Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Verteiltes Subgraph-Counting mit Colorcoding

Emre Güzel

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Kurt Rothermel
Betreuer/in:	Michael Schramm, M.Sc.
Beginn am:	16. Juni 2021
Beendet am:	16. Dezember 2021

Kurzfassung

Für das Subgraph-Counting-Problem und das Subgraph-Enumeration-Problem sind die Anzahlen und Häufigkeiten der Isomorphismen für bestimmte Subgraphen in größeren Graphen von Interesse. Zur Lösung dieser Probleme gibt es verschiedene Ansätze. Einer dieser Ansätze basiert auf dem Colorcoding [AYZ95] Verfahren von Alon et al. Die beiden Algorithmen PARSE [ZKKM10] und CC [BCK+17] [BCK+18] versuchen diese Probleme mit diesem Ansatz zu lösen. Dabei hat PARSE eine verteilte Version, wohingegen CC lediglich auf einer einzigen Maschine ausführbar ist. Zusätzlich zu diesen beiden Algorithmen wird in dieser Arbeit DistCC, eine verteilte Version CC's, vorgestellt. DistCC orientiert sich bei der Art und Weise seiner Verteilung am Beispiel von PARSE. Dank der Verteilung sind die Laufzeiten DistCC's teilweise deutlich kürzer, als die Laufzeiten CC's. Außerdem wurden neben der natürlichen Sortierung der Graphen auch zwei neue Sortierungen eingeführt, die die Laufzeiten CC's und somit auch DistCC's teilweise verbessern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Bei dem Subgraph-Counting-Problem wird die Anzahl der Einbettungen eines bestimmten Subgraphen, auch Template genannt, in einem großen Graphen gesucht. Da die Suche nach der exakten Anzahl der Einbettungen jedoch zu schwierig und rechenintensiv ist, wurden Approximationsbasierende Verfahren entwickelt, die die Anzahl mit einem bestimmten Grad an Genauigkeit abschätzen. Alle Algorithmen, die in dieser Arbeit vorgestellt werden sind approximationsbasiert. PARSE [ZKKM10] ist ein Algorithmus, der versucht das Subgraph-Counting-Problem zu lösen.

Das Subgraph-Enumeration-Problem beschäftigt sich mit den am häufigsten vorkommenden Subgraphen einer bestimmten Größe in einem Graphen. Hier müssen im Gegensatz zum Subgraph-Counting-Problem die Anzahl eines bestimmten Subgraphen also nicht genau bestimmt werden. Auch bei diesem Problem werden Approximations-basierende Verfahren eingesetzt. CC [BCK+17][BCK+18] versucht das Subgraph-Enumeration-Problem zu lösen.

Zu den Anwendungsgebieten von Subgraph-Counting und Subgraph-Enumeration gehören Mustererkennung in Netzwerken [Wer06], Soziale Netzwerke [LSK06][CLWL16][HS17], Bioinformatik [ADH+08a][MSl+11], Chemoinformatik [HWP03], Data Mining [KK02][KK04] und Graph-Mining [BRRA12][CKM+16][JSP14a][SM13b].

Bei der Suche nach Communities in sozialen Netzwerken spielt die Identifikation des Kerns einer Community eine wichtige Rolle. Eine Community wird gängig als eine Teilmenge der Knoten eines Graphen mit einer hohen Anzahl an Kanten innerhalb der Teilmenge und einer niedrigen Anzahl an Kanten die von der Teilmenge aus nach außen gehen definiert. Den Kern der Community bilden dabei oft fünf bis sieben Knoten, die alle zueinander eine Kante besitzen. Anders ausgedrückt Subgraphen mit fünf bis sieben Knoten. Durch Subgraph-Counting können die Anzahlen dieser Subgraphen bestimmt werden, um so über die Anzahl unterschiedlicher Communities innerhalb eines Graphen Auskunft zu geben [GN02].

Graphen können entweder gelabelt oder nicht-gelabelt sein. Ein Graph ist gelabelt, falls seine Knoten und Kanten Zusatzinformationen besitzen, die für bestimmte Berechnungen und Verfahren nützlich sind. Für solche Graphen ist das Zählen der Anzahlen und Häufigkeiten der Einbettungen der Subgraphen grundsätzlich einfacher und schneller, da diese Zusatzinformationen die potentiellen Einbettungen der Subgraphen stark einschränken. Nicht jeder Graph besitzt jedoch solche Zusatzinformationen für seine Knoten und Kanten. Bei solchen Graphen orientiert man sich deshalb viel mehr an der Struktur der Subgraphen, um Einbettungen zu finden. Diese Arbeit fokussiert sich auf Algorithmen, die diese Probleme in nicht-gelableten Graphen lösen können.

Was den Subgraph-Counting-Algorithmus PARSE mit dem Subgraph-Enumeration-Algorithmus CC verbindet ist die Tatsache, dass beide Algorithmen auf dem Colorcoding-Verfahren [AYZ95] basieren, und die daraus folgende Ähnlichkeit der Algorithmen. Colorcoding ist ein Verfahren, der den Knoten eines Graphen künstlich Zusatzinformationen zuweist. Da es sich bei diesen Algorithmen

um Algorithmen handelt, die die entsprechenden Probleme in nicht-gelabelten Graphen lösen, sind diese Zusatzinformationen bei der Suche nach den Anzahlen und Häufigkeiten der Einbettungen von Subgraphen sehr nützlich.

PARSE ist ein verteilter Algorithmus. CC ist nicht verteilt und somit nur auf einer einzigen Maschine ausführbar. In dieser Arbeit wird eine verteilte Version CC's vorgestellt, die sich DistCC nennt. DistCC orientiert sich bei der Verteilung nach der Verteilung des Algorithmus' PARSE. Zusätzlich zum Algorithmus werden in dieser Arbeit auch verschiedene Partitionierungsverfahren, die zur Verbesserung der Kommunikationszeit DistCC's beitragen vorgestellt.

Darüber hinaus werden durch geschickte Sortierungen der Graphen die Laufzeiten CC's und DistCC's verbessert. Aufgrund der Kommunikation zwischen den Threads CC's entsteht eine Verzögerung der Laufzeiten. Je nach Anzahl und Sortierung der Knoten in dem Graphen kann diese Verzögerung einen großen Anteil der Laufzeiten ausmachen. Zur Beseitigung dieser Verzögerung wird eine Modifizierung CC's namens NewCC präsentiert.

Zum Schluss werden dann DistCC und CC miteinander verglichen. Es wird auf die Laufzeiten und den Speicherverbrauch eingegangen. Zudem werden die verschiedenen Sortierungen und die daraus resultierenden Laufzeiten CC's analysiert. Auch die Laufzeiten NewCC's werden vorgestellt und mit denen CC's verglichen. Des Weiteren wird gezeigt, dass sich die Laufzeiten DistCC's mit einer zunehmenden Anzahl von Maschinen verbessern. Anschließend folgt dann die Analyse der verschiedenen Partitionierungsverfahren.

Zu Beginn werden in Kapitel 2 die für den Rest der Arbeit wichtigen Grundkenntnisse vermittelt. Eines der Verfahren, die in diesem Kapitel dargestellt werden, ist das bereits erwähnte Colorcoding-Verfahren. Darauf folgend werden in Kapitel 3 die auf Colorcoding basierenden Algorithmen PARSE und CC beschrieben und an einigen Stellen miteinander verglichen. Zudem wird auf die Unterschiede der Algorithmen eingegangen. In Kapitel 4 wird dann schließlich DistCC vorgestellt. Danach werden in Kapitel 5 der Versuchsaufbau, die Ergebnisse und Evaluationen der Experimente diskutiert. Daraufhin folgt in Kapitel 6 eine kurze Präsentation verwandter Publikationen. Kapitel 7 beinhaltet schließlich die Schlussfolgerung und einen Ausblick auf zukünftige Arbeiten.

2 Grundvoraussetzungen

In diesem Kapitel werden die für diese Arbeit notwendigen Grundvoraussetzungen eingeführt. In 2.1 wird formal definiert, was unter einem Graphen zu verstehen ist. Daraufhin wird in 2.2 der Subgraph-Isomorphismus beschrieben. In 2.3 wird erklärt, wie sich induzierte Subgraph-Isomorphismen von nicht-induzierten Subgraph-Isomorphismen unterscheiden. Anschließend wird in 2.4 definiert, was ein Automorphismus ist. In 2.5 wird die Colorcoding-Technik eingeführt. Zum Schluss werden in 2.6 und 2.7 das Subgraph-Counting-Problem und das Subgraph-Enumeration-Probleme erläutert.

2.1 Graph

Ein Graph ist formal ein Tupel aus einer Knotenmenge und einer Kantenmenge. Sei $G = (V, E)$ ein Graph. V ist dabei die Menge der Knoten und E die Menge der Kanten. Eine Kante $e = (u, v) \in E$ ist eine Verbindung zweier Knoten $u, v \in V$. Es wird zwischen gerichteten und ungerichteten Graphen unterschieden. Bei gerichteten Graphen haben die Kanten bestimmte Richtungen. Die Richtung der Kante $e = (u, v)$ richtet sich in diesem Fall von Knoten u nach Knoten v. In ungerichteten Graphen besitzen die Kanten keine Richtungen. Das bedeutet, dass die beiden Kanten (u, v) und (v, u) äquivalent sind. In gerichteten Graphen ist das nicht der Fall.

Die Menge $N(v)$ beinhaltet die Nachbarn des Knoten v. Dies sind alle Knoten $m \in V$ für die eine Kante $(v, m) \in E$ existiert.

In Abbildung 2.1 ist ein Graph mit sechs Konten und sechs Kanten zu sehen. Hierbei handelt es sich um einen ungerichteten Graphen. Der Knoten c hat in diesem Graphen die drei Nachbarn a, e und f, also $N(c) = \{a, e, f\}$.

2.2 Subgraph-Isomorphismus

Ein Subgraph-Isomorphismus M ist eine injektive Abbildung eines Subgraphen $T = (\hat{V}, \hat{E})$ in einem Graphen $G = (V, E)$ mit $|V| \geq |\hat{V}|$, die alle Knoten aus $\hat{V}$ unter Berücksichtigung der Kanten aus $\hat{E}$ auf Konten aus V abbildet. Dabei muss für jede Kante $(u', v') \in \hat{E}$ eine entsprechende Kante $(u, v) \in E$ existieren, sodass die Konten u' und v' auf die beiden Knoten u und v abgebildet werden. In anderen Worten muss sich also für jede Kante in T eine entsprechende Kante im Isomorphismus von T in G befinden.

Formal wird ein Subgraph-Isomorphismus wie folgt definiert:

$$M : \hat{V} \rightarrow V \text{ sodass } \forall (u', v') \in \hat{E} \Leftrightarrow (M(u'), M(v')) \in E$$

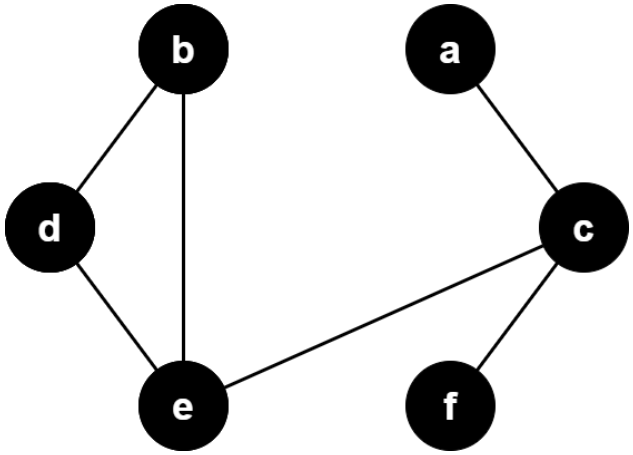

Abbildung 2.1: Beispielgraph G mit $V = \{a, b, c, d, e, f\}$ und $E = \{(a, c), (b, d), (b, e), (c, e), (c, f), (d, e)\}$

In Abbildung 2.2 ist ein Isomorphismus zwischen den Knoten $1, 2, 3, 4$ von T und den Knoten a, b, c, d, e, f von G zu erkennen. Die gestrichelten grünen Linien veranschaulichen die einzelnen Abbildungspaare der Konten.

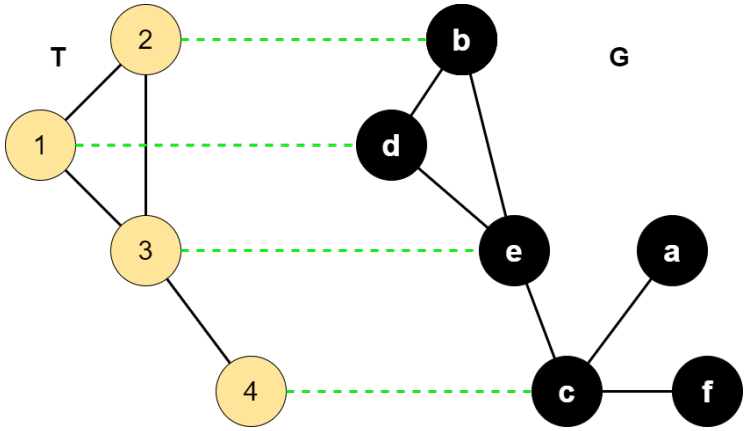

Abbildung 2.2: Beispiel Isomorphismus $M = \{(2 \to b), (1 \to d), (3 \to e), (4 \to c)\}$.

2.3 Induziert vs. nicht-induziert

Subgraph-Isomorphismen lassen sich in induzierte und nicht-induzierte unterteilen. Bei induzierten Subgraph-Isomorphismen dürfen sich in der Abbildung $M(\hat{V})$ der Knotenmenge $\hat{V}$ des Subgraphen T im Graphen G keine weiteren Kanten befinden als die Abbildungen $M(\hat{E})$ der Kanten $\hat{E} \in T$. Zwischen den Knoten in $M(\hat{V})$ dürfen sich demnach keine weiteren Kanten befinden, als die Kanten die in $M(\hat{E})$ existieren. Formal lässt sich das so definieren:

$$\forall (u, v) \notin \hat{E} : (M(u), M(v)) \notin E$$

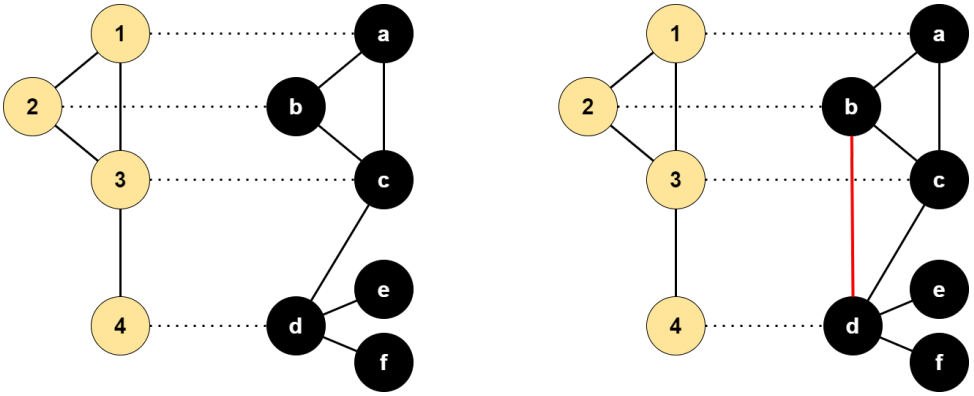

Abbildung 2.3: Links: zwischen den Knoten der Abbildung existieren keine zusätzlichen Kanten ⇒ induziert. Rechts: zwischen den beiden Knoten b und d befindet sich eine zusätzliche Kante ⇒ nicht-induziert.

Bei nicht-induzierten Subgraph-Isomorphismen gibt es diese Beschränkung nicht. Zusätzliche Kanten zwischen den Knoten in der Knotenmenge $M(\hat{V})$ sind erlaubt. Es genügt, wenn die Bedingung, die in 2.2 eingeführt wurde erfüllt ist. Aus diesem Grund ist es grundsätzlich einfacher, nicht-induzierte Subgraphen in einem Graphen zu finden als induzierte.

In Abbildung 2.3 wird dies anhand von zwei Beispielen veranschaulicht. Links ist eine induzierter Subgraph-Isomorphismus zu erkennen. Sowohl der Subgraph, als auch seine Abbildung besitzen genau vier Kanten zwischen ihren Knoten. Rechts dagegen ist ein nicht-induzierter Subgraph-Isomorphismus erkennbar. Hier besitzt die Knotenmenge der Abbildung eine zusätzliche, rot markierte Kante zwischen den beiden Knoten b und d.

2.4 Automorphismen

Ein Subgraph-Isomorphismus, bei dem ein Graph auf sich selbst abgebildet wird, wird Automorphismus genannt. Für die Subgraph-Counting- und Subgraph-Enumeration-Probleme sind Automorphismen äußert wichtig. Denn bei jedem gefundenen Isomorphismus, besteht die Möglichkeit, dass es sich um einen Automorphismus handelt. Deshalb ist es wichtig die Anzahl der Automorphismen pro Isomorphismus festzustellen und diese dann anschließend von der Anzahl des Vorkommens dessen Einbettungen abzuziehen.

