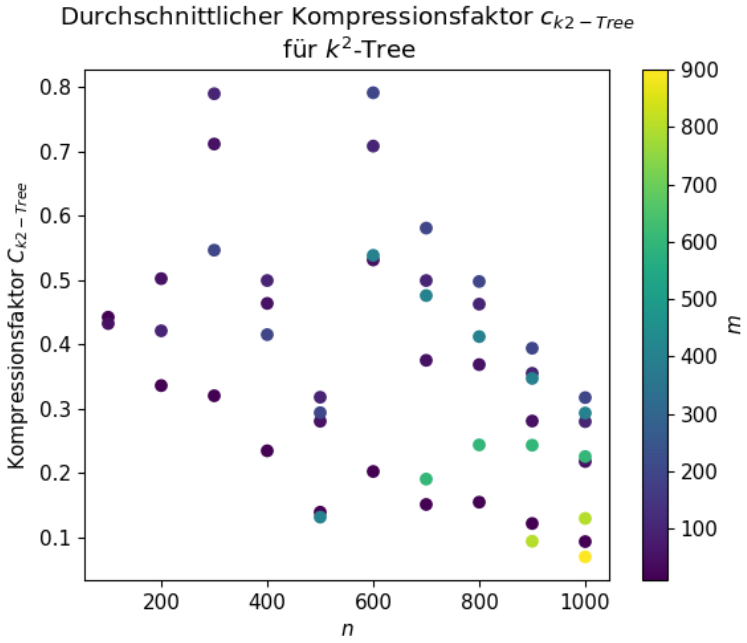

Abbildung 4.4: Der Kompressionsfaktor $C_{k2-Tree}$ in Abhängigkeit des Graph-Parameters n (Anzahl Knoten) für die Kompression mittels k^2-Tree

Analog zur Auswertung in Kapitel 4.2.2 wird in Abbildung 4.5 der Kompressionsfaktor $C_{k2-Tree}$ in Abhängigkeit der Dichte d gezeigt. Insgesamt ist der Kompressionsfaktor sehr variabel, steigt mit steigender Dichte jedoch leicht an.

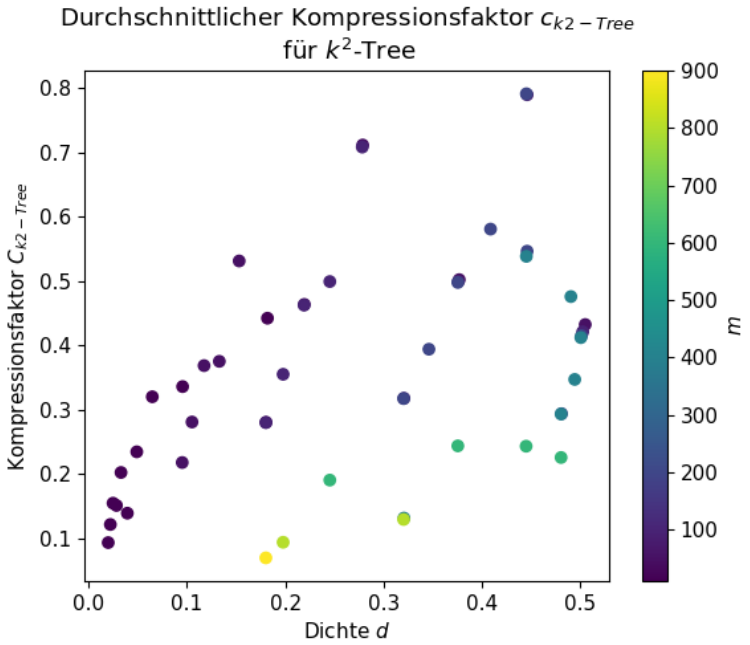

Abbildung 4.5: Der Kompressionsfaktor $C_{k2-Tree}$ in Abhängigkeit der Graphdichte d für die Kompression mittels k^2-Tree

5. Diskussion

Die im Rahmen dieser Arbeit durchgeführten Messungen zeigen für beide verwendeten Kompressionsalgorithmen Stärken und Schwächen bezüglich der untersuchten Messgrössen Kompressionsfaktor und Kompressionszeit.

5.1. Kompressionsfaktor

Der Kompressionsfaktor hängt bei der Verwendung von Huffman Encoding von der Dichte d des Graphen ab und erreicht für die untersuchten Graphen bei einer Dichte von ungefähr 0.4 das Maximum. Der Kompressionsfaktor, welcher durch die Verwendung von k^2-Tree erreicht wird, liegt für den gesamten Messbereich unter den durch Huffman maximal erreichbaren Werte (vgl. Abbildung 5.1).

