Karlsruhe Institute of Technology

Engineering von Algorithmen zur Berechnung von Betweenness-Varianten

Bachelorarbeit
von

Yvonne Braun

An der Fakultät für Informatik
Institut für Theoretische Informatik(ITI)

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuender Mitarbeiter:	Dipl.-Inform. Andrea Kappes
Zweiter betreuender Mitarbeiter:	Dipl.-Inform./-Math. Tanja Hartmann

Bearbeitungszeit: 13. August 2012 – 12. November 2012

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 12.November 2012

Danksagung

Dank an Prof. Dr. Dorothea Wagner für die Annahme der Arbeit und dafür dass ich die Arbeit am Institut schreiben konnte.
Außerdem möchte ich Andrea Kappes und Tanja Hartmann für die zeitintensive Betreuung, die Korrekturen und regelmäßigen Treffen danken. Danke auch nochmal an Andrea für das konvertieren eines Teiles der Graphen und Tanja für die Bereitstellung des Parsers.

Zusammenfassung
In dieser Arbeit wird ein schon vorhandenes Verfahren zur Berechnung der Paarbetweenness $\mathcal{PB}$ von je zwei Knoten eines Netzwerkes durch eine Reduktion des Graphen auf seinen 2-Core im Hinblick auf den Speicherverbrauch und die Laufzeit verbessert. Dabei wird das vorhandene Verfahren modifiziert, so dass bei der Paarbetweennessberechnung im 2-Core auch Wege zu Knoten, die außerhalb des 2-Cores liegen berücksichtigt werden. Es wird gezeigt, dass mit Hilfe der Paarbetweennesswerte im 2-Core und wenigen zusätzlich vorberechneten Werten auch die Paarbetweennesswerte von Knotenpaaren, bei denen mindestens ein Knoten nicht im 2-Core liegt, zu jeder Zeit mit geringem Aufwand berechnet werden können. Der Aufwand für die Vorberechnung ist linear in der Anzahl der Kanten, die zu mindestens einem Knoten außerhalb des 2-Cores inzident sind. In der Evaluation zeigt sich, dass das Verfahren für Graphen mit kleinem 2-Core tatsächlich schneller ist und dass durch den geringeren Speicherbedarf auch größere Netzwerke verarbeitet werden können

Inhaltsverzeichnis

1. Einleitung **3**

2. Speichereffiziente Berechnung der Paarbetweenness **7**
2.1. Schnellere Berechnung der Betweenness 7
2.2. Platzsparendere Berechnung der Paarbetweenness 9
2.2.1. Betrachtung von Paaren (a,b), mit $a, b \in T$ und $w(a) = w(b)$ . . . 10
2.2.2. Betrachtung von Paaren (a,b), mit $a, b \in 2 - \text{Core}$ 12
2.2.3. Betrachtung von Paaren (a,b) mit $a \in T \leftrightarrow b \notin T$ 16
2.2.3.1. erste Richtung: vom 2-Core nach T 16
2.2.3.2. zweite Richtung: von T in den 2-Core 17
2.2.4. Betrachtung von Paaren (a,b), mit $a, b \in T$ und $w(a) \neq w(b)$ . . . 18
2.2.5. Analyse und Bewertung 19

3. Evaluation **21**
3.1. Darstellung der Ergebnisse 21
3.2. Interpretation der Ergebnisse 23
3.3. Kritische Betrachtungen 24

4. Zusammenfassung und Ausblick **25**

Literaturverzeichnis **27**

Anhang **29**
A. First Appendix Section 29

Notation

$G(V, E)$	Graph mit Knotenmenge V und Kantenmenge E	3
$\|V\| = n$	Anzahl Knoten von G	3
$\|E\| = m$	Anzahl Kanten von G	3
$\mathcal{B}(v)$	Betweenness eines Knotens v	3
$d(s, t)$	Länge des kürzesten Weges von s nach t	3
σ_{st}	Anzahl kürzeste Wege von s nach t	3
$\sigma_{st}(v)$	Anzahl kürzeste Wege von s nach t über v	3
$\delta_{s\bullet}(v)$	Dependency des Knotens v bezüglich Startknoten s	3
$\delta_{st}(v)$	Paardependency des Knotens v bezüglich Endknoten s,t	12
$\delta_{s\bullet}^{mod}(v)$	modifizierte Dependency $\equiv \mathcal{PB}(vs)$	12
$\mathcal{GB}(C)$	Gruppenbetweenness einer Gruppe C	4
$\mathcal{PB}(a, b)$	Paarbetweennes des geordneten Knotenpaares (a,b)	4
s − a − t − Weg	kürzeste Wege von s nach t über a	5
$P^-(a, b)$	$\{s \mid \exists$ s − a − b − Weg$\}$	5
$P^+(a, b)$	$\{t \mid \exists\ a - b - tWeg\}$	5
T	$\{t \in V \mid t \notin 2Core\}$	7
$t(v)$	$\{t \in T \mid$ t ist Nachfolger von v$\}$	10
$T(v)$	$\{t \in V \mid t(v) \wedge v\}$	8
$w(v)$	Knoten w ist Wurzel von v und $v \in T$	8
U	$\{u \in V \mid \exists t \in V \wedge w(t) = u\}$	12
$a(v)$	Knoten a ist Vaterknoten von v und $v \in T$	8
dfs_num(v)	DFS-Nummer des Knotens v	11
tree(v)	ID des Baumfortsatzes in dem sich Knoten v befindet, $v \in T$	8

1. Einleitung

Die *Betweenness* wurde ursprünglich als Maß zur Analyse von sozialen Netzwerken entwickelt, und kann sinngemäß auch für andere Netzwerke, wie beispielsweise Straßennetze oder Internetverbindungen verwendet werden. Die Modellierung des Netzwerkes erfolgt dafür durch einen *Graphen* $G = (V, E)$, wobei die Knotenmenge V für die Teilnehmer des Netzwerkes steht, und die Kantenmenge E Verbindungen zwischen diesen Teilnehmern repräsentiert. Zur Beschreibung der Größe eines Graphen werden im allgemeinen die beiden Maßzahlen $|V| = n$ und $|E| = m$ verwendet. Die Betweenness eines Knotens v beschreibt den Anteil der kürzesten Wege zwischen je zwei Knoten des Netzwerkes, die über v führen. Die *Distanz* $d(s, t)$ zwischen zwei Knoten s und t ist bei gewichteten Graphen die Summe der Kantengewichte aller Kanten, die einen kürzesten Weg zwischen s und t bilden. Für ungewichtete Graphen ist $d(s, t)$ somit gleich der Summe der Kanten eines kürzesten Weges von Knoten s zu Knoten t. Mit σ_{st} wird die Anzahl der kürzesten Wege von einem Startknoten s zu einem Zielknoten t bezeichnet. $\sigma_{st}(v)$ ist die Anzahl der kürzesten Wege von s nach t, die zusätzlich über den Knoten v führen. Damit lässt sich die Betweenness $\mathcal{B}(v)$ eines Knotens v auf die folgende Weise definieren:

$$\mathcal{B}(v) := \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1.1}$$

Die schnellsten bekannten Algorithmen zur exakten Berechnung der Betweenness basieren auf dem Algorithmus von Brandes [Bra01]. Dieser beruht auf der n-fachen Ausführung des Single-Source-Shortest-Path-Problems, mit anschließender Berechnung der *Dependency* $\delta_{s\bullet}(v)$ eines Knotens v zum Startknoten s:

$$\delta_{s\bullet}(v) := \sum_{t \in V} \frac{\sigma_{st(v)}}{\sigma_{st}} \tag{1.2}$$

Für gewichtete Graphen wird das Single-Source-Shortest-Path-Problem standardmäßig, mit dem Algorithmus von Dijkstra gelöst, für ungewichtete Graphen kann eine einfache Breitensuche durchgeführt werden. Die Berechnung der Dependencies ist wesentlicher Bestandteil von Brandes Algorithmus [Bra01] und erfolgt für jeden Startknoten in

$\mathcal{O}(m)$. Die Laufzeit wird also, vor allem im gewichteten Graphen durch die Berechnung der kürzesten Wege verursacht. Da das Single-Source-Shortest-Path-Problem für jeden Knoten des Netzwerkes gelöst werden muss, ergibt sich insgesamt, ein Aufwand von $\mathcal{O}(n^2 log(n) + nm)$ für den gewichteten Graphen und ein Aufwand von $\mathcal{O}(nm)$ für den ungewichteten Graphen. In dieser Arbeit werden nur ungerichtete, ungewichtete Graphen betrachtet. Aufgrund der recht hohen Laufzeit von Brandes Algorithmus, findet man in der Literatur einige Lösungsmöglichkeiten, um die Betweenness auch für sehr große Graphen berechnen zu können. Dies wird in [GSS08] durch einen approximativen Ansatz gelöst, denkbar ist es aber auch parallele Varianten zu entwickeln. Eine Idee, die auch in dieser Arbeit weiterverfolgt wird ist es, den Graphen zu reduzieren [BGPL11], und den Algorithmus danach auf einer kleineren Instanz und somit mit weniger Aufwand durchzuführen.
Um die Betweenness einer Gruppe C auszurechnen, betrachtet man Wege von einem Startknoten s zu einem Zielknoten t, die über mindestens einen Knoten der Gruppe führen. Die Anzahl dieser Wege wird mit $\sigma_{st}(C)$ bezeichnet. Die Zentralität der Gruppe läßt sich dann wie folgt berechnen:

$$\mathcal{GB}(C) := \sum_{s,t \in V \setminus C | s \neq t} \left(\frac{\sigma_{st}(C)}{\sigma_{st}} \right) \tag{1.3}$$

Die *Gruppenbetweenness* beschreibt den Anteil der Informationswege, die über mindestens einen Knoten der betrachteten Gruppe führen. Die Betweenness einer einzelnen Gruppe der Größe k, kann durch eine Variante von Brandes Algorithmus aus [Bra01] berechnet werden [Bra08]. Für die Problemstellung eine Gruppe mit maximaler Gruppenbetweenness zu finden, wird in der Literatur das Verfahren aus [PES07] vorgeschlagen, welches sich dadurch auszeichnet die Gruppenbetweenness von mehreren Gruppen relativ schnell berechnen zu können. Das Verfahren basiert auf der Verwendung der *Paarbetweenness* $\mathcal{PB}(a, b)$, welche den Anteil der kürzesten Wegen zwischen allen Knotenpaaren beschreibt, die das geordnete Knotenpaar (a, b) enthalten. Die Anzahl der kürzesten Wege von einem Startknoten s zu einem Zielknoten t welche gleichzeitig über das geordnete Knotenpaar (a, b) führen, wird mit $\sigma_{st}(a, b)$ bezeichnet. Die Paarbetweenness $\mathcal{PB}(a, b)$ zweier Knoten a und b berechnet sich dann wie folgt:

$$\mathcal{PB}(a, b) := \sum_{s,t \in V \setminus \{a,b\}} \left(\frac{\sigma_{st}(a, b)}{\sigma_{st}} \right) \tag{1.4}$$

Der Algorithmus aus [PES07] beruht auf einer recht aufwendigen Vorberechnung. Diese besteht zum einen darin den Algorithmus von Brandes für die Berechnung der Betweenness [Bra01] einmalig durchzuführen und alle Distanzen $d(s, t)$, die Anzahl der kürzesten Wege σ_{st} und die Dependencies $\sigma_{s\bullet}(v)$ zu speichern, was Speicherplatz im Bereich von $\mathcal{O}(n^2)$ erfordert. In in einem weiteren Schritt muss die Paarbetweenness $\mathcal{PB}(a, b)$ für alle geordneten Knotenpaare (a, b) des Netzwerkes berechnet werden. Mit der Methode aus [PES07] wird dadurch ein Rechenaufwand im Bereich von $\mathcal{O}(n^3)$ verursacht, wenn folgende Formel verwendet wird:

$$\mathcal{PB}(a, b) := \sum_{s \in V} \delta_{s\bullet}(b) \left(\frac{\sigma_{sb}(a)}{\sigma_{sb}} \right) \tag{1.5}$$

Die Berechnung pro Knotenpaar geht in $\mathcal{O}(n)$, wenn die Paarbetweenness für je zwei Knoten des Netzwerkes berechnet wird, resultiert daraus die oben angegebene kubische Laufzeit. Sind die Werte einmal vorberechnet, kann die Gruppenbetwenness einer Gruppe der Größe k in $\mathcal{O}(k^3)$ berechnet werden.