Für einen Subgraphen $T = (\hat{V}, \hat{E})$ lässt sich ein Automorphismus formal folgendermaßen definieren:

$$A : \hat{V} \to \hat{V} \text{ sodass } \forall (u, v) \in \hat{E} \Leftrightarrow (A(u), A(v)) \in \hat{E}$$

Jeder Knoten aus $\hat{V}$ wird auf einen anderen (oder denselben) Knoten aus $\hat{V}$ abgebildet und jede Kante aus $\hat{E}$ wird auf eine andere (oder dieselbe) Kante aus $\hat{E}$ abgebildet. Jeder Subgraph hat mindestens den einen Automorphismus, bei dem jeder Knoten und jede Kante auf sich selbst abgebildet werden. Betrachtet man den Subgraphen T aus der Abbildung 2.2 kann man feststellen,

dass er zwei Automorphismen besitzt. Zunächst einmal der Automporhismus, der den Subgraphen auf sich selbst abbildet:
$A_1 = ((1 \Rightarrow 1), (2 \Rightarrow 2), (3 \Rightarrow 3), (4 \Rightarrow 4))$

Beim zweiten Automorphismus werden dann die beiden Knoten 1 und 2 vertauscht:
$A_2 = ((1 \Rightarrow 2), (2 \Rightarrow 1), (3 \Rightarrow 3), (4 \Rightarrow 4))$.

Da es viel zu aufwändig ist, bei jedem gefundenen Isomorphismus die Anzahl der Automorphismen zu berechnen, werden die Automorphismen in der Praxis grundsätzlich zu Beginn der Algorithmen einmal gezählt und abgespeichert.

2.5 Colorcoding

Die Suche nach der genauen Anzahl der Einbettungen eines Subgraphen T in einem Graphen G ist ein NP-vollständiges Problem. Aus diesem Grund wurden verschiedene Approximationsverfahren entwickelt. Eines dieser Verfahren stammt von Alon et al. und heißt Colorcoding [AYZ95] [ADH+08b]. Mithilfe dieses Verfahrens kann die Suche nach der approximierten Anzahl der Einbettungen bei Subgraphen, die bis zu $O(log(V))$ groß sind, in polynomialer Zeit abgeschlossen werden.

Zunächst wird G zufällig gefärbt. Jeder Konten aus G enthält eine von k vielen Farben, wobei k die Größe von T ist.

$$\forall v \in V : c_v = [1..k]$$

c_v ist die Farbe des Knotens v. Anschließend werden die Einbettungen von T gesucht. Damit eine Einbettung T's gefunden werden kann, muss diese vollständig gefärbt sein. Sei H eine gefundene Einbettung von T in G. H ist vollständig gefärbt, wenn gilt:

$$|V_H| = k \wedge \forall v \forall v' \in V_H : c_v \neq c_{v'}$$

Jeder Knoten in H muss also eine einzigartige Farbe besitzen. Nachdem man nun alle vollständig gefärbten Isomorphismen gefunden hat, kann die tatsächliche Anzahl e der zu T isomorphen Subgraphen in G abgeschätzt werden. Da jeder Knoten eine einzigartige Farbe besitzen muss beträgt die Wahrscheinlichkeit, dass ein Subgraph gefärbt ist $\frac{k!}{k^k}$. Somit ist die erwartete Anzahl der gefärbten Subgraphen $e * \frac{k!}{k^k}$.

In der Praxis werden die Färbung des Graphen, das Zählen der gefärbten Einbettungen und das Abschätzen der tatsächlichen Anzahl häufig in mehreren Iterationen wiederholt, um so die Genauigkeit des Ergebnisses zu verbessern und der Tatsache, dass Colorcoding ein Zufallsbasierender Algorithmus ist entgegenzuwirken.

In Abbildung 2.4 sieht man, dass lediglich eine vollständig gefärbte Einbettung T's in G existiert. Bei den restlichen Knoten kann die Suche nach einem Isomorphismus abgebrochen werden, sobald eine bereits vorhandene Farbe erneut erreicht wird. Das führt zu einem signifikanten Ersparnis an Laufzeit. Mehrere Iterationen ermöglichen es, sich der tatsächlichen Anzahl mit einer Abweichung von ϵ zu nähern. Dabei beträgt die Anzahl der benötigten Iterationen $O(\frac{e^k log(1/\delta)}{\epsilon^2})$ [ADH+08b].

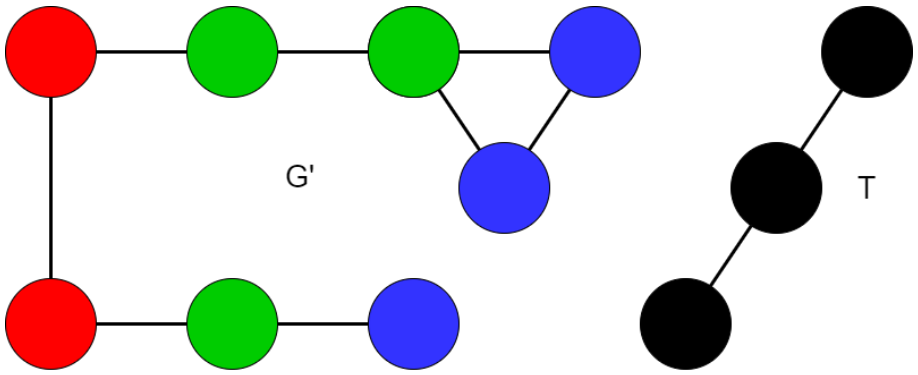

Abbildung 2.4: Gefärbter Graph G' und Template T

2.6 Subgraph-Counting-Problem

Bei dem Subgraph-Counting-Problem wird nach der Anzahl der Isomorphismen eines Subgraphen T in einem Graphen G gesucht. Subgraphen werden in Algorithmen, die dieses Problem lösen häufig Template genannt. Bei den meisten Algorithmen gibt es dabei gewisse Beschränkungen für die Templates, wie zum Beispiel, dass sie eine Baumstruktur besitzen müssen.

Da es vor allem für größere Templates und Graphen viel zu aufwändig ist, die genauen Anzahlen der Isomorphismen herauszufinden, wird in der Regel auf approximative Verfahren zurückgegriffen. Zusätzlich werden oft nur die nicht-induzierten Subgraphen gezählt, da das Zählen der induzierten Subgraphen deutlich aufwändiger ist.

Es gibt unterschiedliche Ansätze, auf denen Subgraph-Counting-Algorithmen basieren. Diese Arbeit fokussiert sich auf dem zuvor in 2.5 beschriebenen Colorcoding-Ansatz.

2.7 Subgraph-Enumeration-Problem

Das Subgraph-Enumeration-Problem befasst sich mit den Häufigkeiten der vorkommenden Subgraphen T der Größe k in einem Graphen G. Subgraphen werden in Subgraph-Enumeration-Algorithmen auch Graphlet genannt. Hierbei ist es nicht das Ziel, die genauen Anzahlen des Vorkommens dieser Graphlets herauszufinden. Viel wichtiger ist es zu erkennen, mit welcher Frequenz die Graphlets im Graphen auftauchen. Da es jedoch bei größeren k's zu sehr vielen möglichen Graphlets kommt, werden in den meisten Fällen lediglich die Graphlets mit den höchsten Häufigkeiten ausgewählt.

Auch beim Subgraph-Enumeration-Problem gibt es unterschiedliche Ansätze, auf denen die Algorithmen basieren. Diese Arbeit beschäftigt sich mit dem Colorcoding-Ansatz.

3 Algorithmen

Dieses Kapitel beschreibt die beiden Algorithmen PARSE und CC. Diese beiden Algorithmen bilden gemeinsam die Basis für DistCC. Es wird auf den Ablauf und die Details eingegangen. Zudem werden bestimmte Aspekte der Algorithmen miteinander verglichen. Zunächst folgt in 3.1 die Vorstellung vom Subgraph-Counting-Algorithmus PARSE. Anschließend wird in 3.2 der Subgraph-Enumeration-Algorithmus CC beschrieben.

3.1 PARSE

PARSE [ZKKM10] ist ein paralleler und verteilter Subgraph-Counting-Algorithmus. Als Eingabe empfängt der Algorithmus einen Graphen und ein Template, für welches die nicht-induzierten Isomorphismen gezählt werden sollen. Diese Anzahl wird dann schließlich ausgegeben. Dazu werden die Maschinen in einen Master und mehrere Worker unterteilt. Der Graph wird in mehreren Iterationen entsprechend dem Colorcoding-Verfahren gefärbt. Anschließend werden die farbigen Einbettungen des Templates mittels eines Backtracking-Verfahrens von den Workern gezählt. Die Anzahlen werden an den Master gesendet, damit dieser zum Schluss aus diesen das Gesamtergebnis berechnen kann.

3.1.1 Templates

Anders als bei vielen anderen Subgraph-Counting-Algorithmen, kann PARSE die Anzahlen der Isomorphismen auch für Templates approximieren, die nicht zwingend eine Baumstruktur besitzen müssen. Es genügt, wenn eine sogenannte *Cut-Edge* existiert. Dabei handelt es sich um eine Kante, die die beiden Subtemplates T_1 und T_2 vom Template T miteinander verbindet. T_1 und T_2 können beliebige Graphen sein. Die einzige Kante, die diese beiden Subtemplates miteinander verbindet darf die *Cut-Edge* sein. Diese Strukturklasse schließt die Klasse der Templates mit Baumstrukturen mit ein, da jede Kante in einem Graphen mit Baumstruktur eine potentielle *Cut-Edge* sein kann. Zur Veranschaulichung werden in Abbildung 3.1 Beispielgraphen dieser Templateklasse dargestellt.

3.1.2 Farbenmengen

Wie bereits in Kapitel 1 beschrieben, wird bei der Colorcoding-Technik jedem Knoten des Graphen eine bestimmte Farbe zugewiesen. Eine Farbe c ist eine natürliche Zahl die größer als 0 und kleiner als die Templategröße k ist. Um gefärbte Isomorphismen im Graphen zu finden werden Farbenmengen verwendet. Eine Farbenmenge enthält für jeden Knoten eines Templates eine einzigartige Farbe, denn es handelt sich nur dann um einen vollständig gefärbten Isomorphismus, wenn alle Farben des gefundenen Isomorphismus' einzigartig sind. Somit wird eine Farbenmenge im

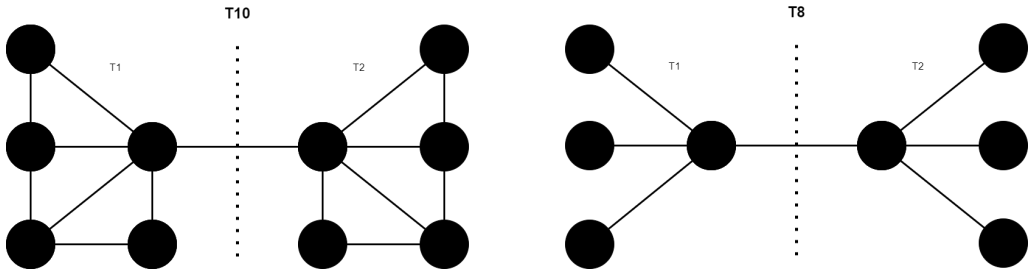

Abbildung 3.1: Unterteilung der Templates in T_1 und T_2

Algorithmus von einer Menge von natürlichen Zahlen repräsentiert. Da sich das Template T in die beiden Subtemplates T_1 und T_2 unterteilen lässt, müssen auch für diese Subtemplates Farbenmengen generiert werden. Um die Isomorphismen der beiden Subtemplates zusammenzuführen, damit ein Isomorphismus für T gefunden werden kann, müssen die Farbenmengen vereinigt und geschnitten werden. Dies sind im Grunde einfache Schnitt- und Vereinigungsoperationen für Zahlenmengen.

3.1.3 Verteilung

In PARSE werden die Maschinen in eine Master-Maschine und mehrere Worker-Maschinen unterteilt. Jedem Worker wird eine Teilmenge der Knoten im Graphen zugewiesen. Die Teilmengen aller Worker sind zueinander disjunkt, die Vereinigung aller Teilmengen ist die Menge der Knoten im Graphen. Die Aufgabe des Workers ist es, die Anzahlen der Isomorphismen des Templates in seiner eigenen Teilmenge zu zählen, und anschließend diese an den Master zu senden. Die Aufgabe des Masters ist es, von jedem Worker eine Nachricht mit den entsprechenden Anzahlen zu empfangen und diese dann zu einer Gesamtanzahl zusammenzufassen. Die Worker müssen untereinander nicht kommunizieren, da die Teilmenge der Knoten, die einem Worker zugewiesen wird, alle für das Zählen der Isomorphismen notwendigen Knoten enthält.

3.1.4 Graphpartitionierung

Jeder Worker muss die Anzahlen der Isomorphismen für eine Teilmenge der Knoten des Graphen berechnen. Es ist also unnötig, den gesamten Graphen in den Arbeitsspeicher der Worker zu laden. Deshalb wird der Graph partitioniert. In Algorithmus 3.1 wird beschrieben, wie das gemacht wird. Die Partitionierung wird von jedem Worker für sich vorgenommen. Jede Partition besitzt einen Kern. Dieser Kern wird *nCore* genannt. *nCore* ist der Quotient der Gesamtknotenanzahl und der Anzahl der Worker. Die Kerne der Partitionen sind also gleich groß. Zur Identifikation seines eigenen Kerns iteriert ein Worker über alle Zeilen bzw. Knoten der Adjazenzliste des Graphen. Abhängend von seiner $WorkerID = p$ bestimmt er den ersten Knoten seines Kerns. Die darauf folgenden $nCore - 1$ Knoten fügt er ebenfalls dem Kern seiner Partition hinzu. Nachdem der Kern einer Partition gebildet wurde, wird der Partition seine r-Nachbarschaft hinzugefügt. In der r-Nachbarschaf eines Kerns liegen alle Knoten, die von jedem Knoten des Kerns in r Schritten erreicht werden können. Somit bildet sich eine Hülle um den Kern herum, dessen Durchmesser r ist.

r muss dabei mindestens so groß wie der größte Radius der beiden Subtemplates sein, damit auch alle Knoten des Templates vom Kern heraus bei der Suche nach einem Isomorphismus erreicht werden können. Somit besitzt die Partition alle Knoten, die für das Zählen der Isomorphismen in dieser Partition notwendig sind. Einem Worker können in PARSE mehrere Partitionen zugewiesen werden. Dies ist dann sinnvoll, wenn eine der Maschinen deutlich rechen-stärker als der Rest ist. Falls jedoch alle Maschinen sich in der selben Rechenklasse befinden, ist eine Partition pro Worker der Idealfall. Für den Rest dieses Kapitels wird angenommen, dass jedem Worker lediglich eine Partition zugewiesen wird.

Algorithmus 3.1 Partitionierung des Graphen [ZKKM10]

GENERATEPARTITION$(G(V, E), p, r, P)$

1: $nCore \leftarrow \frac{n}{p}$
2: Skip $nCore \times (p-1)$ nodes
3: $V_p \leftarrow$ the set of next $nCore$ nodes
4: $E_p \leftarrow \emptyset$
5: **for** i from 1 to r **do**
6: **for** each edges (v, u) in G **do**
7: **if** $v \in V_P$ and $u \notin V_p$ **then**
8: $V_p \leftarrow V_p \cup \{u\}$
9: **end if**
10: **end for**
11: **end for**
12: **for** each edges (v, u) in G **do**
13: **if** $v, u \in V_p$ **then**
14: add (v, u) to E_p
15: **end if**
16: **end for**
17: **return** $G_p(V_p, E_p)$

3.1.5 Zählung der Templates

Die Templates werden in den inneren Schleifen des Algorithmus' gezählt. Dazu kommen die beiden Funktionen **COUNTTEMPLATE** (Algorithmus 3.2) und **MATCHSUBGRAPH** (Algorithmus 3.3) zum Einsatz.

COUNTTEMPLATE

Zunächst wird die BFS-Reihe des Templates T bestimmt. Anschließend wird über alle Knoten v der Partition V_p und alle Permutationen S' der Farbenmenge S iteriert. Falls der erste Knoten u_1 der BFS-Reihe dieselbe Farbe in S' hat, wie der Knoten v in der Partition, bedeutet das, dass für den Knoten u_1 des Templates T ein passender Knoten v in der Partition gefunden wurde. In diesem Fall können nun die restlichen Knoten T's betrachtet werden. Dazu wird die Methode **MATCHSUBGRAPH** mit der entsprechenden Permutation S' und dem als nächstes zu betrachtenden BFS-Reihenindex 2 aufgerufen.