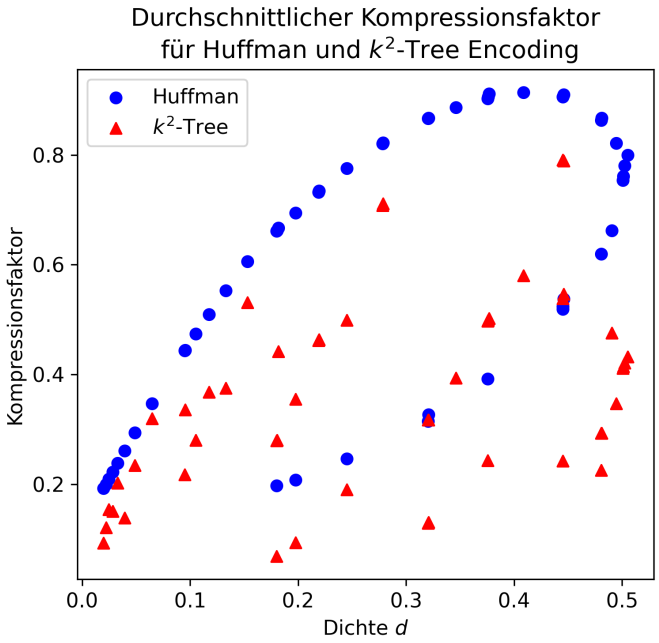

Abbildung 5.1: Vergleich der erreichten Kompressionsfaktoren für die beiden untersuchten Algorithmen Huffman Encoding und k^2-Tree

Die Abhängigkeit des Kompressionsfaktors von der Dichte beim Einsatz von Huffman Encoding und dessen Maximum bei einer Dichte von 0.4 hängt vermutlich von der Entropie der Kantenliste ab, welche für die Graphen gemäss Gleichung 2.2 auf Seite 6 ein Minimum besitzen muss. Diese Vermutung müsste in einer weiterführenden Arbeit untersucht werden.

5.2. Kompressionszeit

Die Daten zeigen eine hohe Korrelation zwischen der Anzahl der Knoten n in den untersuchten Graphen und der Zeit t_{Huffman}, die Huffman Encoding für die Kompression benötigt. Die Abhängigkeit der Kompressionszeit von der Anzahl Knoten lässt sich für die untersuchten Barábasi-Albert Graphen mit einer quadratischen Funktion darstellen.

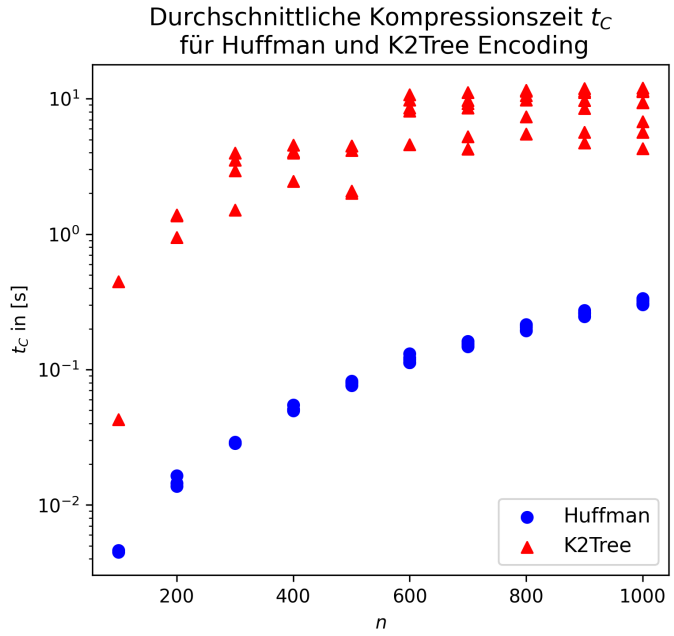

Abbildung 5.2: Vergleich der Kompressionszeiten für die beiden untersuchten Algorithmen Huffman Encoding und k^2-Tree

Huffman Encoding ist also bezüglich der Kompressionszeit weniger effizient für grössere Graphen. Im Gegensatz dazu hängt k^2-Tree etwas weniger stark von der Grösse der Graphen ab, die Kompressionszeit steigt jedoch in der Tendenz mit n an, was auch in der relativ hohen Korrelation von 0.759 zur Kompressionszeit sichtbar wird. Der direkte

Vergleich zwischen den beiden Algorithmen (vgl. Abbildung 5.2) zeigt, dass die Kompressionsfaktoren von Huffman Encoding auf den untersuchten Graphen durch k^2-Tree nicht erreicht werden.