Motivation

Beschleunigung der Berechnung der Paarbetweenness aller Knotenpaare eines Netzwerkes. Die Paarbetweenness kann unter anderem zur Berechnung der Gruppenbetweenness verwendet werden.

In [KCB09] wird ein Algorithmus beschrieben, der die Paarbetwenness von je zwei Knotenpaaren, auf ungewichteten Graphen, in einer Laufzeit von $\mathcal{O}(nm + n^{2+p}logn)$ berechnet. Der Algorithmus basiert ebenfalls auf der Technik des Algorithmus von Brandes. Nachdem von einem Startknoten s aus, alle kürzesten Wege berechnet worden sind, wird der Anteil aller Wege, die zu einem Zielknoten t führen, und dabei das geordnete Knotenpaar (a, b) überqueren, zur Paarbetweenness $\mathcal{PB}(a, b)$ aufaddiert. Dafür ist es allerdings erforderlich alle im Graph vorkommenden kürzesten Wege, zwischen je zwei Knotenpaaren (s, t), vom Zielknoten t zum Startknoten s zurück zu verfolgen. Zur Laufzeitanalyse wurde die Anzahl der kürzesten Wege, mit n^{2+p} abgeschätzt, wobei der Wert p von den Autoren empirisch gefunden wurde. Die maximale Länge eines kürzesten Weges wird mit $\log n$ abgeschätzt, was allerdings nur für Smallworld-Graphen zutrifft.
In dieser Arbeit wird das Verfahren aus [Sch09] verwendet (siehe Anhang: 6). Dort wird durch geschickte Verwendung einiger Eigenschaften von kürzesten Wegen ein Verfahren zur Berechnung der Paarbetweenness $\mathcal{PB}(a, b)$, für alle geordneten Knotenpaare (a, b) eines Netzwerkes, verwendet, welches eine Laufzeit von $\mathcal{O}(nm + n^2 \log n)$ erzielt. Das Verfahren basiert darauf, Brandes Algorithmus zwei mal auf dem gesamten Netzwerk auszuführen. Im ersten Durchlauf werden die Dependencies zwischen allen Knotenpaaren des Netzwerkes berechnet und gespeichert. Mit diesen Werten kann dann im zweiten Durchlauf die Paarbetweenness $\mathcal{PB}(a, b)$ eines Startknotens b zu allen Knoten a im Netzwerk berechnet werden. Im folgenden wird in Anlehnung an [Sch09] ein kürzester Weg der von einem Knoten s aus über einen Knoten a zu einem Knoten t führt als $\mathrm{s} - \mathrm{a} - \mathrm{t} -$ Weg bezeichnet. $P^-(a, b)$ ist dann, ebenfalls in Anlehnung an [Sch09], die Menge der Knoten s, für die ein $\mathrm{s} - \mathrm{a} - \mathrm{b} -$ Weg existiert. Analog beschreibt die Menge $P^+(a, b)$ alle Knoten t, die auf einem kürzesten $\mathrm{a} - \mathrm{b} - \mathrm{t} -$ Weg liegen. Mit diesen Definitionen kann die Paarbetweenness $\mathcal{PB}(a, b)$ zweier Knoten a und b auf folgende Weise berechnet werden:

$$PB(a, b) = \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}} \tag{1.6}$$

Der Beweis ist in [Sch09] nachzulesen.
Es bleiben dennoch zwei Nachteile, zum einen erfordert die Durchführung des Algorithmus von Brandes eine relativ hohe Laufzeit und zum anderen liegt der benötigte Speicheraufwand in $\mathcal{O}(n^2)$.
In dieser Arbeit wird deshalb untersucht, ob sich eine Reduktion des Graphen auf den

2-Core positiv auf diese Parameter auswirkt. Weiterhin wird untersucht welche Laufzeit sich insgesamt ergibt, wenn der Graph in einem ersten Schritt reduziert und die Paarbetweenness aller Paare im Netzwerk anschließend auf einer kleineren Instanz berechnet wird. Die Idee, den Graph auf seinen 2-Core zu reduzieren um den eigentlichen Algorithmus auf einer kleineren Instanz auszuführen und dadurch Laufzeit zu sparen, ist [BGPL11] entnommen. Dort wird diese Technik zur Beschleunigung der Berechnung der Betweenness verwendet. Das Verfahren kann als eine Variante des Algorithmus von Brandes [Bra01] betrachtet werden. In dieser Arbeit soll untersucht werden ob sich dieses Verfahren auch für die Berechnung der Paarbetweenness eignet.

2. Speichereffiziente Berechnung der Paarbetweenness

Im folgenden wird eine Reduktion des Graphen auf seinen 2-Core durchgeführt, mit dem Ziel, den Algorithmus aus [Sch09] zur Berechnung der Paarbetweenness von je zwei Knoten des Netzwerkes auf einer kleineren Instanz auszuführen (siehe Anhang: 4, 5). Was man will, ist die Paarbetweenness von allen Knotenpaaren des ursprünglichen Netzwerkes. Deshalb ist die Reduktion nur sinnvoll, wenn auch die Paarbetweenness $\mathcal{PB}(a, b)$ von zwei Knoten a und b, bei denen mindestens einer nicht im 2-Core des Graphen liegt, mit möglichst geringem Aufwand berechnet werden kann. Dass dies möglich ist wird in den Unterabschnitten von 2.2 gezeigt.
Da das Verfahren, welches in dieser Arbeit untersucht wird, auf der Arbeit von [BGPL11] zur Beschleunigung der Betweenness aufbaut, wird dieses in Abschnitt 2.1 noch mal erklärt. Die beiden Varianten können problemlos kombiniert werden, so dass man die Betweenness von allen Knoten im Netzwerk und die Paarbetweenness $\mathcal{PB}(a, b)$ von allen Knotenpaaren des Netzwerkes gleichzeitig berechnen kann.
In Abschnitt 2.2 wird gezeigt, wie die Reduktion erfolgen muss, um alle Informationen zu erhalten, die zur Berechnung der Paarbetweenness auch außerhalb des 2-Cores, benötigt werden. Weiterhin wird beschrieben, welche Modifikationen im Algorithmus aus [Sch09] notwendig sind, wenn die Berechnung der Paarbetweenness auf Paare, die im 2-Core liegen, reduziert wird, aber auch Wege berücksichtigt werden sollen, die von und zu Knoten führen, welche außerhalb des 2-Cores liegen. Am Ende des Abschnittes wird dann auf die Laufzeit und den Speicherverbrauch des Algorithmus eingegangen.

2.1. Schnellere Berechnung der Betweenness

Da die Berechnung der Betweenness sehr zeitaufwändig ist, sucht man nach Lösungen diese zu beschleunigen. In [BGPL11] wird der Graph dafür auf seinen 2-Core reduziert (siehe Anhang: 2, 3). Der 2-Core eines Graphen ist diejenige Knotenmenge, die übrig bleibt, wenn man iterativ alle Knoten mit Grad 1 entfernt. Knoten die außerhalb des

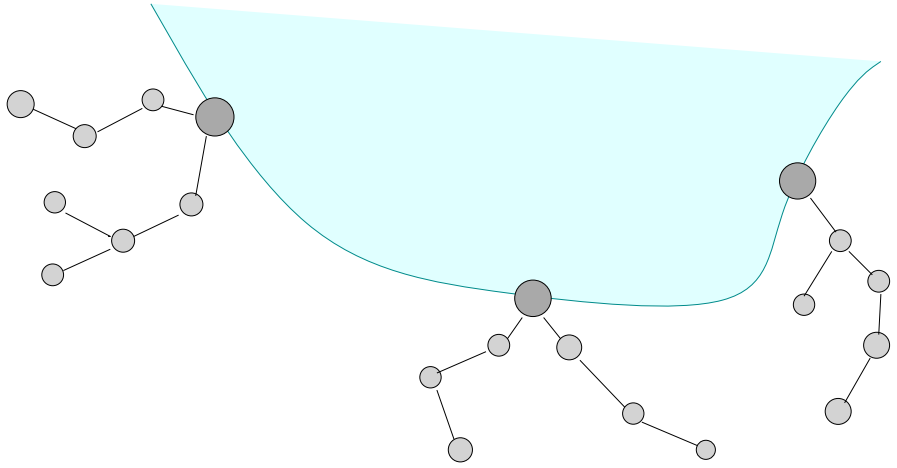

Abbildung 2.1.: Der 2Core eines Graphen entsteht, wenn iterativ Knoten mit Grad 1, entfernt werden. Diese Knotenmenge wird in dieser Arbeit mit T bezeichnet und besteht aus den Baumfortsätzen des Graphen. Jeder Baumfortsatz ist durch einen Wurzelknoten, hier großen dunklen Knoten, mit dem 2Core verbunden.

2-Cores liegen, werden im folgenden der Knotenmenge T zugeordnet. Der Kern des Algorithmus ist die iterative Berechnung der Betweenness $\mathcal{B}(v)$ eines Knotens v aus T bei der Reduktion auf den 2-Core, so dass Brandes Algorithmus auf einer kleineren Instanz ausgeführt werden kann. Wege, die von einem Knoten s im 2-Core zu einem Knoten t in T führen und umgekehrt, können durch leichte Modifikation des Algorithmus von Brandes zur Betweenness der Knoten im 2-Core addiert werden [BGPL11]
Da hier nur zusammenhängende Graphen betrachtet werden, sind alle Knoten v aus T Teil eines *Baumfortsatzes* tree(v), welcher durch eine *Wurzel* w mit dem 2-Core verbunden ist (siehe Abb.: 2.1). Die Wurzel w ist selbst Teil des 2-Cores. Allen Knoten v aus T wird mit $w(v)$ eine Wurzel zugeordnet. Auf der Menge V der Knoten, wird im folgenden eine Funktion p definiert. Einem Knoten v aus T und den Wurzelknoten, wird durch $p(v)$ die Anzahl ihrer Nachfolger zugeordnet. Diese wird ebenfalls iterativ während der Reduktion des Graphen auf den 2-Core berechnet. Für alle Knoten im 2-Core, außer den Wurzelknoten, gilt $p(v) = 0$. Entsprechend wird jedem Knoten v, der Teil eines Baumes ist, mit $T(v)$ die Knotenmenge zugeordnet, die den Teilbaum bildet, welcher in v wurzelt. Die Betweenness der Knoten aus T wird bei der Reduktion iterativ berechnet. Im folgenden wird der direkte Nachfolger eines Knotens v, welcher oberhalb von v, auf dem Pfad zur Wurzel $w(v)$ liegt, als Vaterknoten $a(v)$ bezeichnet. Wenn ein Knoten v iterativ entfernt wird, wird die Anzahl der kürzesten Wege, welche über v und $a(v)$ führen zur Betweenness von $a(v)$ addiert. Da die Wege in einem Baumfortsatz eindeutig sind, deckt man somit alle Wege, die von Startknoten aus $T(v)$ stammen ab. Die Endknoten dieser Wege liegen also entweder in $T(v)$ oder sind von v aus betrachtet über $a(v)$ erreichbar (siehe Abb.: 2.2. Die Anzahl der kürzesten Wege, die über den Knoten v und Vaterknoten $a(v)$ führen, wird bei der iterativen Entfernung des Knotens v auf die Betweenness

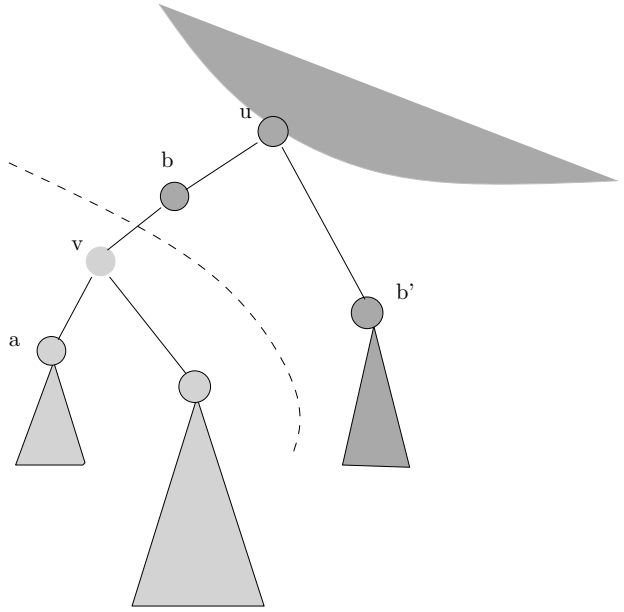

Abbildung 2.2.: Bei der Berechnung von $\mathcal{PB}(a, b)$ liegen die Startknoten der Wege, die das Paar (a, b) enthalten in $T(a)$, dem Teilbaum der in a wurzelt. Zielknoten sind genau die Knoten oberhalb der gestrichelten Linie. Diese Menge erhält man, wenn man von der Gesamtknotenmenge V, den Teilbaum $T(v)$ entfernt. $\mathcal{PB}(a, b')$ besteht aus Wegen die das Paar (a, b') enthalten. Die Startknoten dieser Wege befinden sich in $T(a)$ und die Zielknoten in $T(b')$. Von einem Startknoten in T(a) führen $|T(b')|$ Wege nach b'.

$\mathcal{B}(u)$ von $a(v) = u$ addiert:

$$\mathcal{B}(u) = \mathcal{B}(u) + 2 * (p(v) * (V \setminus u \setminus T(v)) - p(u)) \tag{2.1}$$

Wenn man von der gesamten Knotenmenge V $a(v)$ und die Knoten, die in $T(v)$ liegen, entfernt, erhält man die Menge der Knoten, welche von v aus über $a(v)$ erreichbar sind. Da hier ungerichtete Graphen betrachtet werden, können Start - und Endknoten vertauscht werden. Deshalb wird die Anzahl der kürzesten Wege, die von v aus über $a(v)$ führen mit 2 multipliziert. Um Wege, die über $a(v)$ zu bereits entfernten Knoten führen nicht doppelt zu zählen muss diese Anzahl, welche in $p(u)$ gespeichert ist, von der Menge der Endknoten, die von v aus betrachtet auf der anderen Seite von $a(v)$ liegen, subtrahiert werden.

2.2. Platzsparendere Berechnung der Paarbetweenness

Im folgenden wird analog zu [BGPL11] eine Reduktion des Graphen auf den 2-Core durchgeführt. Es wird gezeigt, dass man durch eine zusätzliche Vorberechnung, die einen

Aufwand in $\mathcal{O}(m)$ erfordert, den Speicherverbrauch für die Berechnung der Paarbetweenness $\mathcal{PB}(a,b)$ zwischen je zwei Knoten a und b des Netzwerkes deutlich reduzieren kann. Dies kann erreicht werden, indem man den Algorithmus aus [Sch09] für die Berechnung der Paarbetweenness zwischen allen im Netzwerk vorkommenden Paaren, auf einer kleineren Instanz, dem 2-Core ausführt. Es ist möglich die Paarbetweenness von Paaren (a,b) mit mindestens einem Knoten in T zu jeder Zeit mit relativ geringem Aufwand zu ermitteln. wenn man die vorberechneten Werte verwendet.
Im folgenden wird gezeigt wie man die Paarbetweenness $\mathcal{PB}(a,b)$ eines geordneten Paares (a,b) des Netzwerkes speichereffizient berechnen kann, indem man den Graphen auf seinen 2-Core reduziert. Dadurch entstehen verschiedene Rechenvorschriften für die folgenden Paarungen:

- $\mathcal{PB}(a,b)$, $a,b \in T$ und $w(a) = w(b)$
- $\mathcal{PB}(a,b)$, a und b liegen im 2-Core des Graphen
- $\mathcal{PB}(a,b)$, a liegt im 2-Core und b in T oder umgekehrt
- $\mathcal{PB}(a,b)$, $a,b \in T$ und $w(a) \neq w(b)$

In den folgenden Abschnitten wird nun gezeigt, wie die Paarbetweenness für verschiedene Paarungen effizient berechnet werden kann. Dabei wird auch auf den Speicherbedarf und die Laufzeit der Algorithmen eingegangen.

2.2.1. Betrachtung von Paaren (a,b), mit $a,b \in T$ und $w(a) = w(b)$

Die Berechnung der Paarbetweenness von Knoten in T, die im gleichen Baumfortsatz liegen kann durch die Vorberechnung bestimmter Werte zu jeder Zeit und mit relativ geringem Aufwand berechnet werden. Im folgenden werden zwei verschiedene Fälle von Paaren (a,b) unterschieden. Im ersten Fall liegt das Paar (a,b) auf einem gemeinsamen Pfad zur Wurzel, mit $w(a) = w(b)$. Der zweite Fall behandelt Paare (a,b), für die das nicht zutrifft. Im ersten Fall muss man für die Berechnung von $\mathcal{PB}(a,b)$ den Knoten v, mit $a(v) = b$ kennen. Insbesondere benötigt man die Anzahl der Nachfolger $t(v) :=| T(v) \setminus \{v\} |$ von v . Dann gilt für die Paarbetweenness $\mathcal{PB}(a,b)$ des geordneten Knotenpaares (a,b):

Lemma 1. *Seien a,b und v Knoten in T und $a(v) = b$, w sei Wurzel mit $w(a) = w(b)$, weiterhin gibt es einen $a-b-w-Weg$ in T, dann lässt sich die Paarbetwenness $\mathcal{PB}(a,b)$ der Knoten a und b durch folgende Formel berechnen:*

$$\mathcal{PB}(a,b) = \mathcal{PB}(b,a) = t(a) * |V \setminus T(v) \setminus b| \tag{2.2}$$

Proof. Betrachtet man s-a-b-t-Wege, dann erkennt man mit Formel 1.6, dass Startknoten genau die Knoten in T(a) sind. Zielknoten liegen von a aus betrachtet "auf der anderen Seite" von b. Deshalb erhält man die Anzahl der Zielknoten, indem man von der Summe aller Knoten des Netzwerkes diejenigen abzieht, die erreicht werden können ohne b passieren zu müssen (siehe Abb.: 2.1). Das sind genau die Knoten in T(v). Die Anzahl der Paarungen von Start- und Zielknoten, erhält man durch Multiplikation der beiden Mengen. □

Der zweite Fall betrachtet Paare (a, b), die nicht auf einem gemeinsamen Pfad zur Wurzel liegen. Wege innerhalb von T sind immer eindeutig (siehe Abb.: 2.2) und Wege, die das Paar (a, b) enthalten beginnen in $T(a)$ und enden in $T(b)$.

Lemma 2. *Seien a und b Knoten in T. Die Anzahl der Nachfolger des Knotens a sei $t(a)$, $t(b)$ analog. Dann lässt sich die Paarbetwenness $\mathcal{PB}(a, b)$ von a und b wie folgt berechnen:*

$$\mathcal{PB}(a, b) = t(a) * t(b) \tag{2.3}$$

Proof. Wege in einem Baumfortsatz haben die Eigenschaft eindeutig zu sein. Der Weg zwischen a und b führt über u. Betrachtet man s-a-b-t-Wege, dann beginnen diese in t(a) und enden in t(b). Die Anzahl der Kombinationen verschiedener Start-Endknoten-Paare erhält man durch Multiplikation der beiden Werte. □

Man erkennt dass die Formeln zur Berechnung der Paarbetweenness sich auf wenige Werte reduzieren lassen. Die meisten können durch eine Vorberechnung gewonnen werden und erfordern einen Speicherplatz, der linear in der Anzahl der Knoten in T ist. Man erkennt, dass es notwendig ist die Anzahl der Nachfolger aller Knoten aus T zu kennen. Diese kann analog zu [BGPL11] bei der Reduktion des Graphen auf den 2-Core, iterativ, berechnet werden. Formel 2.3 gilt für den Fall, dass zwei Knoten a und b auf unterschiedlichen Pfade zur Wurzel $w(a) == w(b)$ liegen. Wenn die Werte einmal vorberechnet sind kann diese in konstanter Zeit angewendet werden. Wenn die Knoten a und b einen gemeinsamen Pfad zur Wurzel haben, wird Formel 2.2 für die Berechnung der Paarbetweenness $\mathcal{PB}(a, b)$ verwendet. Da der Knoten v, welcher für die Berechnung benötigt wird, nicht von vorne herein bekannt ist und erst ermittelt werden muss, kann diese Paarbetweenness nicht in konstanter Zeit berechnet werden. Nimmt man an, dass der Knoten b die kleinere Distanz zur Wurzel hat, dann gilt $a(v) = b$ und es gibt einen a-v-b-Weg. Um Knoten v zu finden Bedarf es der Vorberechnung weiterer Werte.
Da es zwei verschiedene Formeln gibt, ist es außerdem erforderlich zu wissen welcher der beiden Fälle vorliegt. Man muss also mit möglichst wenig Aufwand für zwei Knoten a und b herausfinden, ob der Knoten a in $T(b)$ liegt, oder umgekehrt. In diesem Fall gibt es einen gemeinsamen Pfad zum Wurzelknoten und Formel 2.2 wird verwendet. Andernfalls wird die Paarbetweenness von Knoten a und Knoten b mit Formel 2.3 berechnet. Die Idee ist, eine DFS-Nummerierung [KP08] für jeden Baumfortsatz zu erstellen. Diese wird durch eine Tiefensuche von jedem Wurzelknoten nach T erstellt und erfordert somit einen einmaligen Rechenaufwand im Bereich von $\mathcal{O}(m)$.
Im folgenden wird einem Knoten v mit dfs_num(v) seine DFS-Nummer zugeordnet. Mit der DFS-Nummerierung kann für zwei Knoten a und b in konstanter Zeit geprüft werden, ob es einen gemeinsamen Pfad zur Wurzel $w(a) = w(b)$ gibt. Anschließend kann die entsprechende Formel zur Berechnung der Paarbetweenness verwendet werden.

Lemma 3. *Seien a, b und v Knoten in T, für Knoten a und b, gelte* dfs_num(a) > dfs_num(b)*, außerdem sei $a(v) = b$, dann kann die Paarbetweenness $\mathcal{PB}(a, b)$, mit 2.3 und 2.2 wie folgt berechnet werden:*

$$\mathcal{PB}(a, b) = \begin{cases} t(a) * |V - T(v) - b| & , \text{dfs\_num}(b) + t(b) \geq \text{dfs\_num}(a) \\ t(a) * t(b) & , sonst \end{cases}$$

Proof. Wenn b bei der Tiefensuche zum ersten mal besucht wird, wird dessen DFS - Nummer gesetzt, im weiteren Verlauf werden alle Knoten in $T(b)$ besucht und durchnummeriert, wobei die DFS-Nummer jedes mal inkrementiert wird. Der letzte Knoten vor dem Verlassen von $T(b)$ hat demnach den Wert $\text{dfs\_num}(b) + t(b)$ erhalten. □

Will man Enknoten bei der Berechnung der Betweenness berücksichtigen, kann die Formel wie folgt angepasst werden:

$$\mathcal{PB}(a,b) = \begin{cases} (t(a)+1) & * & |V - T(v)| & , \text{dfs\_num}(b) + t(b) \geq \text{dfs\_num}(a) \\ (t(a)+1) & * & (t(b)+1) & , sonst \end{cases}$$

Mit der DFS-Nummerierung lässt sich jetzt auch Knoten v ermitteln, welcher für die Berechnung der Paarbetwenness mit Formel 2.2 benötigt wird. Dazu ist es erforderlich, Knoten b habe k Kanten, über alle ausgehenden Kanten von b, mit $a(v) = b$ zu iterieren. Der gesuchte Knoten v ist derjenige zu b adjazente Knoten, welcher die Bedingung $\text{dfs\_num}(v) + t(v) \quad \geq \quad \text{dfs\_num}(a)$ erfüllt.

Theorem 1. *Für zwei Knoten a und b aus T gelte* $\text{dfs\_num}(a) > \text{dfs\_num}(b)$*, dann lässt sich die Paarbetweenness $\mathcal{PB}(a,b)$ des geordneten Paares (a,b), mit Formel 2.2 und 2.3 in einer Laufzeit von $\mathcal{O}(k)$ berechnen.*

Die benötigten Werte werden bei der Reduktion des Graphen auf den 2-Core und einer Tiefensuche, in der die DFS - Nummerierung erstellt wird, berechnet. Der Aufwand für die Vorberechnung liegt also im Bereich von $\mathcal{O}(m)$. Um einen Knoten v vollständig charakterisieren zu können, benötigt man zusätzlich die Information darüber, welchem *Baumfortsatz* $\text{tree}(v)$ dieser angehört. Dies kann innerhalb der Tiefensuche, in der die DFS-Nummerierung erstellt wird, ermittelt werden.
Insgesamt benötigt man für die in der Vorberechnung gewonnenen Werte einen Speicherplatz, der linear in der Anzahl der Knoten n des Graphen ist. Gespeichert werden müssen die DFS-Nummer, die Id des Baumfortsatzes und die Anzahl der Nachfolger, für jeden Knoten in T.

Theorem 2. *Durch die Berechnung der Paarbetweeness $\mathcal{PB}(a,b)$ von zwei Knoten aus T in Echtzeit, spart man Speicherplatz, der quadratisch in $|T|$ ist.*

Im folgenden wird gezeigt welche Modifikationen im Algorithmus aus [Sch09] notwendig sind um korrekte Paarbetweennesswerte $\mathcal{PB}(a,b)$ von zwei Knoten a und b aus dem 2-Core zu erhalten.

2.2.2. Betrachtung von Paaren (a,b), mit $a, b \in 2 - \text{Core}$

Der Algorithmus, der hier für die Berechnung der Paarbetweenness verwendet wird stammt aus [Sch09] und basiert auf Formel 1.6. Diese kann im ungerichteten Graphen auf die folgende Weise umgestellt werden:

$$PB(a,b) = \sum_{s \in P^+(b,a)} \frac{\sigma_{bs}(a)}{\sigma_{bs}} * \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}} \tag{2.4}$$

Die Formel berechnet die Paarbetweenness aller geordneten Knotenpaaren (a, b) des Netzwerkes, wobei in dieser Schreibweise $s - a - b - t -$ Wege betrachetet werden. Die erste Summe addiert alle *Pairdependencies* des Knotens a bezüglich einem Startknoten s und Knoten b. Die Pairdependency beschreibt den Anteil aller kürzesten Wege, die von einem Startknoten s zu einem bestimmten Zielknoten t führen, und ist wie folgt definiert [Bra01]:

$$\delta_{st}(v) := \frac{\sigma_{st(v)}}{\sigma_{st}} \tag{2.5}$$

Man erkennt dass die Paarbetweenness die Summe über alle Pairdependencies des Knotens a bezüglich b eines Startknotens s, multipliziert mit der Dependency von b bezüglich des Knotens s ist. In [Sch09] wird gezeigt dass man durch Initialisierung jedes Knotens t mit einem Wert $f(t)$ und anschließender Durchführung von Brandes Algorithmus mit Startknoten s eine *modifizierte* Dependency $\delta^{\text{mod}}_{s,t}$ jedes Knotens t bezüglich des Startknotens s erhält:

$$\delta^{\text{mod}}_{s\bullet}(v) = \sum_{t \in V} \frac{\sigma_{st(v)}}{\sigma_{st}} * f(t) \tag{2.6}$$

Die Paarbetweenness zweier Knoten a und b kann mit 2.4 und 2.6, durch zweimalige Durchführung von Brandes Algorithmus berechnet werden. Im ersten Durchlauf berechnet man die dependencies von je zwei Knoten des Netzwerkes. Wenn b im zweiten Durchlauf von Brandes Algorithmus Startknoten ist wird die Dependency des Knotens s, mit $\delta_{s\bullet}(b)$ initialisiert. Diese hat man ja im ersten Durchlauf von Brandes Algorithmus berechnet. Im zweiten Durchlauf des Algorithmus erhält man jetzt die Paarbetweenness des Startknotens b und allen anderen Knoten im Netzwerk.
Per Definition ist die Paarbetweenness von zwei Knoten a und b, der Anteil von allen kürzesten Wegen zwischen je zwei Knoten eines Netzwerkes, auf denen a und b liegen. Um auch kürzeste Wege betrachten zu können, die von Knoten in T ausgehen, oder zu Knoten in T führen, muss der Algorithmus aus [Sch09] leicht modifiziert werde. Zuerst wird gezeigt, wie sich die Berechnung der Dependency auf den kontrahierten Graphen anpassen lässt. Für die Dependency auf dem reduzierten Graphen gilt dann:

Lemma 4. *Sei V die Menge der Knoten des Originalgraphen und $V \setminus T$ der 2-Core des Graphen. Sei Knoten s Startknoten für das Single-Source-Shortest-Path-Problem. Die Dependency $\delta_{s\bullet}(v)$ eines Knotens v bezüglich eines Startknotens s lässt sich auf folgende Weise berechnen:*

$$\delta_{s,\bullet}(v) = \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} * (t(t) + 1) \tag{2.7}$$

Proof.

$$\begin{aligned}
\delta_{s,\bullet}(v) &\overset{1.2}{=} \sum_{t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \\
&= \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{t \in T} \frac{\sigma_{st}(v)}{\sigma_{st}} \\
&= \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{t \in T} \frac{\sigma_{sw(t)}(v) * \sigma_{w(t)t}}{\sigma_{sw(t)} * \sigma_{w(t)t}} \\
&= \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{t \in T} \frac{\sigma_{sw(t)}(v)}{\sigma_{sw(t)}}
\end{aligned}$$

Im ersten Schritt werden Wege, die im 2-Core enden, von Wegen, die nach T führen, getrennt. In Schritt 2 wird verwendet, dass alle Knoten nach T über den entsprechenden Wurzelknoten laufen und Knoten v sich im 2-Core befindet. Deshalb entspricht die Pairdependency des Knotens v bezüglich s und einem Knoten $t \in T$ der Pairdependency dieses Knotens bezüglich des Startknotens und der Wurzel w(t) und kann in Zeile 4 entspechend ersetzt werden.

$$\begin{aligned}
\delta_{s,\bullet}(v) &= \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{u \in U} t(u) * \frac{\sigma_{su}(v)}{\sigma_{su}} \\
&= \sum_{t \in V \setminus T} \frac{\sigma_{st}(v)}{\sigma_{st}} * (t(t) + 1)
\end{aligned}$$

Eine korrekte Dependency enthält alle Pairdependencies eines Knotens v bezüglich des Startknotens s und allen Zielknoten t, wenn es einen kürzesten s-v-t Weg gibt. Da $\delta_{st}(v) = \delta_{sw(t)}(v)$ für alle Knoten $t \in T$ gilt, enthält $\delta_{s\bullet}(v)$ insgesamt t(w(t)) + 1 mal den Wert der Pairdependency einer Wurzel w(t). □

Im folgenden sei $U := \{u \mid \exists t \in P^-(a,b), u = w(t)\}$, dann kann die Paarbetweenness wie folgt berechnet werden:

Lemma 5. *Seien a und b Knoten im 2-Core, weiter sei $u \in U$, dann gilt für die Paarbetweenness $PB(a,b)$ von a und b:*

$$PB(a,b) = \sum_{s \in P^+(b,a) \setminus T} \frac{\sigma_{bs}(a)}{\sigma_{bs}} \quad * \quad \delta_{s\bullet}(b) \quad * \quad (t(s)+1) \tag{2.8}$$

Proof.

$$\begin{aligned} PB(a,b) &\overset{1.6}{=} \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}} \\ &= \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \quad \delta_{s\bullet}(b) \end{aligned}$$

Der erste Schritt verwendet dass der zweite Faktor genau die Dependency des Knotens b bezüglich des Knotens s ist.

$$\begin{aligned} PB(a,b) &= \sum_{s \in P^-(a,b) \setminus T} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \quad \delta_{s\bullet}(b) \quad + \quad \sum_{s \in P^-(a,b) \cup T} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \quad \delta_{s\bullet}(b) \\ &= \sum_{s \in P^-(a,b) \setminus T} \frac{\sigma_{sb}(a)}{\sigma_{sb}} \quad * \quad \delta_{s\bullet}(b) + \sum_{s \in P^-(a,b) \cap T} \frac{\sigma_{sw(s)}\sigma_{w(s)b}(a)}{\sigma_{sw(s)}\sigma_{w(s)b}} \quad * \quad \delta_{w(s)\bullet}(b) \end{aligned}$$

Im zweiten Schritt werden die Pairdependencies des Knotens a bezüglich Startknoten aus dem 2-Core von Pairdependencies von a bezüglich eines Startknotens aus T und dem Zielknoten b getrennt. Danach wird gezeigt, dass die Pairdependency bezüglich eines Startknoten s aus T mit der pairdependency bezüglich w(s) übereinstimmt.

$$\begin{aligned} PB(a,b) &= \sum_{s \in P^-(a,b) \setminus T} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \quad \delta_{s\bullet}(b) \quad + \quad \sum_{u \in U} \frac{\sigma_{ub}(a)}{\sigma_{ub}} * \quad \delta_{u\bullet}(b) * t(u) \\ &= \sum_{s \in P^-(a,b) \setminus T} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \quad \delta_{s\bullet}(b) * \quad (t(s)+1) \\ &= \sum_{s \in P^+(b,a) \setminus T} \frac{\sigma_{bs}(a)}{\sigma_{bs}} * \quad \delta_{s\bullet}(b) * \quad (t(s)+1) \end{aligned}$$

Da die modifizierte Dependency eines Knotens $s \in T$ der Dependency von $w(s) = u$ entspricht, wird diese in Schritt 4 ersetzt. Die Dependency jedes Wurzelknotens muss deshalb insgesamt, $t(s)$ ist Anzahl der Nachfolger von s, $t(s)+1$ mal aufsummiert werden. Für alle Knoten s mit $s \notin U$ gilt $t(s) = 0$. □

2.2.3. Betrachtung von Paaren (a,b) mit $a \in T \leftrightarrow b \notin T$

Bei der Berechnung der Paarbetweenness $\mathcal{PB}(a,b)$ des geordneten Knotenpaares (a,b), mit genau einem Knoten im 2-Core, sind zwei Richtungen zu betrachten. Im folgenden wird gezeigt, wie sich die Berechnung der Werte unter Ausnutzung der Graphstruktur vereinfacht und welche Werte dafür bei der Durchführung der Berechnung der Paarbetweenness auf dem 2-Core gespeichert werden müssen.

2.2.3.1. erste Richtung: vom 2-Core nach T

Für die Berechnung der Paarbetweenness zwischen einem Knoten a aus dem 2-Core und einem Baumknoten b aus T werden der Anteil aller kürzesten Wege, auf denen b liegt, mit dem Anteil von allen kürzesten Wegen zum Knoten b, die über a laufen, multipliziert. Da alle Pfade über die Wurzel laufen und es innerhalb des Baumes einen eindeutigen Weg von der Wurzel zum Knoten b gibt, ist es möglich, die Paarbetweenness $\mathcal{PB}(a,b)$ durch den Anteil aller kürzesten Wege zur Wurzel u, die über a führen, und $t(b)$ zu berechnen. Denn $t(b)$ ist die Anzahl der Nachfolger des Knotens b und somit Endknoten der kürzesten Wege, die b enthalten.

Lemma 6. *Sei a ein Knoten im 2-Core und b ein Knoten aus T, weiter sei u = w(b), dann lässt sich die Paarbetweenness $\mathcal{PB}(a,b)$ auf folgende Weise berechnen:*

$$PB(a,b) = \delta_{u,\bullet}(a) * t(b) \tag{2.9}$$

Proof.

$$\begin{aligned} PB(a,b) &\overset{1.6}{=} \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}} \\ &= \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * t(b) \end{aligned}$$

Der erste Vereinfachungsschritt, ergibt sich dadurch, dass Zielknoten in T genau die Nachfolger des Knotens b sind.

$$\begin{aligned} PB(a,b) &= \sum_{s \in P^-(a,u)} \frac{\sigma_{su}(a)}{\sigma_{su}} * t(b) \\ &= t(b) * \sum_{s \in P^-(a,u)} \frac{\sigma_{su}(a)}{\sigma_{su}} \\ &= t(b) * \sum_{s \in P^+(u,a)} \frac{\sigma_{us}(a)}{\sigma_{us}} \end{aligned}$$

Da alle Wege nach b über die Wurzel führen, der Weg von der Wurzel u nach b eindeutig ist, und a im 2-Core liegt, kann man b durch u ersetzen (Schritt 2). Da t(b) unabhängig von s ist, lässt sich dieser Faktor vor die Summe ziehen. In ungerichteten Graphen gibt es Pfade immer in beide Richtungen, weshalb Schritt 4 erlaubt ist. Man erkennt jetzt dass die verbleibende Summe genau die Dependency des Knotens a bezüglich der Wurzel $w(b) = u$ ist. □

Da in dieser Arbeit Endknoten bei der Betwenness berücksichtigt werden, wird die Formel wie folgt angepasst:

$$P(a,b) = \delta_{u,\bullet}(a) * (t(b)+1)$$

Innerhalb der Dependency werden die Endknoten berücksichtigt, wenn man vor der Berechnung des Single-Source-Shortest-Path-Problems, die Dependencies aller Knoten v mit 1 initialisiert.

2.2.3.2. zweite Richtung: von T in den 2-Core

Im Gegensatz zum letzen Kapitel werden nun Pfade betrachtet, die in die andere Richtung, also von T in den 2-Core führen. Als Startknoten kommen hier genau die Knoten der Menge T(b) in Frage. Da alle Wege in T eindeutig sind, entspricht die dependency des Knotens a bezüglich eines Knotens v in T der Dependency von a bezüglich w(v). Multipliziert man diese mit der Anzahl der Nachfolger von b erhält man den Anteil der kürzesten Wege, die über a *und* b laufen.

Lemma 7. *Sei a ein Knoten im 2-Core und b ein Knoten aus T, weiter sei $u = w(b)$, dann gilt für die Paarbetweenness $\mathcal{PB}(b,a)$:*

$$PB(b,a) = t(b) * \delta_{u,\bullet}(a) \tag{2.10}$$

Proof.

$$\begin{aligned}
PB(b,a) &\overset{1.6}{=} \sum_{s\in P^-(b,a)} \frac{\sigma_{sa}(b)}{\sigma_{sa}} * \sum_{t\in P^+(s,a)} \frac{\sigma_{st}(a)}{\sigma_{st}} \\
&= \sum_{s\in P^-(b,a)} 1 \quad * \sum_{t\in P^+(s,a)} \frac{\sigma_{st}(a)}{\sigma_{st}} \\
&= \sum_{s\in P^-(b,a)} 1 \quad * \sum_{t\in P^+(s,a)} \frac{\sigma_{sw(s)}\sigma_{w(s)t}(a)}{\sigma_{sw(s)}\sigma_{w(s)t}} \\
&= \sum_{s\in P^-(b,a)} 1 \quad * \sum_{t\in P^+(s,a)} \frac{\sigma_{ut}(a)}{\sigma_{ut}}
\end{aligned}$$

Startknoten s die zusammen mit b und a einen kürzesten Weg der Form s-b-a bilden, sind genau die Knoten der Menge $t(b)$. Wege in T sind eindeutig, so dass b auf allen kürzesten Wegen von $t(b)$ in den 2-Core liegt. Da a "hinter" der Wurzel liegt, kann die Pairdependency von a bezüglich $s \in T$ durch die Pairdependency bezüglich u ersetzt werden.

$$PB(a,b) \overset{1.6}{=} t(b) * \sum_{t\in P^+(u,a)} \frac{\sigma_{ut}(a)}{\sigma_{ut}}$$

Man erkennt, dass der zweite Faktor wieder genau die Dependency des Knotens a bezüglich $w(b) = u$ ist. Da Endknoten genau die Knoten in $T(b) \setminus b$ sind, erhält man $\mathcal{PB}(a,b)$, indem man $\delta_{u\bullet}(a)$ mit diesem Wert multipliziert. □

Da in dieser Arbeit Endknoten bei der Betwenness berücksichtigt werden ist b selbst auch Startknoten und die Formel wird wie folgt angepasst:

$$P(a,b) = (t(b)+1) * \delta_{u,\bullet}(a)$$

Dependencies berücksichtigen dann Endknoten, wenn man vor der Berechnung des Single-Source-Shortest-Path-Problems, die Dependencies aller Knoten v mit 1 initialisiert.

Auch hier zeigt sich, dass für die Berechnung der Paarbetweenness eines Knotens b aus T und eines Knotens a aus dem 2-Core genau die Dependency bezüglich w(b) mit der Anzahl der Nachfolger des Knotens b multipliziert werden kann. Daraus ergibt sich allgemein.

Theorem 3. *Die Paarbetweenness $\mathcal{PB}(a,b)$ von Paaren (a,b) mit $a \in T \leftrightarrow b \notin T$ lässt sich in $\mathcal{O}(1)$ Berechnungschritten ermitteln, wenn die Dependencies aller Knoten $v \in 2-\text{Core}$ bezüglich aller Knoten $u \in U$ und die Anzahl der Nachfolger $t(v)$ eines Knotens $v \in T$ vorberechnet werden.*

2.2.4. Betrachtung von Paaren (a,b), mit $a, b \in T$ und $w(a) \neq w(b)$

Betrachtet man zwei Knoten a und b aus T, mit $w(a) \neq w(b)$, zeigt sich, dass Start - und Zielknoten nur in t(a) und t(b) liegen können. Im folgenden werden die Wege betrachtet, die von a in Richtung von b führen. Da Wege in T eindeutig sind, laufen alle Wege über a und b. Deshalb muss für jedes Paar (s,t) mit Startknoten s und Zielknoten t der Anteil 1 zu PB(a,b) addiert werden. Um die Anzahl der Kombinationen von Start - und Zielknoten zu erhalten, multipliziert man t(a) und t(b).

Lemma 8. *Seien a und b Knoten in T mit $w(a) \neq w(b)$, dann lässt sich die Paarbetweenness $\mathcal{PB}(a,b)$ auf folgende Weise berechnen:*

$$P(a,b) = t(a) * t(b) \tag{2.11}$$

Proof.

$$\begin{aligned} PB(a,b) &\overset{1.6}{=} \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \sum_{t \in P^+(s,b)} \frac{\sigma_{st}(b)}{\sigma_{st}} \\ &= \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * \sum_{t \in P^+(s,b)} 1 \\ &= \sum_{s \in P^-(a,b)} \frac{\sigma_{sb}(a)}{\sigma_{sb}} * t(b) \\ &= t(a) * t(b) \end{aligned}$$

Da Start - und Zielknoten in $T(a)$ und $T(b)$ liegen und alle Wege über a und b laufen, sind von einem Startknoten in $T(a)$ genau $t(b)$ Endknoten zu erreichen. Deshalb kann die zweite Summe für jeden Startknoten durch diesen Wert ersetzt werden. Insgesamt gibt es t(a) viele Startknoten und die Anzahl von Paaren (s,t) ergibt sich durch Multiplikation der Nachfolger der Knoten a und b □

Will man kürzeste Wege von und zu Endknoten berücksichtigen, kommt jeweils die gesamte Knotenmenge von T(a) beziehungsweise T(b) als Start - und Endknoten in Frage, so dass man die korrekte Anzahl von Paarkombinationen durch:

- $(t(a)+1) \quad * \quad (t(b)+1)$

erhält. In dieser Arbeit werden Endknoten immer berücksichtigt.

Die Paarbetweenness $\mathcal{PB}(a,b)$ des geordneten Knotenpaares (a,b), mit $a,b \in T$ und $w(a) \neq w(b)$ kann in $\mathcal{O}(1)$ berechnet werden und Theorem **??** wird somit bestätigt. Voraussetzung dafür ist dass man die Anzahl der Nachfolger jedes Knotens $v \in T$ kennt. Diese Werte werden bei der Reduktion des Graphen berechnet.

2.2.5. Analyse und Bewertung

Im folgenden wird untersucht, ob es einen Gewinn bezüglich der Laufzeit und des Speicherverbrauches gibt, wenn man vor der Durchführung des Algorithmus aus [Sch09] zur Berechnung der Paarbetweenness aller Knotenpaare des Netzwerkes eine Reduktion des Graphen auf seinen 2-Core durchführt. Hierbei ist es notwendig Werte, die man zur Berechnung der Paarbetweenness von Paaren mit mindestens einem Knoten in T benötigt, vorzuberechnen und zu speichern.
Die Laufzeit des Originalalgorithmus liegt im ungerichteten, ungewichteten Graphen in $\mathcal{O}(nm)$, da 2n-mal das Single-Source-Shortest-Path-Problem gelöst werden muss. Die Berechnung der Dependencies, die in jeder Iteration von Brandes Algorithmus durchgeführt wird, geht ebenfalls in $\mathcal{O}(m)$ pro Startknoten. Denn dafür werden einmalig alle Kanten des Graphen traversiert.
Das hier vorgestellte Verfahren gliedert sich in drei Teile:

- die Reduktion des Graphen auf den 2-Core
- die Vorberechnung
- die Berechnung der Paarbetweenness auf dem 2-Core

Die Reduktion des Graphen erfolgt durch iteratives Entfernen von Knoten mit Grad 1. Hierfür muss zuerst der Knotengrad aller Knoten des Netzwerkes bestimmt werden. Mit Algorithmus 1 geht dies in $\mathcal{O}(n)$ Schritten, wenn der Graph als Adjazenzarray [KP08] vorliegt. Dabei wird eine Liste der Knoten mit Grad 1 erstellt. Bei der Iteration wird jeweils der erste Knoten, im folgenden Knoten v, aus der Liste entnommen. Die einzige Kante $e = (u,v)$ von v wird aus E entfernt und der adjazente Knoten wird, falls er nach dem entfernen der Kante Grad 1 hat, in die Liste eingefügt. Dies wird solange durchgeführt, bis sich kein Knoten mehr in der Liste befindet. Der Aufwand liegt somit in $\mathcal{O}(|T|)$.
Die Anzahl der Nachfolger $t(v)$ eines Knotens v wird bei der Iteration automatisch mitberechnet. Die restlichen Werte können durch eine Tiefensuche innerhalb jedes Baumfortsatzes gewonnen werden, was mit einem Aufwand, der linear zur Anzahl der Kanten m_{T} in T ist, durchgeführt werden kann. Die Berechnung der Paarbetweenness aller Knotenpaare des 2-Cores, kann mit dem Algorithmus aus [Sch09] in $\mathcal{O}(|2C|(m-m_{\mathrm{T}}))$ berechnet werden.

Insgesamt benötigt das hier untersuchte Verfahren eine Laufzeit von $\mathcal{O}(n+ \mid T \mid +m_{\mathrm{T}}+ \mid 2C \mid (m - m_{\mathrm{T}}))$. Berechnet man die Paarbetweenness aller Paare eines Netzwerkes mit dem gleichen Algorithmus, aber ohne den Graphen vorher zu reduzieren, geht das in einer Laufzeit von $\mathcal{O}(nm)$. Im nächsten Kapitel wird evaluiert, wie sich die Laufzeiten der Algorithmen auf echten Daten verhalten.
Im restlichen Abschnitt wird der Speicherbedarf des hier verwendeten Verfahrens zur Berechnung der Paarbetweenness von allen Knotenpaaren des Netzwerkes analysiert. Um die Paarbetweenness $\mathcal{PB}(a, b)$ von Paaren (a, b) mit $a, b \in T$ berechnen zu können, ist es erforderlich die Anzahl der Nachfolger jedes Knotens $v \in T$, tree(v) und dfs_num(v) zu kennen. Für diese Werte wird Speicherplatz, der linear in der Anzahl der Knoten in T ist benötigt.
Für die Berechnung der Paarbetweenness $\mathcal{PB}(a, b)$ von Paaren (a, b) mit genau einem Knoten in T muss die Dependency jedes Knotens im 2-Core und allen Wurzelknoten des Graphen gespeichert werden (siehe Abs.: 2.2.2). Diese Werte entstehen, als Nebenprodukt, während der Berechnung der Paarbetweenness auf dem 2-Core mit dem Algorithmus aus [Sch09] und benötigen einen Speicherplatz von $\mid U \mid \times \mid 2C \mid$. Schließlich erfordet die Speicherung der Paarbetwennesswerte aus dem 2-Core einen Speicherplatz vom $\mid 2C \mid \times \mid 2C \mid$.
Im Vergleich dazu benötigt man, wenn man keine Reduktion auf dem Graphen durchführt, einen Speicherplatz von $n \times n$, für die Speicherung der Paarbetweennesswerte aller Paare im Graphen.

Wenn man eine Reduktion auf dem Graphen durchführt,spart man $\mid T \mid \times \mid T \mid$ Speicherplatz zur Speicherung der Paarbetweenness $\mathcal{PB}(a, b)$ zweier Knoten a und b, mit $a, b \in T$ und einen Speicherplatz von $\mid T \mid \times \mid 2C \mid$ zur Speicherung der Paarbetweenness $\mathcal{PB}(a, b)$ zweier Knoten a und b, mit $a \in T \leftrightarrow b \notin T$. Dies dürfte im allgemeinen wesentlich mehr sein, als für die Speicherung der Werte aus der Vorberechnung benötigt wird. Eine Reduktion des Graphen auf seinen 2-Core, ist also im Hinblick auf den, für die Daten des Programmes, benötigten Speicher als sinnvoll zu bewerten, wie sich die Reduktion und die notwendige Vorberechnung auf die Laufzeit auswirkt, wird im nächsten Kapitel evaluiert.

3. Evaluation

Die Experimente wurden auf einem Rechner, mit einem 2Core - Mikroprozessor der Familie Intel x86.64, mit einer Verarbeitungsgeschwindigkeit von 1.6 GHz durchgeführt. Der Rechner verfügt über 4Gbyte RAM und einen L2-Cache von 3072K. Gearbeitet wurde auf einem Linux-System mit Kernelversion 2.6.37.6. Die Algorithmen wurden mit der Programmiersprache C++ implementiert und mit einem GCC Compiler der Version 4.5.1 kompiliert.
Für Teile der Graphrepräsentation wurde die Open-Source-Bibliothek Lemon [lem] verwendet.

3.1. Darstellung der Ergebnisse

Die Graphen des ersten Datensatzes (siehe Tab.: 3.1), stammen von Seite [dim], die im Rahmen des 10. DIMACS - Programmierwettbewerbes entstanden ist. DIMACS - Programmierwettbewerbe wurden mit dem Ziel ins Leben gerufen Praxis und Theorie näher zusammenzubringen. Innerhalb der Wettbewerbe sollen Algorithmen auf realen Daten evaluiert werden, da sich die worst-case Laufzeit eines Algorithmus deutlich von dessen realer Laufzeit unterscheiden kann. Im Rahmen des zehnten DIMACS - Programmierwettbewerbs wurden die Probleme Graphpartitionierung und Graphclustering behandelt. Da die Berechnung der Betweenness auch als Vorverarbeitungsschritt für das Clustering verwendet werden kann, wurden diese Instanzen für die Evaluation des Verfahrens zur Berechnung der Paarbetweenness für alle Knotenpaare des Netzwerkes verwendet. Die Clusteringgraphen liegen in ungerichteter und ungewichteter Form im Metis - Format [Dep11] vor.

Graph	Knoten	Kanten	T_Anteil[%]	Zeit_{red}[s]	Zeit_{org}[s]	ratio
karate	34	78	2.9	0	0.004	-
dolphins	62	159	15.5	0.005	0.009	0.56
lesmis	77	254	23.3	0.01	0.01	1
polbooks	105	441	0.0	0.03	0.024	1.25
adjnoun	112	425	8.9	0.028	0.026	1.08
football	115	613	0.0	0.038	0.03	1.27
jazz	198	2742	2.5	0.199	0.156	0.13
celegansneural	297	2148	5.0	0.305	0.253	1.21
celegans_metabolic	453	2025	1.7	0.555	0.455	1.22
email	1133	5451	13.6	3.108	3.081	1.01
polblogs	1490	16715	9.4	7.287	6.552	1.11
netscience	1589	2742	20.1	0.583	0.555	1.05
power	4941	6594	32.1	21.983	36.113	0.61
hep-th	8361	15751	23.7	57.25	66.6	0.86
astro-ph	16709	121251	8.4	869.27	-	-
cond-mat	16726	47594	14.0	431.55	430.73	1.00
cond-mat-2003	31163	120029	-	-	-	-

Tabelle 3.1.: Die Tabelle zeigt die Evaluation der Daten von [dim]. Die Instanzen wurden nach der Anzahl ihrer Knoten sortiert. Die einzelnen Spalten enthalten, in dieser Reihenfolge, die Anzahl der Knoten des Graphen, die Anzahl der Kanten, den prozentualen Anteil der Knoten in T, die Laufzeit des Algorithmus in der reduzierten Variante in Sekunden, die Laufzeit des Algorithmus auf dem Originalgraphen in Sekunden und schliesslich das Verhältnis der beiden Laufzeiten.

Graph	Knoten	Kanten	T_Anteil[%]	$\text{Zeit}_{\text{red}}[s]$	$\text{Zeit}_{\text{org}}[s]$	ratio
as-20000102	6474	13233	37.83	40.88	75.2	0.544
as-caida20071105	26475	106762	38.44	503.29	-	-
ca-GrQc	5242	28980	25.2	29.59	32.85	0.900
ca-HepTh	9877	51971	22.9	139.93	162.94	0.866
ca-HepPh	12008	237010	12.89	520.3	438.31	1.129
ca-AstroPh	18772	396160	7.1	1820.33	-	-
ca-CondMat	23.133	186936	10.89	-	-	-
cit-HepTh	27770	352807	6.0	-	-	-

Tabelle 3.2.: Die Tabelle zeigt die Evaluation der Daten von [sna]. Die Instanzen wurden zuerst nach der Art des Netzwerkes und anschließend nach der Anzahl ihrer Knoten sortiert. Die einzelnen Spalten enthalten, in dieser Reihenfolge, die Anzahl der Knoten des Graphen, die Anzahl der Kanten, den prozentualen Anteil der Knoten in T, die Laufzeit des Algorithmus in der reduzierten Variante in Sekunden, die Laufzeit des Algorithmus auf dem Originalgraphen in Sekunden und schließlich das Verhältnis der beiden Laufzeiten.

Der zweite Datensatz (siehe Tab.: 3.2) stammt von der Stanford Network Analysis Platform [sna]. Das Projekt beinhaltet eine Open-Source-C++- Bibliothek für die Repräsentation und Manipulation von Netzwerken, und weiterhin eine Datenbank, die eine Reihe von mittleren bis sehr großen Netzwerken beinhaltet. Das Projekt stellt verschiedene Arten von Netzwerken bereit, unter anderem sind dies soziale Netzwerke, Kommunikationsnetzwerke oder Wep-Graphen. Die Netzwerke werden je nach Semantik als ungerichtete oder gerichtete Graphen repräsentiert. Für diese Arbeit wurde ein Zitationsnetzwerk, mehrere Collaborationsnetzwerke und zwei Netzwerke, welche die Kommunikation zwischen Routernetzen repräsentieren, verwendet. Um mehr Daten zu erhalten wurden gerichtete Graphen in ungerichtete Graphen transformiert.

3.2. Interpretation der Ergebnisse

Bei Graphen mit geringem Baumknoten-Anteil sind beide Varianten etwa gleich schnell, die Variante ohne Reduktion des Graphen auf seinen 2-Core ist hier sogar etwas schneller. Es hat sich während der Evaluation gezeigt dass die Berechnung der Paarbetweennesswerte den überragenden Anteil der Laufzeit ausmacht, so dass der Laufzeitoverhead der reduzierten Variante in dieser Phase zu suchen ist. Möglicherweise fällt die Initialisierung der Dependencies hier entsprechend ins Gewicht. Es ist aber auch nicht auszuschließen, dass der Grund in der Implementierung zu suchen ist, da jedes mal wenn auf eine Struktur des Originalgraphen zugegriffen werden muss ein mapping der ID des reduzierten Graphen auf die ID des Originalgraphen statt findet.
In beiden Versuchsreihen (siehe Tab.: 3.1 und Tab.: 3.2) ist zu erkennen, dass die Reduktion die Berechnung auf Graphen mit kleinem 2-Core beschleunigt. Diese Beschleunigung beträgt bis zu 45%. Bei einer Netzwerkgrösse von 15000-20000 Knoten, kann es entscheidend sein ob man die Reduktion des Graphen durchführt, da der Speicherbereich des Programmes in diesem Bereich offensichtlich ausgeschöpft wird. Beispiel-Instanzen

hierfür sind die Graphen *ca-AstroPh* und *as-caida20071105* in Tabelle 3.2. Für diese Netzwerke kann die Paarbetweenness zwischen allen Knotenpaaren des Netzwerkes bei Reduktion des Graphen berechnet werden. Während es, wenn man die Berechnung auf dem gesamten Netzwerk durchführt, zu einer Überschreitung des Speicherbereiches kommt.

Größere Netzwerke können von dem Algorithmus, aufgrund der Speicherbelastung, nicht verarbeitet werden (siehe Tab.: 3.1, Graph *cond-mat-2003* und Tab.: 3.2, Graph *cit-HepTh*)

3.3. Kritische Betrachtungen

Die Ergebnisse zeigen, dass der Algorithmus tendenziell dazu geeignet ist, die Berechnung der Paarbetweenness aller Knotenpaare eines Netzwerkes zu beschleunigen. Eine Vorraussetzung dafür, dass die Variante Gewinn bringt ist, dass $|T|$ groß ist. Die Reduktion erhöht die Verwendbarkeit des Algorithmus auch im Hinblick auf die Grösse der Netzwerke, die untersucht werden können. Dennoch bleibt die maximale Eingabegröße begrenzt.
Es sollte auch erwähnt werden, das hier nur wenige Instanzen getestet wurden, um ein sicheres Bild über das Verhalten des Verfahrens zu gewinnen sollte man sicher noch mehr Graphen testen. Das wäre auch deshalb sinnvoll, da diese Evaluation nicht frei von Unregelmäßigkeiten ist, der Graph *netscience* (in Tab.: 3.1) hat beispielsweise einen Verhältnismäßig hohen Anteil von Baumknoten und trotzdem findet keine Beschleunigung durch die Reduktion statt.

4. Zusammenfassung und Ausblick

Die Gruppenbetweenness $\mathcal{GB}(C)$ einer Gruppe C läßt sich mit dem Algorithmus aus [PES07] berechnen. Dieser Algorithmus ist insbesondere dafür geeignet die Gruppe mit der höchsten Betweenness zu finden. Zur Berechnung der Gruppenbetweenness werden vorberechnete Werte benötigt. Unter anderem sind dies die Paarbetweenesswerte aller Knotenpaare des Netzwerkes. Die Berechnung der Paarbetweenness aller Knotenpaare wird mit dem Verfahren aus [PES07] in kubischer Zeit und mit quadratischem Speicheraufwand durchgeführt.

Hier wird nun gezeigt, dass, m_{T} sind im folgenden diejenigen Kanten, die zu mindestens einem Knoten aus T inzident sind, in $\mathcal{O}(n + |T| + m_{\mathrm{T}} + |2C|m_{\mathrm{T}})$ eine Datenstruktur erstellt werden kann, anhand derer sich die Paarbetweenness eines Knotenpaares in den meisten Fällen in $\mathcal{O}(1)$ berechnen lässt. Die verbleibenden Fälle können in $\mathcal{O}(k)$ berechnet werden, wenn k der maximale Ausgangsgrad aller Knoten in T ist. Der Speicherplatz für die Paarbetwennesswerte kann durch die Reduktion deutlich gesenkt werden und liegt in $\mathcal{O}(|2C|^2)$, wenn die Größe des 2-Cores $|2C|$ beträgt.
Das hier verwendete Verfahren, basiert auf dem Algorithmus zur Berechnung der Paarbetweenness aller Paare eines Netzwerkes aus [Sch09]. Diesem wird zur Optimierung eine Reduktion des Graphen auf seinen 2-Core vorgeschaltet. Für die Berechnung der Paarbetweenness von Knotenpaaren außerhalb des 2-Cores sind außerdem weitere Vorberechnungsschritte notwendig.

Die Evaluation zeigt, dass das Verfahren aus [Sch09] zur Berechnung der Paarbetweenness aller Paare eines Netzwerkes, durch die Reduktion, für Graphen mit kleinem 2-Core Anteil beschleunigt, und die Speicherbelastung verbessert werden kann. Dadurch können größere Graphen verarbeitet werden. Der Algorithmus stellt aufgrund seiner besseren worst-case Laufzeit eine Alternative zu dem Verfahren für die Berechnung der Paarbetweenness aus [PES07] dar.

Ein weiterer Schritt könnte sein zu untersuchen, ob sich die Idee des reduzierten Graphen auf den Algorithmus zur Berechnung der Gruppenbetweenness in [PES07] übertragen

lässt. Dieser Algorithmus berechnet die Gruppenbetweenness iterativ, indem Knoten für Knoten einer Menge M zugeordnet werden, solange bis diese alle Knoten der Gruppe C, deren Betweenness berechnet werden soll, enthält. Bei jedem Iterationsschritt muss die Betweenness aller Knoten in C und die Paarbetweenness aller Knotenpaare in C aktualisiert werden. Zu untersuchen wäre dann, ob es ausreicht Werte im 2-Core zu aktualisieren und daraus auf die Knotenpaare außerhalb des 2-Cores zu schließen und somit Rechenzeit und Speicherplatz zu sparen.

Der Gruppenbetweennessalgorithmus aus [PES07] kann für sehr große Netzwerke aufgrund des hohen Speicherbedarfs nicht verwendet werden. Deshalb wäre ein Ziel, Verfahren zu entwickeln die mit geringerem Speicherbedarf auskommen. Hierbei könnte man sich beispielsweise parallele Varianten vorstellen, die auf Brandes Algorithmus aufbauen.

Literaturverzeichnis

[BGPL11] M. Baglioni, F. Geraci, M. Pellegrini und E. Lastres: *Fast exact computation of betweenness centrality in social networks.* Technischer Bericht, Institut für Telematik,National Research Council, Rom, 2011.

[Bra01] Ulrik Brandes: *A Faster Algorithm for Betweenness Centrality.* Journal of Mathematical Sociology, 25:163–177, 2001.

[Bra08] Ulrik Brandes: *On variants of shortest-path betweenness centrality and their generic computation.* Social Networks, 30:136–145, 2008.

[Dep11] Department of Computer Science and Engeneering, University of Minnesota: *Metis, A Software Package for Partitioning Unstructured Graphs,Partitioning Meshes, and Computing Fill-Reduced Orderings of Sparse Matrices*, August 2011.

[dim] `http://snap.stanford.edu/snap/`.

[GSS08] Robert Geisberger, Peter Sanders und Dominik Schultes: *Better Approximation of Betweenness Centrality.* In: *Conference: Workshop on Algorithm Engeneering and Experiments (ALENEX 2008)*, Seiten 90–100, 2008.

[KCB09] Eric Kolaczyk, David Chua und Marc Barthelmy: *Group betweenness and co-betweenness: Inter-related notions of coalition centrality.* Social Networks, 31:190–203, 2009.

[KP08] K.Mehlhorn und P.Sanders: *Algorithms and Data Structures - The Basic Toolbox.* Springer Verlag, 2008.

[lem] `http://http://lemon.cs.elte.hu/trac/lemon`.

[PES07] R. Puzis, Y. Elovici und S.Dolev: *Fast Algorithm for successive computation of group betweenness centrality.* Physical Review E, 76:056709[9]pages, 2007.

[Sch09] Andrea Schumm: *Heuristic Algorithms for the Shortcut Problem.* Diplomarbeit, Institut für Theoretische Informatik, Universität Karlsruhe (TH), 2009.

[sna] `http://www.cc.gatech.edu/dimacs10/`.

Anhang

A. First Appendix Section

Algorithm 1 Compute_Node_Degrees

```
1: Input: V: Array of NodeIds, E: Array of NodeIds
2: Output: degree[v]   ∀v ∈ V, deg_1 = {v | v hat Grad 1}
3: int d = 0;
4: for  i = 0 to (n-1) do
5:    d = V[i+1] - V[i];
6:    if d == 1 then
7:       deg_1 ← deg_1 ∪ {NodeFromId(i)}
8:    end if
9:    degree[i] = d
10: end for
```

Algorithm 2 SPVB

Input: Graph g: undirected, unweightet
Output: $b = Array[0 \cdots R] : int$;
$p = Array[0 \cdots R] : int$;
$i = 0; \quad G^i = G; \quad deg_1 = \{v \in V^i | deg(v) = 1\}$
repeat
  $v \leftarrow deg_1$;
  remove v from deg_1
  for all $u : (v, u) \in E$ **do**
    b[u] = b[u] + 2(N - p[v] - p[u] -2)(p[v] + 1);
    p[u] = p[u] + p[v] + 1;
    i++;
    $V^i = V^{i-1} \setminus v$
    $E^i = E^{i-1} \setminus (v, u)$
  end for
  if deg(u) = 1 **then**
    $deg_1 \rightarrow u$
  end if
until $deg_1 = \emptyset$
for all $n : node \in g$ **do**
  $Brandes(G^i, b, p)$
end for

Algorithm 3 Brandes_modified

Input: Graph g:undirected, unweighted
$b = Array[0 \cdots R] : float$;
$p = Array[0 \cdots R] : int$; $s : Node$
$\delta = Array[0 \cdots R] : float$;
$d = Array[0 \cdots R] : int$; $\sigma = Array[0 \cdots R] : int$
Output: $b = Array[0 \cdots R] : int$;
pred[R] = Array:Edge
Q = queue:Node
S = stack:Node
$d[t] \leftarrow -1$, $t \in V$; $d[s] \leftarrow 0$
$\sigma[t] \leftarrow 0$,$t \in V$; $\sigma[s] \leftarrow 1$
$\delta[t] \leftarrow 0$,$t \in V$;
$Q \leftarrow s$
while Q not empty **do**
  $v \leftarrow Q$
  $v \rightarrow S$
  for all neighbors of w **do**
    if $d[w] < 0$ **then**
      $Q \leftarrow w$
      $d[w] \leftarrow d[v] + 1$
    end if
    if $d[w] = d[v] + 1$ **then**
      $\sigma[w] = \sigma[w] + \sigma[v]$
      $pred[w] \leftarrow v$
    end if
  end for
end while
while s not empty **do**
  $w \leftarrow S$
  for all p:pred[w] **do**
    if !PB **then**
      $\delta[p] = \delta[p] + \frac{\sigma[p]}{\sigma[w]} * (\delta[w] + p[w] + 1)$
    end if
  end for
  if w!=s **then**
    $b[w] = b[w] + \delta[w] * (p[s] + 1)$;
  end if
end while

Algorithm 4 PB_Reduced

1: **Input:** $Graph\ g : V = Array[0 \cdots R] : int; \quad E = Array[0 \cdots R] : Node$
2: **Output:** $b = Array[0 \cdots R] : int; \quad pb = Matrix[0 \cdots R][0 \cdots R] : int$
3: $dpcy = Matrix[0 \cdots R][0 \cdots R] : int$
4: $dist = Matrix[0 \cdots R][0 \cdots R] : int$
5: $\sigma = Matrix[0 \cdots R][0 \cdots R] : int$
6: $root\_dpcy = Matrix[0 \cdots ROOTS][0 \cdots R] : int$
7: $dfs\_num = Array[0 \cdots R] : int; \quad tree = Array[0 \cdots R] : int$
8: $i = 0; \quad G^i = G; \quad deg_1 = \{v \in V^i | deg(v) = 1\}$
9: **repeat**
10: $v \leftarrow deg_1$;
11: $processed[v] = true$
12: $degree[v] = degree[v] - 1$
13: b[u] = b[u] + (2N - 1)+1;{Endknoten addieren}
14: **for all** $u \leftarrow E[V[v]]$ **do**
15: **if** !processed[u] **then**
16: $processed[u] = true$
17: $degree[u] = degree[u] - 1$
18: b[u] = b[u] + 2(N - p[v] - p[u] -2)(p[v] + 1);
19: p[u] = p[u] + p[v] + 1;
20: i++;
21: $V^i = V^{i-1} \setminus v$
22: $E^i = E^{i-1} \setminus (v, u)$
23: **end if**
24: **end for**
25: **if** deg(u) = 1 **then**
26: $deg_1 \rightarrow u$
27: **end if**
28: **until** $deg_1 = \emptyset$
29: **for** i = 0 to N **do**
30: **if** $processed[i] == true$ && $degree[i] > 0$ **then**
31: $roots \leftarrow V[i]$
32: **end if**
33: **end for**
34: **for all** $r \in roots$ **do**
35: DFS(r,dfs_num,tree)
36: **end for**
37: **if** $|V^i| > 1$ **then**
38: **for all** $n : node \in g$ **do**
39: $Brandes(G^i, b, p, pb[n], dpcy[n], \sigma[n], dist[n])$
40: **end for**
41: **for all** $n : node \in g$ **do**
42: $Brandes(G^i, b, p, pb[n], dpcy[n], \sigma[n], dist[n])$
43: **end for**
44: **end if**
45: **for all** $r \in roots$ **do**
46: **for** i = 0 to N **do**
47: root_dependencies[id(r)][i] = dependencies[id(r)][i];
48: **end for**
49: **end for**

Algorithm 5 Brandes_PB_Reduced

1: **Input:** $Graph\ g: \quad V = Array[0 \cdots R] : int; \quad E = Array[0 \cdots R] : Node$
2: $b = Array[0 \cdots R] : int; \quad dpcy = Array[0 \cdots R] : int$
3: $p = Array[0 \cdots R] : int; \quad pb = Array[0 \cdots R] : int, s : Node$
4: $d = Array[0 \cdots R] : int; \quad \sigma = Array[0 \cdots R] : int, s : Node$
5: **Output:** $b = Array[0 \cdots R] : int;\ dep = Array[0 \cdots R] : int;\ pb = Array[0 \cdots R] : int$
6: pred[R] = Array:Edge
7: Q = queue:Node
8: S = stack:Node
9: $d[t] \leftarrow -1,\ t \in V;\ d[s] \leftarrow 0$
10: $\sigma[t] \leftarrow 0, t \in V;\ \sigma[s] \leftarrow 1$
11: **if** PB **then**
12: $pb[t] \leftarrow dpcy[t] * (1 + p[t]), t \in V;$
13: **else**
14: $dpcy[t] \leftarrow dpcy[t] + p[t], t \in V;$
15: **end if**
16: $Q \leftarrow s$
17: {BFS}
18: **while** s not empty **do**
19: $w \leftarrow S$
20: **for all** p:pred[w] **do**
21: **if** !PB **then**
22: $dpcy[p] = dpcy[p] + \frac{\sigma[p]}{\sigma[w]} * dbcy[w]$
23: **else**
24: $pb[p] = pb[p] + \frac{\sigma[p]}{\sigma[w]} * pb[w]$
25: **end if**
26: **end for**
27: **if** !PB **then**
28: **if** w!=s **then**
29: **if** p[w]==0 **then**
30: $b[w] = b[w] + dpcy[w] * (p[s] + 1);$
31: **else**
32: $b[w] = b[w] + (dpcy[w] - p[w]) * (p[s] + 1);$ {Korrektur: bei Reduktion schon berücksichtigt}
33: **end if**
34: **else**
35: b[w] = b[w]+ dpcy[w];
36: b[w] = b[w]+ p[w]; {Endknoten addieren}
37: **end if**
38: **else if** $PB \wedge w == s$ **then**
39: pb[w] = b[w]
40: **end if**
41: **end while**

Algorithm 6 Brandes_PB

1: **Input:** $Graph\ g: \quad V = Array[0 \cdots R] : int; \quad E = Array[0 \cdots R] : Node$
2: $b = Array[0 \cdots R] : int; \quad dpcy = Array[0 \cdots R] : int$
3: $p = Array[0 \cdots R] : int; \quad pb = Array[0 \cdots R] : int, s : Node$
4: $d = Array[0 \cdots R] : int; \quad \sigma = Array[0 \cdots R] : int, s : Node$
5: **Output:** $b = Array[0 \cdots R] : int;\ dep = Array[0 \cdots R] : int;\ pb = Array[0 \cdots R] : int$
6: pred[R] = Array:Edge
7: Q = queue:Node
8: S = stack:Node
9: $d[t] \leftarrow -1,\ t \in V;\ d[s] \leftarrow 0$
10: $\sigma[t] \leftarrow 0, t \in V;\ \sigma[s] \leftarrow 1$
11: **if** PB **then**
12: $pb[t] \leftarrow dpcy[t], t \in V;$
13: **end if**
14: $Q \leftarrow s$
15: **while** Q not empty **do**
16: $v \leftarrow Q$
17: $v \rightarrow S$
18: **for all** neighbors of w **do**
19: **if** $d[w] < 0$ **then**
20: $Q \leftarrow w$
21: $d[w] \leftarrow d[v] + 1$
22: **end if**
23: **if** $d[w] = d[v] + 1$ **then**
24: $\sigma[w] = \sigma[w] + \sigma[v]$
25: $pred[w] \leftarrow v$
26: **end if**
27: **end for**
28: **end while**
29: **while** s not empty **do**
30: $w \leftarrow S$
31: **for all** p:pred[w] **do**
32: **if** !PB **then**
33: $dpcy[p] = dpcy[p] + \frac{\sigma[p]}{\sigma[w]} * dbcy[w]$
34: **else**
35: $pb[p] = pb[p] + \frac{\sigma[p]}{\sigma[w]} * pb[w]$
36: **end if**
37: **end for**
38: **if** !PB **then**
39: b[w] = b[w]+ dpcy[w];
40: **end if**
41: **end while**