Algorithmus 3.2 Zählung der Templates in jeder Partition [ZKKM10]

COUNTTEMPLATE$(T(V^T, E^T), G_p(V_p, E_p), S)$

1: Perform a breadth-first search (BFS) in T. Let $u1, u2, ...$ be the nodes in T in the order of their discovery in BFS.
2: **for** each node $v \in V_p$ **do**
3: **for** each permutation S' of S **do**
4: **if** MATCHVERTEX(u_1, v, S') **then**
5: MATCHSUBGRAPH$(2, S')$
6: **end if**
7: **end for**
8: **end for**
9: α = number of automorphisms of T with root $\rho(T)$
10: **return** $count/\alpha$

MATCHSUBGRAPH

Algorithmus 3.3 Erkennung eines Isomorphismus' [ZKKM10]

MATCHSUBGRAPH(i, S')

1: **if** $i = |V^T| + 1$ **then**
2: $count \leftarrow count + 1$
3: **return**
4: **end if**
5: Find M = $\{f(u_k) | k < i \wedge (u_i, u_k) \in E^T\}$, where $f(u_k)$ denotes the node in the main graph that u_k has been mapped to.
6: Compute $K = \cap_{v \in M} N(v)$
7: **for** each $v \in K$ **do**
8: **if** MATCHVERTEX(u_i, v, S') **then**
9: MATCHSUBGRAPH$(i + 1, S')$
10: **end if**
11: **end for**
12: **return**

In **MATCHSUBGRAPH** werden alle farbigen Subgraphen gesucht. Zu diesem Zweck bedient sich diese Funktion einem Backtracking-Verfahren, der alle Möglichkeiten durchprobiert. Noch bevor diese Funktion aufgerufen wird, wurde in **COUNTTEMPLATE** bereits der erste passende Knoten $v \in V_p$ für $u_1 \in T$ gefunden. Dieser befindet sich in der Menge M, die alle Knoten beinhaltet, für die bereits passende Knoten gefunden wurden. Nun müssen auch für die restlichen Knoten des Templates der BFS-Reihe nach passende Konten aus der Partition gefunden werden. Aus diesem Grund werden alle Kandidatsknoten der Partition in der Menge K zusammengefasst. Ein Knoten v' ist ein Kandidatsknoten, falls $v' \in N(v)$ und $v' \notin M$. Daraufhin wird über diese Knoten in K iteriert. Falls ein passender Knoten gefunden wird, wird **MATCHSUBGRAPH** erneut mit dem nächsten BFS-Reihenindex aufgerufen.

Nachdem für alle Knoten aus dem Template ein passender Knoten aus der Partition gefunden wurde, wird die Variable *count* um 1 inkrementiert und die Backtracking-Rekursion terminiert.

Algorithmus 3.4 PARSE($(T(V^T, E^T), G_p(V, E), \epsilon, \delta)$ [ZKKM10]

for $p \leftarrow me - 1; p < P; p \leftarrow p + Q - 1$ **do**
  $G_p = GeneratePartition(G(V, E), p, r, P)$ {me is the current processors ID}
end for
for j from 1 to N_1 **do**
  for i from 1 to N_2 **do**
    if $me = 0$ **then**
      Color(G(V, E), S) {randomize coloring the original graph}
    end if
    barrier {synchronizing}
    if $me > 1$ **then**
      LoadTemplate(T)
      for $p \leftarrow me - 1; p < P; p \leftarrow p + Q - 1$ **do**
        LoadPartition(G_p)
        for each vertex $v \in core(G_p)$ **do**
          for each $S_1^\pi \cup S_2^\pi = S, S_1^\pi \cap S_2^\pi = \emptyset, \pi \in [1, \binom{k}{k_1}]$ **do**
            $c_1 \leftarrow COUNTTEMPLATE(v, T_1, S_1^\pi)$
            $c_2 \leftarrow COUNTTEMPLATE(v, T_2, S_2^\pi)$
            $val[\pi] \leftarrow (c_1, c_2)$
          end for
          Send$(val, 0)$ {send to processor 0}
        end for
      end for
    end if
    if $me = 0$ **then**
      $Count \leftarrow 0$
      for each $v \in G$ **do**
        Recv(val)
        $valMatrix[v] \leftarrow v$
      end for
      for each $v \in G$ **do**
        for each $u \in N(v)$ **do**
          if u > v **then**
            for π from 1 to $\binom{k}{k_1}$ **do**
              $Count \leftarrow Count + valMatrix[v][\pi].c1 \times valMatrix[u][\pi].c2$
              $Count \leftarrow Count + valMatrix[v][\pi].c2 \times valMatrix[u][\pi].c1$
            end for
          end if
        end for
      end for
      $Count \leftarrow Count/\beta$
      $Count \leftarrow Count \times \frac{k^k}{k!}$
      $X_i = Count$
    end if
  end for
  if $me = 0$ **then**
    $Y_j \leftarrow \sum_i X_i / N_2$
  end if
end for
Output the median of $\{Y_1, ..., Y_{N_1}\}$

3.1.6 Der Algorithmus

Algorithmus 3.4 ist eine genaue Beschreibung von PARSE. Zu Beginn werden in den ersten beiden Zeilen die Partitionen von den Worker gebildet. Daraufhin werden in jeder der $N_1 \times N_2$ Iterationen jedem Knoten eine zufällige Farbe vom Master zugewiesen. N_1 und N_2 sind abhängig von den beiden Inputvariablen ϵ und δ, welche den Genauigkeitsgrad und die Fehlerquote der Ergebnisse bestimmen. Nach der Färbung der Knoten beginnt das eigentliche Zählen der Subtemplates von den Workern. Dazu iteriert jeder Worker über die Kernknoten $v \in core(G_p)$ seiner Partition und zählt die Isomorphismen für alle möglichen Permutationen der Farbenmenge S, welche in die beiden Farbenmengen S_1 und S_2 aufgeteilt wird. S_1 und S_2 sind dabei die Farbenmengen der beiden Subtemplates T_1 und T_2. Für beide dieser Subtemplates werden die Anzahlen c_1 und c_2 nacheinander berechnet und abgespeichert. Nachdem über alle Permutationen iteriert wurde und die Anzahlen abgespeichert wurden, können diese in Zeile 20 für den jeweiligen Knoten v an den Master gesendet werden. Diese werden dann in Zeile 27 vom Master empfangen. Sobald der Master die Anzahlen für jeden Knoten aus dem gesamten Graphen empfängt, beginnt er in Zeile 30 mit dem Zusammensetzen der Teillösungen zu einer Gesamtlösung. Dazu iteriert er über alle Kanten (v, u) im Graphen und kombiniert die Anzahlen für T_1 an v mit den Anzahlen für $T2$ an u und umgekehrt. Die Kante (v, u) stellt dabei die weiter oben beschriebene *Cut-Edge* dar. Schließlich wird die finale Anzahl durch die Anzahl der Automorphismen β geteilt und mit $\frac{k^k}{k!}$ multipliziert, welche die Wahrscheinlichkeit ist, dass ein Isomorphismus gefärbt ist. Somit wird in jeder der N_2 Iterationen eine Anzahl berechnet. Zum Schluss wird dann der Median dieser ganzen Anzahlen als finale Lösung ausgegeben. Der Median ist dabei die approximierte Anzahl der Isomorphismen in dem Graphen G für das Template T.

3.2 CC

CC [BCK+17][BCK+18] ist ein paralleler Subgraph-Enumeration-Algorithmus. Das Ziel CC's ist es, für einen Graphen und eine ganze Zahl k, die beide als Eingabe übergeben werden, die Häufigkeiten der am häufigsten vorkommenden Graphlets der Größe k auszugeben. Graphlets sind induzierte Subgraphen, für die es keine Strukturbeschränkungen gibt. Der Algorithmus setzt sich aus zwei Phasen zusammen. Die Building Phase und die Sampling Phase. In der Building Phase werden die gefärbten Einbettungen sogenannter Treelets gezählt. Treelets sind Subgraphen, die eine Baumstruktur besitzen. Auch hier kommt das Colorcoding-Verfahren zum Einsatz. Da CC sich aber nicht für die genauen Anzahlen der Graphlets interessiert, sondern nur die Häufigkeiten berechnen möchte, genügt bei CC im Gegensatz zu Subgraph-Counting-Algorithmen eine Colorcoding-Iteration. Der Graph wird also lediglich einmal gefärbt und die gefärbten Einbettungen werden einmal gezählt. Darafhin folgt die Sampling Phase. Hier werden aus den während der Building Phase gezählten Treelets die entsprechenden Graphlets gebildet und die Häufigkeiten berechnet.

3.2.1 Graphlets und Treelets

In CC werden zunächst in der Building Phase die nicht-induzierten Isomorphismen der Treelets gezählt. Diese sind Subgraphen, die eine Baumstruktur besitzen. Jedes Treelet bildet somit den Spannbaum eines Graphlets. Aus einem gefundenen nicht-induzierten Isomorphismus eines Treelets

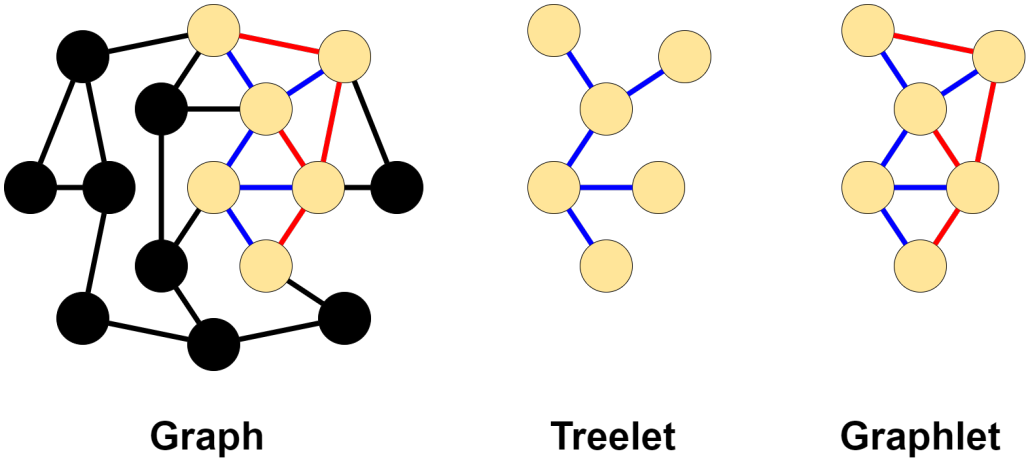

Abbildung 3.2: Die gelben Koten bilden zusammen mit den blauen Kanten einen Isomorphismus des Treelets. Indem die roten Kanten zwischen den gelben Knoten hinzugefügt werden, erhält man das Graphlet.

kann dann in der Sampling Phase ein induziertes Graphlet gebildet werden, indem alle Kanten, die sich zwischen den Knoten des Isomorphismus befinden, zu den bereits in dem Isomorphismus vorhandenen Kanten hinzugefügt werden. Somit kann CC, anders als viele andere Subgraph-Enumeration-Algorithmen [BCK+17] [SM13a] [SM14] die Häufigkeiten für induzierte Graphlets berechnen. In Abbildung 3.2 wird dies veranschaulicht.

3.2.2 Farbenmengen

Wie beim PARSE-Algorithmus verwendet CC Farbenmengen. Diese werden benötigt, um den Knoten der Treelets Farben zuzuweisen, damit dann entsprechende Isomorphismen im Graphen erkannt werden können. Dabei wird jedem Knoten eines Treelets eine einzigartige Farbe c zugewiesen, mit $0 < c \leq k$. Die Implementierung der Farbenmengen müssen beim Colorcoding-Verfahren die Schnittmengen- und Vereinigungsoperatoren für Mengen unterstützen. Anders als bei PARSE handelt es sich bei CC jedoch nicht um Mengen von ganzen Zahlen. Die einzelnen Farben werden also nicht von ganzen Zahlen repräsentiert. Stattdessen stellen einzelne Bits einer ganzen Zahl die einzelnen Farben einer Farbenmenge dar, und die ganze Zahl somit die Farbenmenge selbst. Sei beispielsweise $k = 5$ und die Farbenmenge $F = \{1, 2, 5\}$. Die entsprechende Repräsentation in Form einer ganzen Zahl besitzt die Bitfolge 10011 welche gleich 19 ist. Zum Vereinigen zweier Farbenmengen kommt ein Bitweises-Oder zum Einsatz. Zur Berechnung der Schnittmenge wird ein Bitweises-Und verwendet. Dies führt insgesamt zu einem viel effizienteren Speicherverbrauch.

3.2.3 Der Algorithmus

Der Algorithmus setzt sich aus der Building Phase und der Sampling Phase zusammen. Die Treeltes T werden in diesem Algorithmus mit den Farbenmengen C verknüpft. Daher besitzt jedes Treelet zusätzlich zur Struktur der Knoten und Kanten eine Farbenmenge. Für jede Farbenmenge einer

bestimmten Größe muss also ein eigenes Treelet erstellt werden, wobei die Strukturen zweier Treelets gleich sein können, aber die Farbenmengen unterschiedlich sein müssen. Sei beispielsweise $|T| = 4$. In einem ungerichteten Graphen existieren zwei mögliche Strukturen für ein Treelet der Größe 4. Ein Stern und ein Pfad. Es gibt jedoch 24 verschiedene Farbenmengen, mit denen sich die Knoten eines solche Treelets färben lassen. Demnach müssen $2(AnzahlderStrukturen) * 24(AnzahlderFarbenmengen) = 48$ verschiedene Treelet-Farbenmengen Kombinationen T_C gebildet werden. Solche T_C werden im Rest dieser Arbeit Treelet genannt.

In der Building Phase werden für jede Kombination von einem gefärbten Treelet T_C und einem Knoten v der Wert $C(T_C, v)$ berechnet. $C(T_C, v)$ ist die Count-Tabelle, die für jedes T_C und v die Anzahl der nicht-induzierten Isomorphismen T_C's für die Wurzel v und der Farbenmenge C angibt. Anschließend werden in der Sampling Phase die Treelets die am häufigsten vorkommen identifiziert und die entsprechenden Graphlets gebildet.

Building Phase

Algorithmus 3.5 Building Phase [BCK+18]

1: input graph G, graphlet size k
2: **for** v in G **do**
3: $c(v)$ = color drawn u.a.r. from $[k]$
4: $T_C(c(v))$ = trivial treelet with the according color of v
5: $C(T_C, v) = 1$
6: **end for**
7: **for** $h = 2$ to k **do**
8: **for** v in G **do**
9: **for** each $T_C : |T_C| = h$ **do**
10: $C(T_C, v) = d^{-1} \sum_{(v,u)\in E} \sum_{C', C'' \subset C \wedge C' \cup C'' = \emptyset} C(T_{C'}, v) * C(T_{C''}, u)$
11: **end for**
12: **end for**
13: **end for**

In den ersten drei Zeilen des Algorithmus 3.5 wird zunächst der Graph gefärbt. Jedem Knoten wird eine zufällige Farbe zwischen 1 und k zugewiesen. In der Count-Tabelle werden für jeden Knoten die Anzahlen der entsprechenden trivialen Treelets, also den Treelets mit derselben Farbe wie v, auf 1 gesetzt.

Darauf folgend werden mittels Dynamischem Programmieren die Anzahlen größerer Treelets der Reihe nach berechnet und in die Count-Tabelle eingetragen. Dazu wird in der achten Zeile über jeden Knoten v des Graphen iteriert. In Zeile 9 wird dann über alle möglichen Treelets T_C der Größe h iteriert. In Zeile 10 des Algorithmus gelten folgende zwei Tatsachen:

(i) Die Einträge der Count-Tabelle für alle Treelets die kleiner als h sind wurden bereits berechnet.

(ii) Da es sich bei den Treelets zudem um Subgraphen mit Baumstrukturen handelt, können diese in zwei Subtreelets $T_{C'}$ und $T_{C''}$ aufgespalten werden, indem eine Kante aus T_C entfernt wird. Dabei muss gelten, dass $C' \cup C'' = C$ und $C' \cap C'' = \emptyset$. Die Wurzel $T_{C'}$' s entspricht dabei der Wurzel T_C's. Die Wurzel $T_{C''}$'s ist ein Nachbar u von v.

Aus (i) und (ii) folgt, dass sich die Anzahl der Isomorphismen für T_C somit aus den Anzahlen der Subtreelets zusammensetzen kann. Dazu wird über jeden Nachbar u von v und alle Aufspaltungen der Farbenmenge C in C' und C'' iteriert, damit alle Subtreelets mit den Wurzeln v und u gebildet werden können. Die Summe der Produkte der Anzahlen für jede Kombination solcher Subtreelets $T_{C'}$ und $T_{C''}$ ergibt die Anzahl der Isomorphismen T_C's für v.

$$C(T_C, v) = d^{-1} \sum_{(v,u)\in E} \sum_{C', C'' \subset C \wedge C' \cup C'' = \emptyset} C(T_{C'}, v) * C(T_{C''}, u)$$

Die Korrektheit und Vollständigkeit dieser Formel wurde in [AYZ95] gezeigt. d ist dabei die Anzahl der Automorphismen von $T_{C''}$ unter allen Nachbarn u von v.

Sampling Phase

Sobald die Building Phase abgeschlossen ist und somit alle Werte der Count-Tabelle berechnet wurden, beginnt die Sampling Phase. Zu Beginn der Sampling Phase wird zufällig ein Knoten v mit Wahrscheinlichkeit proportional zu der Gesamtanzahl der Isomorphismen für Treelets der Größe k am Knoten v ausgewählt. Je größer $\sum_{T_C:|T_C|=k} C(T_C, v)$, desto größer ist also die Wahrscheinlichkeit, dass v ausgewählt wird. Anschließend wird zufällig ein Treelet T_C proportional zu der Anzahl seiner Isomorphismen am Knoten v ausgewählt.