6. Schlussfolgerungen

Die in dieser Arbeit gestellten Forschungsfragen (vgl. Kapitel 1.2) konnten anhand der durchgeführten Untersuchungen wie folgt beantwortet werden:

1. Die Effektivität der beiden Kompressionsalgorithmen Huffman Encoding und k^2-Tree in Bezug auf Kompressionsfaktor und -Zeit bei der Kompression von Barábasi-Albert Graphen hängt stark von den spezifischen Eigenschaften des Graphen ab. In Bezug auf den Kompressionsfaktor erreicht Huffman Encoding bei einer Dichte von ca. 0.4 das Maximum. Der durch k^2-Tree erreichte Kompressionsfaktor bleibt über alle untersuchten Graphen unter den Maximalwerten von Huffman Encoding. Die Kompressionszeit von Huffman Encoding steigt mit dem Quadrat der Anzahl Knoten an, bleibt jedoch über den gesamten Messbereich bis $n = 1000$ stark unter den durch k^2-Tree benötigten Kompressionszeiten.

2. Die Leistung der beiden Algorithmen unterscheiden sich stark. Beide hängen bezüglich der Kompressionszeit von der Grösse der Graphen ab, wobei die Abhängigkeit bei Huffman Encoding stärker ausgeprägt ist. Die Dichte hat einen grossen Einfluss auf den Kompressionsfaktor.

3. Die Leistung von Huffman Encoding hängt stark von Dichte und Grösse der Graphen ab. Huffman Encoding eignet sich bei den untersuchten Graphen bei einer Dichte um den Maximalwert von 0.4. Weitere Untersuchungen mit grösseren Graphen müssten zeigen, ob sich die Leistung von k^2-Tree bei noch grösseren Graphen mit kleinerer Dichte verbessern würde.

Zusammenfassend lässt sich sagen, dass von den beiden Algorithmen nur Huffman Encoding wirklich gut zur Kompression von Barábasi-Albert Graphen geeignet ist. k^2-Tree kann im untersuchten Bereich seine Stärke durch seine Abhängigkeit von leeren Bereichen in der Adjazenzmatrix bei zufällig erzeugten, skalenfreien Graphen nicht ausspielen.

7. Zukünftige Arbeiten

Die in dieser Arbeit durchgeführten Untersuchungen haben Erkenntnisse zur Leistungsfähigkeit von Huffman Encoding und k^2-Tree bei der Kompression von Barábasi-Albert-Graphen geliefert. Die gewonnenen Erkenntnisse werfen aber gleichzeitig neue Fragen auf, welche in zukünftigen Forscungsarbeiten untersucht werden können:

- Eventuell könnte sich k^2-Tree bei noch grösseren und weniger dichten Barábasi-Albert Graphen als etwas effizienter herausstellen, eine weitere Untersuchung könnte zeigen, in welchem Bereich das Optimum für k^2-Tree liegen könnte.
- Für die Kompression von Barábasi-Albert Graphen müssten zwingend weitere Algorithmen untersucht werden, welche auf die speziellen Eigenschaften dieser Graphen optimiert sind.
- In der Arbeit wurde nicht untersucht, wie gut sich die unterschiedlichen Kompressionsarten für Analysen der Graphen oder Operationen auf den Graphen eignen. Diese Aspekte sind jedoch bei der Beurteilung eines Kompressionsalgorithmus zentral.

Die vorliegenden Forschungsergebnisse bilden eine Grundlage für zukünftige Untersuchungen in diesem Bereich. Die gewonnenen Erkenntnisse können dazu verwendet werden, Algorithmen zur Kompression von Barábasi-Albert Graphen zu optimieren.

Literaturverzeichnis

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[3] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv*, pages 1806–01799, 2018.

[4] Ulrik Brandes. Graphentheorie. *Handbuch Netzwerkforschung*, pages 345–353, 2010.

[5] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *SPIRE*, volume 9, pages 18–30. Springer, 2009.

[6] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[7] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[8] Tomasz Luczak, Abram Magner, and Wojciech Szpankowski. Structural information and compression of scale-free graphs. *Urbana*, 51:618015, 2017.

[9] Muhammad Naeem, Tauseef Jamal, Jorge Diaz-Martinez, Shariq Aziz Butt, Nicolo Montesano, Muhammad Imran Tariq, Emiro De-la Hoz-Franco, and Ethel De-La-Hoz-Valdiris. Trends and future perspective challenges in big data. In *Advances in Intelligent Data Analysis and Applications: Proceeding of the Sixth Euro-China Conference on Intelligent Data Analysis and Applications, 15–18 October 2019, Arad, Romania*, pages 309–325. Springer, 2022.

[10] Xia Zhu, Jing Zhang, and Hongbo Zhu. Research of data compression using huffman coding and arithmetic coding. In *Proceedings of the 12th International Conference on Computer Engineering and Networks*, pages 954–961. Springer, 2022.