Nun müssen die entsprechenden Knoten des Graphen gesamplet werden. An dieser Stelle ist nur bekannt, dass v einer dieser Knoten ist, und die restlichen Knoten per Isomorphismus auf die Knoten T_C's abgebildet werden können. Deshalb wird nun T_C in die beiden Subtreelets $T_{C'}$ und $T_{C''}$ aufgespalten. Somit kann der nächste Knoten, welcher ein Nachbar u des Knoten v's ist gefunden werden. Die Anzahlen der Isomorphismen für alle Subtreelets und alle Wurzeln u und v sind in der Count-Tabelle enthalten. u wird daher proportional zu der Gesamtanzahl seiner Isomorphismen ausgesucht und gesamplet. Um alle weiteren Knoten auch finden zu können wiederholt sich dieser Prozess rekursiv, bis es sich bei den Subtreelets um die trivialen Treelets handelt.

Wenn alle Knoten des Treelets gefunden wurden, kann das induzierte Graphlet $G'(V', E')$ mit $V' \subset V$ und $(v, u) \in E' : v, u \in V' \wedge (v, u) \in E$ gebildet werden. Ein Treelet ist nämlich der Spannbaum eines oder mehrerer Graphlets. Um einen Graphlet aus einem Treelet zu generieren müssen also die restlichen Kanten zwischen den Knoten des Treelets im Graphen zum Graphlet hinzugefügt werden.

Dies kann beliebig oft wiederholt werden, um mehrere Graphlets zu samplen.

3.2.4 Parallelisierung und Implementierungsdetails

Die Zeile 8 des Algorithmus 3.5 wird parallel ausgeführt. Jeder Thread berechnet die Anzahlen der Isomorphismen für eine Teilmenge der Knoten. Der Algorithmus wurde in der Programmiersprache JAVA implementiert. Um die Knoten unter den Threads aufzuteilen, wurde die Klasse $AtomicInteger$ verwendet. Ein Thread der mit der Berechnung eines Knotens fertig ist und sich den nächsten Knoten holen möchte ruft die $getAndIncrement()$-Methode dieser Klasse auf. Bei dieser Methode handelt es sich um eine blockierende Methode. Falls also zwei oder mehrere Threads gleichzeitig den nächsten Knoten zur Berechnung anfordern müssen sie aufeinander warten.

4 DistCC

In dem vorangegangenen Kapitel wurden die beiden Algorithmen PARSE [ZKKM10] und CC [BCK+17] [BCK+18] präsentiert. In diesem Kapitel wird DistCC, welches eine verteilte Version CC's ist vorgestellt. Die Verteilung beschränkt sich auf die Building Phase CC's. Die Verteilung wird ähnlich umgesetzt wie bei PARSE. Auch bei der Graphpartitionierung gibt es Ähnlichkeiten zu PARSE. Zusätzlich zu der Verteilung CC's werden in diesem Kapitel weitere Optimierungen des Algorithmus' diskutiert. Es wird hauptsächlich auf verschiedene Sortierungen des Graphen eingegangen, die teilweise zu einer Verbesserung der Laufzeiten bei CC führen.

4.1 Verteilung CC's

Bei PARSE werden die Maschinen in einen Master und mehrere Worker unterteilt. Den Workern wird jeweils eine Partition des Graphen zugeteilt. Für diese Partition berechnen dann die Worker die Anzahlen der Isomorphismen des Templates und senden diese anschließend an den Master. Der Master fasst alle Anzahlen zu einer Gesamtanzahl zusammen und gibt das Ergebnis schließlich aus.

4.1.1 Master und Worker bei DistCC

Eine ähnliche Verteilung ist auch in CC möglich. Deshalb werden die Maschinen auch bei DistCC in einen Master und mehrere Worker unterteilt. Die Aufgaben des Masters und der Worker unterscheiden sich hier jedoch nicht so sehr voneinander wie bei PARSE. Die Berechnung der Anzahlen der Isomorphismen wird in DistCC sowohl vom Master als auch von den Workern durchgeführt. Der wesentliche Unterschied zwischen dem Master und den Workern bei DistCC ist, dass die Worker lediglich die Building Phase ausführen, wohingegen der Master den Algorithmus anschließend mit der Sampling Phase fortsetzt und die finale Ausgabe produziert.

4.1.2 Verteilungsdetails

Auch bei DistCC wird der Graph partitioniert. Jeder Maschine wird eine Partition zugewiesen. Zunächst färbt jede Maschine seine eigene Partition und trägt die entsprechenden Werte in die Count-Tabelle ein. Dies entspricht den ersten 5 Zeilen des Algorithmus 4.1. Anschließend wird in Zeile 7 die eigene Count-Tabelle an alle anderen Maschinen gesendet, und die Count-Tabellen jeder anderen Maschine empfangen. In Zeile 11 werden dann die Anzahlen der Isomorphismen für alle Treelets der Größe $h = 2$ berechnet. Diese werden daraufhin in Zeile 14 unter allen Maschinen ausgetauscht, da die Berechnung in Zeile 11 für $h = 3$ von diesen Anzahlen abhängig ist. Dies ist notwendig, da der Algorithmus über alle Nachbarn u von v iteriert.

Algorithmus 4.1 Building Phase von DistCC

1: input partition G_p, graphlet size k
2: **for** v in G_p **do**
3: $c(v)$ = color drawn u.a.r. from $[k]$
4: $T_C(c(v))$ = trivial treelet with the according color of v
5: $C(T_C, v) = 1$
6: **end for**
7: sendAndReceive(C)
8: **for** $h = 2$ to k **do**
9: **for** v in G_p **do**
10: **for** each $T_C : |T_C| = h$ **do**
11: $C(T_C, v) = d^{-1} \sum_{(v,u)\in E} \sum_{C', C'' \subset C \wedge C' \cup C'' = \emptyset} C(T_{C'}, v) * C(T_{C''}, u)$
12: **end for**
13: **end for**
14: sendAndReceive(C)
15: **end for**

So werden für jedes $h < k$ die Anzahlen berechnet und ausgetauscht. Für $h = k$ müssen die Anzahlen aber nicht unter allen Maschinen ausgetauscht werden, da die Building Phase hier endet. Nur der Master muss die Anzahlen empfangen, damit er seine Berechnung mit der Sampling Phase fortsetzen kann. Deshalb sendet in Zeile 14 für $h = k$ jeder Worker seine Count-Tabelle nur an den Master und beendet somit seine eigene Berechnung.

4.1.3 Nachrichtenaustausch

Die Funktion $sendAndReceive(C)$, die in den beiden Zeilen 7 und 14 aufgerufen wird, sorgt dafür, dass die für die Berechnung der Anzahlen der nächsten Treeletgröße benötigten Anzahlen von jeder Maschine empfangen werden. Jede Maschine benötigt die Anzahlen der Knoten u, für die gilt $(v, u) \in E \wedge v \in G_p$. u kann also in einer anderen Partition liegen.

In der Count-Tabelle $C(T_C, v)$ befinden sich die Anzahlen der Isomorphismen für jedes Treelet T_C mit der Wurzel v. Eine Maschine trägt für jeden seiner Knoten die entsprechenden Anzahlen ein. Nicht jeder dieser Einträge bzw. Anzahlen für die Wurzel der Knoten dieser Maschine werden aber von der Empfängermaschine benötigt.

Eine Nachricht einer Maschine muss deshalb nicht die gesamte Count-Tabelle beinhalten. Es genügt und ist auch effizienter, wenn die Nachricht nur die Anzahlen für die Knoten u beinhalten, die auch in der Hülle der Partition der Empfängermaschine liegen. Eine Hülle einer Partition beinhaltet alle Knoten, die eine Kante zu mindestens einem der Knoten in der Partition haben. In Abbildung 4.1 wird dies veranschaulicht.

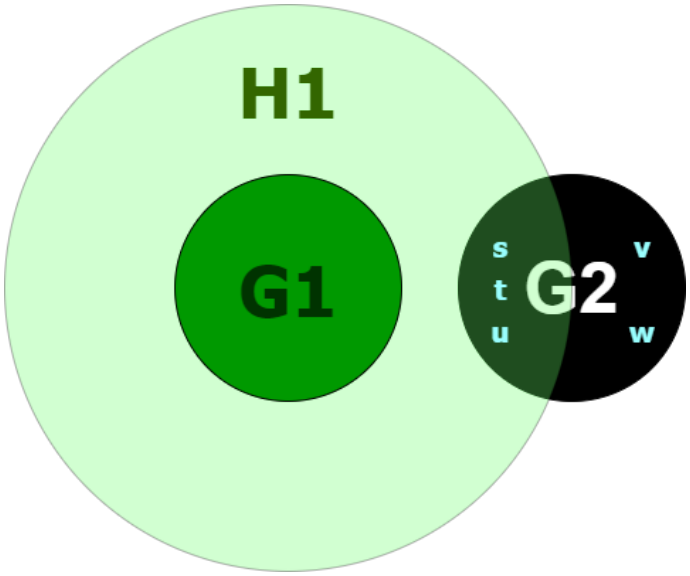

Abbildung 4.1: $G1$ und $G2$ sind zwei Partitionen. $H1$ ist die Hülle von $G1$. Die Knoten s, t und u liegen in der Partition $G2$ und auch in der Hülle $H1$. Die Knoten v und w liegen in der Partition $G2$, aber nicht in der Hülle $H1$. Die Maschine mit der Partition $G2$ muss also nur die Anzahlen für die drei Knoten s, t und u an die Maschine mit der Partition $G1$ senden. Die Knoten v und w werden nicht benötigt.

4.1.4 Partitionierung des Graphen

Die Partitionierung von PARSE (Algorithmus 3.1) lässt sich sehr einfach auf die DistCC übertragen. Es wird über die Knoten des Graphen iteriert. Zuerst werden die Kerne der Worker gebildet. Anschließend die Hüllen. Anders als bei PARSE, genügt es bei DistCC, wenn $r = 1$ ist, denn in Zeile 11 des Algorithmus' 4.1 werden nur die direkten Nachbarn des Kerns benötigt. Im Folgenden wird diese Partitionierung ParsePart genannt.

Diese Partitionierung weist jedoch einige Schwächen auf. Zum einen können die Hüllen sehr groß ausfallen. Dies führt dann zu großen Nachrichten, die ausgetauscht werden müssen und somit zu einer längeren Kommunikationszeit. Zum anderen können sich die Anzahlen der Kanten der Partitionen stark voneinander unterscheiden. Dies wirkt sich sehr nachteilig auf DistCC auf, da in Zeile 11 über jede Kante der Partition iteriert wird. Bei unterschiedlichen Kantenanzahlen der Partitionen werden die Maschinen der Partitionen mit weniger Kanten früher mit ihrer Berechnung für eine Treeletgröße fertig sein als die Maschinen der Partition mit mehr Kanten. Dies führt zu langen Wartezeiten zwischen den Berechnungen der verschiedenen Treeletgrößen.

Optimal wäre es also, wenn die Kantenanzahlen der Partition gleich groß sind und gleichzeitig der Schnitt aller Partition minimal ist. Im Rahmen dieser Arbeit wurde eine Heuristik für die Partitionierung entwickelt, die auf diesen Erkenntnissen beruht. Es wurde ein Partitionierungsverfahren namens RandPart entwickelt, das dafür sorgt, dass die Partitionen ungefähr gleich viele Knoten und Kanten haben. Dies beseitigt die eine Schwäche der zuvor genannten Partitionierung von PARSE. Zusätzlich wurde auch ein eine Partitionierung von Metis [KK97] getestet, die für einen guten Schnitt der Partitionen sorgt. Auch hierbei handelt es sich um eine Heuristik.

RandPart

Wie der Name es bereits verrät, werden bei RandPart die Partitionen zufällig erstellt. Das Ziel RandParts ist es, Partitionen zu produzieren, die ähnliche Knoten- und Kantenanzahlen haben. Zu diesem Zweck werden wie in den ersten vier Zeilen des Algorithmus' 4.2 eine obere und eine untere Schranke für den Partitionsgrad bestimmt. Die Abweichung d spielt dabei eine wichtige Rolle. Denn die Schranken sollten weder zu nah zueinander sein, noch zu weit voneinander entfernt sein.

Wenn sie zu nah zueinander sind, kann es sein, dass mindestens eine der Partitionen nicht gebildet werden kann, da der letzte Knoten der der Partition hinzugefügt werden soll immer einen zu hohen Grad haben kann und somit der Grad der Partition niemals innerhalb der beiden Schranken landen kann. Im Falle, dass die Schranken zu weit voneinander entfernt sind, unterscheiden sich die Kantenanzahlen der resultierenden Partitionen zu stark voneinander.

In der Praxis hat sich gezeigt, dass für Kantenanzahlen $m < 10^5 \rightarrow d \approx 7^{-4}$, für $m < 10^6 \rightarrow d \approx 2^{-4}$, für $m < 10^7 \rightarrow d \approx 7^{-5}$ und für $m < 10^8 \rightarrow d \approx 2^{-5}$ eine gute Abweichung bilden.

Algorithmus 4.2 RandPart

```
 1: input graph G(V, E), machine count mc, deviation d
 2: partitionDegree ← |E|/mc
 3: upperBound ← partitionDegree + d * partitionDegree
 4: lowerBound ← partitionDegree − d * partitionDegree
 5: for i = 1 to mc do
 6:     while deg(G_i) < lowerBound do
 7:         vertex = V.get(random)
 8:         if deg(G_i) + deg(vertex) < upperBound then
 9:             G_i ← G_i ∪ {vertex}
10:             V.remove(vertex)
11:         end if
12:     end while
13: end for
```

Anschließend wird über jede Partition iteriert. Einer Partition werden so lange zufällig neue Knoten hinzugefügt, bis sich der Grad der Partition zwischen der unteren und oberen Schranke befindet. Dies führt dazu, dass die Partitionen über sehr ähnliche Kantenanzahlen verfügen. Obwohl die Knotenanzahlen der Partitionen für die Laufzeiten DistCC's nicht sehr wichtig sind, sind auch diese in der Praxis sehr gleichmäßig verteilt.

4.2 Sortierung des Graphen

Die Knoten werden in der Zeile 9 von Algorithmus 4.1 und der Zeile 8 von Algorithmus 3.5 der Reihe nach abgearbeitet. Die Reihe der Knoten entspricht der Sortierung der Kantenliste. Diese Sortierung kann sich auf die Laufzeiten der Building Phase auswirken. Zwei Sortierungen, die sich negativ auf die Laufzeiten auswirken und zwei dazugehörige Verbesserungen werden vorgestellt.

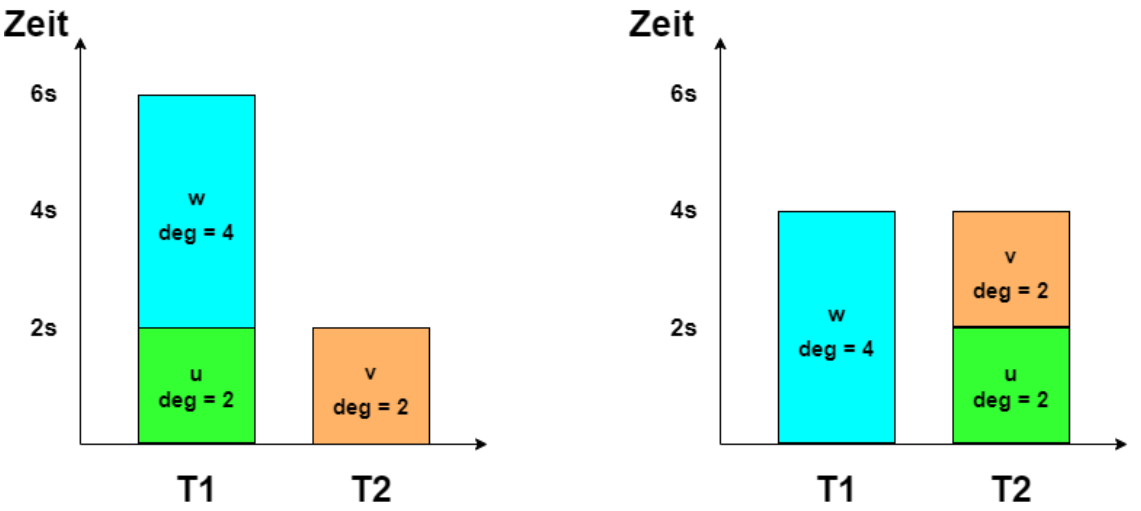

Abbildung 4.2: Zwei Diagramme, die die Laufzeiten bei zwei unterschiedlichen Sortierungen für zwei Threads darstellen. Links, die natürliche Sortierung. Rechts die SBD Sortierung.

4.2.1 Knoten mit hohem Grad

In der Zeile 11 von Algorithmus 4.1 wird über alle Nachbarn u des Knotens v iteriert. Je mehr Nachbarn v hat, desto länger benötigt der zuständige Thread für seine Berechnung. Falls sich ein Knoten mit sehr hohem Grad, also mit sehr vielen Nachbarn sehr weit hinten in der Kantenliste befindet, kann es passieren, dass alle Threads mit ihrer Berechnung fertig sind, bis auf den einen Thread, der für diesen hochgradigen Knoten zuständig ist.

Bildlich kann man sich das wie im linken Diagramm der Abbildung 4.2 dargestellt vorstellen. Hier sind die Knoten in ihrer natürlichen Ordnung sortiert, also $u \rightarrow v \rightarrow w$. Es wird angenommen, dass ein Knotengrad einer Rechensekunde entspricht. Dies führt dazu, dass der Thread $T1$ den Knoten u bekommt. $T2$ bekommt den Knoten v. Nach dem Abschluss der Berechnung für u bekommt $T1$ den Knoten w. Somit wird eine Gesamtlaufzeit von sechs Sekunden erreicht.

Im Gegensatz dazu sind im rechten Diagramm die Knoten nach ihrem Grad sortiert, also $w \rightarrow u \rightarrow v$. Man kann sehr gut erkennen, dass die Gesamtlaufzeit sich durch diese Sortierung um zwei Sekunden verbessert, denn hier sind die Threads gleichmäßig ausgelastet. Diese Sortierung wird im Rest dieser Arbeit SBD (sorted by degree) genannt.

Anhand von Experimenten wurde deutlich, dass sich die Laufzeiten CC's für manche Graphen mit dieser Sortierung im Vergleich zur natürlichen Sortierung verbessern. Bei Graphen, bei denen sich die Laufzeiten nicht verbesserten, waren keine signifikanten Veränderungen der Laufzeiten erkennbar. Somit führt diese Sortierung im besten Fall zu einer Verbesserung der Laufzeiten CC's und im schlechtesten Fall zu keiner signifikanten Veränderung.

Da die einzelnen Maschinen bei DistCC intern auch den CC Algorithmus ausführen, werden bei DistCC die Partitionen nach diesem Schema sortiert.

4.2.2 Aufeinanderfolgende Knoten mit niedrigem Grad

Bei Graphen mit sehr vielen niedriggradigen Knoten die aufeinander folgen kann die blockierende *getAndIncement*()-Methode aus Zeile 9 des Algorithmus 4.1 zu einer signifikanten Verzögerung führen. Die Threads sind in diesem Fall nämlich sehr schnell mit der Berechnung für einen Knoten fertig und fordern sofort den nächsten Knoten. Wenn mehrere Threads das ständig wiederholen, müssen sie auch ständig aufeinander warten. Um solche Blöcke von niedriggradigen Knoten in der Sortierung zu beseitigen werden in dieser Arbeit die Knoten zufällig gemischt. Dies führt zu einer insgesamt besseren Verteilung der Knoten innerhalb der Sortierung und wirkt dem Problem entgegen.

Außerdem wurde auch eine leicht modifizierte Version CC's getestet, bei der jedem Thread zu Beginn der Berechnung eine bestimmte Teilmenge der Knoten zugewiesen wird, sodass die Anzahl der Kanten pro Thread ungefähr gleich groß sind. In Zeile 10 von Algorithmus 3.5 wird also nicht mehr die *getAndIncement*()-Methode aufgerufen. Stattdessen wird der nächste Knoten aus der Teilmenge des Threads genommen und die Berechnung wird sofort fortgesetzt, ohne auf andere Threads warten zu müssen. Die durch die *getAndIncement*()-Methode auftretende Verzögerung wird somit eliminiert. Diese Version CC's wird im Rest dieser Arbeit NewCC genannt.

Die letzten Knoten in der SBD-Sortierung besitzen alle einen sehr niedrigen Grad. Diese führen beim Aufruf der *getAndIncement*()-Methode somit auch zu einer Verzögerung. In der Praxis hat sich jedoch gezeigt, dass solch eine Verzögerung in den meisten Fällen vernachlässigbar gering ausfällt. Nur bei Graphen mit sehr vielen Knoten (über fünf Millionen) und einem niedrigen Durchschnittsgrad pro Knoten (unter 10) war eine signifikante Verzögerung erkennbar. Da aber bei DistCC die Knoten unter den Maschinen aufgeteilt werden, reduziert sich somit die Anzahl der Knoten pro Maschine auf ungefähr einen Bruchteil der Gesamtanzahl. Somit führt die SBD-Sortierung bei den Partitionen DistCC's im besten Fall zu einer Verbesserung der Laufzeiten und im schlimmsten Fall zu keiner signifikanten Veränderung.

5 Experimente

Dieses Kapitel besteht aus Experimenten zu den unterschiedlichen Versionen von DistCC, welche mit CC und untereinander verglichen werden. Zunächst werden in 5.3 die Laufzeiten der verschiedenen Sortierungen für CC miteinander verglichen. In 5.4 wird dabei auch auf NewCC eingegangen und das Potenzial dieser kleinen Modifizierung CC's gezeigt. Daraufhin werden in 5.5 die Laufzeiten und der Speicherverbrauch DistCC's mit vier Maschinen mit den Laufzeiten CC's für dieselben Graphen und k's verglichen. In 5.6 werden die Kommunikationszeiten genauer analysiert. Anschließend wird 5.7 die Skalierbarkeit DistCC's vorgestellt, indem die Laufzeiten und der Speicherverbrauch mit vier, drei und zwei Maschinen miteinander verglichen werden. Zum Schluss werden dann in 5.8 die Stärken und Schwächen der unterschiedlichen Partitionierungsverfahren verglichen und diskutiert.

Bevor jedoch die Experimente vorgestellt werden, wird in 5.1 der Datensatz und in 5.2 die verwendete Hardware mit einigen Informationen zur Stabilität CC's präsentiert.

5.1 Datensatz

Für die Experimente wurden öffentlich zugängliche Graphen, die unterschiedliche Größenklassen repräsentieren und sich in ihrer Dichte unterscheiden ausgesucht. Diese werden in der Tabelle 5.1 mit den Knoten- und Kantenanzahlen aufgelistet. Des Weiteren werden auch die größte Graphletgröße k, die getestet werden konnte und der durchschnittliche Knotengrad angegeben.

Dabei wird k entweder durch einen zu hohen Speicherverbrauch, oder durch einen Überlauf der Zählvariable bei zu vielen Isomorphismen beschränkt.

Tabelle 5.1: Auflistung der Graphen mit Knotenanzahl, Kantenanzahl, dem größten möglichen k und dem Durchschnittsgrad eines Knotens

Graph	Knotenanzahl	Kantenanzahl	Größtes k	Durchschnittsgrad pro Knoten
WordAssociation	10,6K	63,8K	10	6,01
Facebook	63,4K	0,8M	9	12,61
Amazon	0,7M	3,5M	9	5,00
Berkstan	0,7M	6,6M	5	9,43
LiveJournal	5,4M	49,5M	6	9,12
Hollywood	1,9M	114,3M	5	60,16
Orkut	3,1M	223,5M	5	72,10

Tabelle 5.2: Darstellung der Standardabweichungen CC's für jeden Graphen und jedes k

Graph	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
WordAssociation	0.39	0.22	0.18	0.10	0.13	0.03
Facebook	0.21	0.13	0.08	0.09	0.02	-
Amazon	0.21	0.21	0.14	0.03	0.02	-
Berkstan	0.18	-	-	-	-	-
LiveJournal	0.03	0.05	-	-	-	-
Hollywood	0.04	0.03	-	-	-	-
Orkut	0.03	-	-	-	-	-

5.2 Verwendete Hardware und Stabilität CC's

Die Experimente wurden auf vier Maschinen durchgeführt. In jeder einzelnen dieser Maschinen sind 2 AMD Epic 7401 mit jeweils 24 Kernen verbaut. Die Kerne unterstützen HyperThreading. Somit sind 96 Threads pro Maschine möglich. Der verfügbare RAM beträgt 256 GB pro Maschine.

Laut [BCK+18] liegt die Standartabweichung der Laufzeiten bei 50 Durchläufen für alle Graphen und k's außer WordAssociation mit $k = 5$ und $k = 6$ unter 0.1. Die Tests, die im Rahmen dieser Arbeit durchgeführt wurden können dies nicht vollständig reproduzieren.

Um zu überprüfen, ob Hyper Threading die Ursache dafür ist, wurden mit dem Befehl **taskset -c 0-47** 48 Kerne für die Ausführung CC's festgelegt. Selbst bei solch einer Ausführung liegen die Werte für WordAssociation jedoch weiterhin höher als in [BCK+18]. Hier liegt die Standartabweichung jedoch nur sehr knapp über 0.1. Der Grund dafür kann die Architektur der Prozessoren oder der durch Hyper Threading generierte Overhead sein. Dies wurde nicht weiter untersucht und für weitere Forschung offen gelassen, da festgelegte Kerne mit 48 Threads dabei nur auf Kosten der Gesamtlaufzeiten möglich sind. Deshalb wurden trotz größerer Standartabweichungen immer 96 Threads mit HyperThreading verwendet. In Tabelle 5.2 werden die Standarabweichungen der einzelnen Graphen und k's für 96 Threads mit HyperThreading aufgelistet.

5.3 Einfluss der Sortierungen auf die Laufzeiten

Bei den meisten Graphen hatten die in Kapitel 4 beschriebenen Sortierungen keinen signifikanten Einfluss auf die Laufzeiten der Building Phase. Nur bei Wordassociation und LiveJournal konnten signifikante Verbesserungen erreicht werden. Bei WordAssociation erzielte die SBD Sortierung bessere Laufzeiten, bei LiveJournal die gemischte.

Für $k = 9$ und $k = 10$ benötigt CC bei 100 Durchläufen für WordAssociation durchschnittlich 140 und 1962 Sekunden. Bei der SBD Sortierung sinken diese Zeiten auf 125 und 1859 Sekunden. Für $k = 10$ führt die SBD Sortierung also zu einer Verbesserung von mehr als 100 Sekunden. Dass sich solch eine Verbesserung nur bei WordAssociation zeigt, liegt an der natürlichen Sortierung dieses Graphen. Der Graph enthält 10617 Knoten. Der Knoten an der 10451. Stelle der Sortierung hat einen Grad von 277. Der Durchschnittsgrad eines Knotens beträgt 6.01. Somit handelt es sich bei dem Knoten an der 10451. Stelle um einen der Knoten mit den höchsten Graden des Graphen. In

Zeile 10 von Algorithmus 3.5 muss also über 277 Nachbarn und alle disjunkten Teilmengen C' und C'' der Farbenmenge C iteriert werden, sodass $C' \cup C'' = C$ gilt. Der Thread, der für diesen Knoten zuständig ist, benötigt somit vor allem bei größeren k's sehr lange für seine Berechnung, sodass alle anderen Threads terminieren, bevor dieser Thread mit seiner Berechnung abgeschlossen ist. Dies führt dann schließlich zu einer signifikanten Verzögerung, welche durch die SBD Sortierung behoben wird, da dieser Knoten bei dieser Sortierung eine der ersten Stellen einnimmt.

Bei LiveJournal wird die Verzögerung der Laufzeit nicht durch einen Knoten mit sehr hohem Grad, der sich ganz hinten bei der natürlichen Sortierung befindet, verursacht. Denn obwohl LiveJournal der Graph mit den meisten Knoten, der getestet wurde ist, ist der Durchschnittsgrad eines Knotens der drittkleinste von allen Graphen. Es existieren demnach sehr viele Knoten mit geringem Grad in diesem Graphen. Da viele solcher Knoten in der natürlichen Sortierung diese Graphen aufeinanderfolgenden, sorgen sie dafür, dass die Threads bei dem Aufruf der $getAndIncrement()$-Methode lange aufeinander warten müssen. CC benötigt in der Building Phase für die natürliche Sortierung dieses Graphen 219 Sekunden bei $k = 5$ und 400 Sekunden bei $k = 6$. Bei der gemischten Sortierung sinken diese Zeiten auf 193 und 385 Sekunden. Hier wird die Wahrscheinlichkeit für solche Folgen von niedriggradigen Knoten und der damit verbundenen Verzögerung durch die $getAndIncrement()$-Methode gesenkt.

5.4 Potential NewCC's

Eine weitere Lösungsmöglichkeit für die Verzögerung durch aufeinanderfolgende Knoten mit niedrigem Grad ist NewCC. Hier wird jedem Thread zu beginn eine Teilmenge der Knoten zugeteilt, sodass die Anzahl der Kanten pro Thread ungefähr gleich ist. Somit ist der Aufruf der $getAndIncrement()$-Methode nicht mehr notwendig. Diese Modifizierung hat jedoch bei den meisten Graphen zu einer Verschlechterung der Laufzeiten geführt. Die Hauptursache hierfür ist, dass sich die Laufzeiten der Threads bedingt durch die Stabilität stark voneinander unterscheiden, trotz ungefähr gleicher Kanten- und Knotenanzahl. Dies sorgt für eine geringe Auslastung der Threads im Vergleich zur $getAndIncrement()$-Methode, bei der ja die Auslastung bei nahezu 100% liegt.

Trotzdem gibt es einen Graphen, für den NewCC deutlich bessere Laufzeiten als CC erzielt. Dieser ist der bereits wegen seiner hohen Anzahl von niedgriggradigen Knoten erwähnte LiveJournal, bei dem die $getAndIncrement()$-Methode eine signifikante Verzögerung verursacht. Die Laufzeit für $k = 5$ sinkt von 219 Sekunden auf 162 Sekunden und für $k = 6$ von 400 Sekunden auf 359 Sekunden. Bei einer stabileren Implementierung CC's sollten diese Zeiten sogar noch besser ausfallen und auch bei den anderen Graphen Verbesserungen erkennbar werden.

5.5 CC vs DistCC

Hier werden nun die Laufzeiten und der Speicherverbrauch von CC und DistCC miteinander verglichen. Für die Experimente wurden jeweils 100 Durchläufe für einen Graphen und jedes mögliche k mit CC und DistCC ausgeführt. Die folgenden Ergebnisse sind jeweils die Durchschnitte dieser 100 Durchläufe.

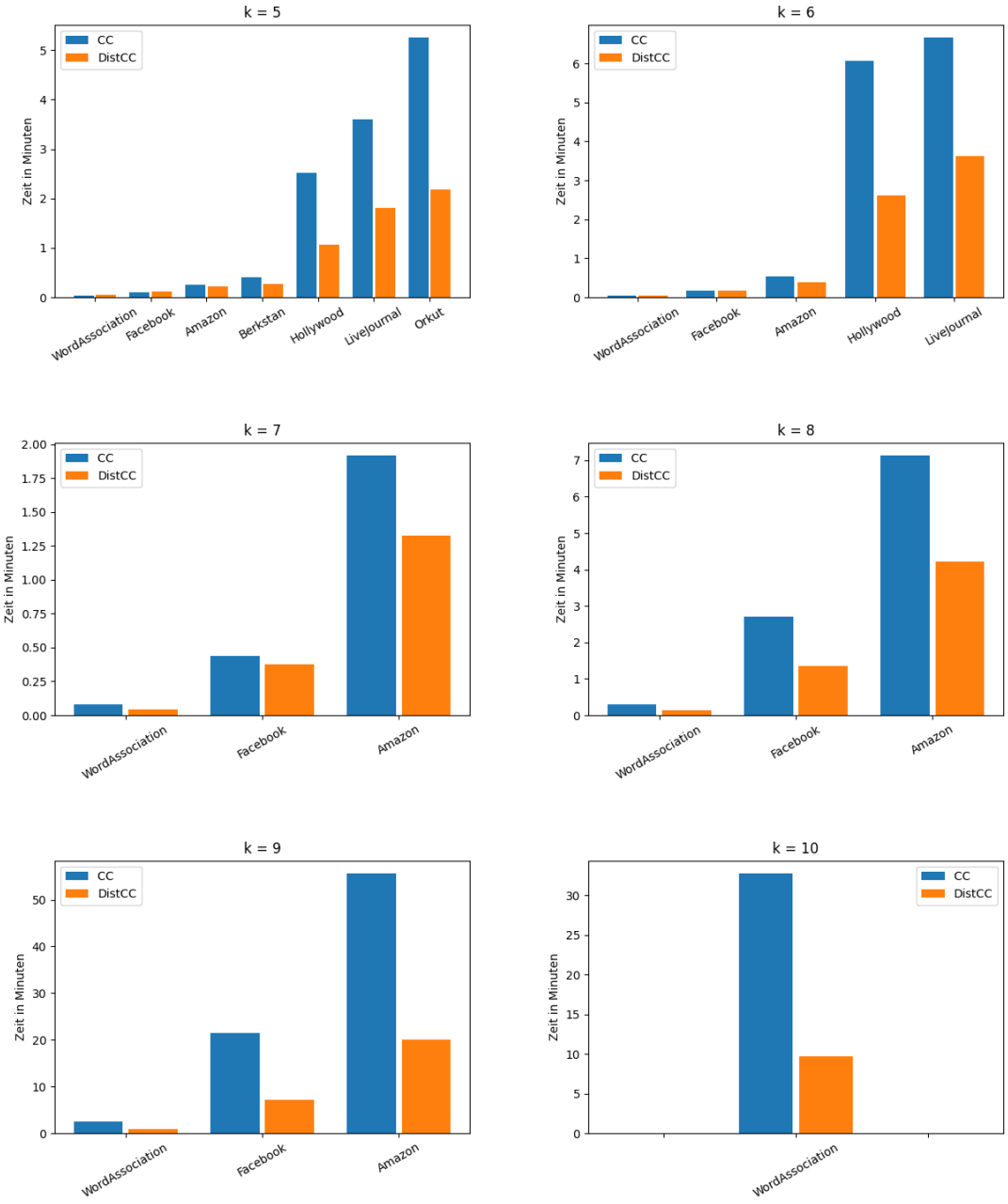

Abbildung 5.1: Laufzeiten CC's und DistCC's mit vier Maschinen für $k = 5$ bis $k = 10$. Die Verbesserung durch DistCC ist deutlich zu erkennen.