A. Code

A.1. Code zum Generieren der Graphen

```
def generate_graphs():
    graph_list = []
    adj_list = []
    nm_list = []
    n_values = [100,200,300,400,500,600,700,800,900,1000]
    m_values = [10,50,100,200,400,600,800,900,1000]

    # Create directory for saving graphs
    if not os.path.exists('Graphs'):
        os.makedirs('Graphs')

    for n in n_values:
        for m in m_values:
            if m < n:
                for i in range(10):  # Repeat 10 times for each
                ↪  combination of n and m
                    graph = nx.barabasi_albert_graph(n, m)
                    adj = nx.adjacency_matrix(graph).todense().tolist()
                    graph_list.append(graph)
                    adj_list.append(adj)
                    nm_list.append((n, m))  # Save n, m used for
                    ↪  generating this graph

                    # Save graph
                    nx.write_graphml(graph,
                    ↪  f'Graphs/graph_{n}_{m}_{i}.graphml')
    return graph_list, adj_list, nm_list
```

A.2. Code Messung Kompressionszeiten

```
def compare_algorithms(graph_list, adj_list, nm_list):
    results = []

    for (graph, adj, nm) in zip(graph_list, adj_list, nm_list):
        n_nodes = graph.number_of_nodes()
        m_edges = graph.number_of_edges()
        density = nx.density(graph)
        n_value, m_value = nm  # n, m used for generating this graph

        # Measure time for k2tree compression
        start_time = time.time()
        k2_tree = k2t.k2_tree(adj, 2)
        k2tree_time = time.time() - start_time
        k2tree_compression = k2t.count_nodes(k2_tree)/(n_nodes**2)  #
        ↪  correct here

        # Measure time for Huffman compression
        adj_bin = ''.join(str(e) for row in adj for e in row)
        start_time = time.time()
        huffman_compress = huff.compress_binary_data(adj_bin)
        huffman_time = time.time() - start_time
        huffman_compress = len(huffman_compress)/(n_nodes**2)  # correct
        ↪  here

        results.append((n_value, m_value, n_nodes, m_edges, density,
        ↪  k2tree_time, k2tree_compression, huffman_time,
        ↪  huffman_compress))

    return results
```

A.3. Code Implementierung Huffman Encoding

```
from collections import defaultdict
import heapq
```

```
class Node:
    def __init__(self, symbol, frequency, left=None, right=None):
        self.symbol = symbol
        self.frequency = frequency
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.frequency < other.frequency

def calculate_frequency(data):
    frequency = defaultdict(int)
    for i in range(len(data) - 7):
        pattern = data[i:i+8]
        frequency[pattern] += 1
    return frequency

def build_huffman_tree(frequency):
    heap = []
    for pattern, freq in frequency.items():
        heapq.heappush(heap, Node(pattern, freq))

    while len(heap) > 1:
        node1 = heapq.heappop(heap)
        node2 = heapq.heappop(heap)
        merged = Node(None, node1.frequency + node2.frequency, node1,
        ↪  node2)
        heapq.heappush(heap, merged)

    return heap[0]

def assign_codes(node, code='', codes={}):
    if node.symbol:
        codes[node.symbol] = code
    else:
```

```
        assign_codes(node.left, code + '0', codes)
        assign_codes(node.right, code + '1', codes)

    return codes

def encode_data(data, codes):
    encoded_data = ''
    i = 0
    while i < len(data) - 7:
        pattern = data[i:i+8]
        if pattern in codes:
            encoded_data += codes[pattern]
            i += 8
        else:
            encoded_data += pattern
            i += 1
    return encoded_data

def compress_binary_data(data):
    frequency = calculate_frequency(data)
    huffman_tree = build_huffman_tree(frequency)
    codes = assign_codes(huffman_tree)
    encoded_data = encode_data(data, codes)
    return encoded_data
```

A.4. Code Implementierung K2-Tree

```
import numpy as np

# Node class for k2-tree
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []
```

```
    # used for printing the tree
    def __repr__(self, level=0):
        ret = "\t" * level + repr(self.value) + "\n"
        for child in self.children:
            ret += child.__repr__(level+1)
        return ret

# Generate a k2-tree from the adjacency matrix
def k2_tree(matrix, k):
    n = len(matrix)

    matrix_array = np.array(matrix)
    if np.all(matrix_array == 0):
        return Node(0)

    if n == 1:
        return Node(matrix_array[0, 0])

    nodes = []
    size = n // k
    for i in range(k):
        for j in range(k):
            nodes.append(k2_tree(matrix_array[i*size:(i+1)*size,
            ↪  j*size:(j+1)*size].tolist(), k))

    node = Node(any(node.value == 1 for node in nodes))
    node.children = nodes

    return node

def count_nodes(node):
    return 1 + sum(count_nodes(child) for child in node.children)
```