5.5.1 Laufzeiten

Abbildung 5.1 stellt die Laufzeiten der beiden Algorithmen dar. In fast allen Fällen sind signifikante Verbesserungen durch DistCC erkennbar. Zur Analyse dieser Ergebnisse sollen die Graphen in zwei Kategorien eingegliedert werden. Die kleinen Graphen: WordAssociation, Facebook und Amazon. Die großen Graphen: Berkstan, LiveJournal, Hollywood und Orkut.

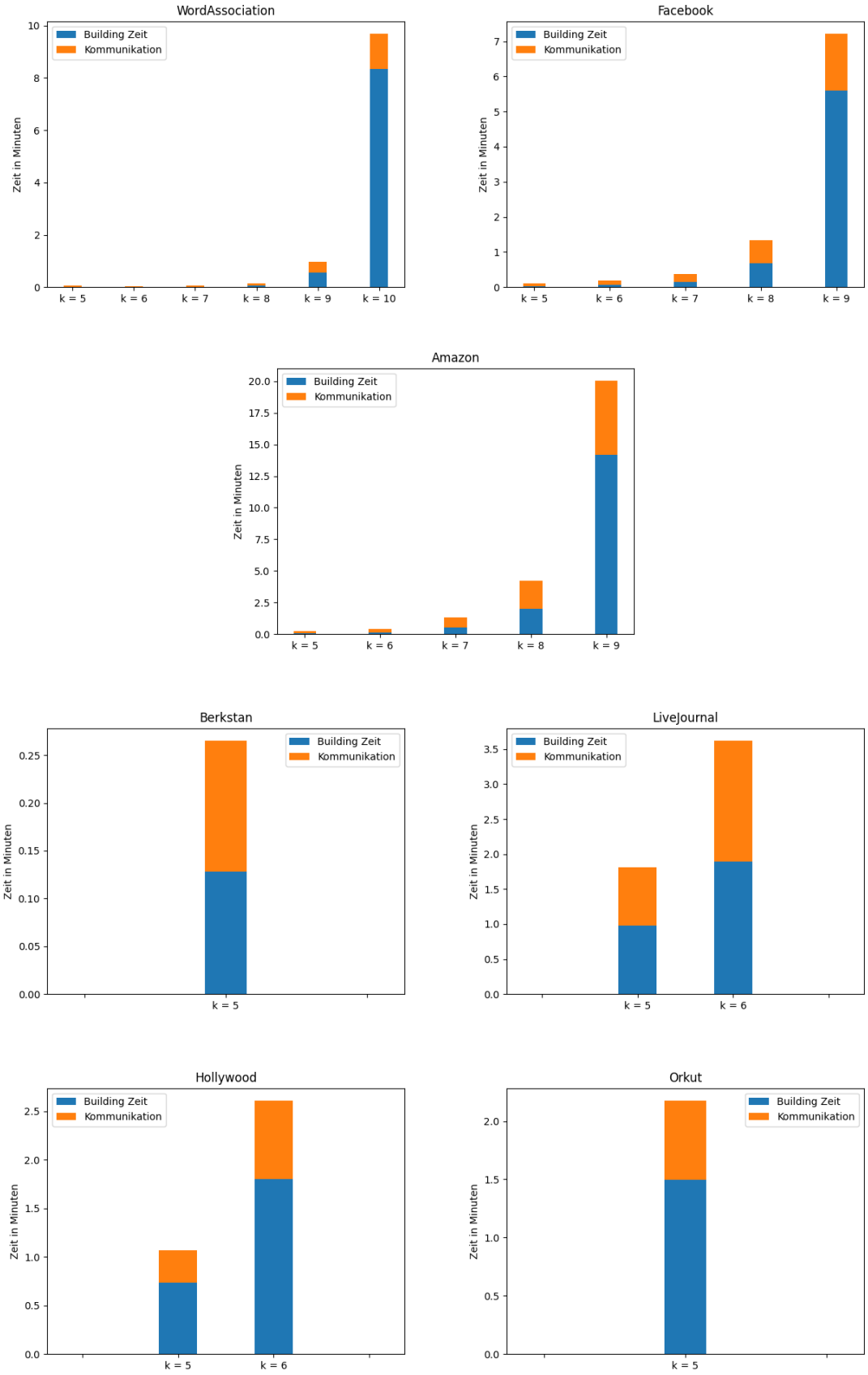

Abbildung 5.2: Anteil der Kommunikationszeit in der Gesamtlaufzeit der Building Phase für jeden Graphen und jedes k.

Tabelle 5.3: Verbesserung der Laufzeiten durch DistCC im Vergleich zu CC für jeden Graphen und jedes k.

Graph	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
WordAssociation	-1.2	-1.03	1.8	2.0	2.6	3.4
Facebook	-1.1	-1.01	1.2	2.0	3.0	-
Amazon	1.2	1.3	1.5	1.7	2.8	-
Berkstan	1.5	-	-	-	-	-
LiveJournal	2.0	1.8	-	-	-	-
Hollywood	2.4	2.3	-	-	-	-
Orkut	2.4	-	-	-	-	-

Bei den kleinen Graphen ist erkennbar, dass DistCC bis $k = 7$ zu keinen großen Verbesserungen führt. Im Gegenteil, es sind sogar Verschlechterungen durch DistCC erkennbar. Dies ist auf den großen Anteil der Kommunikationszeit zwischen den Maschinen zurückzuführen. Doch selbst dann handelt es sich um nur sehr geringe Verschlechterungen. Das Verhältnis der Kommunikationszeit und der Gesamtlaufzeit der Building Phase wird in Abbildung 5.2 veranschaulicht. Für $k = 7$ und größer verbessern sich die Laufzeiten erheblich. Die geringe Anzahl an Knoten, die diese Graphen haben, sorgt dafür, dass die Nachrichten zwischen den Maschinen nicht allzu groß werden. Somit fällt die benötigte Kommunikationszeit klein aus.

Auch bei den großen Graphen führt DistCC zu signifikanten Verbesserungen. So haben sich für $k = 5$ die Laufzeiten aller großen Graphen bis auf Berkstan bei DistCC im Vergleich zu CC mindestens halbiert. Es ist jedoch auch erkennbar, dass für $k = 6$ die Verbesserungen für LiveJournal und Hollywood kleiner werden. Auch das ist auf die großen Nachrichten der Maschinen bei diesen Graphen zurückzuführen. In Abbildung 5.2 erkennt man, dass sich die Kommunikationszeit für diese Graphen erhöht. Trotzdem handelt es sich im Vergleich zu den Laufzeiten CC's um eine bemerkenswerte Verbesserung.

5.5.2 Speicherverbrauch

Beide Algorithmen wurden in JAVA implementiert. Deshalb ist der Speicherverbrauch vom Verhalten des Garbage Collectors abhängig. Abbildung 5.3 stellt den Speicherverbrauch der beiden Algorithmen dar. Die Ergebnisse DistCC' wurden auf dem Master gemessen. Insgesamt ist erkennbar, dass der Speicherverbrauch von der Anzahl der Knoten des Graphen abhängt. Je weniger Knoten ein Graph besitzt, desto weniger Speicher benötigt DistCC im Vergleich zu CC. Besonders gut sieht man das bei Wordassociation. Amazon und Berkstan haben ungefähr gleich viele Knoten und benötigen deshalb auch ungefähr gleich viel Speicher für $k = 5$. Bei Hollywood, LiveJournal und Orkut, den drei größten Graphen, ist ein Umbruch erkennbar. Bei diesen Graphen benötigt DistCC insgesamt etwas mehr Speicher.

Bei DistCC berechnet eine Maschine nur die Anzahlen der Isomorphismen für die Knoten aus ihrer eigenen Partition. Die restlichen Anzahlen der anderen Knoten werden von den anderen Maschinen per Nachricht empfangen. Eine Nachricht enthält dabei zwei Listen. Eine Liste enthält die Treelets und deren Anzahlen, und die andere die entsprechenden Wurzelknoten.

Der niedrige Speicherverbrauch DistCC's bei kleineren Graphen liegt daran, dass bei DistCC nur die Anzahlen der Isomorphismen für die Knoten der eigenen Partition berechnet werden. Nur für diese werden also temporäre Variablen angelegt. Die Nachrichten, die eine Maschine empfängt sind auch verhältnismäßig klein, da nur die benötigten Anzahlen empfangen werden.

Bei größeren Graphen müssen viel mehr temporäre Variablen angelegt werden. Da die Nachrichten zusätzlich zu den eigentlich benötigten Anzahlen eine Liste mit den entsprechenden Knoten enthalten entsteht Speicherbedarf, der bei CC nicht existiert.

Insgesamt ist hier jedoch noch weitere Forschung notwendig. Der Master muss die Anzahlen aller Knoten der Worker empfangen, damit in der Sampling Phase der Algorithmus fortgesetzt werden kann. An dieser Stelle könnten zukünftige Arbeiten anknüpfen und versuchen die vom Master benötigten Anzahlen zu beschränken, um somit sowohl den Speicherverbrauch, als auch die Kommunikationszeiten zu verkürzen.

5.6 Kommunikation und Stabilität

Wie bereits oben demonstriert, bilden die Kommunikationszeiten DistCC's in bestimmten Fällen einen entscheidenden Anteil der Gesamtlaufzeit. Einer der Hauptfaktoren der die Kommunikationszeit bestimmt ist die Anzahl an Knoten im Graphen. Jeder Maschine wird eine Partition des Graphen zugewiesen, wobei nicht alle für die Berechnung benötigten Knoten in dieser Partition enthalten sind. Die Knoten, deren Anzahlen zwar benötigt werden, aber nicht in der Partition enthalten sind, befinden sich in den Partitionen anderer Maschinen. Die benötigten Anzahlen der Isomorphismen dieser Knoten werden nach jeder Berechnung für eine Treeletgröße $h \leq k$ unter den Maschinen ausgetauscht.

Je mehr Knoten ein Graph also hat, desto mehr Anzahlen der Isomorphismen für die Knoten müssen nach jeder Treeletgrößenberechnung ausgetauscht werden und desto größer sind die Nachrichten somit, die unter den Maschinen versendet und empfangen werden. Das ist der Hauptgrund für die überdurchschnittlich hohe Kommunikationszeit von den großen Graphen.

Einen weiteren wichtigen Faktor macht die Stabilität CC's aus. Zu Beginn dieses Kapitels wurde erläutert, dass die Stabilität CC's nicht optimal ist und somit zu hohen Standartabweichungen führt. Bei DistCC werden den Maschinen durch RandPart ungefähr gleich große Partitionen zugewiesen, mit dem Ziel, dass sie zur selben Zeit mit der Berechnung für eine Treeletgröße terminieren und sofort ihre Nachrichten austauschen können. An dieser Stelle wirken sich die starken Schwankungen in der Laufzeit CC's negativ auf die Laufzeiten DistCC's aus. Die Maschinen terminieren nicht wie erwartet gleichzeitig. In Abbildung 5.4 werden die durchschnittlichen Zeitdifferenzen zwischen der schnellsten und der langsamsten Maschine für alle k's dargestellt. Für $k = 9$ bei Facebook muss beispielsweise die schnellste Maschine durchschnittlich ca. 49 Sekunden auf die langsamste Maschine warten. Als Folge davon spiegeln sich diese Zeitdifferenzen in der Kommunikationszeit als Wartezeiten wieder.

Eine Verbesserung der Stabilität CC's durch eine optimierte Implementierung würde deshalb insgesamt auch die Wartezeiten verkürzen und schließlich auch die Laufzeiten DistCC's verbessern, ohne die Laufzeiten CC's signifikant zu verändern. So würden sich die Verbesserungen der Tabelle 5.3 weiter erhöhen.

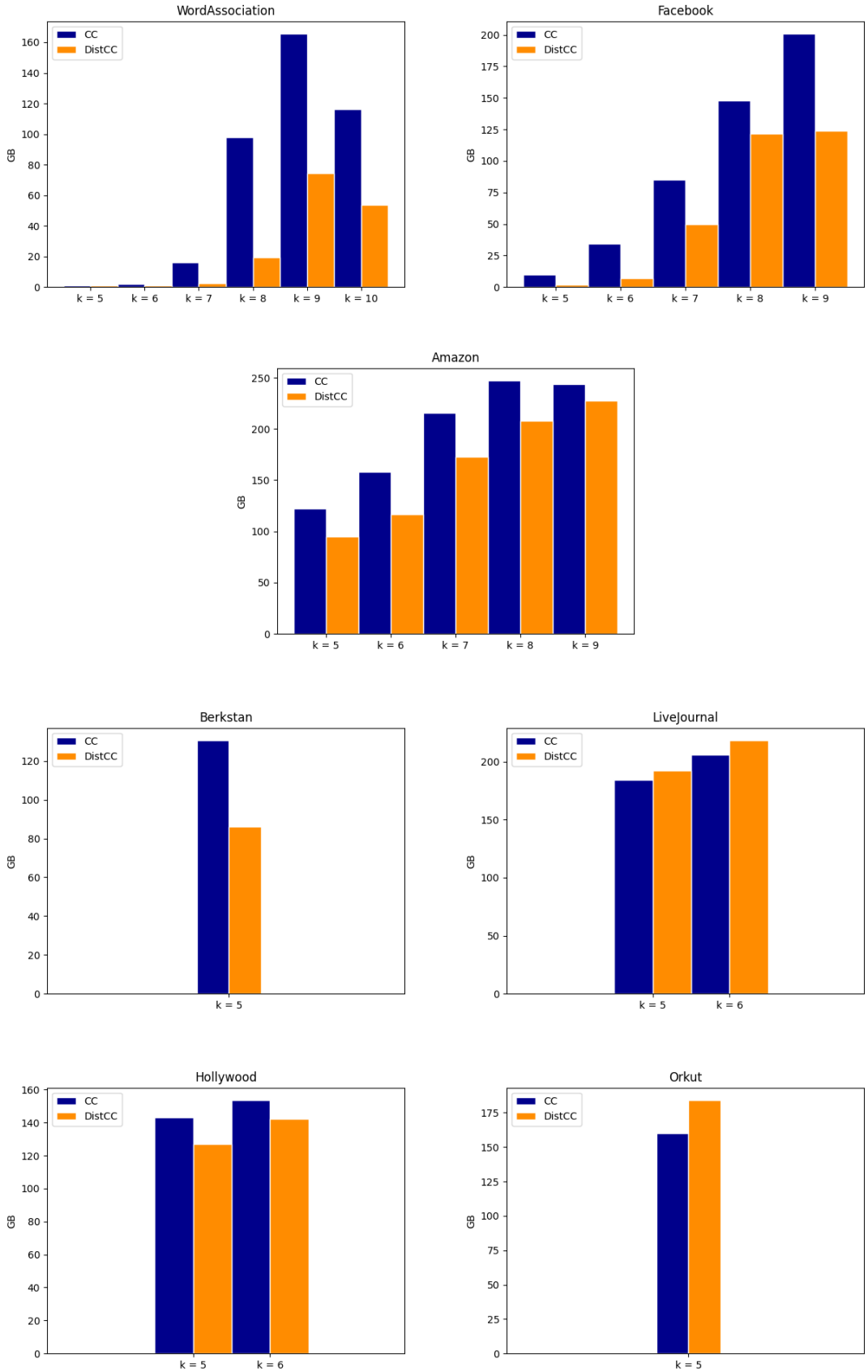

Abbildung 5.3: Speicherverbrauch CC's und des Masters DistCC's für jeden Graphen und jedes k.

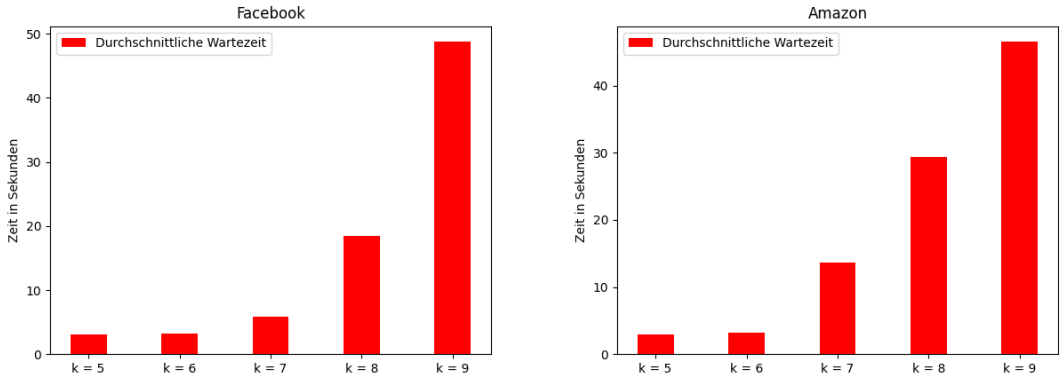

Abbildung 5.4: Wartezeiten, die aufgrund der unterschiedlichen Berechnungszeiten entstehen. Diese Wartezeiten sind in allen Kommunikationszeiten, die bisher vorgestellt wurden enthalten. Die Stabilität des Algorithmus hat einen starken Einfluss auf die Wartezeiten.

5.7 Skalierbarkeit

Um die Skalierbarkeit DistCC's zu analysieren wurde DistCC für drei Graphen mit zwei, drei und vier Maschinen jeweils 50 mal ausgeführt. Bei den drei Graphen handelt es sich um WordAssociation, Amazon und LiveJournal. In den Diagrammen aus Abbildung 5.5 werden die durchschnittlichen Laufzeiten dargestellt. Es ist deutlich zu erkennen, dass sich die Laufzeiten mit mehreren Maschinen verbessern.

Auch beim Speicherverbrauch sind Verbesserungen bei zunehmender Maschinenanzahl erkennbar. Diese werden in Abbildung 5.6 veranschaulicht. An dieser Stelle ist jedoch erneut zu betonen, dass für den Speicher eine weitere Forschung notwendig ist und dass deshalb die hier präsentierten Ergebnisse mit Vorsicht zu genießen sind.

Ausgehend von diesen Ergebnissen kann man festhalten, dass sich sowohl die Laufzeiten, als auch der Speicherverbrauch mit zunehmender Anzahl an Maschinen verbessern. Bis zu einer bestimmten Anzahl an Maschinen sollte das funktionieren. Daraufhin wird vermutlich die Kommunikationszeit zwischen den Maschinen weitere Verbesserungen der Laufzeit nicht zulassen. Auch dies ist offen für zukünftige Forschungen.

5.8 Analyse der Partitionierungsverfahren

Es wurden drei Partitionierungsverfahren miteinander verglichen. ParsePart, das bisher bei allen Ergebnissen vorgestellte RandPart und die Partitionierung aus [KK97], die hier MetisPart genannt wird. In 4.1.4 wurden zwei Schwächen ParseParts beschrieben. Zum einen unterscheiden sich die Kantenanzahlen der Partitionen zu stark voneinander. Zum anderen ist die Größe der Nachrichten zu groß, denn diese Partitionierung produziert keinen guten Schnitt zwischen den Partitionen. RandPart erzielte es, die Kanten gleichmäßig unter den Partitionen zu verteilen und MetisPart

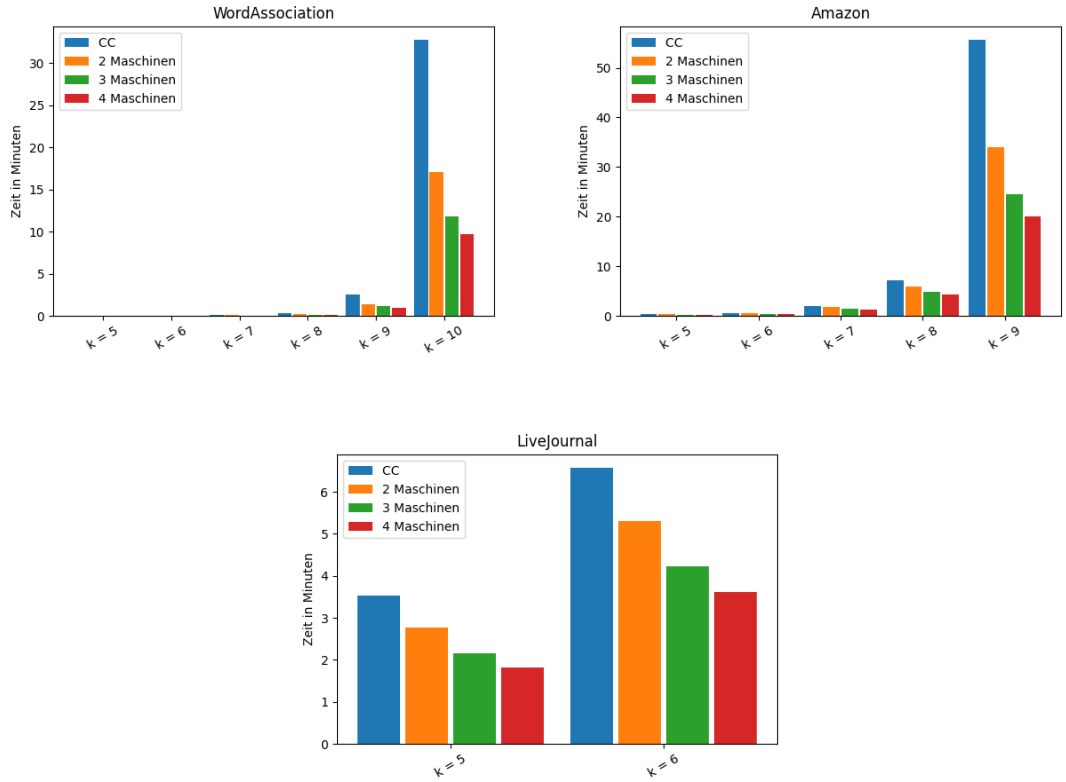

Abbildung 5.5: Durchschnittliche Laufzeiten für eine Maschine (=CC), zwei Maschinen, drei Maschinen und vier Maschinen.

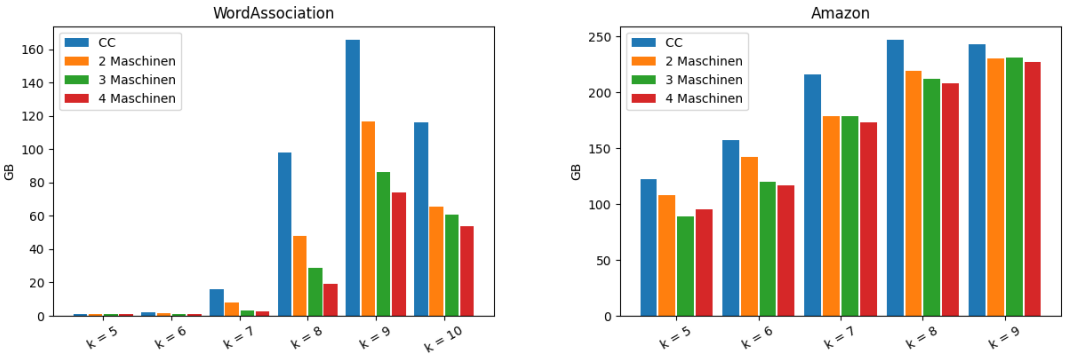

Abbildung 5.6: Durchschnittlicher Speicherverbrauch für eine Maschine (=CC), zwei Maschinen, drei Maschinen und vier Maschinen.

erzielte es, mittels eines besseren Schnitts zwischen den Partitionen die Größe der Nachrichten zu reduzieren. Im Folgenden werden zunächst die Lauf- und Wartezeiten ParseParts mit denen RandParts verglichen. Anschließend folgt der Vergleich von RandPart und MetisPart.

5.8.1 ParsePart vs. MetisPart

Um die folgenden Ergebnisse besser zu verstehen sollten die Kreisdiagramme in Abbildung 5.7 betrachtet werden. Hier werden die relativen Partitionsgrößen bezogen auf die Anzahl der Kanten in einer Partition dargestellt. Bei RandPart werden die Kanten sehr gleichmäßig verteilt, wohingegen sich bei ParsePart große Unterschiede zwischen den Kantenanzahlen aufweisen.

In den Diagrammen aus Abbildung 5.8 können die Laufzeiten von ParsePart und RandPart miteinander verglichen werden. RandPart ist der eindeutige Gewinner dieses Vergleichs. Die Ursache dafür wird in den Diagrammen aus Abbildung 5.9 dargestellt. Da die Kantenanzahlen der Partitionen sich bei ParsePart stark voneinander unterscheiden, benötigen Maschinen, die Partitionen mit verhältnismäßig vielen Kanten bearbeiten, viel mehr Zeit für ihre Berechnung als die restlichen Maschinen. So entstehen diese langen Wartezeiten.

Da die Kanten bei RandPart unter den Partitionen sehr gleichmäßig verteilt sind, entstehen hier geringe Wartezeiten. Jede Maschine hat eine ungefähr gleich große Arbeitslast. Die Unterschiede bei den Rechenzeiten werden von der Stabilität CC's verursacht. Eine stabilere Implementierung sollte diese Wartezeiten also weiter verkürzen.

5.8.2 RandPart vs. MetisPart

MetisPart produziert einen besseren Schnitt als RandPart. Dies resultiert in kleineren Nachriten die unter den Maschinen ausgetauscht werden und somit in einer kürzeren Kommunikationszeit. Die Kommunikationszeit besteht jedoch nicht nur aus dem Nachrichtenaustausch der Maschinen. Die Wartezeiten, die hauptsächlich von unterschiedlich großen Partitionen verursacht werden, sind ein wichtiger Bestandteil der Kommunikationszeit.

Die Experimente haben gezeigt, dass Unterschiede in den Partitionsgrößen die Gesamtlaufzeiten viel stärker beeinflussen als kleinere Schnitte unter den Partitionen. Denn bei fast allen Graphen und k's hat RandPart besser abgeschnitten als MetisPart. In Abbildung 5.10 werden die Laufzeiten beider Partitionierungsverfahren für WordAssociation verglichen. Lediglich bei WordAssociation und $k = 8$ konnte MetisPart eine kürzere Gesamtlaufzeit erzielen. Hier waren also zusätzlich zu den kürzeren Zeiten des Nachrichtenaustausch durch kleinere Nachrichten die Wartezeiten kurz genug, damit MetisPart eine kürzere Laufzeit als RandPart erzielen konnte.

Insgesamt lässt sich festhalten, dass sowohl der Schnitt, als auch die gleichmäßige Verteilung der Kanten unter den Partitionen die Laufzeiten beeinflussen, wobei beim letzteren signifikantere Verbesserungen erkennbar sind. Ein Partitionierungsverfahren, dass Partitionen produziert, die beide dieser Faktoren berücksichtigt könnte die Laufzeiten zusätzlich verbessern.

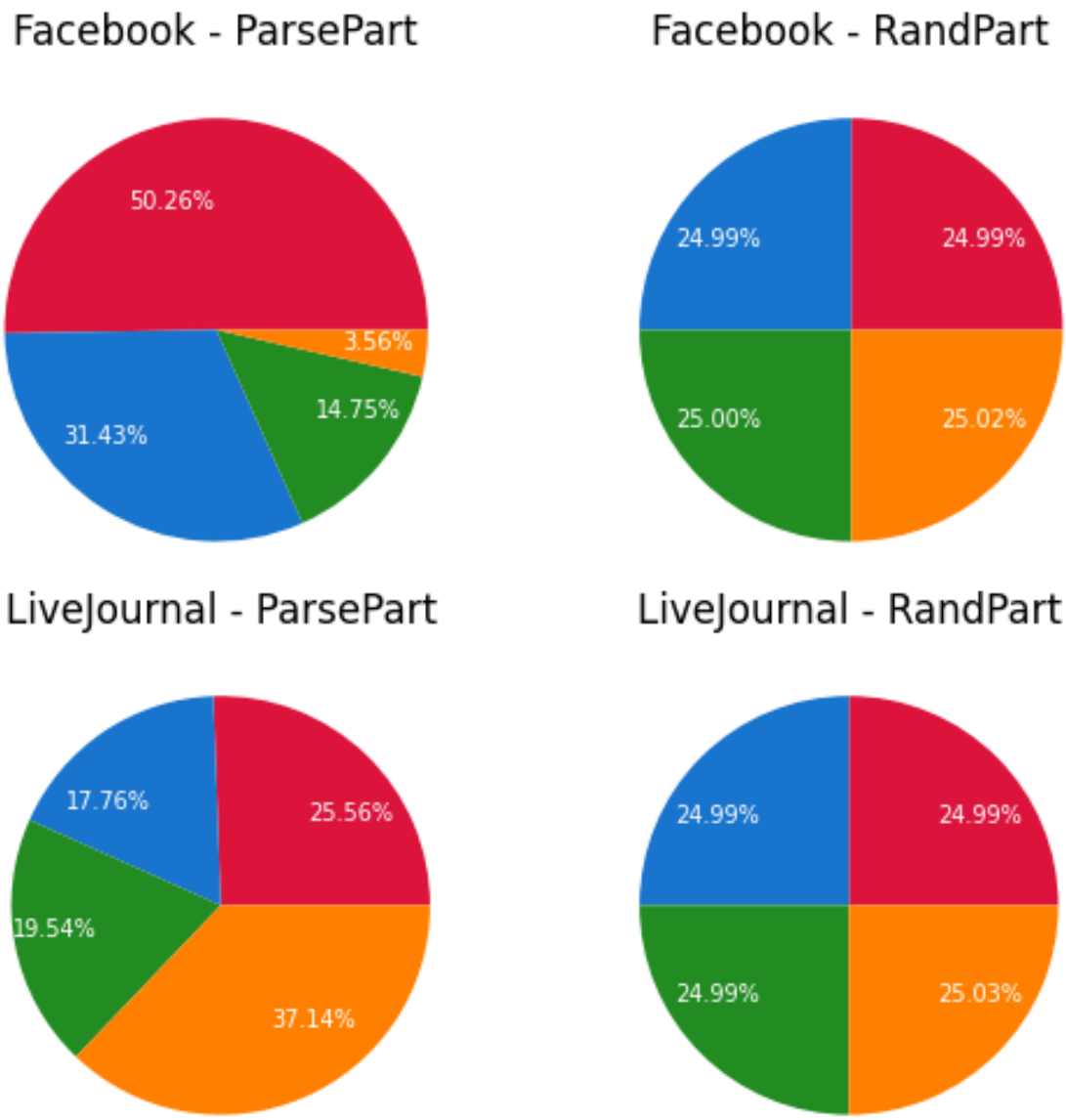

Abbildung 5.7: Verteilung der Kanten unter den Partitionen. Gleichmäßige Verteilung bei RandPart, große Unterschiede der Kantenanzahlen bei ParsePart.

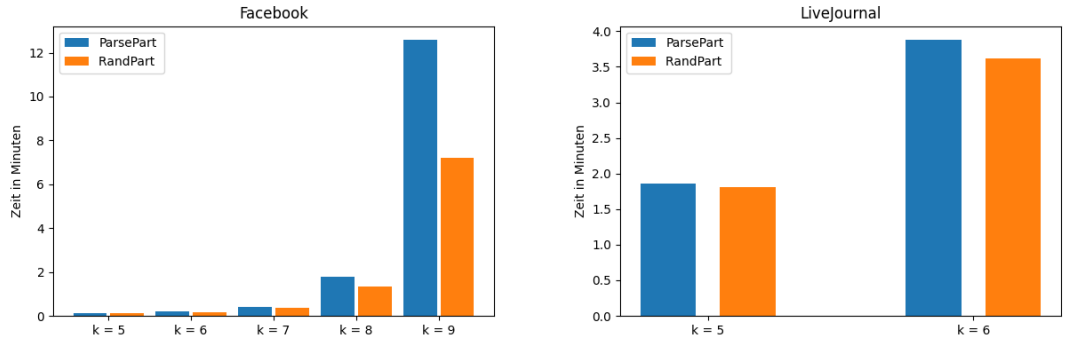

Abbildung 5.8: Laufzeiten von ParsePart und RandPart im Vergleich. RandPart benötigt aufgrund der besseren Verteilung der Kanten unter den Partitionen deutlich weniger Zeit.

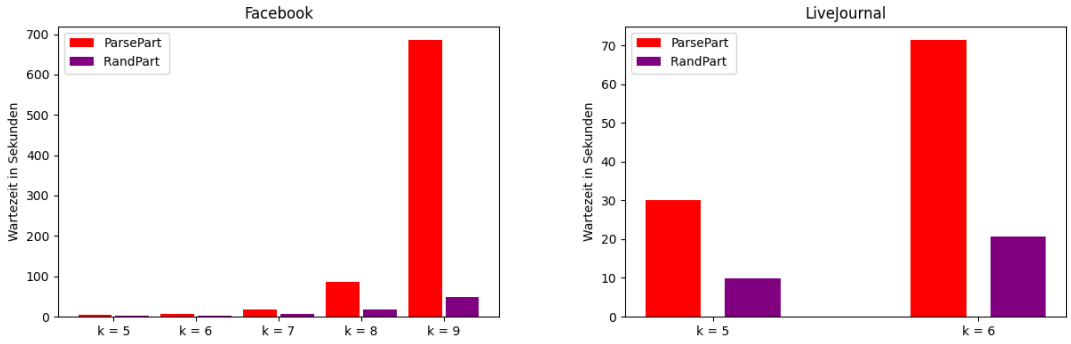

Abbildung 5.9: Wartezeiten von ParsePart und RandPart im Vergleich.

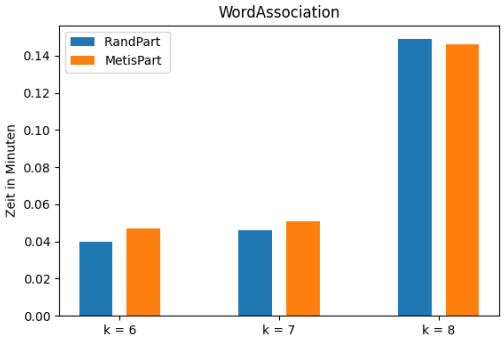

Abbildung 5.10: Laufzeiten von RandPart und MetisPart für WordAssociation. Für $k = 8$ erzielt MetisPart eine geringfügig bessere Laufzeit als RandPart.

6 Verwandte Arbeiten

Viele Subgraph-Counting-Algorithmen, inklusive PARSE, CC und DistCC, basieren auf der Colorcoding-Technik. Diese Arbeit hat sich lediglich auf das Wesentliche dieser Technik beschränkt. Eine detaillierte Erklärung und Beschreibung dieser Technik befindet sich in der ursprünglichen Publikation in [AYZ95].

Der PARSE Algorithmus bildet zusammen mit CC die Grundlage DistCC's. Bei PARSE werden unter den Workern keine Nachrichten ausgetauscht, sondern nur zwischen den Workern und dem Master. Dies hat der Algorithmus seinem Backtracking-Ansatz und seiner Partitionierung zu verdanken, welche beide in [ZKKM10] genauer erläutert werden.

[BCK+17] und [BCK+18] enthalten eine detaillierte Beschreibung CC's. Zudem befinden sich hier auch wichtige Beweise und Herleitungen der in dem Algorithmus verwendeten Formeln. Auch der Speicherverbrauch und die Laufzeit werden mit theoretischen Grenzen näher erläutert.

Ein weiterer parallele und verteilter Subgraph-Counting-Algorithmus ist FASCIA (fast approximate subgraph counting and enumeration) [SM13a] [SM14] [SM15]. Hier werden unter Anderem Optimierungen für den Speicherverbrauch der Count-Tabelle mittels Hash-Tabellen und der CSR (Compressed Sparse Rows) Datenstruktur präsentiert. Außerdem werden zwei unterschiedliche Versionen dieses Algorithmus' für kleine und große Graphen gezeigt.

Sowohl FASCIA, als auch PARSE wurden vor einigen Jahren entwickelt. CC ist 2017 erschienen. Der relativ neuer Subgraph-Enumeration-Algorithmus Motivo [BLP19], der auf CC basiert, ist 2019 erschienen. Bei diesem Algorithmus werden durch prägnante, auf Colorcoding-Operationen spezialisierte Datenstrukturen und einer voreingenommenen Färbung des Graphen deutliche Verbesserungen der Laufzeiten sowohl in der Building Phase, als auch in der Sampling Phase erzielt. Jedoch ist Motivo, genauso wie CC, nicht verteilt.

Ein weiteres Approximation-basierendes Subgraph-Counting-Verfahren ist Path Sampling [JSP14b] [WTZG15] [WZL+16]. Man durchläuft dazu k Knoten des Graphen und betrachtet, den von diesen Knoten induzierten Subgraphen. Dieses Verfahren kann jedoch nur Isomorphismen für Subgraphen bis zur Größe fünf zählen.

Diese Arbeit hat sich auf Approximation-basierende Subgraph-Counting- und Subgraph-Enumeration-Algorithmen beschränkt, da die Suche nach der exakten Anzahl der Isomorphismen eines Subgraphen schwierig ist. In 2016 wurde der Algorithmus namens ESCAPE [PSV16] eingeführt, der für $k \leq 5$ bei hauptsächlich kleineren Graphen die exakten Anzahlen der Isomorphismen berechnen kann. In 2019 ist dann LIGHT [SCWL19] erschienen, der Isomorphismen mit bis zu sechs Knoten exakt zählen kann.

7 Schlussfolgerung

In dieser Arbeit wurde der verteilter Subgraph-Enumeration-Algorithmus DistCC entwickelt. Dieser basiert auf den beiden Algorithmen CC [BCK+17] [BCK+18] und PARSE [ZKKM10]. CC bildet die Basis für den eigentlichen Algorithmus und PARSE bildet die Basis für die Verteilung DistCC's. Die Maschinen werden ähnlich wie bei PARSE in Master- und Workermaschinen unterteilt. Auch die Partitionierung des Graphen wird aus PARSE übernommen. Da diese jedoch zwei signifikante Schwächen aufwies, wurde RandPart entwickelt, welches ein neues Partitionierungsverfahren ist. Darüber hinaus wurde auch ein Partitionierungsverfahren aus [KK97] getestet. Außerdem wurde festgestellt, dass die Laufzeiten CC's in bestimmten Fällen von der Sortierung des Graphen abhängig sind. Deshalb wurden mit dem Zweck die Laufzeiten zu kürzen neben der natürlichen Sortierung der Graphen zwei weitere Sortierungen getestet.

Die Experimente haben gezeigt, dass die Sortierungen der Graphen die Laufzeiten beeinflussen können. Dafür ist hauptsächlich die $getAndIncerement()$-Methode aus der Zeile 10 des Algorithmus' 3.5 verantwortlich. Die SBD Sortierung verbessert die Laufzeiten der Graphen mit Knoten, die einen sehr hohen Grad haben und die sehr weit hinten in der Sortierung ihren Platz einnehmen. Die gemischte Sortierung verbessert die Laufzeiten der Graphen, bei deren Sortierung viele niedriggradige Knoten aufeinander folgen. Es ist möglich, dass eine Sortierung existiert, die die Verzögerung, die durch den Aufruf der $getAndIncerement()$-Methode entsteht, minimiert. Solch eine Sortierung könnte ein Bestandteil einer zukünftigen Arbeit sein.

Auch NewCC lies anhand der Experimente sein Potenzial erkennen. Damit NewCC noch kürzere Laufzeiten erzielt ist eine stabilere Implementierung CC's erforderlich, denn die Auslastung der Threads bei NewCC hängt von dieser Stabilität ab. Ist eine stabilere Implementierung vorhanden, kann die Modifizierung bei NewCC auch auf die einzelnen Maschinen DistCC's übertragen werden.

Eine stabilere Implementierung CC's würde des Weiteren die Laufzeiten DistCC's verbessern, die ohnehin schon teilweise deutlich kürzer sind als die Laufzeiten CC's. Einen signifikanten Anteil der Laufzeiten DistCC's bilden die Wartezeiten der Maschinen während der Kommunikationszeit, da nicht alle Maschinen zur selben Zeit ihre Berechnung abschließen. Das ist eine direkte Konsequenz der Stabilität, denn zwei Durchgänge CC's mit demselben Input können zwei voneinander sehr stark abweichende Laufzeiten produzieren. Dies lässt sich auch auf die einzelnen Maschinen DistCC's übertragen, unter denen die Arbeitslast mit RandPart ungefähr gleichmäßig verteilt wird. Eine stabilere Implementierung würde in geringeren Wartezeiten zwischen den Maschinen während der Kommunikationszeit resultieren, die in einigen Fällen über 60% der Kommunikationszeit ausmachen. Die durchschnittlichen Laufzeiten CC's würden dabei keine signifikanten Unterschiede aufweisen.

Im Speicherverbrauch waren auch Verbesserungen durch DistCC erkennbar. Der beschränkende Faktor ist hier jedoch, dass der Master alle Anzahlen der Isomorphismen von jedem Worker empfängt, um in der Sampling Phase seine Berechnung fortsetzen zu können. Um noch mehr Speicher sparen zu können, könnte in zukünftigen Arbeiten erforscht werden, ob sich diese Beschränkung aufheben lässt.

Die Experimente zur Skalierbarkeit haben gezeigt, dass sich sowohl die Laufzeiten, als auch der Speicherverbrauch mit einer zunehmenden Anzahl von Maschinen verringern. In dieser Arbeit konnte DistCC mit maximal vier Maschinen ausgeführt werden. Die Experimente versprechen aber noch größere Verbesserungen bei einer noch größeren Anzahl von Maschinen.

Einen wichtigen Faktor der Laufzeiten DistCC's bildet die jeweilige Partitionierung des Graphen. Dabei sollte die Partitionierung im Idealfall sowohl einen guten Schnitt produzieren, damit die Nachrichten unter den Maschinen klein sind, als auch eine gute Kantenverteilung unter den Partitionen gewährleisten, damit die Arbeitslast der Maschinen ungefähr gleich groß ist und somit die Wartezeiten minimiert werden. RandPart hat sich als eine relativ gute Lösung erwiesen. Dabei erfüllt RandPart lediglich eine dieser beiden Voraussetzungen. Thema einer zukünftigen Arbeit könnte es daher sein, einen Algorithmus zu entwickeln, der beide dieser Voraussetzungen für einen optimalen Partitionierungsalgorithmus für diesen Anwendungsfall erfüllt.

Die Sampling Phasen CC's und DistCC's sind identisch. Diese Arbeit hat also lediglich die Building Phase CC's verteilt und alle Verbesserungen, die hier vorgestellt wurden, beziehen sich somit auf die Building Phase. Zukünftige Forschungen können sich auf die Sampling Phase fokussieren, und eventuell auch für diese eine verteilte Version des Algorithmus' entwickeln.

Der bereits in Kapitel 6 erwähnte Subgraph-Enumeration-Algorithmus Motivo [BLP19], der ebenso auf CC basiert bildet einen weiteren interessanten Forschungsgebiet für künftige Arbeiten. Dieser Algorithmus führt zu bemerkenswerten Verbesserungen sowhol in der Building Phase, als auch in der Sampling Phase. Genauso wie CC ist jedoch auch Motivo kein verteilter Algorithmus. Künftige Arbeiten können versuchen die Verteilung DistCC's auf Motivo zu übertragen.

Literaturverzeichnis

[ADH+08a] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, C. Sahinalp. „Biomolecular network motif counting and discovery by color coding“. In: *Bioinformatics (Oxford, England)* 24 (Juli 2008), S. i241–9. DOI: 10.1093/bioinformatics/btn163 (zitiert auf S. 13).

[ADH+08b] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, S. C. Sahinalp. „Biomolecular network motif counting and discovery by color coding“. In: *Bioinformatics* 24.13 (2008), S. i241–i249 (zitiert auf S. 18).

[AYZ95] N. Alon, R. Yuster, U. Zwick. „Color-coding“. In: *Journal of the ACM (JACM)* 42.4 (1995), S. 844–856 (zitiert auf S. 3, 13, 18, 29, 51).

[BCK+17] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, A. Panconesi. „Counting Graphlets: Space vs Time“. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. WSDM '17. Cambridge, United Kingdom: Association for Computing Machinery, 2017, S. 557–566. ISBN: 9781450346757. DOI: 10.1145/3018661.3018732. URL: https://doi.org/10.1145/3018661.3018732 (zitiert auf S. 3, 13, 26, 27, 31, 51, 53).

[BCK+18] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, A. Panconesi. „Motif Counting Beyond Five Nodes“. In: *ACM Trans. Knowl. Discov. Data* 12.4 (Apr. 2018). ISSN: 1556-4681. DOI: 10.1145/3186586. URL: https://doi.org/10.1145/3186586 (zitiert auf S. 3, 13, 26, 28, 31, 38, 51, 53).

[BLP19] M. Bressan, S. Leucci, A. Panconesi. „Motivo: fast motif counting via succinct color coding and adaptive sampling“. In: *Proceedings of the VLDB Endowment* 12 (Juli 2019), S. 1651–1663. DOI: 10.14778/3342263.3342640 (zitiert auf S. 51, 54).

[BRRA12] M. A. Bhuiyan, M. Rahman, M. Rahman, M. Al Hasan. „GUISE: Uniform Sampling of Graphlets for Large Graph Analysis“. In: *2012 IEEE 12th International Conference on Data Mining*. 2012, S. 91–100. DOI: 10.1109/ICDM.2012.87 (zitiert auf S. 13).

[CKM+16] V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, B. Schieber. *Subgraph Counting: Color Coding Beyond Trees*. 2016. arXiv: 1602.04478 [cs.DC] (zitiert auf S. 13).

[CLWL16] X. Chen, Y. Li, P. Wang, J. C. S. Lui. *A General Framework for Estimating Graphlet Statistics via Random Walk*. 2016. arXiv: 1603.07504 [cs.SI] (zitiert auf S. 13).

[GN02] M. Girvan, M. E. J. Newman. „Community structure in social and biological networks“. In: *Proceedings of the National Academy of Sciences* 99.12 (2002), S. 7821–7826. ISSN: 0027-8424. DOI: 10.1073/pnas.122653799. eprint: https://www.pnas.org/content/99/12/7821.full.pdf. URL: https://www.pnas.org/content/99/12/7821 (zitiert auf S. 13).

[HS17] G. Han, H. Sethu. *Waddling Random Walk: Fast and Accurate Mining of Motif Statistics in Large Graphs*. 2017. arXiv: 1605.09776 [cs.SI] (zitiert auf S. 13).

[HWP03] J. Huan, W. Wang, J. Prins. „Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism“. In: Dez. 2003, S. 549–552. ISBN: 0-7695-1978-4. DOI: 10.1109/ICDM.2003.1250974 (zitiert auf S. 13).

[JSP14a] M. Jha, C. Seshadhri, A. Pinar. *Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts*. 2014. arXiv: 1411.4942 [cs.DS] (zitiert auf S. 13).

[JSP14b] M. Jha, C. Seshadhri, A. Pinar. *Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts*. 2014. arXiv: 1411.4942 [cs.DS] (zitiert auf S. 51).

[KK02] M. Kuramochi, G. Karypis. „Frequent Subgraph Discovery“. In: (Jan. 2002) (zitiert auf S. 13).

[KK04] M. Kuramochi, G. Karypis. „Finding Frequent Patterns in a Large Sparse Graph“. In: *Data Mining and Knowledge Discovery* 11 (Juli 2004). DOI: 10.1007/s10618-005-0003-9 (zitiert auf S. 13).

[KK97] G. Karypis, V. Kumar. „METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices“. In: (1997) (zitiert auf S. 33, 45, 53).

[LSK06] J. Leskovec, A. Singh, J. Kleinberg. „Patterns of Influence in a Recommendation Network“. In: Bd. 3918. März 2006, S. 380–389. ISBN: 978-3-540-33206-0. DOI: 10.1007/11731139_44 (zitiert auf S. 13).

[MSl+11] R. Milo, S. Shen-Orr, S. ltzkovitz, N. Kashtan, D. Chktovskii, U. Alan. „Network Motifs: Simple Building Blocks of Complex Networks“. In: Dez. 2011. ISBN: 9781400841356. DOI: 10.1515/9781400841356.217 (zitiert auf S. 13).

[PSV16] A. Pinar, C. Seshadhri, V. Vishal. *ESCAPE: Efficiently Counting All 5-Vertex Subgraphs*. 2016. arXiv: 1610.09411 [cs.SI] (zitiert auf S. 51).

[SCWL19] S. Sun, Y. Che, L. Wang, Q. Luo. „Efficient Parallel Subgraph Enumeration on a Single Machine“. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, S. 232–243. DOI: 10.1109/ICDE.2019.00029 (zitiert auf S. 51).

[SM13a] G. M. Slota, K. Madduri. „Fast approximate subgraph counting and enumeration“. In: *2013 42nd International Conference on Parallel Processing*. IEEE. 2013, S. 210–219 (zitiert auf S. 27, 51).

[SM13b] G. M. Slota, K. Madduri. „Fast Approximate Subgraph Counting and Enumeration“. In: *2013 42nd International Conference on Parallel Processing*. 2013, S. 210–219. DOI: 10.1109/ICPP.2013.30 (zitiert auf S. 13).

[SM14] G. M. Slota, K. Madduri. „Complex network analysis using parallel approximate motif counting“. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, S. 405–414 (zitiert auf S. 27, 51).

[SM15] G. M. Slota, K. Madduri. „Parallel color-coding“. English. In: *Parallel Computing* 47.C (2015), S. 51–69. DOI: 10.1016/j.parco.2015.02.004 (zitiert auf S. 51).

[Wer06] S. Wernicke. „Efficient Detection of Network Motifs“. In: *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM* 3 (Okt. 2006), S. 347–59. DOI: 10.1109/TCBB.2006.51 (zitiert auf S. 13).

[WTZG15] P. Wang, J. Tao, J. Zhao, X. Guan. *Moss: A Scalable Tool for Efficiently Sampling and Counting 4- and 5-Node Graphlets*. 2015. arXiv: 1509.08089 [cs.SI] (zitiert auf S. 51).

[WZL+16] P. Wang, X. Zhang, Z. Li, J. Cheng, J. C. S. Lui, D. Towsley, J. Zhao, J. Tao, X. Guan. *A Fast Sampling Method of Exploring Graphlet Degrees of Large Directed and Undirected Graphs*. 2016. arXiv: 1604.08691 [cs.SI] (zitiert auf S. 51).

[ZKKM10] Z. Zhao, M. Khan, V. A. Kumar, M. V. Marathe. „Subgraph enumeration in large social contact networks using parallel color coding and streaming“. In: *2010 39th International Conference on Parallel Processing*. IEEE. 2010, S. 594–603 (zitiert auf S. 3, 13, 21, 23–25, 31, 51, 53).

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift