Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Pascal Jäger

Community Tracking mit Hilfe von Link Prediction in sozialen Graphen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Pascal Jäger

Community Tracking mit Hilfe von Link Prediction in sozialen Graphen

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Julia Padberg

Eingereicht am: 28. Mai 2014

Pascal Jäger

Thema der Arbeit

Community Tracking mit Hilfe von Link Prediction in sozialen Graphen

Stichworte

Community Tracking, Social Graph Mining, Link Prediction, Node Prediction

Kurzzusammenfassung

Die Beobachtung von Gruppen in sozialen Graphen ist eine komplexe Aufgabe. Zunächst werden diskrete Zeitabschnitte des Graphen erstellt. Anschließend wird für jeden Zeitabschnitt, die darin enthaltenen Gruppen gesucht und über die Zeitabschnitte hinweg in Beziehung gebracht. Dabei unterliegen Gruppen Änderungen in ihrer Größe, sowie den internen und externen Verbindungen. Um die Zuordnung zu verbessern, soll mit Hilfe von Link- und Node Prediction, die Änderungen vorhergesagt werden. Die verwendeten Algorithmen werden auf dem Graph Processing System Apache Giraph ausgeführt.

Pascal Jäger

Title of the paper

Community Tracking in Social Graphs using Link Prediction

Keywords

Community Tracking, Social Graph Mining, Link Prediction, Node Prediction

Abstract

Tracking communities in social graphs requires the detection of communities in each timestep and the matching of communities between these timesteps. Communities change over time, they split or merge and single nodes enter or leave the graph. Link and node prediction shall ease the matching of communities leading to more accurate tracking results. The used algorithms are implemented on top of the Apache Giraph framework.

Inhaltsverzeichnis

1. Einführung **1**
1.1. Motivation . . . 1
1.2. Ziele der Arbeit . . . 2
1.3. Aufbau der Arbeit . . . 2

2. Grundlagen **3**
2.1. Graphen . . . 3
2.1.1. Soziale Graphen . . . 4
2.1.2. Time Evolving Graphs . . . 5
2.2. Community Detection . . . 6
2.2.1. Gruppen . . . 6
2.2.2. Qualitätsfunktionen . . . 7
2.2.3. Klassische Ansätze . . . 9
2.2.4. Alternative Ansätze . . . 10
2.3. Node Prediction . . . 11
2.4. Link Prediction . . . 12
2.4.1. Common Neighbors . . . 13
2.4.2. Resource Allocation . . . 13
2.4.3. Local Path Index . . . 14
2.4.4. Jaccard Koeffizient . . . 14
2.4.5. Weitere gruppenbasierte Kennzahlen . . . 14
2.4.6. Qualitätsfunktion . . . 15
2.5. Community Tracking . . . 16
2.6. Verwandte Arbeiten . . . 18
2.7. Apache Giraph . . . 19

3. Analyse **22**
3.1. Problembeschreibung . . . 22
3.2. Community Detection . . . 24
3.3. Node Prediction . . . 27
3.3.1. Erstellen neuer Knoten . . . 27
3.3.2. Erstellen initialer Kanten . . . 28
3.3.3. Entfernen existierender Knoten . . . 29
3.4. Link Prediction . . . 29
3.4.1. Zeitliche Relevanz von Kanten . . . 29

3.4.2. Dezentrales Link Prediction . . . 31
3.5. Community Tracking . . . 32
3.5.1. Vergleich der Ansätze . . . 33
3.5.2. Erweiterung für Prediction-Daten . . . 33
3.5.3. Definition der zu vergleichenden Partitionen . . . 34
3.6. Zusammenfassung . . . 35

4. Realisierung **36**
4.1. Giraph . . . 36
4.2. Community Detection . . . 36
4.2.1. Phase 1: Label Propagation . . . 37
4.2.2. Phase 2: Aufräumen . . . 42
4.2.3. Ein- und Ausgabe . . . 46
4.3. Node Prediction für Zitationsnetzwerke . . . 47
4.4. Link Prediction für Zitationsnetzwerke . . . 48
4.4.1. Ein- und Ausgabe . . . 50
4.5. Community Tracking . . . 50

5. Experimente **51**
5.1. Testdaten . . . 51
5.1.1. Zitationsnetzwerke . . . 52
5.1.2. Synthetische Daten . . . 52
5.2. Validierung der Community Detection . . . 54
5.3. Prediction Experimente . . . 59

6. Diskussion und Ausblick **61**
6.1. Ergebnisse . . . 62
6.1.1. Community Detection . . . 62
6.1.2. Node Prediction . . . 63
6.1.3. Link Prediction . . . 64
6.1.4. Community Tracking . . . 64
6.1.5. Überprüfung der Zielerreichung . . . 64
6.2. Ausblick . . . 65
6.2.1. Community Detection . . . 65
6.2.2. Node- und Link Prediction . . . 66

A. Inhalt der DVD **67**

Abbildungsverzeichnis

2.1. Beispiel für 3 Gruppen (Fortbestand, Geburt und Tod)(Greene u. a., 2010) . . . 16
2.2. Beispiel für 4 Gruppen (Zusammengehen und Aufteilen)(Greene u. a., 2010) . . 17
2.3. Die Block-Kombinationen von Jung und Segev (2013) 19
2.4. Map-only Job (Ching, 2012) 20

3.1. a) der reguläre Ablauf für Daten mit einer ausreichenden Qualität b) das Generieren eines Zwischenschritts mit Hilfe von Node- und Link Prediction . 23
3.2. Zentrale und dezentrale Komponenten in Giraph 24
3.3. Zeitliche relevant von Kanten. Ein Punkt repräsentiert den Zeitpunkt des Erstellens oder Löschens einer Kante. 31
3.4. Vergleich der Gruppen aus den verschiedenen Zeitschritten. 34

4.1. Die Interfaces und Klassen für die MasterComputation am Beispiel des Copra-Master. 37
4.2. Die Computation-Klassen in Giraph für die Implementierung der Knoten-Logik. 38
4.3. Zeitlicher Ablauf und Informationsfluss in Giraph für die Label Propagation Variante 1. 39
4.4. Berechnung des Labels in Variante 1. 40
4.5. Berechnung des Labels in Variante 2. 43
4.6. Vergleich der Gruppen aus den verschiedenen Zeitschritten. 49

5.1. Wachstum der Zitationsnetzwerke. 53
5.2. Nicht überlappende Gruppen (vgl. Gregory (2010, Abbildung 11)) 55
5.3. Netzwerke mit 1000 Knoten und $\mu = 0,1$ (vgl. Gregory (2010, Abbildung 12 oben rechts und 13 oben rechts)) 56
5.4. Netzwerke mit 1000 Knoten und $\mu = 0,3$ (vgl. Gregory (2010, Abbildung 12 unten rechts und 13 unten rechts)) 57
5.5. Netzwerke mit 5000 Knoten und und $\mu = 0,1$ (vgl. Gregory (2010, Abbildung 14 und 15 oben rechts)) 58
5.6. Netzwerke mit 5000 Knoten und $\mu = 0,3$ (vgl. Gregory (2010, Abbildung 14 unten rechts und 15 unten rechts)) 59
5.7. Laufzeit bei steigender Graph-Größe (vgl. (Gregory, 2010, Abbildung 16) . . . 60
5.8. Laufzeit bei steigender Graph-Größe (vgl. (Gregory, 2010, Abbildung 16) . . . 60

Listings

2.1. MaxValueVertex (Martella, 2012): 21

4.1. Vereinfachte Darstellung des CopraMaster der Variante 1 40
4.2. Subgruppen entfernen nach Gregory (2010) 42
4.3. Zustände innerhalb eines Knotens 45

1. Einführung

1.1. Motivation

Community Tracking befasst sich mit der Untersuchung der zeitlichen Entwicklung von Gruppen in sozialen Graphen. Gesucht werden dabei Ereignisse wie das Aufteilen einer Gruppe oder das Verschmelzen zweier Gruppen zu einer. Dazu werden Graphen mit temporalen Daten in Zeitabschnitte unterteilt. Für jeden Zeitschritt müssen die darin vorhandenen Gruppen bestimmt werden (Community Detection). Anschließend werden die Gruppen aus den einzelnen Zeitschritten einander zugeordnet. Dadurch lässt sich die Entwicklung einer Gruppe über alle Zeitschritte hinweg beobachten (Hopcroft u. a., 2004; Asur u. a., 2007; Palla u. a., 2007; Greene u. a., 2010) und bietet somit Einblicke in die Gruppendynamik.

Die für das Finden von Gruppen verwendeten Algorithmen hängen in ihrer Qualität stark von den zu Grunde liegenden Daten ab. Sind die Daten beispielsweise lückenhaft, weil eventuell immer nur Daten aus dem ersten Quartal eines Jahres vorliegen, so kann die Änderung innerhalb der Gruppen zwischen diesen beiden Zeitabschnitten so großen sein, dass es nicht mehr möglich ist, die Gruppen im vorherigen Zeitabschnitt wieder zu Finden.

Mit Hilfe von Node- und Link Prediction, also Verfahren zur Vorhersage der Entstehung neuer Knoten und Kanten, soll daher untersucht werden, ob sich Teilschritte der Gruppenentwicklung zwischen zwei Zeitabschnitten nachbilden lassen. Dadurch soll eine höhere Wiedererkennungsrate zwischen den ursprünglichen Zeitschritten erreicht werden. Darüber hinaus, kann ein detaillierter Einblick in die Entwicklung der Gruppen gewonnen werden.

Auf Grund der immer größer werdenden Datenmengen und der daraus resultierenden Notwendigkeit einer Parallelisierung, untersucht diese Arbeit die Umsetzung von Algorithmen zur Gruppenfindung, sowie Node- und Link Prediction, für das dezentrale Graph Processing System Apache Giraph. Algorithmen für Apache Giraph werden aus der lokalen Sicht eines einzelnen Knoten geschrieben und müssen daher mit dezentralen und einer lokal begrenzten Menge an Daten auskommen.

1.2. Ziele der Arbeit

Diese Arbeit verfolgt zwei wesentliche Ziele. Das erste Ziel ist die Realisierung eines Algorithmus für das Finden von Gruppen (Community Detection), aufbauend auf dem Apache Giraph Framework. Als zweites Ziel werden Ansätze für Node- und Link Prediction untersucht, welche ebenfalls mit Hilfe des Apache Giraph Frameworks umgesetzt werden sollen. Aus diesen Zielen lassen sich die folgenden Teilaufgaben ableiten:

1. Implementieren eines Community Detection Algorithmus für Apache Giraph
2. Implementierung von Node Prediction Algorithmen für Apache Giraph
3. Implementierung eines Link Prediction Algorithmus für Apache Giraph
4. Erzielen einer höheren Genauigkeit bei der Wiedererkennung von Gruppen, durch das Generieren von Zwischenschritten zwischen den bekannten Datensätzen im Gegensatz zu einem Vergleichen ohne Zwischenschritt.

Dabei ist es nicht Ziel dieser Arbeit, einen verbesserten Algorithmus für das Finden von Gruppen ,Link- oder Node Prediction zu entwickeln. Vielmehr sollen bestehende Konzepte auf das Apache Giraph Framework übertragen und deren Skalierbarkeit untersucht werden.

1.3. Aufbau der Arbeit

Nach der Einführung in diesem Kapitel, werden in Kapitel 2 die Grundlagen für das weitere Vorgehen erläutert. Es wird auf die wichtigsten Eigenschaften von Graphen eingegangen, um anschließend die darauf aufsetzenden Konzepte für Community Detection, Link- und Node Prediction sowie Community Tracking anwenden zu können. Auch Apache Giraph und dessen konzeptueller Ursprung Pregel werden erklärt.

In Kapitel 3 werden die Anforderungen der Fragestellung an die Grundlagen untersucht und das Vorgehen für die Umsetzung geplant. In Kapitel 4 werden Aspekte der Realisierung beschrieben. In Kapitel 5 werden Experimente mit den implementierten Algorithmen durchgeführt, um die Eigenschaften und Effektivität der Algorithmen zu ermitteln.

2. Grundlagen

In diesem Kapitel werden Grundlagen definiert, die für das Verständnis der Problemstellung und der Umsetzung notwendig sind. Neben der zu Grunde liegenden Datenstruktur eines Graphen (Kapitel 2.1) werden verschiedene Ansätze für das Finden von Gruppen (Community Detection) besprochen (Kapitel 2.2). Anschließend werden Aspekte für die Vorhersage neuer Knoten (2.3 und Kanten (2.4) beleuchtet. Nachdem gezeigt wurde, wie Gruppen über mehrere Zeitabschnitte hinweg beobachtet werden können (Community Tracking, 2.5) wird ein Blick auf verwandte Arbeiten geworfen (Kapitel 2.6). Abschließend werden die Grundkonzepte von Apache Giraph vorgestellt (Kapitel 2.7).

2.1. Graphen

Ein Graph ist eine mathematische Struktur bestehend aus einer Menge von Knoten V (engl.: vertices) $\{v_1, \ldots, v_n\}$ und einer Menge von Kanten E (engl.: edges) $\{e_1, \ldots, e_m\}$. Eine Kante ist ein Knoten-Paar $\{v_i, v_j\}$ mit $i, j \in \{1 \ldots n\}$. In einem *Multi-Graph* können mehrere Kanten zwischen den Knoten i und j existieren (Diestel, 2010), beispielsweise verschiedene Telefonate zwischen zwei Personen. Knoten und Kanten können beide mit zusätzlichen Attributen belegt werden. Für Knoten könnte dies eine Telefonnummer sein, für Kanten wäre dementsprechend die Anrufdauer ein mögliches Attribut.

In einem *ungerichteten Graphen* wird eine Kante $\{v_i, v_j\}$zwischen zwei Knoten v_i und v_j interpretiert, als eine Verbindung zwischen v_i und v_j die, sowohl von v_i nach v_j, als auch von v_j nach v_i *traversiert* werden kann. Bei *gerichteten Graphen* kann eine Kante $\{v_i, v_j\}$ nur von v_i nach v_j abgeschritten werden. Für die Rückrichtung bedarf es dann einer Kante $\{v_j, v_i\}$ (Diestel, 2010).

Der Grad (engl.: degree) eines Knoten definiert die Anzahl der Kanten des Knotens (für ungerichtete Graphen). Für gerichtete Graphen wird zwischen *in-degree* und *out-degree* unterschieden. Der *in-degree* ist die Anzahl aller Kanten, die an diesem Knoten enden. Der *out-degree* umfasst die Anzahl aller Kanten die von diesem Knoten aus traversiert werden können. Ein Pfad ist eine Reihenfolge von Knoten, für die es möglich ist, von einem Knoten zum nächsten

Knoten zu traversieren. Kanten können mit einer Gewichtung versehen werden, welche je nach Kontext unterschiedlich interpretiert werden kann.

Die Graphentheorie geht zurück bis zu Leonard Euler im Jahre 1736 und dem bekannten Puzzle der Königsberger Brücken (Euler, 1736). Fast 300 Jahre später sind die Informationen, die in Graphen gespeichert werden beinahe schon unvorstellbar groß geworden und damit auch die Graphen selbst. Bereits seit den 1930ern werden Graphen im Form sozialer Netzwerke untersucht (Scott, 2000; Wasserman und Faust, 1994). Doch spätestens mit dem Erfolg des sozialen Netzwerks *Facebook* ist klar, das diese Menge an Daten nicht mehr mit den klassischen Methoden untersucht und verarbeitet werden können.

2.1.1. Soziale Graphen

Der Begriff sozialer Graph (engl.: social graph) wird in der Literatur oft verwendet. Google Ngramms[1] verzeichnet, in den von Google bis 2008 digitalisierten Büchern, einen starke Anstieg in der Nutzung des Begriffs *social graph* ab 2003/4. Dennoch wird der Begriff in den meisten Veröffentlichungen nicht definiert. Aus dem Zusammenhang lässt sich jeweils schließen, dass es dabei um Daten geht, welche in der Struktur eines Graphen vorliegen und die Informationen aus sozialen Netzwerken widerspiegeln. Für Knoten ist das Verständnis im Allgemeinen das Gleiche: Knoten sind Nutzer oder Dinge mit denen ein Nutzer in eine Beziehung gebracht werden kann, beispielsweise Gruppen, Institutionen oder Orte. Bei Kanten hängt das Verständnis dann vom jeweils verwendeten sozialen Netzwerk ab, aus welchem die Daten stammen. *Facebook*, welches ebenfalls den Begriff *social graph* verwendet, versteht darunter die Beziehungen zwischen Nutzern und Dingen (Curtiss u. a., 2013, Tabelle 1), meint damit aber nicht die Beziehungen, die sich über den Versand von Nachrichten oder Postings auf Nutzerseiten herleiten lassen. Aus diesem Grund untersuchen Wilson u. a. (2012) den Unterschied zwischen einem *social graph* und einem *interaction graph*, welcher ebendiese Interaktionen als Kanten besitzt. Dabei leitet sich das Gewicht der Kanten aus der Häufigkeit der Interaktion ab.

Bei Kim und Bonneau (2009, Abschnitt 2) vermischt sich die Unterscheidung einer Relation. Je nach Datensatz ist die Relation durch eine explizit Beziehung zwischen den Nutzern definiert (beispielsweise durch eine Freundschaft), oder eben implizit durch Interaktionen (beispielsweise den Austausch von Nachrichten). Besteht ein Graph jedoch ausschließlich aus Kanten, welche eine Interaktion repräsentieren, wird der Graph automatisch zu einem *social graph*.

[1] http://tinyurl.com/o6cbsut

Diese Arbeit definiert einen sozialen Graphen daher, ähnlich wie in Kim und Bonneau (2009), als einen ungerichteten Graphen $G = (V, E)$, bei dem die Knoten V Nutzer und die Kanten E soziale Beziehungen repräsentieren. Diese Beziehungen können durch Interaktionen zwischen den Knoten zum Ausdruck kommen. Eine Unterscheidung zwischen einem *social-* und einem *interaction graph* findet somit nicht statt.

Für diese Arbeit werden nur Netzwerke untersucht, bei denen alle Knoten vom gleichen Typ sind. So ist es beispielsweise möglich ein Zitationsnetzwerk nur über die veröffentlichten Arbeiten aufzuspannen. Hier gehören alle Knoten zum Typ *veröffentlichte Arbeit*. Es ist aber auch möglich, zusätzlich die Autoren einer Arbeit mit dieser zu verknüpfen. Knoten in einem solchen Netzwerk sind dann entweder vom Typ *veröffentlichte Arbeit* oder *Autor*. Dadurch entsteht ein bipartiter Graph.

Soziale Netzwerke und somit soziale Graphen zeigen häufig *Kleine-Welt Eigenschaften* (engl.: *small world phenomenon*) auf. Dieser Begriff wurde von Stanley Milgram geprägt, ein sozialpsychologischer Begriff, der innerhalb der sozialen Vernetzung in der modernen Gesellschaft den hohen Grad abkürzender Wege durch persönliche Beziehungen bezeichnet (Milgram, 1967). Später haben unter anderem Watts und Strogatz (1998) dieses Konzept aufgegriffen und in die Graphentheorie übernommen. Die Kleine-Welt-Eigenschaft spielt oft eine Rolle bei der Planung von Algorithmen, da auf Grund dieser Eigenschaft, die Laufzeit deutlich reduziert werden kann.

Kleine-Welt-Netzwerke sind häufig skalenfrei (bzw. skaleninvariant). Das heißt, dass der Grad (die Anzahl der Kanten eines Knoten) gemäß einem Potenzgesetz über die Knoten verteilt ist. Das Verhältnis der Menge $P(k)$ von Knoten, die einen Grad k haben zu den anderen Knoten kann beschrieben werden durch $P(k) \sim k^{-\gamma}$, wobei γ zwischen 2 und 3 liegt. Zitationsnetzwerke sind Beispiele für skalenfreie Netzwerke.

2.1.2. Time Evolving Graphs

Im Zusammenhang des zeitlichen Fortschritts wird die Momentaufnahme eines Graphen auch als ein *statischer Graph* bezeichnet. Ein Time Evolving Graph (TEG) ist entsprechend in Liu u. a. (2012) als ein *dynamischer Graph* definiert, der aus einer Abfolge von statischen Graphen $\{G_1, \ldots G_T\}$ besteht. Hierbei gilt, dass $G_t = (V, E, w(e, t))$ ist und mit $w(e, t)$ eine Funktion existiert, die einer Kante $e \in E$ genau dann ein Gewicht zuordnet, wenn die Kante zum Zeitpunkt $t \in T$ existiert.

Das Vorhalten mehrerer Momentaufnahmen eines Graphen stellt erhöhte Anforderungen an die bereitgestellte Hardware, wie zum Beispiel Speicher. Der Unterschied zwischen zwei

Momentaufnahmen lässt sich leicht als Menge von Kanten und Knoten beschreiben, die entweder Hinzugefügt oder Entfernt werden müssen (Fard u. a., 2012).

2.2. Community Detection

Community Detection befasst sich mit dem Finden von Personengruppen in sozialen Graphen, die eine bestimmen Eigenschaft besitzen. In diesem Abschnitt soll zunächst näher auf den Begriff Gruppe und dessen Verständnis eingegangen werden. Anschließend wird ein Überblick über die verschiedenen Ansätze zum Finden von Gruppen gegeben, der sich im Wesentlichen auf die Arbeit von Fortunato (2010) stützt.

2.2.1. Gruppen

In der Literatur existiert keine einheitliche Definition für Gruppen. Was genau eine Gruppe ist, also welche Eigenschaften zur Abgrenzung herangezogen werden, hängt häufig vom verwendeten System oder dem Anwendungskontext ab. So nutzen die in dieser Arbeit beschriebenen Verfahren (siehe 2.2.3 und 2.2.4) unterschiedliche Metriken und Eigenschaften, um die Knoten in Gruppen einzuteilen. Auch wenn keine einheitliche Definition existiert, gibt es dennoch einige generelle Ansätze und Eigenschaften, mit deren Hilfe Definitionen aufgebaut werden können. Nach Fortunato (2010) lassen sich die Definitionen in drei Klassen unterteilen: lokal, global und basierend auf Knotenähnlichkeit.

Lokale Definitionen betrachten nur den jeweiligen Teilgraph und gegebenenfalls die direkten Nachbarschaften. Gruppen werden dann basierend auf diesen Informationen definiert. Globale Definitionen hingegen betrachten Gruppen als wichtigen Teil des ganzen Graphen und setzen daher Graph und Gruppe mit einander in Beziehung. Definitionen basierend auf der Ähnlichkeit von Knoten, versuchen den Knoten eine Position in einem euklidischen Raum zu geben. Durch den Vergleich von Knotenpaaren lässt sich dann ein Abstand zwischen den Knoten bestimmen. Knoten, die nahe bei einander sind, gehören dann zu einem Gruppe.

Aus Sicht der Informatik ist eine solche Gruppe das gleiche wie ein Cluster. Im Kern ist Community Detection also das gleiche wie Clustering, mit dem Ziel, die Knoten des Graphen zu partitionieren.

Überlappende Gruppen

Ist ein Knoten nicht nur Mitglied einer einzigen Gruppe, sonder Teil mehrere Gruppen, so überlappen sich die Gruppen in diesem Knoten. Ein solcher Knoten wird überlappender Knoten (*overlapping node*) genannt. Die Gruppen haben eine gemeinsame Schnittmenge in

diesem Knoten. Entsprechend Xie u. a. (2011) wird im Falle solcher überlappenden Gruppen (*overlapping communities*) die Menge der gefunden Gruppen *cover* genannt und ist definiert als *cover* $C = \{c_1, c_2, ..., c_n\}$, bei dem ein Knoten zu mehr als nur einer Gruppe gehören darf. Ein Zugehörigkeitskoeffizient (*belonging factor*) $[b_{i1}, b_{i2}, ..., b_{ik}]$ gibt an, wie stark die Zuordnung eines Knotens i zu einer bestimmten Gruppe c $\in 1...k$ ist. Ohne Beschränkung der Allgemeinheit werden die folgenden Gesetzmäßigkeiten als gültig angenommen

$$0 \leq b_{ic} \leq 1 \quad \forall i \in V, \forall c \in C \tag{2.1}$$

$$\sum_{c=1}^{|C|} b_{ic} = 1 \tag{2.2}$$

Dabei ist $|C|$ die Anzahl der Gruppen. Für jede Gruppe besitzt der Knoten einen Zugehörigkeitskoeffizienten zwischen 0 und 1. Die Summe aller Faktoren muss jedoch immer 1 ergeben.

2.2.2. Qualitätsfunktionen

Eine *Partition* beschreibt die Einteilung der Daten in Gruppen. Die Partition repräsentiert das Ergebnis der Algorithmen zu Gruppenfindung. Für die Vergleichbarkeit verschiedener Partitionen eines Graphen, müssen diese mit Hilfe einer Funktion auf einen Zahlenwert abgebildet werden.

Eine noch immer auf Intuition beruhende Definition für Gruppen basiert auf den Kennzahlen *density* und *sparsity*. *Density* beschreibt die Dichte der Kanten innerhalb einer Gruppe, *sparsity* bedeutet die 'Seltenheit' der Kanten zwischen den Gruppen.

Modularität

Als Modularität bezeichnet man die Aufteilung eines Ganzen in mehrere Teile (Module), Sie ist in der Graphentheorie ein Maß für die Struktur eines Graphen. Modularität vergleicht die Dichte eines Subgraphen mit der Dichte eines Nullmodells des Graphen. Ein Nullmodell ist eine Kopie des originalen Graphen, die einige seiner Eigenschaften behält, jedoch keine Gruppenstruktur mehr aufweist, da die Existenz jeder Kante gleich wahrscheinlich ist.

Die Modularität wird definiert als (Newman, 2004)

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - P_{ij}) \delta(C_i, C_j) \tag{2.3}$$

wobei m die Anzahl der Kanten ist. Die Summe iteriert über alle Paare von Knoten i und j, A_{ij} ist die Adjazenzmatrix und P_{ij} die Anzahl der zu erwartenden Kanten aus dem Nullmodell. Die δ-Funktion liefert den Wert 1, wenn die Knoten i und j zur selben Gruppe gehören, ansonsten 0. Die Modularität ist gut, wenn in einer gefundenen Gruppe mehr Kanten verlaufen, als dies im Nullmodell der Fall ist.

Gegenseitige Information

Ein weiteres Maß für die Beurteilung von Partitionen ist gegenseitige Information (*mutual information*) $I(x, y)$. Sie beschreibt den statistischen Zusammenhang zweier Zufallsvariablen x und y und ist definiert als (MacKay, 2002)

$$I(x, y) = \sum_x \sum_y P(x, y,)log \frac{P(x, y)}{O(x)P(y)} \tag{2.4}$$

Sie gibt an, wie viel mit dem Wissen über y für x gelernt werden kann (und anders herum). Würde man die Gruppe einer gefundenen Partition in weitere Gruppen unterteilen, so wäre auf Grund der bedingten Wahrscheinlichkeit die gegenseitige Information immer gleich. Daher wurde die normalisierte gegenseitige Information entwickelt.

Normalisierte gegenseitige Information Die *normalized mutual information* (NMI) wurde von Danon u. a. (2005) entwickelt

$$I_{norm}(x, y) = \frac{2I(x, y)}{H(x) + H(y)} \tag{2.5}$$

mit der Shanon Entropy $H(x) = -\sum_x P(x)logP(x)$.

Diese Formel liefert 1, wenn die Partitionen gleich sind und 0, wenn die Partitionen unabhängig von einander sind.

Formel 2.5 berücksichtigt jedoch nicht den Fall überlappender Gruppen. Für die Bewertung von *covers* haben Lancichinetti u. a. (2009) daher im Anhang ihrer Arbeit die Definition entsprechend erweitert zur *lfkNMI*[2]. Sie liefert für nicht überlappende Gruppen ähnliche Werte, wie Formel 2.5. McDaid u. a. (2011) besprechen in ihrer Arbeit Unzulänglichkeiten der *lfkNMI* von Lancichinetti u. a. (2009) und stellen daher eine verbesserte Variante[3] vor, die sie *NMI<MAX>* nennen.

[2] L,F und K sind die Anfangsbuchstaben der Autoren
[3] https://github.com/aaronmcdaid/Overlapping-NMI

2.2.3. Klassische Ansätze

Generell ist das Finden von Gruppen, d.h. die Partitionierung eines Graphen, kein neues Problem und es existieren bereits viele Ansätze. In Fortunato (2010) wird ein Überblick über unterschiedliche klassische Ansätze zum Finden von Gruppen gegeben. Diese lassen sich unterteilen in divisive Algorithmen, modularitätsbasierte Ansätze, Spektral-Algorithmen und dynamische Algorithmen. Für ein ausführliche Diskussion dieser Ansätze vergleiche Fortunato (2010).

Für die Wahl eines Ansatzes können folgende Kriterien herangezogen werden

- Skalierung für große Graphen
- Anzahl möglicher Gruppen im Vorfeld festgelegt
- Möglichkeit überlappender Gruppen
- Determinismus

Nachfolgend wird mit dem Louvain-Algorithmus ein klassischer Ansatz vorgestellt, bevor dann der alternative Ansatz des Label Propagation besprochen wird. Es ist beispielsweise nicht möglich *Spectral-Clustering* auf große Graphen anzuwenden, da dieser Ansatz nicht gut skaliert. Andere Ansätze ,wie *Graph Partitioning* und *Partitional Clustering* benötigen eine vorher festgelegte Anzahl von Gruppen, in welche die Knoten einsortiert werden sollen.

Hierarchisches Clustering skaliert und benötigt keine vorher bestimmte Anzahl an Gruppen, hat jedoch den Nachteil, dass es schwer ist, aus der Menge der Gruppen auf den verschiedenen Hierarchie-Ebenen, die richtigen auszuwählen.

Eine andere Sorte von Algorithmen kann Knoten immer nur genau einer Gruppe zuordnen und sind daher auch nicht immer geeignet. In diese Gruppe zählt der Algorithmus von Girvan und Newman (Girvan und Newman, 2002). Hierfür wurden aber mittlerweile Erweiterungen entwickelt, die überlappende Gruppen finden können. Je nach Ansatz liefern die Algorithmen nicht-deterministische Ergebnisse. So sind sowohl der Louvain-Algorithmus (Blondel u. a., 2008) als auch der Label Propagation Ansatz (Raghavan u. a., 2007; Gregory, 2010) nicht deterministisch. Der Louvain Algorithmus wird im nächsten Abschnitt besprochen, der Label Propagation Ansatz in Abschnitt 2.2.4.

Modularitätsbasierte Ansätze

Wie in 2.2.2 besprochen, ist die Modularität ein mögliches Maß, um die Partitionen verschiedener Algorithmen vergleichen zu können. Je besser die Modularität, desto besser ist

die Einteilung der Knoten in Gruppen. Modularitätsbasierte Ansätze versuchen daher über die Optimierung der Modularität eine verbesserte Einteilung in Gruppen zu erreichen. Die Optimierung der Modularität ist NP vollständig (Brandes u. a., 2006).

Der 2008 von Blondel u. a. vorgestellte Louvain Algorithmus ist ein sog. gieriger Algorithmus, der in jedem Schritt die Modularität optimiert und bereits nach wenigen Schritten stoppt. Zunächst ist jeder Knoten in einer eigenen Gruppe. In jeder Runde wird für jeden Knoten und alle seine Nachbarknoten, die Änderung der Modularität berechnet für den Fall, dass der Knoten mit einem seiner Nachbarknoten eine Gruppe bildet. Die Konstellation mit der größten positiven Änderung wird beibehalten. Eine so gebildete Gruppe wird dann im nächsten Durchlauf durch einen Superknoten repräsentiert. Eine Kante zu anderen Superknoten existiert dabei dann, wenn es aus der Gruppe mindestens eine Kante zu einem Knoten der Gruppe des anderen Superknoten gibt. Dieser Vorgang wird so lange wiederholt, bis sich keine Verbesserung der Modularität einstellt. Auch für große Graphen stoppt der Louvain-Algorithmus bereits nach 5-10 Iterationen. Der Nicht-Determinismus entsteht durch die Reihenfolge, in der die Knoten in unterschiedlichen Gruppen eingeteilt werden.

2.2.4. Alternative Ansätze

Fortunato (2010) führt auch alternative Ansätze an. Nachfolgend werden einige Ansätze kurz vorgestellt, die sich für eine Umsetzung in Giraph anbieten. Eine vollständiger Überblick findet sich in Fortunato (2010, Kapitel X).

Label Propagation Algorithm (LPA)

Label Propagation wurde von Raghavan u. a. (2007) entwickelt. Zunächst werden alle Knoten mit einem zufälligen aber einzigartigen Label versehen. Zu Beginn jeder Iteration betrachtet ein Knoten die Labels seiner Nachbarknoten. Der Knoten übernimmt das Label, welches die meisten seiner Nachbarn besitzen. Sind zwei oder mehr Label gleich oft vertreten, wählt der Knoten zufällig eines dieser Label aus und der Algorithmus geht in die nächste Iteration. Auf Grund der zufälligen Auswahl ist der Algorithmus nicht deterministisch. Der LPA stoppt, wenn sich kein Label eines Knotens mehr ändert. Es kann passieren, dass die Labels einiger Knoten oszillieren. Um dies zu verhindern, speichert jeder Knoten auch die Labels seiner Nachbarn aus vorherigen Iterationen und greift mit einer definierbaren Häufigkeit auf vergangene Labels zurück, um das jeweils aktuelle Label zu bestimmen.

Es ist möglich, dass am Ende des Algorithmus zwei oder mehr getrennte Gruppen mit dem selben Label existieren. Das liegt daran, dass sich ein Label über einige Knoten hinweg

durchsetzen kann, dann aber die Gruppe durch ein anderes Label getrennt wird. Dadurch entstehen zwei unverbundene Gruppen, die sich im Verlauf weiter entwickeln können. Mit einer einfachen Tiefensuche lassen sich diese beiden Gruppen aber zuverlässig erkennen, so dass jeder Gruppe eindeutige Label zugeordnet werden können. Der LPA läuft daher in zwei Phasen ab: in der ersten Phase werden die Label durch den Graph propagiert, in der zweiten Phase werden die gefundenen Gruppen überprüft und ggf. neu beschriftet.

Barber und Clark (2009) und Leung u. a. (2009) haben Verbesserungen entwickelt, da der Algorithmus oft eine große und viele kleine Gruppen findet (siehe Fortunato (2010)).

Community Overlap PRopagation Algorithm (COPRA)

Mit COPRA (Gregory, 2010) wrde eine Erweiterung des LPA entworfen, die es erlaubt auch überlappende Gruppen zu finden. Statt einem einzigen Label nutzt COPRA die in 2.2.1 beschriebenen Zugehörigkeitskoeffizienten. Jeder Knoten speichert nun mehrere Tupel (c, b), wobei c das Label einer Gruppe ist und b der Zugehörigkeitskoeffizient, der angibt, wie stark der Knoten mit der jeweiligen Gruppe verbunden ist. Übersteigt b den Schwellwert $1/v$, wird das Tupel in die Menge der Label des Knoten aufgenommen. Dabei ist v der Parameter, welcher angibt, zu wie vielen Gruppen ein Knoten maximal gehören kann. Auch COPRA ist ein nichtdeterministischer Algorithmus, da hier ebenfalls bei Labels mit gleichen Zugehörigkeitskoeffizienten zufällig ein Label ausgewählt wird.

2.3. Node Prediction

Node Prediction beschäftigt sich mit dem Entstehen und Verschwinden von Knoten in einem Graphen. Leskovec u. a. (2008) haben einige Netzwerke untersucht und festgestellt, dass das Erscheinen neuer Knoten in einem sozialen Netzwerk sehr stark von externen Faktoren abhängt. Daher lässt sich kein einheitliches Modell entwickeln, wie es beispielsweise für Link Prediction möglich ist.

Leskovec u. a. nutzen deshalb eine vorher definierte *node arrival* Funktion. Diese bestimmen sie, indem sie den Zuwachs aus existierenden Zeitschritten extrapolieren. Jung und Segev (2013) verfahren ähnlich. Sie vergleichen die Änderung zwischen dem aktuellen und dem vorhergehenden Zeitschritt und verwenden dies als Richtwert für den kommenden Zeitschritt. Neben der Frage, wie viele Knoten dem Graph beitreten bzw. ihn verlassen, muss eine Node Prediction auch die Frage beantwortet, wo im Graph die Knoten entstehen oder verschwinden.

2.4. Link Prediction

Link Prediction versucht, basierend auf existierenden Kanten und den Eigenschaften von Knoten, auf die Wahrscheinlichkeit der zukünftigen Existenz von fehlenden Kanten zu schließen. Das heißt, sei ein Graph $G(V, E)$ mit der Menge an Knoten V und der Kantenmenge E gegeben, dann ist U die Menge aller möglichen Kanten in diesem Graphen, definiert als $|V| \cdot |V| - 1$ für gerichtete Graphen und $\frac{|V| \cdot |V| - 1}{2}$ für ungerichtete Graphen. Die Menge der fehlenden Kanten entspricht dann $U - E$ (Lü und Zhou, 2011).

Lü und Zhou (2011) geben einen Überblick über verschiedene Link Prediction Ansätze, sowie den dort zum Einsatz kommenden Kennzahl. Sie beschreiben Link Prediction als das Problem der Schätzung der Wahrscheinlichkeit für die Existenz einer Kante zwischen zwei Knoten, basierend auf den topologischen Eigenschaften eines Graphen und eines betrachteten Knotens. Sie führen an, dass in der Vergangenheit strukturelle Informationen des Graphen, wie die hierarchische Organisation und die Strukturen von Gruppen, wenig genutzt wurden. Die verschiedenen Ansätze können wie folgt unterteilt werden (Lü und Zhou, 2011):

Ähnlichkeitsbasierte Verfahren können weiter unterteilt werden in lokale Ansätze wie Common Neighbors (2.4.1), den Jaccard Koeffizient (2.4.4) und den Ressource Allocation Index (2.4.2). Auf Grund des Ansatzes lokaler Daten in Apache Giraph werden globale Ansätze , zu denen der Katz Index und Random Walk with Restart (RWR) gehören, sowie quasi lokale Verfahren wie der Local Path Index (LP) und Local Random Walk (LRW) in dieser Arbeit nicht betrachtet (Lü und Zhou, 2011; Liben-Nowell und Kleinberg, 2007).

Maximum likelihood Verfahren sind rechenintensive Verfahren und funktionieren daher nur für ein begrenzte Anzahl Knoten < 1000. Zudem sind sie nicht sehr genau, liefern im Gegenzug aber einen tieferen Einblick in die Netzwerkstruktur. Sie lassen sich weiter aufteilen in hierarchische Strukturmodelle und stochastische Blockmodelle.

Probabilistische Modelle versuchen vom zugrunde liegenden Graphen zu abstrahieren und die Vorhersagen auf Basis des gelernten Modells zu machen. Sie werden weiter unterteilt in Probabilistisch-Relationale Modelle, probabilistische Entity Relationship Modelle und stochastische Relationalmodelle.

Nachfolgend werden die lokalen ähnlichkeitsbasierten Ansätze näher betrachtet, da sich diese für den Einsatz mit Giraph anbieten.

2.4.1. Common Neighbors

Common Neighbors ist die Menge, welche beide Knoten gemeinsam als Nachbarn haben

$$CN(a,b) = |\Gamma(a) \cap \Gamma(b)| \tag{2.6}$$

für $\Gamma(a)$ als die Menge der Nachbarknoten von a und entspricht dem Local Path Index der Länge 2 (vgl. 2.4.3). Soundarajan und Hopcroft (2012) haben die Common Neighbors Metrik um Gruppeninformationen für Knoten (CN1) erweitert.

Common Neighbors 1 (CN1)

Basiert auf $CN(a,b)$, addiert aber plus 1 für jeden Nachbarknoten, der in der gleichen Gruppe wie a und b ist. Dadurch werden Verbindungen zwischen Knoten mit großen Schnittmengen zwischen den Gruppen hervorgehoben.

$$CN1(a,b) = CN(a,b) + \sum_{i \in \Gamma(a,b)} |\Gamma(i) \cap \Gamma(a) \cap \Gamma(b)| \tag{2.7}$$

2.4.2. Resource Allocation

Resource Allocation (RA) nimmt an, dass jeder Knoten eine Resource gleichmäßig über alle seine Kanten versenden kann. RA berechnet die Menge an Ressourcen, die ein Knoten von einem anderem empfangen kann.

$$S_{ab}^{RA} = \sum_{z \in \Gamma(a) \cap \Gamma(b)} d_z^{-1} \tag{2.8}$$

wobei d_z der Grad des Knoten z ist. Auch hier haben Soundarajan und Hopcroft (2012) die Metrik wieder um Gruppeninformationen für Knoten (RA1) erweitert.

Resource Allocation 1 (RA1)

S_{ab}^{RA1} passt S_{ab}^{RA} dahingehend an, dass nur die Nachbarn i berücksichtigt werden, welche in der gleichen Gruppe wie a und b sind und gewichtet den Beitrag der Knoten i durch die Anzahl gemeinsamer Gruppen mit a und b

$$S_{ab}^{RA1} = \sum_{i \in \Gamma(a,b)} \frac{|\Gamma(i) \cap \Gamma(a) \cap \Gamma(b)|}{d(i)} \tag{2.9}$$

$\Gamma(a, b)$ repräsentiert die Menge der gemeinsamen Nachbarn von A und B.

2.4.3. Local Path Index

Der Local Path Index ist ein in Summe und Länge verkürzter Katz Index (Lü und Zhou, 2011). Dieser zählt die Anzahl von Pfaden einer Länge n, von einem Knoten zu allen anderen Knoten im Graph. Die Zahl der Pfade wird dann gewichtet und über alle Pfadlängen aufsummiert. Der Local Path Index ist definiert als

$$S^{LP(n)} = A^2 + \in A^3 + \in^2 A^4 + \cdots + \in^{n-2} A^n \tag{2.10}$$

wobei A^n die Summer aller Pfade der Länge n ist, und die betrachteten Knoten verbinden. Experimente haben gezeigt, dass der Index gute Ergebnisse für ein n gleich der mittleren durchschnittlichen Pfadlänge des Netzwerkes liefert (Lü u. a., 2009).

2.4.4. Jaccard Koeffizient

Der Jaccard Koeffizient arbeitet auf der Menge direkter Nachbarn zweier Knoten. Er beschreibt das Verhältnis der gemeinsamen Knoten zur Gesamtanzahl aller Nachbarn der beiden Knoten (Zheleva u. a., 2010; Lü und Zhou, 2011):

$$Jaccard(a, b) = \frac{\Gamma(a) \cap \Gamma(b)}{\Gamma(a) \cup \Gamma(b)} \tag{2.11}$$

Die Funktion $\Gamma(a)$ ist nicht nur beschränkt auf die direkten Nachbarn, sondern lässt sich bei Bedarf auch für Nachbarn zweiten, dritten oder n-ten Grades definieren.

2.4.5. Weitere gruppenbasierte Kennzahlen

In ihrer Arbeit haben Zheleva u. a. (2010) weitere Kennzahlen vorgestellt, die das Verhältnis eines Knoten zu einem anderen Knoten basierend auf Gruppeninformationen ausdrücken und für Link Prediction genutzt werden können. Dies sind unter anderen:

- **Dichte gemeinsamer Nachbarn**
  Die Dichte der gemeinsamen Nachbarn beschreibt den Grad der Verknüpfung von Knoten in der Menge gemeinsamer Nachbarn (*common neighbors*) zweier Knoten. Es ist das Verhältnis der vorhandenen Kanten dieser Menge, zur Anzahl der möglichen Kanten zwischen all diesen Knoten. Die Dichte ist auch bekannt als Cluster Koeffizient.

- **Gruppengröße**
  Die Größe der Gruppen, zu welchen der Knoten gehört.

- **Nachbarn in der Gruppe**
  Die Anzahl an Nachbarn, die in der gleichen Gruppe sind.

- ***Nachbarn-Gruppen-Verhältnis***
  Das Verhältnis zwischen der Anzahl der Nachbarn in der Gruppe und der Gruppengröße.

2.4.6. Qualitätsfunktion

Für die Überprüfung der Qualität einer Link Prediction eignen sich, die aus dem Information Retrieval bekannten Funktionen *precision* und *recall*[4]. Precision ist im Information Retrieval definiert, als

$$precision = \frac{|\{\text{relevante Dokumente}\} \cap \{\text{gefundene Dokumente}\}|}{|\{\text{gefundene Dokumente}\}|} \tag{2.12}$$

und beschreibt den Anteil relevanter Dokumente in der Ergebnismenge einer Abfrage. Recall, definiert als

$$recall = \frac{|\{\text{relevante Dokumente}\} \cap \{\text{gefundene Dokumente}\}|}{|\{\text{relevante Dokumente}\}|} \tag{2.13}$$

und ist der Anteil relevanter Dokumente aus der Ergebnismenge, im Verhältnis zur Menge aller relevanter Dokumente.

Für Link Prediction lassen sich diese Funktionen definieren als (Jung und Segev, 2013):

$$precision = \frac{|\{\text{Kanten aus der Vorhersage}\} \cap \{\text{Kanten aus dem org. Datensatz}\}|}{|\{\text{Kanten aus dem org. Datensatz}\}|} \tag{2.14}$$

so dass *precision* hier der Anteil der generierten Kanten im Verhältnis zu der Menge der Kanten ist, die hätten generiert werden sollen. Recall ist definiert, als

$$recall = \frac{|\{\text{Kanten aus der Vorhersage}\} \cap \{\text{Kanten aus dem org. Datensatz}\}|}{|\{\text{Kanten aus der Vorhersage}\}|} \tag{2.15}$$

und ist der Anteil korrekt vorhergesagter Kanten aus der Menge der vorhergesagten Kanten.

[4] http://www.springerreference.com/docs/html/chapterdbid/66221.html

2.5. Community Tracking

Inhalt des Community Trackings ist es, Gruppen über verschiedene Zeitintervalle hinweg zu beobachten. Hierzu werden Gruppen, die in verschiedenen Zeitintervallen gefunden werden, einander zugeordnet. In der Entwicklung einer Gruppe werden sieben Ereignisse unterschieden: Geburt, Fortbestand, Tod, Wachsen, Schrumfen, Aufteilen und Zusammengehen. Abbildung 2.1 zeigt Geburt der Gruppe D2 in Zeitschritt $t = 2$, den Tod von Gruppe D3 sowie den Fortbestand von Gruppe D1. Abbildung 2.2 zeigt, wie sich eine Gruppe (D3) aufteilt in eine kleinere Gruppe D3 und eine neue Gruppe D4. Ebenfalls zu sehen ist das Zusammengehen der Gruppen D1 und D2 im Zeitschritt $t = 2$. Die Ereignisse Vergrößern und Schrumpfen sind in den Abbildungen nicht dargestellt. Nachfolgend werden verschiedene Ansätze vorgestellt, mit

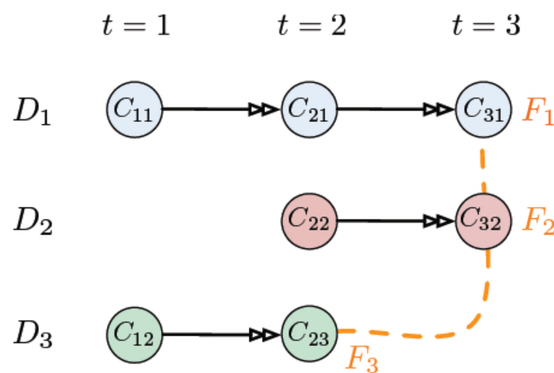

Abbildung 2.1.: Beispiel für 3 Gruppen (Fortbestand, Geburt und Tod)(Greene u. a., 2010)

denen sich Gruppen einander zuordnen lassen.

Evolution of Communities in Dynamic Social Networks

Greene u. a. (2010) unterscheiden zwei Sorten von Gruppen: *step communities* C und *dynamic communities* D. *Step communities* sind Gruppen innerhalb eines bestimmten Zeitintervalls des Graphen, *dynamic communities* D sind Gruppen, die über den gesamten Beobachtungsvorgang existieren. Für jedes Zeitintervall wird versucht, eine *step community* einer existierenden *dynamic community* zuzuordnen In den Abbildung 2.1 und 2.2 werden die Gruppen C_i den dynamischen Gruppen D_i zugeordnet. Konnte eine *dynamic community* über einige Zeitschritte hinweg keiner *step community* zugeordnet werden, kann davon ausgegangen werden, dass die *dynamic community* nicht mehr existiert. Konnte eine *step community* keiner existierenden *dynamic community* zugeordnet werden, so ist eine neue *dynamic community* entstanden.

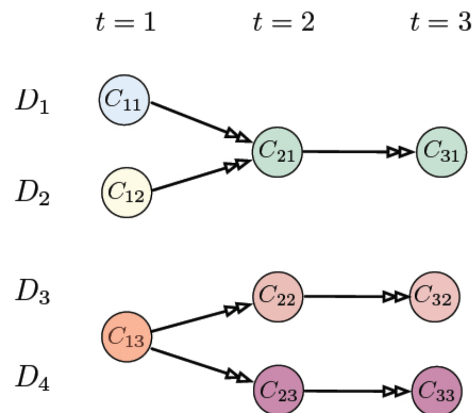

Abbildung 2.2.: Beispiel für 4 Gruppen (Zusammengehen und Aufteilen)(Greene u. a., 2010)

Zum Testen ihres Vorgehens habe Greene u.a. den Graph-Generator von Lancichinetti und Fortunato (2009) angepasst, so dass auf den generierten Graphen gezielt bestimmte Veränderungen der Gruppen simuliert werden können, wodurch verschiedene Zeitschritte erzeugt werden. Für das Finden der *step communities* wurde der MOSES Algorithmus (McDaid und Hurley, 2010) verwendet.

Community Dynamics in Evolutionary Networks

Ein sehr ähnlicher Ansatz für Community Detection kommt von Chen u. a. (2010). Auch hier werden die sieben möglichen Ereignisse identifiziert, wobei Fortbestand kein Ereignis im Verständnis der Autoren ist und daher nur sechs Fälle betrachtet werden. Die Autoren beschränken sich bei der Definition ihrer Gruppen allerdings auf Cliquen und lassen die Abgrenzung über dichtere Subgraphen außen vor. Der Grund hierfür sind die untersuchten Daten, in diesem Fall biologischen Netzwerke. Die Autoren erkennen den erheblichen Rechenaufwand, der entsteht, wenn man jede gefundene Gruppe aus dem einen Zeitschritt, mit jeder anderen Gruppe aus dem folgenden Zeitschritt vergleichen muss. Daher berechnet der Ansatz von Chen u. a. sogenannte *graph*- und *community representatives*, wodurch der Rechenaufwand reduziert werden soll. *Graph representatives* sind Knoten, die in beiden Zeitschritten vorkommen. Knoten, die nur in einem Zeitschritt vorkommen, heißen *graph dependent*. Besteht eine Gruppe nur aus *graph dependent* Knoten, wird die Gruppe nicht weiter betrachtet. *Community representatives* sind Knoten, die eine Gruppe repräsentierten. Eine solcher Knoten ist in möglichst wenig anderen Gruppen gleichzeitig. Trifft diese Eigenschaft auf mehrere Knoten zu, wird zufällig einer ausgewählt.

Durch *graph representatives* wird die Anzahl der möglichen Gruppen reduziert, da *graph dependent* Gruppen aussortiert werden. Durch *community representatives* wird der Vergleich auf die Gruppen reduziert, die den *community representative* enthalten.

Evolutionary Clustering

Ein weiterer Ansatz findet sich in der Arbeit von Chakrabarti u. a. (2006). Das dort entwickelte *evolutionary clustering* Framework berechnet eine *Snapshot-Qualität* sowie *historische Kosten* für eine Gruppe. Die *Snapshot-Qualität* beschreibt die Qualität der Partition des jeweiligen Zeitintervalls, vergleichbar mit der Modularität. Die *historische Kosten* sind ein Maß für den Abstand einer Gruppe aus dem Zeitintervall $t - 1$ zu ihrer Version in Intervall t. Beide Funktionen müssen vom Anwender bereitgestellt werden. Zudem führen Chakrabarti u. a. den Begriff der *zeitlichen Ähnlichkeit* ein. Sie bezeichnen damit das Muster der zeitlichen Anwesenheit einer Gruppe über die Zeitintervalle hinweg.

An einem Beispiels erläutern sie den Grund für die Notwendigkeit historischer Kosten (Chakrabarti u. a., 2006):

> "For intuition, we consider an extreme example to show why the introduction of history cost may have a significant impact on the clustering sequence. Consider a data set in which either of two features may be used to split the data into two clusters: feature A and feature B. Each feature induces an orthogonal split of the data, and each split is equally good. However, on odd-numbered days, feature A provides a slightly better split, while on even-numbered days, feature B is better. The optimal clustering on each day will shift radically from the previous day, while a consistent clustering using either feature will perform arbitrarily close to optimal. In this case, a clustering algorithm that does not take history into account will produce a poor clustering sequence."

Während die Community-Detection-Ansätze von Greene u. a. und Chen u. a. unabhängig des vorgeschalteten Gruppenfindungsalgorithmus arbeiten, beeinflussen Chakrabarti u. a. den jeweiligen Algorithmus zur Gruppenfindung, basierend auf den Informationen aus den vorangegangenen Zeitschritten, um ein möglichst konsistentes Bild zu erhalten.

2.6. Verwandte Arbeiten

Jung und Segev (2013) untersuchen die Anwendung von Node- und Link Prediction, um damit die Entwicklung von Gruppen in einem Zitationsnetzwerk vorherzusagen. Die Algorithmen

werden dabei als eigenständige Blöcke betrachtet, die dann wie in Abbildung 2.3 kombiniert werden. Dabei steht N für Node Prediction, L für Link Prediction und C für Community Detection.

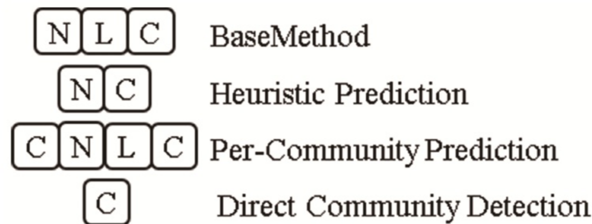

Figure 4. Outline of methods

Abbildung 2.3.: Die Block-Kombinationen von Jung und Segev (2013)

Für den NC-Ansatz wird untersucht, wie viele Kanten im Node Prediction hinzugefügt werden können, bevor die Qualität der Vorhersage abfällt. Bei diesem Ansatz findet keine Link Prediction statt. Für die untersuchten Datensätze wird ein Optimum bei zehn Kanten pro neuem Knoten erreicht.

Die beiden Ansätze NLC und CNLC verwenden verschiedene Verfahren zur Link Prediction, da im CNLC Ansatz auf die Community-Informationen zurückgegriffen werden kann. Der C Ansatz berechnet und vergleicht die Gruppen auf Basis des aktuellen Zeitpunkts. Im Vergleich zu den anderen Ansätzen liefert der C Ansatz, für die verwendeten Datensätze, beste Prognose für den nächsten Zeitschritt (entspricht einem Jahr). Allerdings werden die anderen Ansätze mit steigender Größe des Graphs besser. Für eine Prognose über 5 Jahre sind die Ansätze NC, NLC und CNLC besser.

2.7. Apache Giraph

Apache Giraph[5] ist ein Graph Processing System. Es realisiert das von Google vorgestellte Konzept, Algorithmen aus der Sicht einzelner Knoten zu implementieren (Pregel, Malewicz u. a. (2010)). Giraph dient als Plattform für die Ausführung der zu entwickelnden AlgorithmenGiraph und Pregel basieren beide auf dem Bulk Synchronous Parallelism Prinzip (EncyclopediaParallelComputing, 2011). Dabei wird für jeden Knoten ein aktueller Zustand errechnet. Erst wenn alle Knoten aus der Iteration einen aktuellen Zustand berechnet haben, ist die Synchronisationsbarriere erreicht und einen neue Iteration kann beginnen. Das Giraph-Framework

[5] http://giraph.apache.org

selbst erzeugt einen *Job*, welcher als MapReduce[6] Job auf Apache Hadoop[7] ausgeführt wird. Giraph-Jobs bestehen allerdings nur aus einem Mapper. Ein Reducer wird nicht benötigt, da Giraph versucht die Daten vollständig im Hauptspeicher zu halten, um langsame Zugriffe auf Massenspeicher zu vermeiden. Somit wird auch ein großer Nachteil einer Hadoop Anwendung vermieden.

Giraph als Graph Processing System arbeitet prinzipiell auf dem ganzen Graphen, wobei es in der Regel darum geht, den Zustand der Knoten zu ändern. Graph Datenbanken hingegen ermöglichen Abfragen auf großen Datenmengen mit der generischen Struktur eines Graphen. Bei Graphdatenbanken liegt der Fokus auf dem Finden von Knoten oder Pfaden mit bestimmten Eigenschaften (*pattern matching*). Das heißt, es sollen Teile eines Graphen gefunden und selektiert werden.

Giraph nutzt Apache Hadoop und dessen verteiltes Dateisystem für die Datenvorhaltung. Mit Input-Formaten können verschiedenste Daten für die Map-Tasks in Hadoop eingelesen werden und am Ende der Berechnungen mit Output-Formaten wieder in das verteilte Dateisystem geschrieben werden. Da in einem solchen verteilten System immer Fehler auftreten

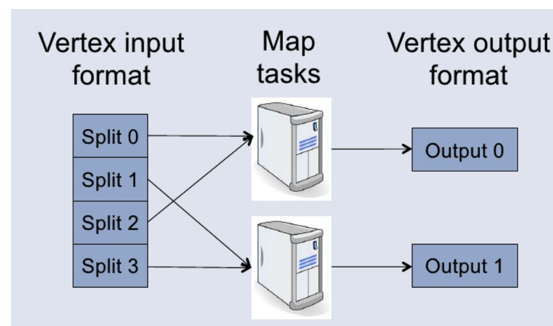

Abbildung 2.4.: Map-only Job (Ching, 2012)

können, erlaubt Giraph mit dem Konzept des Checkpointing einen Zwischenstand der Berechnungen persistent im Hadoop Cluster zu speichern und im Fehlerfall die Berechnung wieder am Checkpoint aufzunehmen.

Um Cluster-Ressourcen zu schonen, kann sich ein Knoten selbst deaktivieren, wenn keine weiteren Berechnungen durchzuführen sind (vgl. Listing 2.1 Zeile 17). Sendet ein Knoten eine Nachricht an den deaktivierten Knoten, wird dieser reaktiviert. Haben sich alle Knoten

[6](White, 2012, Kapitel 2)
[7]http://hadoop.apache.org

deaktiviert und wird kein Knoten durch eine Nachricht aktiviert, terminiert die Berechnung. Natürlich können auch andere Abbruchbedingungen implementiert werden.

Giraph bietet die Möglichkeit Aggregatoren zu implementieren. Dies sind eigenständige Einheiten, an welche Knoten Daten senden können, um sie dort vereinigen zu lassen. Am Anfang der nächsten Iteration können die aggregierten Werte aus den vorherigen Iterationen abgefragt werden. Da Aggregatoren kommutativ sind, können beliebig viele hintereinander geschaltet werden und jeweils die Ergebnisse der vorgelagerten Aggregatoren zu kumulieren. Dieses Vorgehen wird *sharding* genannt. Giraph verwaltet dies automatisch und verhindert so, dass Aggregatoren zu einem Flaschenhals werden.

Listing 2.1 zeigt den Java-Quellcode für die Berechnung des maximalen Knotenwerts im Graphen aus der lokalen Sicht eines Knotens.

```
1  class MaxValueVertex extends Vertex<I,V,E,M> {
2
3    void compute(Iteratble<M> messages){
4      V maxValue = getVertexValue();
5      for (M message : messages){
6        M msgValue = message.getValue();
7        if( msgValue.compareTo(maxValue) > 0) {
8          maxVaule = msgValue;
9        }
10     }
11     if (maxValue.compareTo(getVertexValue()) > 0) {
12       setVertexValue(maxValue);
13       for(I endpoint : getOutNeigbhors()){
14         sendMessageTo(endpoint, getVertexVaue());
15       }
16     }
17     voteToHalt();
18   }
19 }
```

Listing 2.1: MaxValueVertex (Martella, 2012):

3. Analyse

Dieses Kapitel untersucht zunächst, welche Anforderungen an die Algorithmen durch die Verwendung von Apache Giraph gestellt werden. Anhand dieser Analyse werden Algorithmen für Gruppenfindung, Link- und Node Prediction tiefergehend und ausgewählt.

3.1. Problembeschreibung

Community Tracking basiert in vielen Ansätzen (Greene u. a., 2010; Chen u. a., 2010) auf dem Vergleich zweier Gruppen aus zwei verschiedenen Zeitschritten. Sind beiden Gruppen ausreichend ähnlich, so wird angenommen, dass es sich bei der Gruppe aus dem späteren Zeitschritt um eine Weiterentwicklung der Gruppe handelt. Abbildung 3.1 a zeigt die Teilschritte des Community Tracking. Zuerst wird der Ausgangsdatensatz in Zeitintervalle eingeteilt. Im zweiten Schritt werden dann, mittels Community Detection, Gruppen in den jeweiligen Zeitintervallen bestimmt. In einem dritten Schritt werden, mit Hilfe von Community Tracking, die Gruppen verglichen, um so auf eine zeitliche Entwicklung der Gruppen schließen zu können.

Für sehr große Zeitabschnitte, für Datensätze mit fehlenden Zeitschritten oder Daten mit ungünstigen Eigenschaften (Chakrabarti u. a., 2006) kann es vorkommen, dass die gefundenen Gruppen nicht mehr ausreichend ähnlich sind, um sie einander zuordnen zu können. Das Beobachten der Gruppen über die Zeit ist unter diesen Umständen nicht mehr oder nur eingeschränkt möglich. Node- und Link Prediction Verfahren berechnen die Wahrscheinlichkeit für die An- bzw. Abwesenheit eines Knoten oder einer Kante für einen zukünftigen Zeitabschnitt. Mit Hilfe dieser Verfahren soll der Zeitschritt t_z zwischen t_1 und t_2 generiert werden, für den die Veränderung der Gruppe von t_1 nach t_z und t_z nach t_2 klein genug ist, um beim Vergleich im Community Tracking eine ausreichende große Ähnlichkeit zu erhalten.

In Abbildung 3.1 b ist dargestellt, wie mit Node- und Link Prediction, eine Lücke zwischen zwei Zeitschritten geschlossen werden kann, falls der Abstand zwischen den Zeitintervallen zu groß ist für Community Tracking. Für die vorhandenen Datensätze wurden die Gruppen bereits mit Community Detection identifiziert. Nun wird mit Node- und Link Prediction aus dem früheren Zeitintervall ein Zwischenschritt generiert. Hierbei werden mit Node

Prediction neue Knoten erstellt oder entfernt. Link Prediction generiert neue Kante zwischen den Knoten. In einem zweiten Schritt werden dann auch für das generierte Zeitintervall Gruppen mit Community Detection ermittelt. Im dritten Schritt ist es nun möglich die Gruppen mit Community Tracking zunächst zwischen dem frühen und dem generierten Intervall zu vergleichen. Anschließend können auch die Gruppen zwischen dem generierten und dem späteren Intervall verglichen werden. Nun ist es möglich eine Gruppe aus dem späteren Intervall, einer Gruppe im früheren Intervall zuzuordnen, was ohne den Zwischenschritt nicht möglich gewesen wäre.

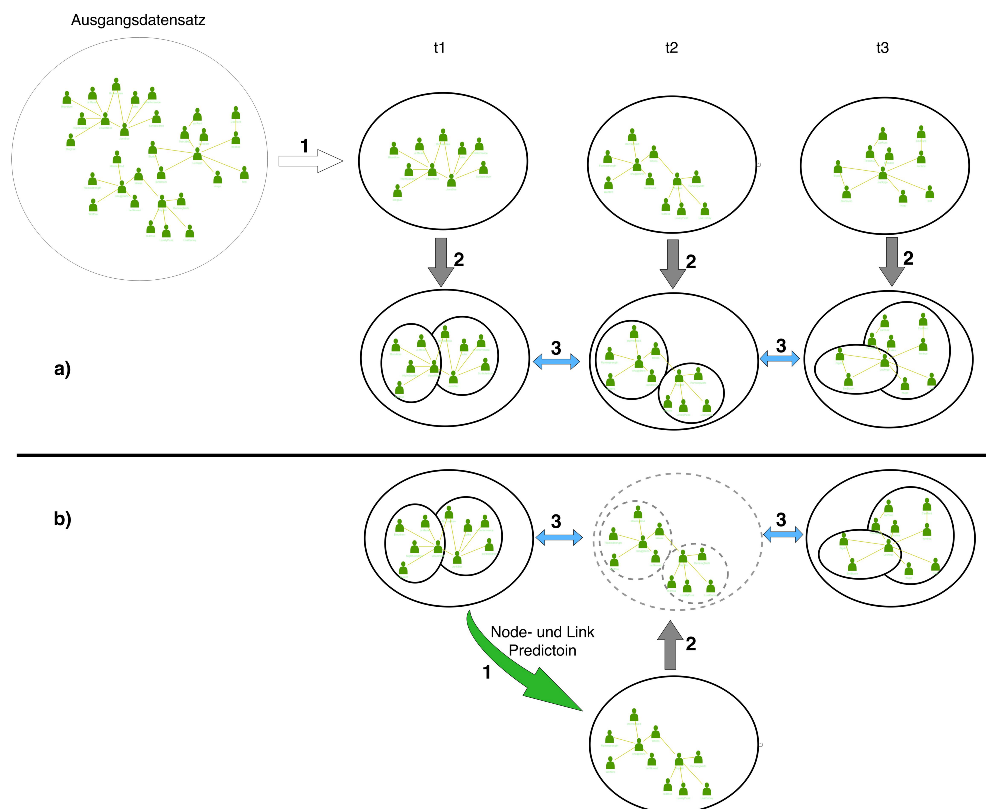

Abbildung 3.1.: a) der reguläre Ablauf für Daten mit einer ausreichenden Qualität b) das Generieren eines Zwischenschritts mit Hilfe von Node- und Link Prediction

Apache Giraph ermöglicht es Graphen mit Millionen von Knoten und Billionen von Kanten zur verarbeiten. Algorithmen in Giraph werden aus der lokalen Sicht eines Knoten geschrieben, weshalb ein Knoten nur auf die Informationen zurückgreifen kann, die ihm lokal zur Verfügung stehen. Für Giraph eignen sich daher Algorithmen, die ohne eine globale Sicht auskommen

oder sich über einige wenige und nicht komplexe, globale Werte steuern lassen. Abbildung 3.2 zeigt die Verteilung der Knoten auf verschiedene Worker. Worker sind die Maschinen eines Cluster, welche für Berechnungen zur Verfügung stehen. Jeder Worker berechnet die ihm zugewiesenen Knoten des Graph. Abbildung 3.2 zeigt auch das *sharding* von Aggregatoren. Damit Aggregatoren nicht zum Flaschenhals werden, verteilt Giraph Aggregatoren auf jedem Worker, deren Werte dann später zusammengeführt werden. Der Master und die Aggregatoren *Agg1* und *Agg2* werden auf einem dedizierten Master-Worker im Cluster ausgeführt.

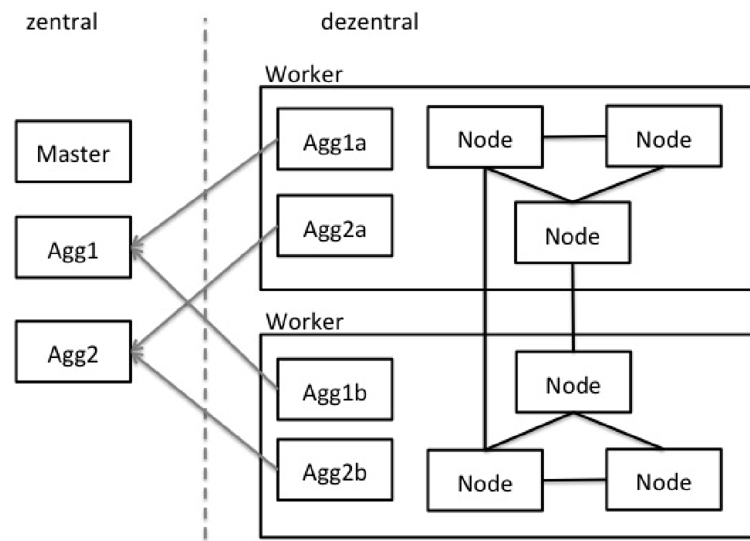

Abbildung 3.2.: Zentrale und dezentrale Komponenten in Giraph

3.2. Community Detection

Für die Wahl eines Gruppenfindungsalgorithmus wurden die folgenden Kriterien herangezogen:

- Der Algorithmus soll in der Lage sein, Knoten nicht nur einer, sondern mehreren (überlappenden) Gruppen zuordnen zu können.
- Soziale Graphen können Millionen Knoten enthalten. Für solche Graphen ist die Abschätzung der vorhandenen Gruppen schwierig. Die Anzahl der gefundenen Gruppen soll sich daher aus dem Algorithmus heraus ergeben und nicht durch den Anwender im Vorfeld festgelegt werden müssen.

- Die Entscheidung über die Zugehörigkeit eines Knotens zu einer Gruppe soll im jeweiligen Knoten getroffen werden.
- Der Algorithmus soll primär auf der Struktur des Graphen arbeiten und ohne zusätzliche Informationen auskommen, um Knoten einer Gruppe zuordnen zu können. Jedoch sollte die Möglichkeit bestehen, zusätzlich vorhandene Daten durch minimale Anpassungen im Algorithmus mit in die Entscheidungsfindung einfließen zu lassen.

Die Wahl viel dabei auf die Label Propagation Erweiterung COPRA (Raghavan u. a., 2007).

Label Propagation

Der in 2.2.4 beschriebene Label Propagation Algorithmus (LPA) (Raghavan u. a., 2007) erlaubt es, einen Knoten, basierend auf den Informationen aus der direkten Nachbarschaft, einer Gruppe zuzuordnen.

Der von Gregory (2010) vorgestellte Community Overlap PRopagation Algorithmus (COPRA) erweitert das Konzept des LPA, um auch überlappende Gruppen finden zu können. In COPRA besteht ein Label aus einer Menge von Paaren (c, b) mit den Gruppen c und den Zugehörigkeitskoeffizienten b. Die Summe über alle b für einen Knoten wird auf 1 normiert. Es wird eine Funktion $b_t(c, x)$ definiert, welche einem Knoten x für eine Gruppe c den entsprechenden Zugehörigkeitskoeffizienten berechnet (Gregory, 2010) :

$$b_t(c, x) = \frac{\sum_{y \in N(x)} b_{t-1}(c, y)}{|N(x)|} \tag{3.1}$$

$N(x)$ liefert die Nachbarn des Knoten x.

Im Vergleich zum LPA, benötigt COPRA einen Eingabeparameter v. Der Parameter definiert, zu wievielen Gruppen ein Knoten maximal gehören kann und somit die Anzahl der Paare (c, b), die ein Knoten in seinem Label speichern kann. Gleichzeitig dient der Wert $1/v$ als Grenzwert, welcher ein Zugehörigkeitskoeffizient b übersteigen muss, damit der Knoten zur Gruppe c aus (c, b) gehören kann.

Erfüllt keines der Paare (c, b) diese Eigenschaft, wird das Paar mit dem größten Wert für b gewählt. Trifft dies auf mehrere Paare zu, so wird zufällig eines der Paare ausgesucht. Anschließend wird das Label des Knoten normalisiert, so dass dessen Summe über alle b gleich 1 ist.

Das Stop-Kriterium wird über eine Minimum-Partition m_t definiert, welche aus der Anzahl und Größe der gefundenen Gruppen zum Zeitpunkt t besteht. Das Stop-Kriterium ist erfüllt, wenn sich die Minimum-Partition m_t zur vorhergehenden Minimum-Partition m_{t-1} nicht

geändert hat. Für jeden Schritt t wird zunächst überprüft, ob sich die Menge c_t der gefundenen Gruppen geändert hat. Ist keine Gruppe verschwunden oder neu hinzugekommen, werden die Größen der Gruppen verglichen. Ist eine Gruppe im Vergleich zum vorherigen Schritt geschrumpft, wird das Minimum für die Gruppe aktualisiert. Die Minimum-Partition hat sich somit geändert und der Algorithmus beginnt die nächste Iteration.

$$\begin{aligned} & m_t = \{(c,i) : \exists p \, \exists q \, (\, (c,p) \in c_{t-1} \wedge (c,q) \in c_t \wedge i = min(p,q) \,)\}, \quad \text{if } i_t = i_{t-1}, \\ & m_t = c_t, \quad \text{otherwise} \end{aligned} \tag{3.2}$$

Es gibt keinen Beweis dafür, dass dieses Stop-Kriterium die besten Lösungen erzeugt, in der Praxis liefert es jedoch akzeptable Ergebnisse(Gregory, 2010).

Der COPRA Algorithmus besteht aus drei Teilaufgaben, die sich in zwei Phasen aufteilen lassen. In der ersten Phase wird das eigentliche Label Propagation durchgeführt. In der zweiten Phase werden zuerst Subgruppen entfernt und anschließend getrennte Gruppen gesucht und umbenannt.

Im Unterschied zum LPA, muss bei COPRA jeder Knoten, in jeder Iteration, Nachrichten versenden, da sich in jedem Schritt die Zugehörigkeitskoeffizienten ändern können. Dieser Mehraufwand muss hingenommen werden, um überlappende Gruppen finden zu können.

Xie und Szymanski (2011) beschreiben eine Optimierung des LPA. Ein *innerer Knoten* ist ein Knoten, bei dem alle Nachbarn das gleiche Label besitzen. Alle anderen Knoten sind *Randknoten*. Ein Knoten ist *passiv*, wenn er bei einem Update sein Label nicht ändert. Ein *aktiver* Knoten hingegen ändert sein Label durch ein Update. Daraus ergeben sich drei Klassen von Knoten: *passive innere Knoten, passive Randknoten, aktive Randknoten*. Der Algorithmus klassifiziert die Knoten entsprechen und bearbeitet anschließend nur noch die aktiven Knoten. Für die LPA Variante wäre hiermit ein Kriterium definiert, um Knoten aus der Berechnung einer Iteration vorübergehend zu entfernen (vgl. *voteToHalt()* in Listing 2.1, Zeile 12). Dieses Kriterium lässt sich jedoch nicht für den COPRA Algorithmus übertragen, da hier bereits Änderungen des Zugehörigkeitskoeffizienten Einfluss auf andere Knoten haben, ohne dass sich dadurch beim Knoten die Gruppen geändert haben müssen.

Vergleich von Partitionen

Für den Vergleich, der mit Label Propagation erstellten Partitionen, können die in 2.2.2 beschriebenen Kennzahlen genutzt werden. Die gegenseitige Information (NMI) eignet sich vor allem für das Testen von Algorithmen. Nämlich dann, wenn der Algorithmus Gruppen findet und diese mit einer vorhandenen Wahrheit (*ground truth*) verglichen werden. Die gegenseitige

Information nutzt die Identifikatoren der Knoten, um deren Gruppenzugehörigkeit zu prüfen. Ändern sich die Identifikatoren im Vergleich zum Ground-Truth-Datensatz, funktioniert die gegenseitige Information nicht mehr. Die Identifikatoren ändern sich, wenn Knoten durch die Node Prediction hinzugefügt werden.

Die Modularität hingegen arbeitet auf der Struktur des Graphen und ist dadurch unabhängig von den Knoten-Identifikatoren. Sie eignet sich daher gut, um Graphen mit vorhergesagten Knoten und Kanten zu vergleichen. Allerdings lässt sich aus der Modularität nicht erkennen, wieviele Gruppen korrekt wiedererkannt werden. Für den Vergleich einer Partition, die vorhergesagten Knoten enthält, mit einer Ground-Truth-Partition, eignen sich die Community Tracking Verfahren besser.

3.3. Node Prediction

Node Prediction ist für das Entfernen vorhandener Knoten sowie das Erstellen neuer Knoten verantwortlich. Während bei klassischen Verfahren eine globale Funktion darüber entscheidet, wo und wieviele Knoten erstellt werde, sollen diese Entscheidungen nun dezentral in jedem Knoten getroffen werden können. Es ergeben sich folgende Anforderungen:

- Ein Knoten entscheidet lokal, ob er den Graphen verlassen soll und entfernt sich ggf. selbst
- Die Entscheidung über das Erstellen eines Knotens soll lokal getroffen werden.
- Der Node Prediction Algorithmus muss nicht nur Knoten erstellen, sondern diese auch initial mit dem Graph verbinden. Diese initialen Kanten werden durch das dem zu Grunde liegenden Modell ausgewählt.

Der Grund für das Entstehen oder Verschwinden von Knoten hängt sehr stark von externen, nicht messbaren Faktoren ab. Zufälliges Raten ist jedoch für einen Graph mit mehreren Millionen Knoten nicht erfolgsversprechend. Daher sollen auch für die Vorhersage neuer oder zu löschender Knoten auf lokale, strukturelle Informationen zurückgegriffen werden.

3.3.1. Erstellen neuer Knoten

Jung und Segev (2013) entwickeln in ihrer Arbeit einen Knoten-Prädiktor (*node predictor*) für Zitationsnetzwerke. Dieser errechnet die Anzahl, der zu erstellenden neuen Knoten und verbindet diese anschließend mit dem Graph. Hierfür werden zunächst die beiden vorangegangenen Zeitschritte betrachtet und daraus der Zuwachs an Knoten errechnet. Dieser

gilt als Anhaltspunkt für die Anzahl neuer Knoten im nächsten Zeitschritt. Der Vorteil an Zitationsnetzwerken ist, dass Knoten nicht verschwinden können. Der Knoten-Prädiktor muss daher nur neue Knoten erstellen, aber keine Knoten löschen. Da diese Arbeit, im Unterschied zu Jung und Segev (2013), keine Vorhersagen in die Zukunft macht, sondern Zwischenschritte zu einer bekannten Zukunft errechnet, werden für diese Arbeit die Anzahl der Knotenänderungen durch den Anwender vorgegeben, da sie sich leicht aus den vorhandenen Datensätzen entnehmen lassen.

Für die Anbindung neuer Knoten an den Graph wählen Jung und Segev die Preferential Attachment Theorie, nach der sich neue Knoten bevorzugt an Knoten mit vielen Kanten bilden. Im Kontext der Zitationsnetzwerke bedeutet das, dass vielzitierte Arbeiten mit einer hohen Wahrscheinlichkeit auch von neuen Arbeiten zitiert werden ("the rich are getting richer")(Jung und Segev, 2013). Die Preferential Attachment Theorie kommt auch in dieser Arbeit zum Einsatz.

3.3.2. Erstellen initialer Kanten

Ein neu erstellter Knoten muss mit dem Graphen verbunden werden, da er sonst in keiner Beziehung zu anderen Knoten steht. Dann gäbe es auch keinen Ansatzpunkt für die anschließende Link Prediction. Für die Qualität der Link Prediction ist die Anzahl, der initial in der Node Prediction erstellten Kanten, eine wichtige Eingangsgröße. Für das Erstellen dieser initialen Kanten, ist die Preferential Attachment Theorie ein naheliegender Ansatz.

Preferential Attachment Theorie

Die Preferential Attachment Theorie (Newman, 2005) berechnet die Wahrscheinlichkeit p_i, das ein Knoten eine neue Kante erhält, auf dem Verhältnis der Kanten k_i des Knoten zur Gesamtanzahl aller Kanten.

$$p_i = \frac{k_i}{\sum_j k_j} \tag{3.3}$$

Je mehr Kanten ein Knoten hat, desto größer die Wahrscheinlichkeit eine neue Kante zu erhalten.

Preferential Attachment Theorie mit Gruppeninformation

Die Wahrscheinlichkeit p_i kann modifiziert werden, um basierend auf zusätzlichen Informationen, eine besser Entscheidung treffen zu können. Nachfolgend steht der Index c in p_i^c für die Wahrscheinlichkeit des Erstellens eines neuen Knotens (*create*) und der Index d entspre-

chend für die Wahrscheinlichkeit des Entfernens (*delete*) eines Knotens (Abschnitt 3.3.3). So kann, entsprechend der Preferential Attachment Theorie, das durch das Community Detection gewonnene Wissen über die vorhandenen Gruppen wie folgt in die Berechnung einbezogen werden

$$p_i^c = \frac{g_i}{|N|} \cdot \frac{k_i}{\sum_j k_j} \tag{3.4}$$

Hierbei steht g_i für die Größe (der größten) Gruppe zu welcher der Knoten i gehört. Die Anzahl aller Knoten im Graph wird durch $|N|$ repräsentiert.

3.3.3. Entfernen existierender Knoten

Für das Entfernen lassen sich die gleichen Überlegungen, wie auch beim Erstellen neuer Knoten anstellen, nur mit umgekehrten Wahrscheinlichkeiten. Das heißt, je wahrscheinlicher ein Knoten eine neue Kante bekommt, desto unwahrscheinlicher ist es, dass der Knoten, auf Grund zu weniger Kanten, entfernt wird. Entsprechend der oben verwendeten Analogie gilt hier "the poor are getting poorer". Die Wahrscheinlichkeit p_i^d für das Entfernen eines Knoten (*delete*) ergibt sich dann zu

$$p_i^d = 1 - p_i^c \tag{3.5}$$

3.4. Link Prediction

Link Prediction befasst sich mit der Vorhersage von Kanten zwischen Knoten. Für die Vorhersage von Kantenbeziehungen zwischen Knoten müssen die Kanten nicht nur hinsichtlich ihrer Richtung (gerichtet oder ungerichtet) unterschieden werden, sondern auch bzgl. ihrer zeitlichen Relevanz. Während die Freundschaftsbeziehung im sozialen Netzwerk *Facebook* beispielsweise explizit durch die Nutzer erstellt wird und (meist) über einen längeren Zeitraum existiert, so können auch Beziehungen über die Anzahl von Interaktionen zwischen Nutzer abgeleitet werden (Wilson u. a., 2012).

3.4.1. Zeitliche Relevanz von Kanten

Je nach Kontext haben Kanten verschiedene Aussagen und müssen daher auch beim Erstellen von zeitlichen Ausschnitten aus einem Graph unterschiedlich behandelt werden. Es kann zwischen drei Typen von Kanten unterschieden werden.

Statische Kanten Einmal erstellt bleiben diese Kanten im Graph bestehen. Die Referenz einer Arbeit auf einer anderen Arbeit in einem Zitationsnetzwerk oder die Autorenschaft

für eine Arbeit sind statische Kanten. Da, einmal erstellt, sie sich auch für spätere Betrachtungen des Graphen nicht ändern.

Semi-statische Kanten Hiermit sind Kanten gemeint, die, einmal erzeugt, über mehrere Zeitschritte bestehen bleiben, bevor sie eventuell wieder entfernt werden. Beispiele hierfür sind Freundschaften in *Facebook* oder *Follower-Beziehungen* in *Twitter*. Der Nutzer kann eine solche Kante erstellen und sie später wieder entfernen.

Dynamische Kanten Dynamische Kanten repräsentieren Interaktionen zwischen Nutzern und haben nur eine gültige Aussage für den jeweiligen Zeitabschnitt, in dem sie auftreten. Beispiele hierfür sind Facebook-Wallposts, Tweets, sowie Telefonanrufe oder Emails.

Beim Aufteilen eines Interaktionsgraphen in Zeitabschnitte ist die Intensität der Beziehungen im jeweiligen Zeitabschnitt von Interesse und nicht die kumulierte Summe aller Aktionen seit der Existenz der Knoten (vgl. *interaction graphs* Wilson u. a. (2012) in 2.1.1).

Im Falle dynamischer Kanten handelt es sich zunächst immer um einen Multigraph. Es gibt zwei Möglichkeiten diesen Umstand für das Link Prediction zu übernehmen.

- Der generierte Zeitabschnitt bleibt ein Multigraph
- Eine gewichtete Kante repräsentiert die Summe aller Interaktionen in diesem Zeitraum

Abbildung 3.3 zeigt Kanten und ihre zeitliche Relevanz. Je nach Fall, sind auch die Anforderungen an die Link Prädiktoren unterschiedlich

Prädiktor für statische Kanten In diesem Fall muss der Prädiktor nur neue Kanten voraussagen. Diese können meist nur nach bestimmten Regeln erstellt werden. So können einmal veröffentlichte Arbeiten nach der Veröffentlichung keine weiteren Arbeiten mehr zitieren. Bestehende Kanten müssen/dürfen nicht entfernt werden.

Prädiktor für semi-statische Kanten Hier muss der Prädiktor nicht nur neue Kanten voraussagen, sondern auch das Verschwinden von Kanten. Ein Großteil der Kanten bleibt jedoch bestehen. So wird in Facebook eine neue Freundschaft geschlossen oder bei Twitter einem Nutzer nicht mehr gefolgt.

Prädiktor für dynamische Kanten Da dynamische Kanten ihre Gültigkeit verlieren, muss der Prädiktor nicht nur die Beziehungen aus dem vorherigen Zeitschritt, sondern auch nicht vorhandene Verbindungen bewerten und diese dann entsprechend erstellen.

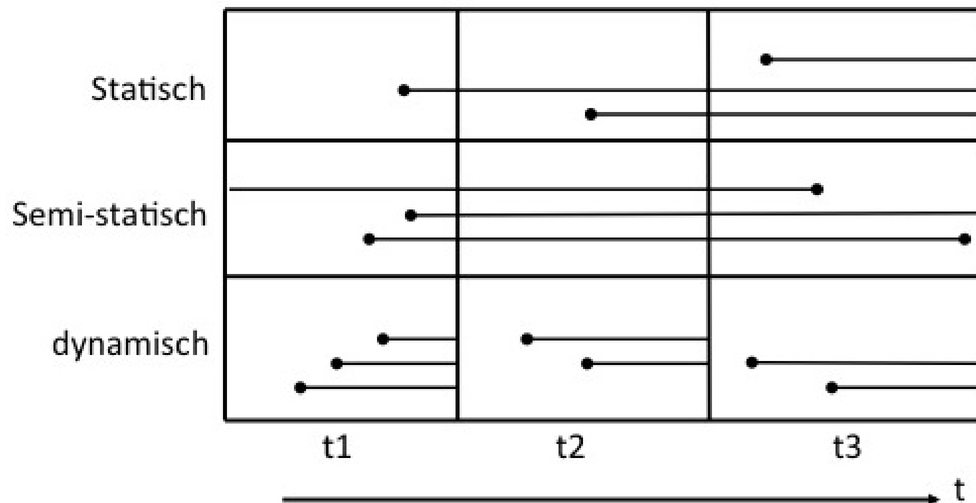

Abbildung 3.3.: Zeitliche relevant von Kanten. Ein Punkt repräsentiert den Zeitpunkt des Erstellens oder Löschens einer Kante.

3.4.2. Dezentrales Link Prediction

Viele Link Prediction Verfahren berechnen einen Kennzahl für jede fehlende Kante und werten diese zentral aus. Je nach verwendeter Kennzahl und Größe des Graphen ein sehr aufwändiger Schritt, da nicht nur sehr viele Kanten berechnet, sondern teilweise Kennzahl aufwendige erhoben werden müssen.

Die direkteste Möglichkeit, fehlende Kanten zu berechnen, ist das Schließen von Dreiecken. Mögliche Metriken hierfür sind *Common Neighbors*, *Resource Allocation* und der *Jaccard Koeffizient*. Die beiden ersten Kennzahlen greifen jedoch etwas kurz, da sie entweder nur die Zahl gemeinsamer Nachbarknoten oder das Produkt der Kanten beider Knoten nutzen. Der *Jaccard Koeffizient* hingegen setzt die Anzahl gemeinsamer Nachbarn in Beziehung zu allen Nachbarn beider Knoten und kombiniert dadurch sowohl die Anzahl gemeinsamen Nachbarn als auch die Anzahl aller Nachbarn zu einer einzigen Kennzahl.

Da die Wahrscheinlichkeiten nun nicht mehr global, sondern lokal in jedem Knoten gesammelt und sortiert werden, kann es sein, dass die Entscheidung eine neue Kante zu erstellen, nur von einem der beiden Knoten getroffen wird. Das ist der Fall, wenn einer der beiden Knoten weitere Möglichkeiten zur Auswahl hat. Aus diesem Grund muss nun zwischen gerichteten und ungerichteten Kanten unterschieden werden. Im Falle gerichteter Kanten muss nun überlegt werden, ob und wie eine Kante in der entgegengesetzten Richtung generiert werden kann.

Haben die Kanten ein Gewicht, so muss auch hier erneut eine Unterscheidung vorgenommen werden.

Gerichtete und ungerichtete Kanten

Giraph kennt nur gerichtete Kanten, die im jeweiligen Ursprungsknoten verwaltet werden. Möchte ein Knoten eine ungerichtete Kante erstellen, so muss der Knoten neben seiner ausgehenden Kante auch eine Kante generieren, deren Ursprung im anderen Knoten liegt und auf ihn gerichtet ist. Für das Entfernen von Kanten muss ebenfalls berücksichtigt werden, ob nur die eine Kante oder auch die Kante der Gegenrichtung entfernt werden muss.

Gewichtete und ungewichtete Kanten

Beim Erstellen gewichteter Kanten benötigt der Knoten eine Funktion für die Berechnung des Kantengewichts. Für Kantengewichte, welche aus dem Anwendungskontext entstehen, muss eine entsprechende Berechnungsvorschrift bereit gestellt werden. Wird die Anzahl der Kanten in einem Multigraph als Gewicht einer einzelnen Kante abgeleitet, lassen sich Zufalls-, Mittel- oder Median-Werte über den Grad des Knoten ableiten, welche dann als Gewicht der neuen Kanten dienen.

Im Fall einer gerichteten Kante, kann der Knoten diese mit der berechneten Gewichtung erstellen und muss nichts weiter berücksichtigen. Bei einem ungerichteten Graph müssen sich der Quellknoten und der Zielknoten auf ein gemeinsames Gewicht einigen. Dazu sendet der Quellknoten eine Nachricht an den Zielknoten, mit seiner initialen Gewichtung. Der Zielknoten errechnet aus diesem und seiner initialen Gewichtung einen neuen Wert, welchen er an den Quellknoten sendet. Dieser akzeptiert den Wert und erstellt eine ausgehende Kante. Initiieren beiden Knoten gleichzeitig das Erstellen einer neuen Kante, bietet es sich an ohne weitere Verhandlungen den Mittelwert aus beiden Gewichten zu nehmen, um zusätzliche Iterationen zu vermeiden.

3.5. Community Tracking

Zunächst werden die in den Grundlagen besprochenen Ansätze aufgegriffen und auf ihre Tauglichkeit hin untersucht. Das gewählte Verfahren wird dann erweitert, um Daten umgehen zu können, welche ein Prediction Verfahren durchlaufen haben. Nach dem Funktion zur Berechnung der Gruppenähnlichkeit gefunden wurde, wird definiert, welche Partitionen mit einander verglichen werden müssen, um die Qualität des gesamten Prozesses überprüfen.

3.5.1. Vergleich der Ansätze

Der *Evolutionary Clustering* Ansatz von Chakrabarti u. a. (2006) lässt sich nicht getrennt vom zugrunde liegenden Gruppenfindungsalgorithmus realisieren und wird daher nicht weiter betrachtet. Der Ansatz von Chen u. a. (2010) arbeitet nur auf Cliquen und ist daher ebenfalls nicht geeignet.

Der Community Tracking Ansatz von Greene u. a. (2010) hingegen ist unabhängig von der Wahl des Community Detection Verfahren und bietet mit dem Jaccard-Koeffizient eine Möglichkeit, beliebige Gruppenstrukturen zu vergleichen. Der Jaccard Koeffizient ist in (Greene u. a., 2010) definiert als

$$sim(C_a, C_b) = \frac{|C_a \cap C_b|}{|C_a \cup C_b|} \tag{3.6}$$

(vgl. Formel 2.11) und berechnet das Verhältnis der übereinstimmenden Knoten zur Anzahl der aller Knoten der beiden Gruppen. Übertrifft $sim(C_a, C_b)$ einem bestimmten Grenzwert, gelten die beiden Gruppen als einander zugeordnet. Je nach Verhältnis der Gruppengrößen von C_a und C_b, kann dann auf den Fortbestand, das Schrumpfen oder Anwachsen der Gruppe geschlossen werden. Zwei Gruppen C_a und C_b aus dem Zeitabschnitt t gehen zusammen zu einer Gruppe C_c im Zeitabschnitt $t + 1$, wenn sowohl $sim(C_a, C_c)$ als auch $sim(C_b, C_c)$ den Grenzwert überschreiten. Das Aufteilen einer Gruppe C_x in zwei Gruppen C_y und C_z ist entsprechend dann der Fall, wenn $sim(C_x, C_y)$ und $sim(C_x, C_z)$ beide über dem Grenzwert liegen, C_x ist aus t und die beiden Gruppen C_y und C_z sind aus $t + 1$. Vergleiche hierzu die Abbildungen 2.1 und 2.2 auf Seite 16.

Wird für eine Gruppe aus t keine passende Gruppe gefunden so stirbt sie. Wird für eine Gruppe aus $t + 1$ keine passende Gruppe in t gefunden, so wurde diese Gruppe in $t + 1$ geboren.

3.5.2. Erweiterung für Prediction-Daten

Durch das Node Prediction Verfahren werden Knoten erzeugt, deren Identifikatoren nicht Teil des Zeitabschnitts $t + 1$ sind. Die Funktion sim muss deshalb erweitert werden, um die, durch das Prediction erzeugten Identifikatoren, trotzdem als Knoten zu erkennen, welche der Gruppe korrekt zugeordnet sind. Dazu werden die Knoten einer Gruppe aufgeteilt in Knoten aus dem alten Zeitabschnitt t und Knoten aus dem neuen Zeitabschnitt $t + 1$. Da die vorhergesagten Knoten-Identifikatoren aus dem neuen Zeitabschnitt, nicht vergleichbar sind mit anderen

Knoten, werden sie auf ihre Anzahl reduziert. Die Ähnlichkeit für neue Knoten wird definiert als

$$sim_{new}(c_1, c_2) = \frac{min(c_1, c_2)}{max(c_1, c_2)} \tag{3.7}$$

Abschließend können die Werte aus sim und sim_{new} gewichtet werden. Die Berechnung der Ähnlichkeit erfolgt dann durch

$$sim(c_1, c_2) = sim_{old}(c_1, c_2) * w_{old} + sim_{new}(c_1, c_2) * w_{new} \tag{3.8}$$

wobei sim_{old} das sim aus Formel 3.6 ist. Jung und Segev (2013) wählen das gleiche Vorgehen.

3.5.3. Definition der zu vergleichenden Partitionen

Um zu überprüfen, ob der mit Node und Link Prediction generierte Zwischenschritt, die Wiedererkennungsrate zwischen zwei Zeitschritten t_1 und t_2 verbessert, reicht es festzustellen, welche Gruppen erkannt wurden. Dabei ist es nicht wichtig, ob sie gewachsen oder geschrumpft sind. Für diesen Zweck sind auch die in Greene u. a. (2010) beschriebene *Front* und *Timeline* nicht notwendig. 1Für den vergleich der vorhandenen Partitionen werden vier Fälle unterschieden, die in Abbildung 3.4 dargestellt sind. Die Zeitintervalle $t1$ und $t2$ entsprechend

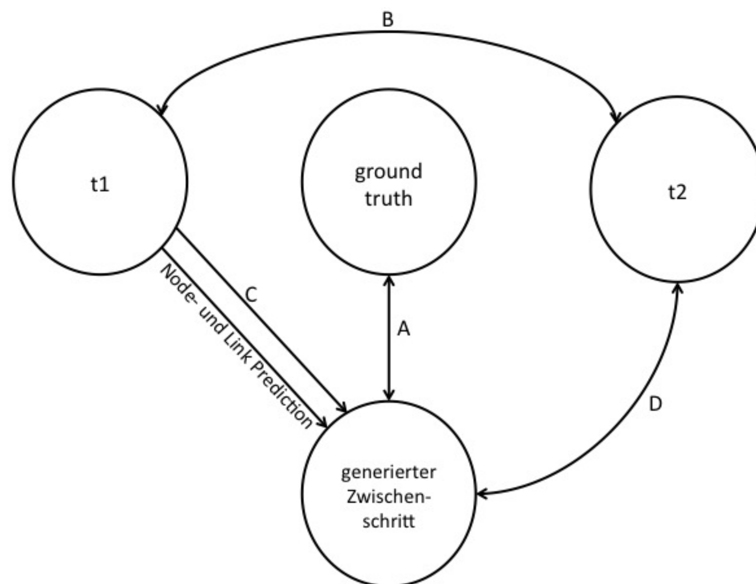

Abbildung 3.4.: Vergleich der Gruppen aus den verschiedenen Zeitschritten.

den Intervallen aus Abbildung 3.1. Für die Testdaten liegen Intervalle vor, die zwischen den

Intervallen $t1$ und $t2$ liegen. Diese dienen als Ground-Truth-Datensätze, um das Ergebnis der Prediction Verfahren zu überprüfen. Für alle 3 Intervalle aus dem Testdatensatz werden die Knoten in Gruppen eingeteilt. Anschließend wird auf Intervall $t1$ die Node- und Link Prediction Verfahren ausgeführt, um den Zwischenschritt zu generieren, welcher abschießend ebenfalls partitioniert wird.

Fall A Fall A überprüft die Qualität des vorhergesagten Zwischenschritts. Hierfür wird gezählt, wieviele Gruppen zwischen dem Ground-Truth-Datensatz und dem Zwischenschritt wiedererkannt werden.

Fall B Für Fall B wird überprüft, wieviele Gruppen zwischen dem ursprünglichen Zeitabschnitt $t1$ und dem nächsten vorhandenen Zeitabschnitt $t2$ wiedererkannt werden können..

Fall C Dieser Fall prüft, wieviele Gruppen zwischen $t1$ und dem Zwischenschritt erkannt werden können.

Fall D Der letzte Fall betrachtet, wieviele Gruppen aus dem Zwischenschritt heraus zum Zeitintervall $t2$ erkannt werden. Ziel ist es eine deutlich höhere Wiedererkennungsrate zu haben, als für den Fall B.

3.6. Zusammenfassung

Mit COPRA wurde ein Community Detection Algorithmus gewählt, der sich gut aus der Sicht eines Knotens implementieren lässt. Die mit COPRA gefundenen Gruppen können dann, mit Hilfe der Node- und Link Prediction weiterentwickelt werden. Gerade für Zitationsnetzwerke ist die Preferential Attachment Theorie ein geeignetes Modell für die Vorhersage neuer Knoten (Jung und Segev, 2013). Die Definition der Gruppenähnlichkeit aus dem Community Tracking lässt sich ebenfalls für die Anwendung auf vorhergesagten Daten anpassen.

4. Realisierung

In diesem Kapitel wird die Umsetzung der in Kapitel 3 besprochenen Algorithmen beschrieben. Nach einem kurzen Überblick über die wichtigsten Klassen in Giraph, wird der Ablauf der einzelnen Algorithmen beleuchtet. Der Fokus liegt dabei auf dem Fluss der Informationen und den Orten für die, durch den Algorithmus zu treffenden Entscheidungen.

4.1. Giraph

Die durch das *Computation* Interface in Giraph bereitgestellte *compute* Methode ist der zentrale Ort für die zu realisierende Logik der Knoten (Abb. 4.2). Für jede Iteration übergibt das Framework der *compute* Methode den zu berechnenden Knoten mit seinen Kanten und seinem Knotenwert (*VertexValue*), sowie allen Nachrichten, die in der vorangegangenen Iteration an den Knoten gesendet wurden.

Darüber hinaus bietet die abstrakte Klasse *MasterCompute* (Abb. 4.1) die Möglichkeit, Dinge zentral zwischen den Iterationen auszuführen. Dadurch lassen sich komplexere Algorithmen steuern, in dem bspw. globale Abbruchkriterien überprüft werden oder die *Compute*-Klasse ausgetauscht wird. Die *MasterCompute*-Klasse implementiert das *MasterAggregatorUsage* Interface (Abb. 4.1), welches ihm erlaubt Aggregatoren im System zu erzeugen, an die Knoten Werte schicken können, um sie dort kumulieren zu lassen.

4.2. Community Detection

Für das Finden von Gruppen wurde der in 2.2.4 und 3.2 besprochene COPRA Algorithmus (Gregory, 2010) implementiert. Der Algorithmus besteht aus zwei Phasen: der Label-Propagation-Phase und der Aufräum-Phase. In der Label-Propagation-Phase, teilt der Algorithmus die Knoten in Gruppen ein. In der zweiten Phase, der Aufräum-Phase, werden die gefundenen Gruppen von Subgruppen bereinigt und auf Zusammenhang überprüft.

Für die drei Teilaufgaben in COPRA - Label Propagation, Entfernung von Subgruppen und Finden disjunkter Gruppen mit identischem Label - wurde je eine eigene *Compute* Klasse implementiert. Der *CopraMaster* überprüft den Fortschritt des Algorithmus und kann die

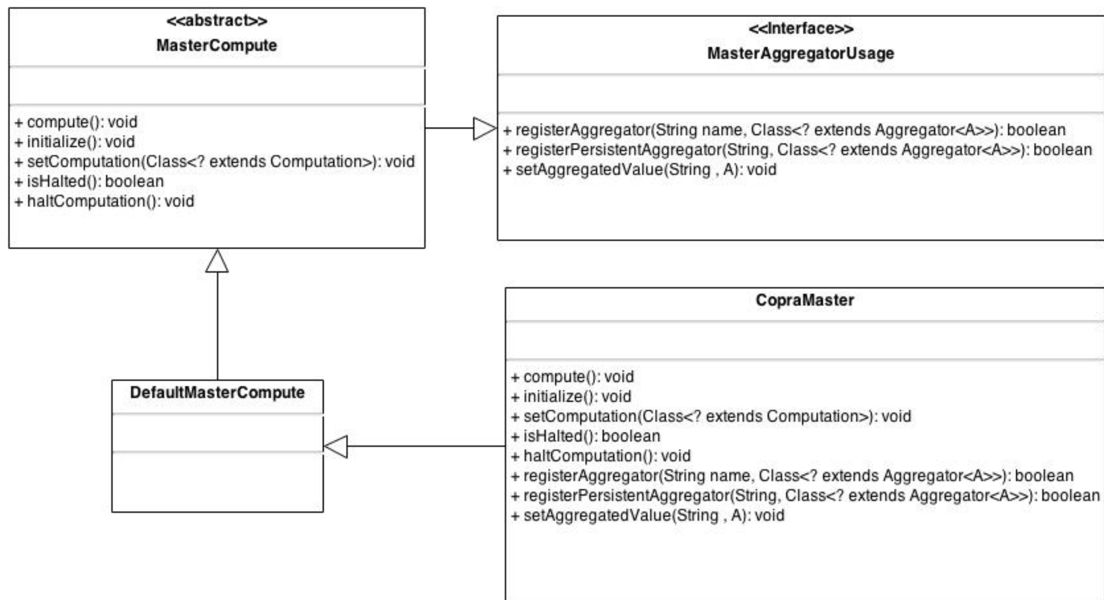

Abbildung 4.1.: Die Interfaces und Klassen für die MasterComputation am Beispiel des CopraMaster.

Compute Klasse für die Knoten entsprechend austauschen. Die drei Computation Klassen für den COPRA Algorithmus sind in Abbildung 4.2 dargestellt.

Die Label-Propagation-Phase wurde in zwei verschiedenen Varianten umgesetzt, da die direkte Umsetzung des Algorithmus (Variante 1) lange Iterationszeiten hatte, die sogar anstiegen[1]. Für Variante 2 wurde der Vergleich von Partitionen als Stoppkriterium aufgegeben und sich näher am Ausgangsalgorithmus LPA (Raghavan u. a., 2007) (siehe ebenfalls in 2.2.4 und 3.2) orientiert. Nachfolgend werden beide Umsetzungen besprochen.

4.2.1. Phase 1: Label Propagation

Variante 1

Für die direkte Umsetzung des COPRA Algorithmus muss nach jeder Iteration die entstandene Partition mit der Partition aus der vorherigen Iteration verglichen werden. Da die Zuordnung eines Knotens zu einer Gruppe aber nur im jeweiligen Knoten gespeichert ist, wurde mit

[1]Der Anstieg der Iterationszeit lag vermutlich an einer ungünstigen Serialisierung der Nachrichten, die den Garbage Collector sehr beansprucht hat

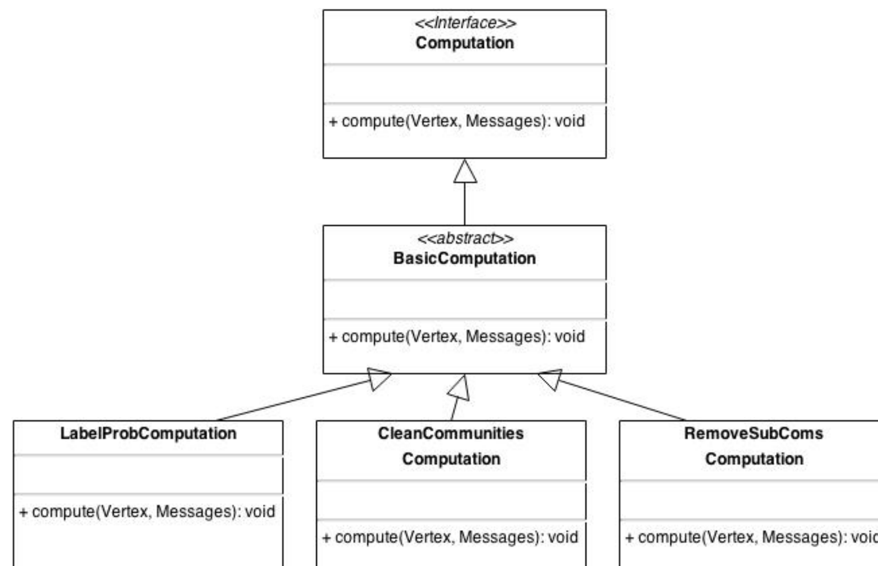

Abbildung 4.2.: Die Computation-Klassen in Giraph für die Implementierung der Knoten-Logik.

HBase[2] eine Tabelle bereitgestellt, auf die jeder Knoten schreibend zugreifen kann. So kann jeder Knoten zentral speichern, zu welchen Gruppen er gerade zugeordnet ist. Nach jeder Iteration konnte der CopraMaster dann die Partitionen auslesen und vergleichen. Da hierbei viele Schreib- und nur wenige Lese-Operationen anfallen, wurde zur Optimierung des HBase-Zugriffs das HBase-HUT[3] Plugin verwendet. Das Plugin zögert das Update der Tabelle solange hinaus, bis lesend auf die Tabelle zugegriffen wird.

Jeder Knoten sendet seine akzeptierten Labels, inklusive der entsprechenden Zugehörigkeitskoeffizient, an alle seine Nachbarn. In jeder Iteration erhält daher jeder Knoten die Labels und Koeffizienten aller Nachbarn und kann daraufhin das eigenen Label aktualisieren. Am Ende jeder Iteration sind die Knoten einer oder mehreren Gruppen zugeordnet und speichern diese Zuordnung zentral in HBase. Zwischen den Iterationen berechnet der CopraMaster das Stoppkriterium. Ist dieses erfüllt, überführt der CopraMaster den Algorithmus von der Label Propagation Phase in die Aufräum-Phase, durch Setzen der entsprechenden Computation Klasse. Abbildung 4.3 zeigt den Ablauf für drei Iterationen. In Iteration $t-1$ können Knoten Informationen sowohl in Aggregatoren (blauer Pfeil) als auch in HBase (grauer Pfeil) serialisieren. Zwischen $t-1$ und t kann der Master die Informationen sowohl aus den Aggregatoren

[2] http://hbase.apache.org/ - Die Hadoop Datenbank
[3] https://github.com/sematext/HBaseHUT

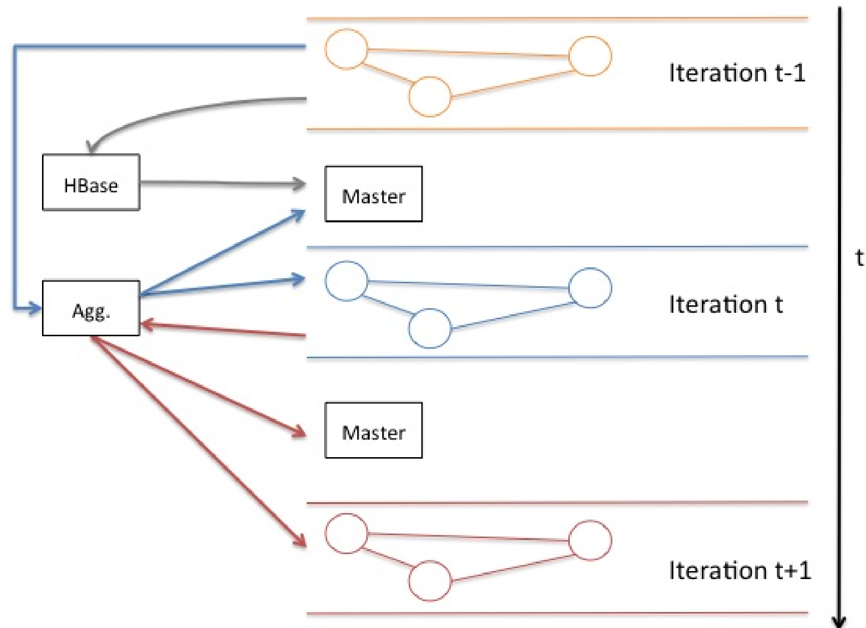

Abbildung 4.3.: Zeitlicher Ablauf und Informationsfluss in Giraph für die Label Propagation Variante 1.

als auch aus der HBase-Tabelle auslesen und auswerten. Je nachdem, ob das Stop-Kriterium erreicht wurde kann der Master die Computation-Klasse für die nächste Iteration bestimmen. In der Iteration t können nun auch Knoten die Informationen aus dem Aggregator auslesen (blau) und neue Informationen an den Aggregator senden (rosa).

Auf Grund der Zugehörigkeitskoeffizienten muss ein Knoten nach jeder Iteration eine Nachricht schicken da sich, auch wenn sich die akzeptierten Labels nicht geändert haben, sich die Koeffizienten geändert haben können. In Abbildung 4.4 a) empfängt der Knoten von seinen Nachbarn 1 bis 5 je eine Nachricht. Die Nachbar gehören entweder ganz zur Gruppe A oder Gruppe B und haben daher je einen Zugehörighkeitskoeffizienten von 1. Der Knoten akzeptiert Label, die einen Wert größer $1/3$ ($v = 3$) haben. In diesem Fall ergeben sich die Label zu (A:$3/5$) und (B:$2/5$) und werden akzeptiert (Labelmenge $\{A, B\}$). In der nächsten Iteration erhält der Knoten erneut Nachrichten seiner Nachbarn, doch nur das Label von Nachbar 3 hat sich im Vergleich zur vorherigen Iteration, geändert. Wieder akzeptiert der Knoten beide Gruppen A und B. Seine eigene Gruppenzuordnung für $A : B$ hat sich von 2:3 zu 3:2 geändert, die Menge der akzeptierten Label sind jedoch identisch ($\{A, B\}$). In beiden Fällen a) und b) sendet der Knoten je eine Nachricht über alle ausgehenden Kanten (im Bild

nicht dargestellt). Dadurch sind alle Knoten durchgehend aktiviert und nach jeder Iteration

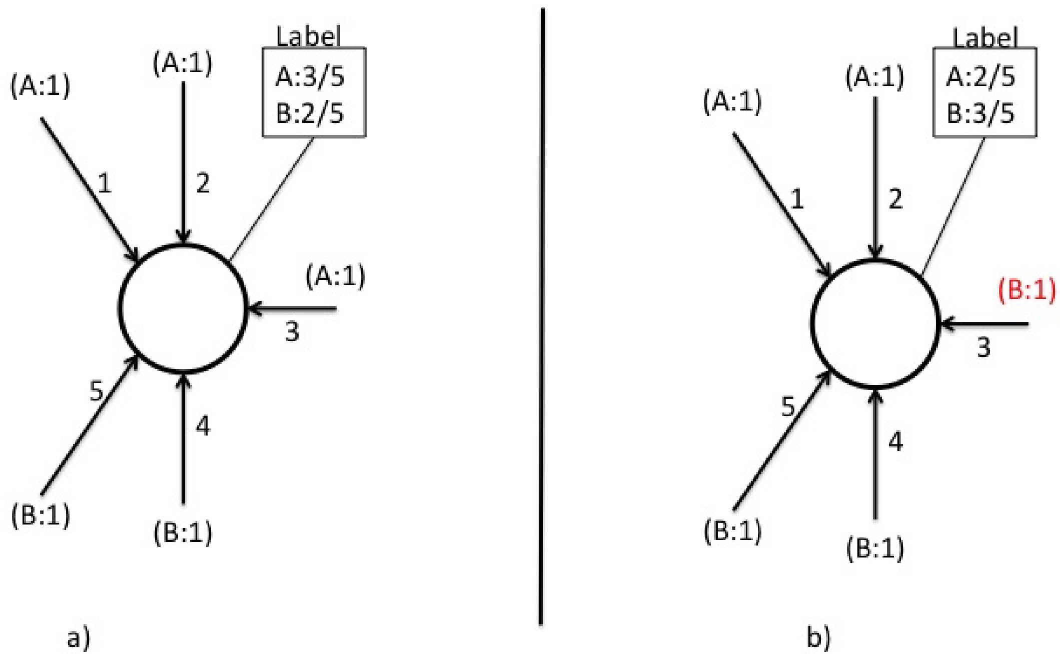

Abbildung 4.4.: Berechnung des Labels in Variante 1.

müssen $|E|$ Nachrichten zugestellt werden.

```
public void compute() {
  finishedNodes = getAggregatedValue(FINISHED_NODES).get();
  activeNodes = getTotalNumVertices() - finishedNodes;

  pendingMsgs = getAggregatedValue(PENDING_MSGS).get();
  currentPhase = getAggregatedValue(PHASE_AGG).getPhase();
  switch (currentPhase) {
    case LabelPropagation:
      labelPropagation();
    }
    break;
  // ...
  case ClusterCleaningFinishing:
    if (activeNodes == 0 && pendingMsgs == 0 && getSuperstep() > 0) {
      haltComputation();
    }
```

```
      break;
    }
}

private void labelPropagation() {
  // Step 3 in the COPRA Algorithm
  Map<Long, Integer> clusterSizes = PIWrapper.getClusterSizes(
      getSuperstep()-1);
  Set<Long> currentClusterIDs = clusterSizes.keySet();

  if (currentClusterIDs.equals(oldClusterIDs)) {
    min = calculateMinClusterSize(min, clusterSizes);
  } else {
    min = clusterSizes;
  }
  // Step 4 in the COPRA Algorithm
  if (!min.equals(oldmin) || getSuperstep() == 0) {
    oldClusterIDs = currentClusterIDs;
    oldmin = min;
    setComputation(LableProbComputation.class);
  } else {
    registerAggregator(REMOVED_COMS_AGG, SubComAggregator.class);
    setNewPhase(COPRA_Phase.LabelPropagation, COPRA_Phase.
        RemoveSubCommunities);
    setComputation(RemoveSubComsComputation.class);
  }
}
```

Listing 4.1: Vereinfachte Darstellung des CopraMaster der Variante 1

In Listing 4.1 ist zu sehen, wie der CopraMaster den aktuelle Zustand aus einem Aggregator ausliest (Zeile 6) und entsprechend des Zustands, in dem er sich befindet, bestimmte Operationen ausführt (Zeilen 7-18). Giraph erlaubt es jedem Knoten, sich zu deaktivieren. Erreicht den Knoten eine Nachricht, wird der Knoten wieder aktiviert. Sind alle Knoten inaktiv und keine Nachrichten mehr vorhanden, wird der Algorithmus gestoppt. Der Fall *ClusterCleaning-Finishing* (Zeile 13 ff) zeigt beispielhaft, wie diese Funktion nachgebildet werden kann, um in komplexeren Algorithmen einen Phasenwechsel oder ähnliches zu realisieren. Statt sich zu deaktivieren, sendet jeder Knoten ein $+1$ an einen Aggregator, um sich inaktiv zu melden und ein $+1$ an einen weiteren Aggregator, um zu signalisieren, dass er Nachrichten versendet hat (vgl. Abbildung 4.3 Zeile 11, 12). Die Klasse *PIWrapper* (Listing 4.1, Zeile 24) bietet Zugriff auf die HBase-Tabellen mit Hilfe des HBaseHUT-Plugins.

Variante 2

Für die zweite Variante der Label Propagation Phase wurde auf das komplexe Stoppkriterium für die Phase - der Vergleich von zwei Partitionen - verzichtet. Die Phase ist nun beendet, wenn alle Knoten inaktiv sind und keine Nachrichten mehr versendet werden (vgl. Listing 4.1 Zeile 15). Die zweite Variante ist daher mehr an eine LPA Implementierung angelehnt, als eine Implementierung des COPRA Algorithmus. Ein Knoten sendet nur dann eine Aktualisierung, wenn sich die Menge der akzeptierten Labels geändert hat. Dies reduziert die Anzahl aktiver Knoten und somit auch die Anzahl versendeter Nachrichten. Dafür muss nun ein Knoten die Label speichern, die er zuletzt vom jeweiligen Nachbarn erhalten hat. Abbildung 4.5 zeigt die gleiche Situation wie in Abbildung 4.3. Im ersten Schritt (Abb. 4.5 a)) senden auch hier alle fünf Nachbarn eine Aktualisierung ihrer Label an den Knoten. Dieser akzeptiert die Gruppen A und B im Verhältnis 3:2. Diesmal speichert der Knoten jedoch für jeden Nachbar das Label, welches ihm geschickt wurde. Im nächsten Schritt ändert sich nur das Label von Nachbar 3 von A auf B und nur dieser sendet eine Aktualisierung. Der Knoten kann nun, aus der gespeicherten Liste, die Label der restlichen Nachbarn auslesen und sein eigenes Label neu berechnen. Da sich die Menge seiner akzeptierten Gruppen $\{A, B\}$ nicht ändert, sendet der Knoten keine Aktualisierung an seine Nachbarn. Würde der Knoten diese Information nicht speichern, währe er nicht in der Lage, die neuen Gruppen berechnen, wenn nur einer der Nachbarn eine Aktualisierung sendet. Gleichzeitig können die gespeicherten Labels ebenfalls genutzt werden, um das in Raghavan u. a. (2007) beschriebene Oszillieren zu verhindern.

4.2.2. Phase 2: Aufräumen

Wurde eine Partition für den Graph gefunden, muss diese nun bereinigt werden. Die Aufräum-Phase lässt sich weiter unterteilen in das Entfernen von Subgruppen und das Finden disjunkter Gruppen mit gleichem Label. Für das Entfernen von Subgruppen wurden ebenfalls zwei Varianten realisiert. Für das Finden getrennter Gruppen wurde eine Broadcast-Funktion implementiert. Die beiden Varianten für das Subgruppen-Problem sowie der Broadcast werden im folgenden beschrieben.

Finden von Subgruppen: Variante 1

Das Finden von Subgruppen ist in den Schritten 5 und 6 des Pseudocodes (Gregory, 2010, Abbildung. 3) beschrieben und in Listing 4.2 leicht angepasst wiedergegeben.

```
1 5. For each vertex x:
2   ids = x.ids
```

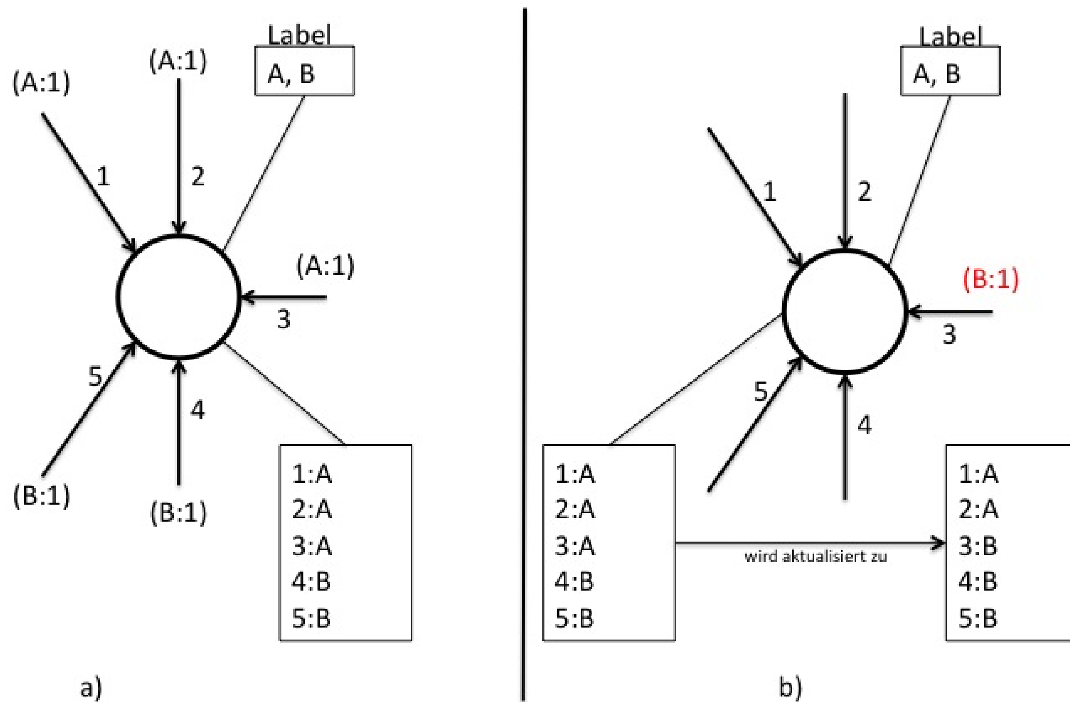

Abbildung 4.5.: Berechnung des Labels in Variante 2.

```
3    For each c in ids:
4      If, for some g, (c,g) is in coms, (c,i) in sub:
5        coms = coms - {(c,g)} union {(c,g union {x})}.
6        sub  = sub - {(c,i)} union {(c,i cut ids)}.
7      Else:
8        coms = coms union {(c,{x})}.
9        sub  = sub union {(c,ids)}.
10 6. For each (c,i) in sub:
11   If i new {}: coms = coms - (c,g).
```

Listing 4.2: Subgruppen entfernen nach Gregory (2010)

In Gregory (2010) sind *coms* und *sub* Mengen von Tupeln. In *coms* ist ein Tupel (c, g) ein Paar aus einer Gruppen-Identifikatoren c und einer Menge von Knoten g. In *sub* besteht das Tupel (c, i) aus einem Gruppen-Identifikatoren c und einer Menge andere Gruppen-Identifikatoren i. Zu Beginn sind die Mengen *coms* und *sub* leer. Für jeden Knoten wird nun folgende Berechnung durchgeführt: Für *coms* wird der Knoten in die Menge der Knoten eingetragen, für die er ein

Label besitzt (Zeile 5). Die Menge *coms* enthält später die Partition des Graphen. Für die Menge *sub* werden, falls die Menge der Label im Tupel leer ist, die Labels des Knoten eingetragen (Zeile 9). Enthält die Menge im Tupel bereits Label, wird eine Vereinigung vorgenommen (Zeile 6). In Schritt 6 wird dann für jedes Tupel aus *sub*, dessen Label-Menge nicht leer ist, das Tupel mit dem Gruppen-Identifikator aus *coms* entfernt. Somit enthält *coms* keine Subgruppen mehr.

Für die Umsetzung in Giraph wurde für die Menge *coms* bereits auf eine HBase Tabelle ausgewichen. Die Menge *coms* muss daher nicht noch einmal berechnet werden. Statt nun mit *sub* zentral eine Tupel-Menge zu speichern, wurde für jedes Tupel ein *Aggregator* in Giraph registriert, der die entsprechende Berechnung vornimmt. Da Aggregatoren nur durch *Master*-Klassen registriert werden können, muss zunächst der *CopraMaster* die vorhandenen Gruppen-Identifikatoren aus HBase auslesen und für jeden Gruppen-Identifikator einen *Aggregator* erstellen. Dieser bekommt als Referenz den Namen des jeweiligen Gruppen-Identifikators. Jeder Knoten kann dann über den Gruppen-Identifikator seiner Gruppen, den jeweiligen *Aggregator* erreichen. Schritt 6, das Prüfen der Tupel, wurde anschließend zentral im *CopraMaster* implementiert. Dort werden die jeweiligen Aggregatoren ausgelesen und über einen weiteren Aggregator den Knoten die gelöschten Gruppen mitgeteilt.

Finden von Subgruppen: Variante 2

Da für die zweite Variante in Phase 1 auf das Vergleichen der Partitionen verzichtet wird, steht für diesen Teilaufgabe keine zentrale Partition bereit, aus der die vorhandenen Gruppen-Identifikatoren erfragt werden könnten. Das Berechnen der Subgruppenbeziehungen muss daher nun vollständig dezentral ablaufen. Statt das der CopraMaster für jede Gruppe einen Aggregator registriert, der für die jeweilige Gruppe die Subgruppen-Verhältnisse prüft, wird die Berechnung nun in den Knoten vorgenommen. Da die Gruppen-Labels aus den Identifikatoren der Knoten entstehen, kann jeder Gruppe immer ein Knoten zugeordnet werden. Statt an den *Aggregator*, senden die Knoten nun an den Knoten, dessen Identifikator dem Gruppen-Identifikatoren entspricht. Nur für das Kommunizieren der gelöschten Subgruppen wird weiterhin ein *Aggregator* verwendet. Der Name dieses Aggregators ist allen Knoten im Vorfeld bekannt. Während in der ersten Variante die Partition *coms* über die HBase Tabelle berechnet wurde, wird nun keine Partition berechnet und muss im Anschluss aus der Ausgabe des Algorithmus, außerhalb von Giraph, berechnet werden.

Listing 4.3 zeigt den vereinfachten Ablauf während des Entfernen von Subgruppen in einem Knoten. Gut zu sehen ist, wie der Zustand durch das Speichern einer Zustandsvariable STEPCOUNT im *VertexValue* realisiert werden kann. Ein Multilabel enthält die Menge der

akzeptierten Labels, deren Zugehörigkeitskoeffizienten sowie die Labels der Nachbarn aus der vorherigen Iteration.

```
public void compute(
  Vertex<LongWritable, CopraVertexValue, DoubleWritable> vertex,
  Iterable<MessageEnvelope> messages) {
  CopraVertexValue value = vertex.getValue();
  long currentstep = getStepCount(value);

  if (currentstep == 0) {
      Set<Long> tmpIDs = value.getMultiLabel().getCommunities();
      MessageEnvelope msgEnv = MessageFactory.createSubCommunityMessage(
          vertex.getId(), tmpIDs);
      sendMessage(vertex.getId(), msgEnv);
      aggregate(CopraMaster.PENDING_MSGS, new IntWritable(1));
    }
    setCurrentStepDone(vertex.getValue());
  } else if (currentstep == 1) {
    for (MessageEnvelope message : messages) {
      CopraSubCommunityMsg msg = message.getLetter();
      // do COPRA step 5 calculation
    }
    setCurrentStepDone(vertex.getValue());
  } else if (currentstep == 2) {
    // get deleted communities from Aggregator
  }
}

private void setCurrentStepDone(CopraVertexValue value) {
  Long newStepC = getStepCount(value) + 1;
  value.setApplicationData(STEPCOUNT, newStepC.toString());
}

private long getStepCount(CopraVertexValue value) {
  if (value.getApplicationData(STEPCOUNT) == null) {
    value.setApplicationData(STEPCOUNT, "0");
    return 0l;
  } else {
    String stepCStr = value.getApplicationData(STEPCOUNT);
    return new Long(stepCStr);
  }
}
```

Listing 4.3: Zustände innerhalb eines Knotens

Finden getrennter Gruppen

In der zweiten Phase (Aufräumen) müssen disjunkte Gruppen identifiziert werden, die das gleiche Gruppenlabel nutzen. Dazu wurde ein Broadcast realisiert. Jeder Knoten sendet an alle Nachbar seinen Identifikator und die Menge seiner akzeptierten Label. Für alle eintreffenden Nachrichten leitet der Knoten nur diejenigen Label weiter, welche der Knoten selbst akzeptiert hat. Dadurch wird sichergestellt, dass der Broadcast eines Knotens nur innerhalb dessen Gruppe abläuft. Da eine zentrale Sicht fehlt, initiiert jeder Knoten einen Broadcast. Dieser ist nach maximal dem längsten der kürzesten Pfade einer Gruppen abgeschlossen. Erhält ein Knoten keine Nachrichten mehr, weiß er, dass für ihn der Broadcast abgeschlossen ist. Da der Knoten für jede empfangene Nachricht, den Sender und dessen Gruppen gespeichert hat, kann er nun für jede Gruppe den Knoten mit dem kleinsten Identifikator nachschlagen und die Gruppe lokal nach diesem Identifikator benennen.

Ist der Knoten der kleinste Knoten in einer Gruppe, überprüft er, ob er nicht bereits der kleinste Knoten einer weiteren Gruppe ist. Für diesen Fall benennt der Knoten die Gruppe nach dem größten Knoten in der Gruppe um. Da alle anderen Knoten dieser Gruppe den kleinsten Knoten-Identifikator als Namen gewählt haben, muss der Knoten nun allen Knoten eine Nachricht schicken, damit diese die Gruppe entsprechend noch einmal umbenennen. Die Wahrscheinlichkeit, dass ein Knoten kleinster Knoten in zwei Gruppen ist, ist jedoch sehr gering. Der Fall, dass der Knoten kleinster Knoten für drei oder mehr Gruppen ist, wird vom Algorithmus ignoriert.

4.2.3. Ein- und Ausgabe

Giraph bietet *InputFormats*, welche die Logik für das Einlesen externen der Daten implementieren. *InputFormats* können entweder Listen von Kanten oder Listen von Knoten einlesen. Eine Kombination der beiden Formate ist möglich. Kanteneingabeformate sind in der Lage für jede gelesene Kante zusätzlich eine Kante in der Gegenrichtung zu erzeugen.

Für die Ausgabe der Community Detection Ergebnisse wurde eine knotenbasierte Ausgabe gewählt, bei der die Daten eines Knoten in Form eines JSONArray[4] ausgegeben werden. Die Struktur des Arrays besteht aus dem Knoten-Identifikator, einer Menge Gruppen-Labels, einer Menge von Gruppen-Labels und ihrer Größe, einer Menge Kanten mit ihren Gewichten, sowie einer Menge an Anwendungsdaten.

$$\left[\text{id}, \left[\text{group-id}^*\right], \left[[\text{community, size}]^*\right], \left[[\text{edgeTargetID, edgeValue}]^*\right], \left[[\text{dataName, data}]^*\right]\right]$$

[4] http://www.json.org

4.3. Node Prediction für Zitationsnetzwerke

In Kapitel 3 wurden verschiedene Arten von Kanten für das Link Prediction besprochen. Unter anderem auch der Spezialfall für Zitationsnetzwerke. Dieser zeichnet sich dadurch aus, dass Kanten nur von neuen Knoten zu bereits existierenden Knoten erstellt werden können. In diesem und dem nächsten Abschnitt wird nun besprochen, wie Node- und Link Prediction für Zitationsnetzwerke mit Giraph realisiert werden kann.

Da Zwischenschritte vorhergesagt werden sollen, ist die Anzahl neuer Knoten bekannt und wird über die Giraph-Konfiguration an den Algorithmus übergeben. Die Anzahl der Knoten muss für diesen Fall nicht vorhergesagt werden.

Die Knoten-Identifikatoren des Zitationsnetzwerkes sind kodiert als $yymmiii$. Die ersten beiden Stellen yy repräsentieren das Jahr, die Stellen mm den Monat. Die drei letzten Stellen iii entsprechen einer eindeutigen Identifizierung innerhalb des Monats. Auf Grund der Repräsentation der Identifikatoren als Zahl fallen führende Nullen gegebenenfalls weg.

In ihrer Arbeit beschreiben Jung und Segev (2013) einen Ansatz für Node Prediction in Zitationsnetzwerken. Der Eingangsgrad eines Knotens dient als Auswahlkriterium für die Preferential Attachment Theorie. Dieser muss in Verhältnis zum Eingangsgrad der restlichen Knoten gesetzt werden. Ein Aggregator berechnet die Summe aller Eingangsgrad und stellt sie den Knoten zur Verfügung.

In der ersten Iteration berechnet der Algorithmus, neben der Summe der Eingangsgrade, auch den größten Knoten-Identifikator ID_{max} im Graph. In der zweiten Iteration generiert dann der Knoten mit dem größten Identifikator, die vorher bestimmte Anzahl neuer Knoten N_{neu}. Aus der vorgegebenen Größe des neuen Zeitabschnitts und ID_{max}, kann so errechnet werden, welchen Ansprüchen neue Identifikatoren gerecht werden müssen. Da die zeitliche Verteilung der Knoten innerhalb eines Zeitraums für die Vorhersage nicht relevant ist, iteriert der Knoten beim Erstellen über $i \in 0..N_{neu}$ und füllt die Monate der Reihe nach mit neuen Knoten auf. Für ein $ID_{max} = 9511425$, einem Vorhersagezeitraum von sechs Monaten und $N_{neu} = 1342$, werden folgende neue Identifikatoren generiert: 9512000 bis 9512999 und 9601000 bis 9601323. Die neuen Knoten sind zu Beginn der nächsten Iteration verfügbar.

In der dritten Iteration entscheidet nun jeder Knoten für jeden neuen Knoten entsprechend seiner Wahrscheinlichkeit, ob es eine Kante zwischen ihm und dem neuen Knoten geben wird. Grundlage zur Berechnung der Wahrscheinlichkeit ist die Preferential Attachment Theorie. Um die Knoten mit dem Graph zu verbinden, werden nun e_{neu} Kanten pro Knoten benötigt. Ein geeigneter Wert für e_{neu} muss später experimentell ermittelt werden Jung und Segev (2013).

In einer ersten Version wurde ein Aggregator registriert, der die Menge neuer Knoten-Identifikatoren verwaltete. Jeder neue Knoten, der e_{neu} Kanten erreicht hat, konnte seine Identifikator beim Aggregator löschen lassen.

So konnten die Knoten am Aggregator erfragen, für welche neuen Knoten noch Kanten benötigt werden, um dann lokal zu entscheiden, ob sie in dieser Iteration eine Kante erhalten können. Ziel des Einsatzes eines solchen Aggregators war es, die Last der Berechnung in den Knoten zu reduzieren, da die Knoten nur noch für die nicht fertiggestellten neuen Knoten Berechnungen durchführen müssen. Dieser Ansatz war jedoch nicht sehr performant.

Daher wird nun nur noch die Anzahl der nicht fertig erstellten Knoten kumuliert, so dass die Knoten nur eine einzige Zahl beim Aggregator erfragen müssen. Stattdessen entscheiden die Knoten jetzt in jeder Iteration immer für alle N_{neu} neue Knoten-Identifikatoren, ob es in dieser Iteration eine Kante für einen Knoten geben wird. Trifft der Knoten eine positive Entscheidung für ein $i \in N_{neu}$, wandelt er i in den entsprechende Knoten-Identifikator um, und sendet dem Knoten den eignen Identifikator und Eingangsgrad. Der empfangende Knoten speichert die eintreffenden Informationen und kann nun für den Fall, dass ihm in einer Iteration mehr als e_{neu} Kanten zur Verfügung stehen, die Zielknoten mit dem größten Eingangsgrad auswählen. Die Knoten iterieren so lange über alle neuen Knoten-Identifikatoren, bis alle neuen Knoten erstellt sind. Es hat sich gezeigt, dass dieses vorgehen deutlich schneller ist, als die erste Realisierung.

4.4. Link Prediction für Zitationsnetzwerke

Für das Label Propagation wurde der eigentlich gerichtete Graph in einen ungerichteten Graph überführt. Ein Knoten kennt daher nicht nur die Referenzen zu anderen Knoten, sondern weiß auch, welche Knoten ihn zitieren. Jeder Nachbar dessen Identifikator größer ist, als der im Node Prediction ermittelte, größte Identifikator ID_{max}, ist ein neuer Knoten. Hat ein Knoten mindestens einen neuen Knoten als Nachbarn, ist der Knoten ein *triangle node*. Ein *triangle node* ist der Vermittler zwischen neuen Knoten und potenziellen Zielen für Referenzen des neuen Knoten. Abbildung 4.6 beschreibt beispielhaft den Ablauf. Der 3-stellige Knoten-Identifikator enthält in der ersten Stelle die Nummer des Zeitintervalls, die beiden letzten Stellen repräsentieren die eindeutige ID innerhalb des Zeitintervalls, ID_{max} ist 189. In der ersten Iteration prüft jeder Knoten, ob er ein *triangle node* ist. Da Knoten 132 mit 205 einen Nachbarknoten aus dem neuen Intervall hat, ist Knoten 132 ein *triangle node*. In diesem Fall sendet er eine Anfrage an seine Nachbarn (Abbildung 4.6 a). In der zweiten Iteration generieren die angesprochenen Knoten ihre Nachbarmenge ($\{A_i\}$ oder $\{B_i\}$) und schicken

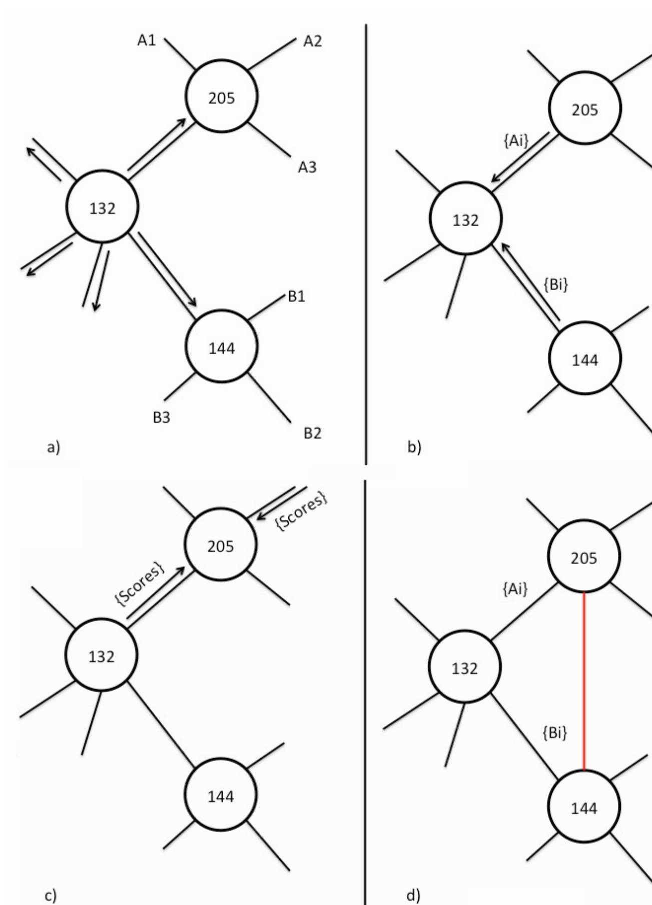

Abbildung 4.6.: Vergleich der Gruppen aus den verschiedenen Zeitschritten.

diese an den *triangle node*, als Antwort auf die Anfrage (Abbildung 4.6 b). In der dritten Iteration kann der *triangle node* für alle Kombinationen aus neuen (hier Knoten 205) und alten Knoten (hier beispielhaft Knoten 144) den Jaccard-Koeffizienten berechnen. Anschließend sendet der*triangle node* jedem neuen Knoten die Menge der für ihn berechneten neuen Kanten, inklusive deren berechneten Kennzahlen (Abbildung 4.6 c). Der neue Knoten speichert alle Links der ihn umgebenden *triangle nodes* und erstellt anschließend Kanten, für die der Wert einen vorgegebenen Grenzwert überschreitet (Abbildung 4.6 d).

4.4.1. Ein- und Ausgabe

Als Eingabeformat dient das in 4.2.3 verwendete JSON[5] Ausgabeformat. Hier muss nicht mehr zwischen gerichteten und ungerichteten Graphen unterschieden werden. Als Ausgabe wird ebenfalls wieder das JSON Ausgabeformat verwendet.

4.5. Community Tracking

Das Community Tracking, also der Vergleich der Gruppen aus den einzelnen Zeitschritten, findet außerhalb von Giraph, in einem Java Programm statt. Als Eingabe dient eine Datei in welcher jede Zeile eine Gruppe repräsentiert. Am Anfang einer Zeile stehen die Knoten aus dem Zeitschritt $t-1$, gefolgt von einem ## als Trennzeichen und einer Zahl, welche die Anzahl der neuen Knoten aus Zeitschritt t widerspiegelt.

$$\begin{array}{cccccc} node1_{t-1} & node2_{t-1} & ... & nodeN_{t-1} & \#\# & X \\ node1_{t-1} & node2_{t-1} & ... & nodeN_{t-1} & \#\# & Y \end{array}$$

Eine solche Datei lässt sich leicht aus der jeweiligen Ausgabe im JSONA-Format erstellen.

Für jeder Gruppe aus dem ersten Zeitschritt iteriert das Programm über alle Gruppen des zweiten Zeitschritts und berechnet die Ähnlichkeit der jeweiligen Gruppen mit

$$sim(c_1, c_2) = sim_{old}(c_1, c_2) * w_{old} + sim_{new}(c_1, c_2) * w_{new} \tag{4.1}$$

für $w_{old} \in \{(0,0); (0,2); (0,4); (0,6); (0,8)\}$ und $w_{new} = 1 - w_{old}$

Für jeder der fünf Kombinationen w_{old} und w_{new} wird gespeichert, welche Gruppen c_1 und c_2 als ähnlich eingestuft werden, so dass eine optimale Kombination der Gewichte gewählt werden kann.

[5] http://www.json.org

5. Experimente

In diesem Kapitel werden die durchgeführten Tests beschrieben und deren Ergebnisse präsentiert. Zunächst werden die verwendeten Datensätze dargestellt und untersucht (5.1). Nachdem das Verhalten des implementierten Community Detection Algorithmus analysiert wurde, werden die Experimente für die Node- und Link Prediction Verfahren besprochen. Abschließend wird auf die Ergebnisse des daraus resultierenden Community Trackings eingegangen.

Aufgrund der Laufzeit-Probleme der ersten Variante des Community Detection Algorithmus COPRA, wurde zwischenzeitlich entschieden zunächst einen anderen Algorithmus für das Finden von Gruppen einzusetzen. Der bereits von Jung und Segev (2013) eingesetzte Louvain-Algorithmus wurde anstatt des COPRA-Algorithmus verwendet. Hierfür wurde eine vorhandene C++ Implementierung[1] genutzt. Die Prediction Verfahren kommen daher auf Daten zur Anwendung, die mit dem Louvain-Algorithmus partitioniert wurden.

Im späteren Verlauf dieser Arbeit konnte eine zweite Variante des COPRA Algorithmus implementiert werden, deren Laufzeit deutlich besser ist. Die Eigenschaften dieser zweite Variante werden in Kapitel 5.2 untersucht. Hierfür werden Szenarien getestet, die denen in Gregory (2010) ähnlich sind. Aus zeitlichen Gründen war es leider nicht mehr möglich, die Prediction Verfahren auf Ergebnissen der verbesserten COPRA Variante auszuführen.

5.1. Testdaten

Für die Experimente werden sowohl Datensätze realer Graphen, als auch synthetisch generierte Graphen verwendet. Die synthetisch generierten Graphen bieten den Vorteil, dass sie in ihren Eigenschaften verändert werden können, sodass mit ihnen bestimmte Eigenschaften gezeigt bzw. überprüft werden können. Dadurch lassen sich Algorithmen auf synthetischen Graphen sehr gut testen. Zudem lässt sich mit synthetischen Graph-Generatoren, für jeden Knoten und jeden Zeitschritt genau definieren, zu welcher Gruppe ein Knoten gehört. Dieses Wissen wird *ground-truth* genannt.

[1] https://sites.google.com/site/findcommunities/

5.1.1. Zitationsnetzwerke

Als Datensatz für Realdaten werden die Zitationsnetzwerke verwendet, die auch in Jung und Segev (2013) untersucht werden. Sie stammen aus dem Stanford Network Analysis Project der Standford University[2] (Leskovec u. a., 2005). Es handelt sich dabei um zwei Netzwerke aus dem arXiv[3]. Nachfolgend wird jedoch nur der HepPh-Datensatz verwendet. Dieser enthält veröffentlichte Arbeiten aus der *high energy physics phenomenology*. Der Datensatz enthält 34546 Knoten und 421578 Kanten und umfasst den Zeitraum Januar 1993 bis April 2003.

Die Knoten repräsentieren veröffentlichte Arbeiten. Kanten stellen die Referenzen einer Arbeit auf eine andere Arbeit dar. Die Kanten sind gerichtet, ungewichtet und statisch, entsprechen ihrer zeitlichen Relevanz (3.4.1), da eine Referenz in einer veröffentlichten Arbeit nicht mehr geändert werden kann und daher dauerhaft Teil des Graphen. Für Link Prediction gilt daher auch, dass neue Kanten nur von neuen Knoten zu bereits existierenden Knoten zeigen können.

Der Datensatz wurde in Zeitschritte mit einer Dauer von sechs Monaten eingeteilt. Das erste Zeitintervall umfasst die Daten bis einschließlich Juni 1994 (1994_6). Spätere Zeitabschnitte enthalten immer auch die Informationen der vorangegangenen Zeitabschnitte. So ist der Datensatz 1994_6 vollständig auch im folgenden Datensatz 1994_12 enthalten. Der letzte Datensatz ist jeweils bis Dezember 2001 (2001_12). Jeder Datensatz enthält nur diejenigen Arbeiten, die bis einschließlich des letzten gültigen Datums für den Zeitabschnitt veröffentlicht wurden. Referenzen von Arbeiten aus einem späteren Zeitabschnitt werden entfernt. Das heißt die Kante eines Knotens aus dem Jahr 2001 zu einen Knoten im Zeitabschnitt 1994_12 wird nicht mit in den Datensatz aufgenommen. Dadurch wird der Grad der Knoten aus den früheren Jahren mit fortschreitender Zeit größer. Das Wachstum des Netzwerks, bezogen auf die Knoten, ist nicht exponentiell (Abbildung 5.1). Die Experimente für das Node- und Link Prediction werden in Abschnitt 5.3 beschrieben.

5.1.2. Synthetische Daten

Zum Testen ihres Community Tracking Ansatzes haben Greene u. a. (2010) den Benchmark-Generator aus Lancichinetti und Fortunato (2009) und Lancichinetti u. a. (2008), um die Generierung von Zeitschritten erweitert. Der Generator kann generische Graphen mit bestimmten Eigenschaften für eine beliebige Anzahl von Zeitschritten generieren. Er erlaubt Einstellungen für

[2] http://snap.stanford.edu/data/index.html#citnets
[3] http://arxiv.org/

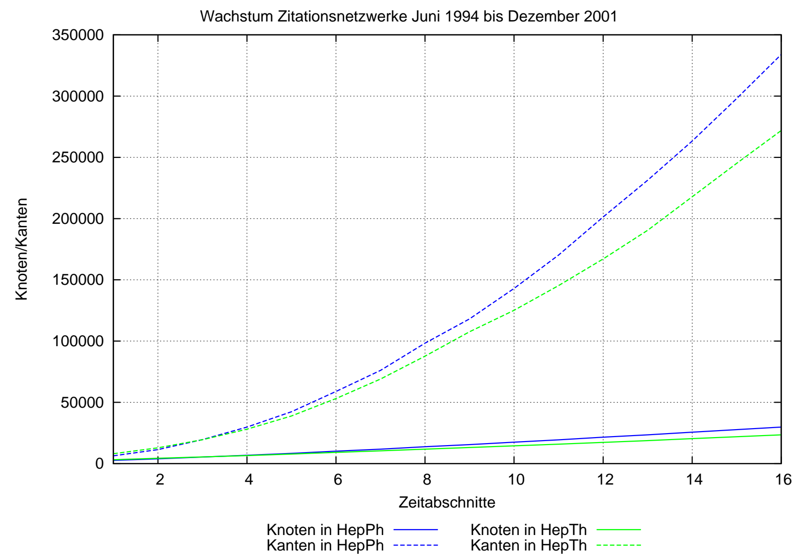

Abbildung 5.1.: Wachstum der Zitationsnetzwerke.

- den durchschnittlichen Grad k und maximalen Grad k_{max} eines Knotens
- die Anzahl an Gruppen o_n, zu denen ein Knoten gehören kann
- die Anzahl der zu erzeugenden Zeitschritte
- Die Veränderung zwischen den Zeitschritten.
- Die Anzahl überlappender Knoten o_m definiert, wieviele Knoten mehr als nur einer Gruppe angehören sollen.

Der Generator enthält fünf verschiedene, eigenständige Programme, wovon jedes jeweils ein gruppendynamisches Ereignis erzeugen kann. Für jeden Zeitschritt liefern die Programme eine Liste der ungerichteten Kanten im Graph, eine Liste der generierten Ereignisse, sowie einen *ground truth* Datensatz, der jeden Knoten für jeden Zeitschritt seinen Gruppen zuordnet. Die Untersuchung des Community Detection Algorithmus in Abschnitt 5.2 findet auf Daten statt, die mit dem *switch* Programm generiert wurden. Dieses simuliert den Wechsel der Knoten von einer Gruppe in eine andere. Für die Tests wurde jedoch immer nur ein Zeitschritt generiert.

Der Generator erzeugt ungerichtete Kanten. Werden diese in Giraph eingelesen, werden entsprechende Gegenkanten erzeugt, weshalb sich der Grad eines Knoten im Vergleich zum generierten Grad, verdoppelt. Dadurch entstehen aber keine neuen Verbindungen zwischen den Knoten, auf denen der Algorithmus Label austauschen könnte.

5.2. Validierung der Community Detection

Für die Untersuchung seines COPRA Algorithmus nutzt Gregory (2010) den ursprünglichen Graph-Generator vonLancichinetti und Fortunato (2009) und bewertet die Ergbenisse mit der lfkNMI Kennzahl. Für die, in dieser Arbeit durchgeführten Experimente, wird der Graph-Generator von von Greene u. a. (2010) genutzt, um Testdaten zu generieren, welche den Experimenten von Gregory (2010) ähnlich sind. Mit dem von McDaid u. a. (2011) Programm[4] werden Werte für lfkNMI und NMI<MAX> erhoben.

Nicht überlappende Gruppen

Es wurden jeweils Netzwerke mit einer Größe von $N = 1000$ und $N = 5000$ generiert. Der Grad der Knoten wurde beim Generieren mit $k_{min} = 20$ und $k_{max} = 45$ angegeben. Ein Knoten kann nur zu einer Gruppe gehören, überlappenden Knoten ($o_m = 0$) sind daher ausgeschlossen. Abbildung 5.2 a zeigt für ein μ zwischen 0 und $0,8$ die Qualität der Gruppenerkennung für $v = 1$. Der Parameter v bestimmt wieviele Gruppen der Algorithmus einem Knoten zuordnen kann. Für Abbildung 5.2 b wurde $v = 4$ gesetzt. Diese Tests entsprechen (Gregory, 2010, Abbildung 11).

Für $\mu > 0,6$ sinkt die Qualität der Gruppenerkennung. Für einige Fälle von NMI = 0 konnte der Algorithmus nur eine Gruppe bestimmen. Das führte dazu, dass Giraph in Phase 2 mit einer *OutOfMemoryException* abgebrochen hat, da beim Finden getrennter Gruppen jeder Knoten versucht hat, den kompletten Graph jeweils lokal für sich zu speichern. Beide NMI Werte fangen unterhalb von 1 an. Die lfkNMI-Werte sind im Vergleich zu den Ergebnissen von Gregory (2010) sogar deutlich schlechter. Für ein μ von 0.4 bis 0.5 nähern sich aber beide Werte dem Optimum an, bevor sie wie auch bei Gregory (2010) auf 0 abfallen.

Netzwerke mit 1000 Knoten

Auf Grund des gewählten Graph-Generators kann kein Einfluss auf die Gruppengröße genommen werden. Daher können die Tests aus (Gregory, 2010, Abbildung 12 und 13) nicht

[4] https://github.com/aaronmcdaid/Overlapping-NMI

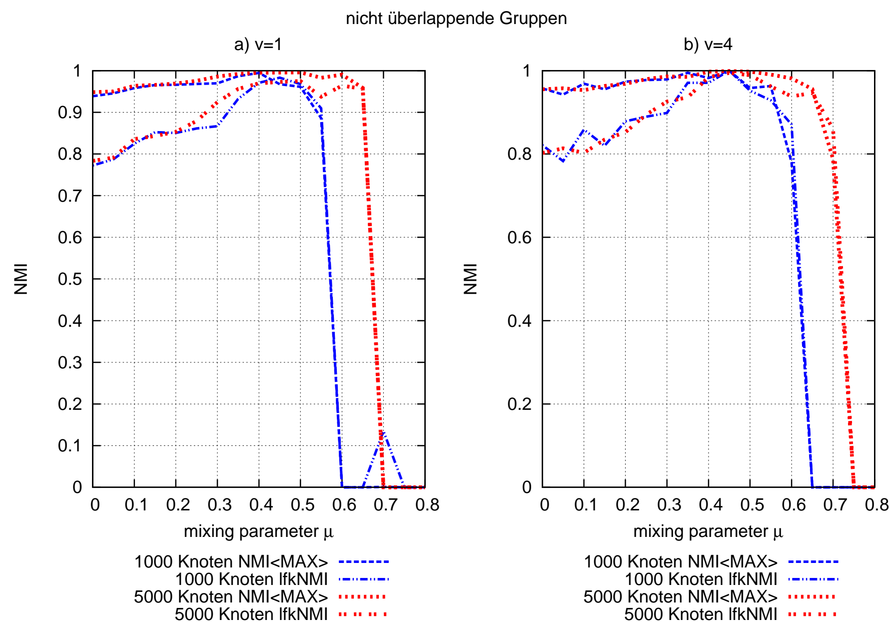

Abbildung 5.2.: Nicht überlappende Gruppen (vgl. Gregory (2010, Abbildung 11))

unterschieden werden. Je größer der Anteil überlappender Knoten o_m, desto größer die durchschnittliche Gruppengröße. Die durchschnittliche Gruppengröße liegt zwischen 13 Knoten für $o_m = 0$ und 24 Knoten für $o_m = 800$ überlappender Knoten (0.8 auf der x-Achse, Abbildung 5.3). Abbildung 5.3 zeigt ein ähnliches Verhalten, wie auch der Algorithmus von Gregory (2010). Jedoch sinkt die Qualität der Gruppenerkennung deutlich schneller und ist nicht so stabil. Dies kann mit der sich ändernden Gruppengröße zusammenhängen. Der direkte Vergleich der lfkNMI-Messwerte zeigt ein Defizit gegenüber den Werten von Gregory (2010). Abbildung 5.4 zeigt ebenfalls ein ähnliche Verhalten, wie der COPRA Algorithmus. Doch auch hier ist die Qualität der Gruppenerkennung nicht so stabil und sinkt schneller.

Netzwerke mit 5000 Knoten

Für die in den Abbildungen 5.5 und 5.6 dargestellten Experimente wurde jeweils nur die Anzahl der Knoten auf 5000 erhöht. Alle anderen Parameter sind unverändert. Da die NMI Werte hier nicht auf 0 fallen, wurde ein kleinere Ausschnitt auf der y-Achse gewählt, um den Verlauf

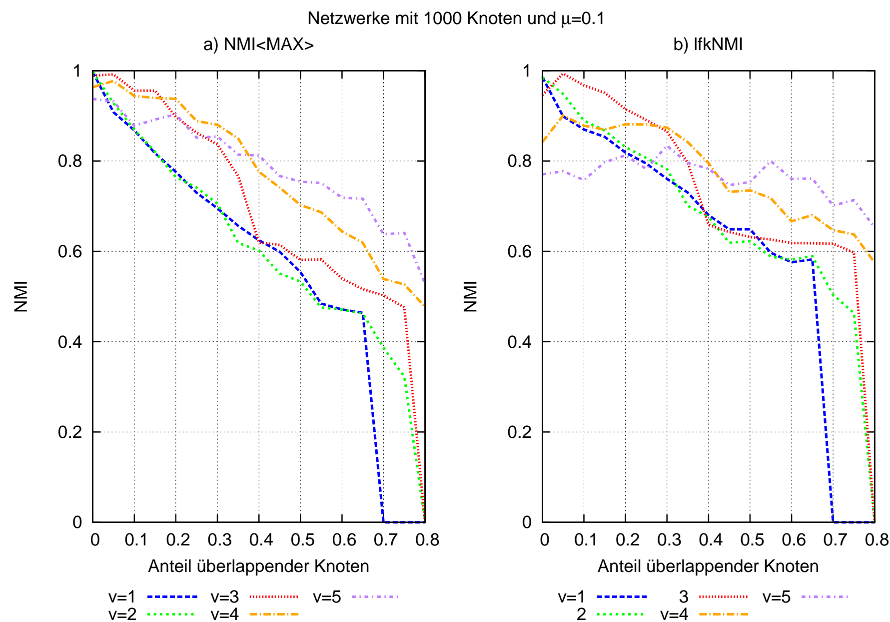

Abbildung 5.3.: Netzwerke mit 1000 Knoten und $\mu = 0,1$ (vgl. Gregory (2010, Abbildung 12 oben rechts und 13 oben rechts))

der Kurven besser darstellen zu können. Interessant ist, dass die lfkNMI-Werte für $v = 4$ und $v = 5$ etwas anders verlaufen als die restlichen Kurven.

Laufzeit für steigende Netzwerkgrößen

Alle Berechnungen wurden auf einem Hadoop Cluster bestehend aus 3 virtuellen Maschinen ausgeführt. Jede Maschine hat Zugriff auf einen 2,3 GHz Quadcore Prozessor und verfügt über 16GB Arbeitsspeicher. Eine Maschine dient als Master-Knoten, zuständig für die Koordination, die beiden anderen Maschinen sind Worker, welche die Berechnungen der Knoten durchführen. Für jeden Worker wurden vier *ComputeThreads* bereitgestellt.

Die vorhergehenden Experimente benötigten im Schnitt zwischen 40 und 80 Sekunden und wurden auf Datensätzen mit maximal 5000 Knoten ausgeführt. Mit den folgenden Szenarien wird nicht nur die Knotenzahl deutlich erhöht, sondern auch die Anzahl der Kanten gesteigert.

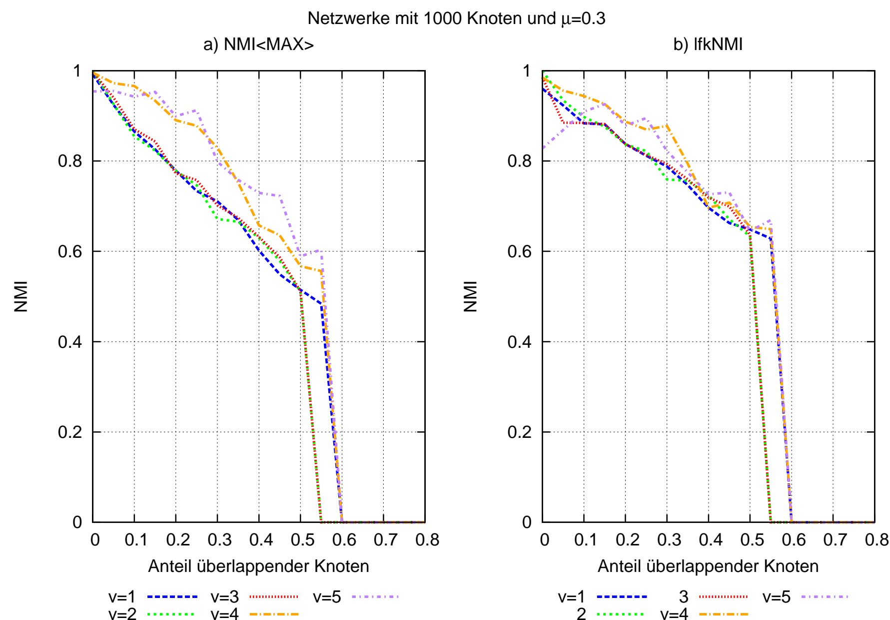

Abbildung 5.4.: Netzwerke mit 1000 Knoten und $\mu = 0,3$ (vgl. Gregory (2010, Abbildung 12 unten rechts und 13 unten rechts))

Für die beiden Szenarien in Abbildung 5.7 werden größer werdende Netzwerke untersucht. Die Anzahl überlappender Knoten ist $o_n = N/2$ und $\mu = 0,1$.

Für Abbildung 5.7 a ist $N = 175000$ und der Grad k ($k = k_{max}$) wird gesteigert von 10 bis 32. Die rote Kurve repräsentiert die Werte, wie sie aus (Gregory, 2010, Abbildung 16 unten rechts) entnommen wurden. Die Laufzeit des für Giraph implementierten Algorithmus wurde mit zwei verschiedenen Konfigurationen getestet. Für beide Konfigurationen wurde jeweils ein Master Worker verwendet.

Die blaue Kurve in Abbildung 5.7 a zeigt die Berechnungen auf zwei Workern (2 w), auf denen jeweils 4 Threads (4 t) für die Berechnung bereitgestellt wurden. Für das Experiment, welches durch die grüne Kurve in Abbildung 5.7 a repräsentiert wird, wurde nur noch ein Worker (1 w) mit 1 Thread (1 t) verwendet. Es ist deutlich zu sehen, dass der Algorithmus durch das Bereitstellen zusätzlicher Rechenleistung skaliert. Für eine Kantenmenge größer 3,5 Millionen, wurde Giraph mit umbestimmtem Fehler beendet, so dass hier keine weiteren

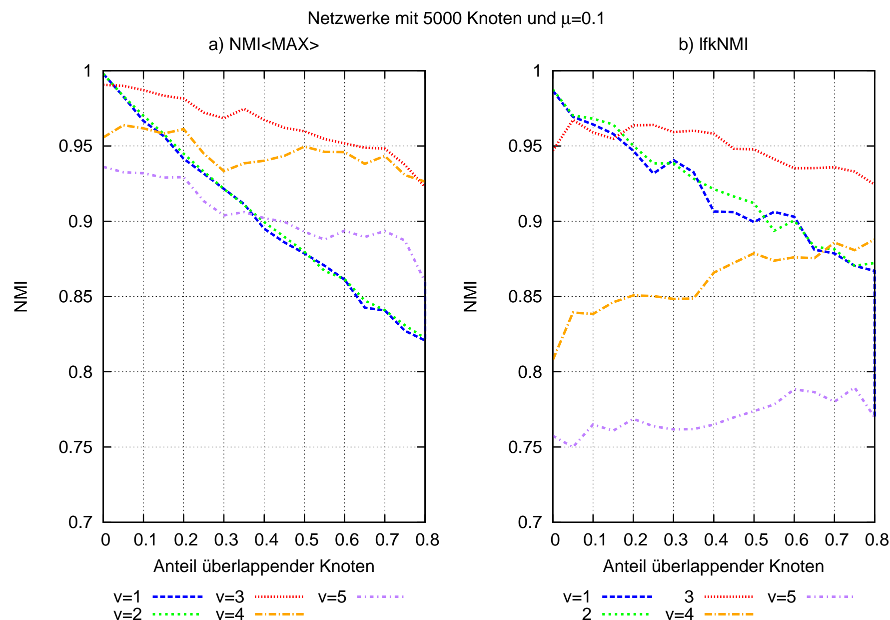

Abbildung 5.5.: Netzwerke mit 5000 Knoten und und $\mu = 0,1$ (vgl. Gregory (2010, Abbildung 14 und 15 oben rechts))

Experimente durchgeführt werden konnten. Der Vergleich der Laufzeit, zu den von Gregory (2010) durchgeführten Experimente zeigt, dass die Laufzeit der Experimente in Giraph stark ansteigt.

Für das zweite Szenario in Abbildung 5.7 b ist der Grad der Knoten $k = 20$ für alle Experimente und die Knoten variieren zwischen $N = 1000$ und $N = 65000$. Wie auch im ersten Szenario wird der Giraph Algorithmus in zwei verschiedenen Konfigurationen getestet und die Ergebnisse den Werten aus (Gregory, 2010, Abbildung 16 oben rechts) gegenübergestellt. Für die in Abbildung 5.7 b abgebildeten Experimente liegt der COPRA Algorithmus zwischen den Laufzeiten der beiden Giraph-Konfigurationen. Für die Experimente auf nur einem Worker (1 w, 1 t) ist zu sagen, dass für Netzwerke > 3 Mio. Kanten Giraph mit unbestimmten Fehlern beendet wurde. Für die Experimente auf zwei Workern konnten keine Netzwerke > 5,8 Millionen Kanten getestet werden, da auch hier Giraph beendet wurde.

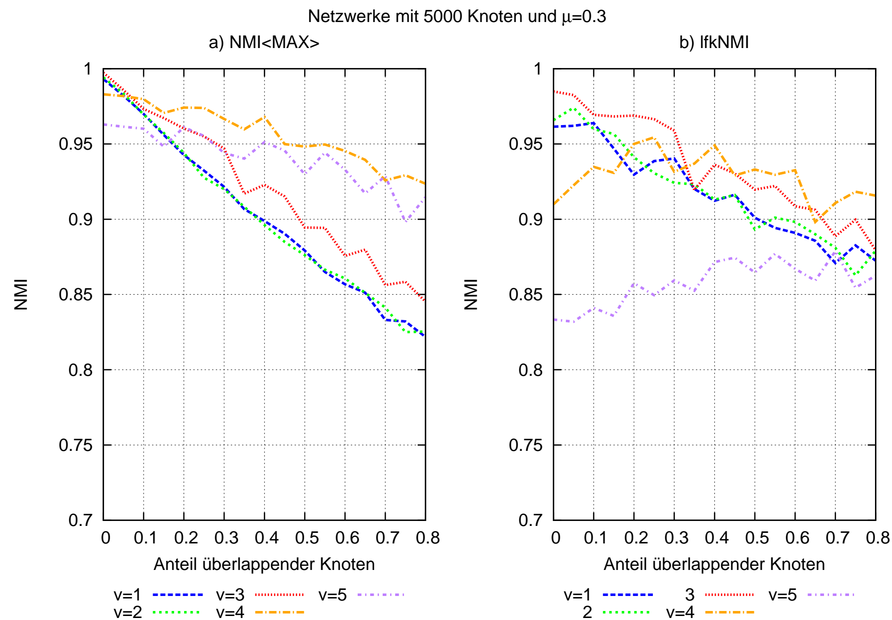

Abbildung 5.6.: Netzwerke mit 5000 Knoten und $\mu = 0,3$ (vgl. Gregory (2010, Abbildung 14 unten rechts und 15 unten rechts))

5.3. Prediction Experimente

Für das Testen der Node- und Link Prediction Verfahren wird der HepPh-Datensatz verwendet. Die Zeitabschnitte erweitern den vorherigen Zeitabschnitt um jeweils sechs Monate. Die erstellten Zeitabschnitte t_i mit $i \in \{1...16\}$ zwischen Januar 1995 und Dezember 2001 werden mit dem Louvain-Algorithmus partitioniert. Der Louvain-Algorithmus kann keine überlappenden Gruppen finden. Anschließend wird aus den Zeitabschnitten t_1 bis t_{15} je ein neuer Zwischenschritt vorhergesagt. Für jeden Zwischenschritt t'_i ist der Vergleichszeitraum mit den *ground truth* Daten t_{i+1} (siehe Abbildung 5.8) Für die ersten drei Vorhersageschritte t'_i, t''_i und t'''_i hat der Vergleich der in A und in B erkannten Gruppen ergeben, dass für jede Vorhersage aus dem Zwischenschritt zum Zielintervall (Vergleich A) weniger Gruppen erkannt wurden, als vom Ursprungsintervall zum Zielintervall (Vergleich B).

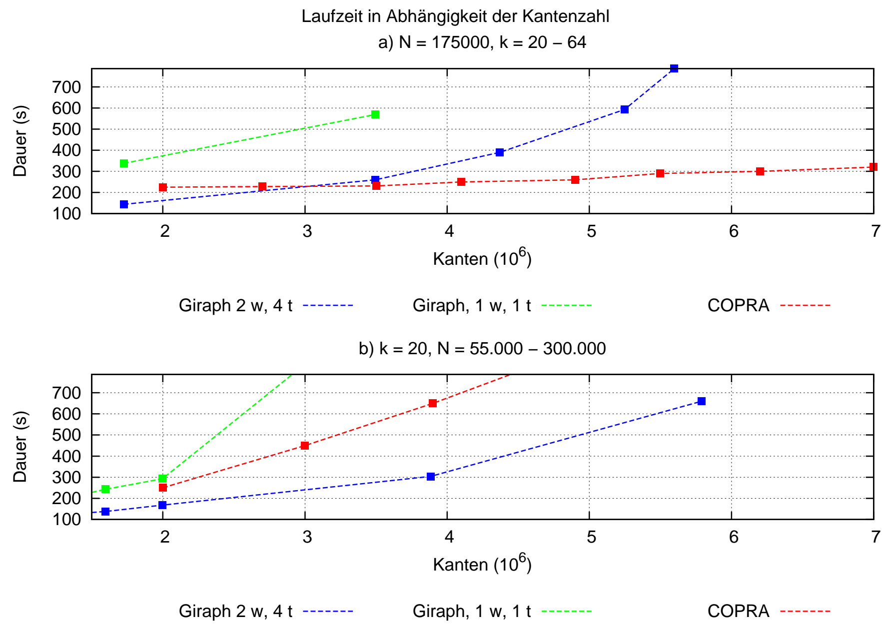

Abbildung 5.7.: Laufzeit bei steigender Graph-Größe (vgl. (Gregory, 2010, Abbildung 16)

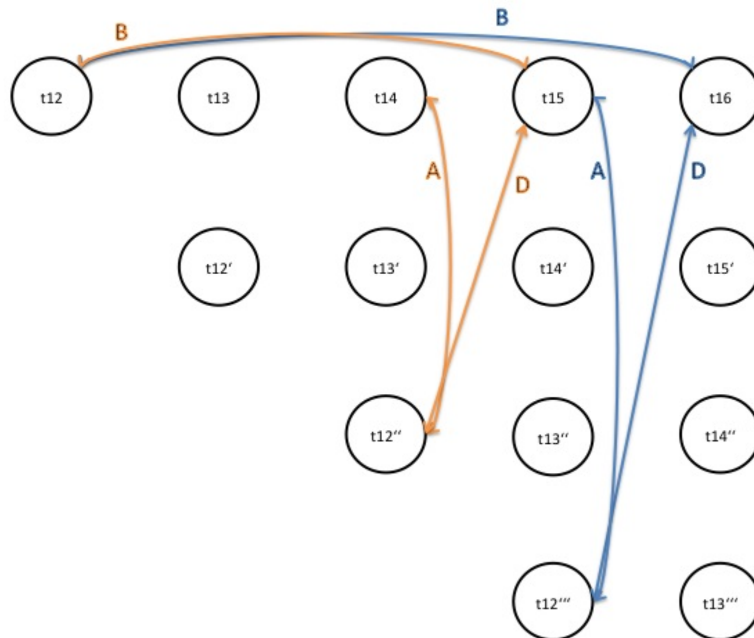

Abbildung 5.8.: Laufzeit bei steigender Graph-Größe (vgl. (Gregory, 2010, Abbildung 16)

6. Diskussion und Ausblick

In diesem Kapitel werden die Ergebnisse der Experimente zusammengefasst und bewertet. Die während der Arbeit aufgetretenen Probleme und Fragen werden erläutert. Nach einer kurzen Zusammenfassung werden die einzelnen Punkte der Arbeit aufgegriffen und bewertet, bevor abschließend ein Ausblick auf offene Punkte gegeben wird.

Ziel der Arbeit ist es, mit Hilfe der Community Detection, sowie der Node- und Link Prediction, Zwischenschritte in der Entwicklung von Gruppen zu generieren und zu analysieren. Auf Grund der immer größer werdenden Datenmengen und der damit einhergehenden Notwendigkeit der Parallelisierung, sollten die Verfahren mit Hilfe des Graph Processing Systems Apache Giraph implementiert werden. Für die Community Detection wurde, in einer frühen Phase der Arbeit, der COPRA Algorithmus (Gregory, 2010) implementiert. Auf Grund der ungünstigen Laufzeiteigenschaften der Implementierung, wurde entschieden für das Bestimmen der Gruppen auf eine existierende Implementierung des Louvain-Algorithmus (Blondel u. a., 2008) zurückzugreifen, da dieser Algorithmus auch in der Arbeit von Jung und Segev (2013) zum Einsatz kommt.

Die Node Prediction generiert neue Knoten und verbindet diese, unter zu Hilfenahme der Preferential Attachment Theorie (Newman, 2005) mit dem Graphen. Hierbei ist die Wahrscheinlichkeit für eine Kante zu einem Zielknoten umso größer, je mehr Kanten dieser Knoten bereits besitzt. Für die Link Prediction kommt der Jaccard-Koeffizient zum Einsatz. Dieser bewertet die Verbindung zweier Knoten durch das Verhältnis der Schnitt- und Vereinigungsmenge der Nachbarn dieser Knoten. Das Community Tracking wurde außerhalb von Giraph realisiert. Hierbei lag der Fokus auf der Wiedererkennung von Gruppen. Bei der Auswertung der Gruppenvergleiche wurde auf eine Unterscheidung der gruppendynamischen Ereignisse verzichtet.

Gegen Ende der Arbeit konnte eine zweite, laufzeitoptimierte Variante des COPRA Algorithmus implementiert werden. Es war jedoch nicht mehr möglich, diese für die Community Detection einzusetzen. Daher wurde der Algorithmus nur auf seine Eigenschaften hin untersucht. Die mit dem Algorithmus generierten Partitionen, fanden keine Verwendung in der Node- und Link Prediction.

Die Ergebnisse der durchgeführten Experimente wurden in Kapitel 5 vorgestellt und werden in diesem Kapitel besprochen. Für die Analyse des implementierten Community Detection Algorithmus wurden Szenarien generiert, die denen in Gregory (2010) ähnlich sind. Die Experimente für die Node- und Link Prediction wurden auf einem Zitationsnetzwerk durchgeführt, welches auch von Jung und Segev (2013) genutzt wird.

6.1. Ergebnisse

In diesem Abschnitt werden einzelnen Ergebnisse aus den Experimenten besprochen und bewertet.

6.1.1. Community Detection

Der für die Community Detection implementierte Algorithmus gliedert sich in drei Teilaufgaben, verteilt auf zwei Phasen, auf die nachfolgend eingegangen wird.

Phase 1: Label Propagation

Die direkte Umsetzung des COPRA Algorithmus in der ersten Variante hatte, neben langen Iterationszeiten, das Problem, dass die Iterationszeiten im Verlauf der Experimente sogar anstiegen. Dies ist auf eine unvorteilhafte Instanzierung von neuen Klassenobjekten bei der Serialisierung von Nachrichten zurückzuführen.. Dadurch wurde der Java Garbage Collector extrem stark beansprucht und hat einenAnstieg der Iterationszeiten zur Folge. Für die Laufzeitoptimierte Variante wurde auf das komplexe Stop-Kriterium , den Vergleich zweier Partitionen, verzichtet. Dadurch bestand keine Notwendigkeit mehr, die Partitionierung jeder Iteration in HBase zu verwalten. Gleichzeitig wurde, neben einer verbesserten Serialisierung der Nachrichten, die Emission neuer Nachrichten reduziert. Dadurch müssen nicht nur weniger Nachrichten serialisiert werden, sondern es ist außerdem möglich, Knoten, welche keine Nachrichten erhalten, zu deaktivieren. Zudem führte die Änderung des Stop-Kriteriums zu einer geringeren Anzahl an Iterationen im Vergleich zu der ersten Implementierung. Während die erste Implementierung, für Netzwerke mit 5000 Knoten stets > 200 Iterationen benötigte, liegt die Zahl der Iterationen nun im Schnitt unter 100[1].

[1]inklusive der Aufräum-Phase

Phase 2: Aufräumen

Diese Phase besteht aus zwei Teilaufgaben, die notwendig sind, um auftretende Nebeneffekte des Algorithmus zu korrigieren.

Entfernen von Subgruppen Für die Berechnung der Subgruppenverhältnisse, wurden in einer ersten Implementierung Aggregatoren eingesetzt. Hierfür registrierte der CopraMaster einen Aggregator je Gruppe. Auf Grund der geänderten Implementierung für Phase 1 wurden auch hier Änderungen vorgenommen. Die Berechnungen der Subgruppenverhältnisse wurde in je einen Knoten pro Gruppe verlagert. Hierfür wurde für jede Gruppe der Knoten gewählt, welcher der Gruppe ursprünglich ihren Namen gegeben hat. Dieses Vorgehen entspricht nun deutlich besser dem Konzept der lokalen Sicht, auf dem Giraph aufbaut.

Finden getrennter Gruppen Für das Finden von Gruppen, die das gleiche Label haben, aber disjunkt sind, wurde ein Broadcast implementiert, der feststellt, welche Knoten entlang der Kanten einer Gruppe erreicht werden können. Terminiert der Broadcast, kennt jeder Knoten den dieser Broadcast erreicht hat, die restlichen Knoten in der Gruppe. Für den Großteil der hier durchgeführten Experimente funktionierte der Broadcast gut. Die Laufzeit ist durch den längsten der kürzesten Pfade einer Gruppe beschränkt. Diese sind, auf Grund der *kleine Welt*-Eigenschaft des Netzwerkes, verhältnismäßig kurz. Jedoch wurde Giraph bei einigen Experimenten für größere Netzwerke, mit unbestimmtem Fehler beendet. Vermutlich war die bereitgestellte Hardware für diese Netzwerke nicht ausreichend dimensioniert. Für solche Fälle bietet Giraph die Möglichkeit sowohl die Daten der Knoten, als auch die generierten Nachrichten auf Massenspeicher zu serialisieren (Out-of-core[2]), um Speicherengpässe zu überbrücken. Durch eine, für jedes Experiment angepasste Konfiguration der Out-of-core-Funktion, hätten Systemabstürze verhindert werden können. Jedoch ist das Nutzen der Out-of-core-Funktion, auf Grund der Serialisierung, mit einer Verlängerung der Laufzeit verbunden. Neben dem horizontalen beziehungsweise vertikalen Skalieren der Infrastruktur, wird in Abschnitt 6.2 eine weitere Optimierung in Betracht gezogen. Als weitere Fehlerquelle muss die Konfiguration des Synchronisationsdienstes Zookeeper[3] in Betracht gezogen werden.

6.1.2. Node Prediction

Wie die Community Detection, hatte auch die erste Implementierung der Node Prediction Laufzeitprobleme. Die zunächst in einem Aggregator zwischengespeicherten, komplexen Daten

[2] http://giraph.apache.org/ooc.html
[3] http://zookeeper.apache.org

waren Grund für lange Iterationszeiten. In dem der Rechenaufwand in die Knoten verlagert wurde und auf die Aggregatoren verzichtet werden konnte, wurde die Iterationsdauer deutlich reduziert. Die für die Berechnung der initialen Kanten benötigten Iterationen, waren für alle Experimente auf den Daten des Zitationsnetzwerkes, konstant. Da die Anzahl der zu generierenden Knoten ein Parameter des Algorithmus ist, welcher durch den Anwender vorzugeben ist, war die Anzahl der generierten Knoten stets korrekt. Eine Erhebung der Precision- und Recall-Werte für die generierten, initialen Kanten der Node Prediction erfolgte nicht.

6.1.3. Link Prediction

Die implementierte Link Prediction versucht, die durch die initialen Kanten der Node Prediction entstandenen Dreiecke, mit Hilfe des Jaccard-Koeffizienten zu schließen. Die hierbei berechneten Werte für die zu erzeugenden Kanten waren dabei stets $< 0,2$, was darauf schließen lässt, das die durch die Node Prediction generierten Kanten entweder qualitativ nicht gut oder quantitativ nicht ausreichend waren. Am Ende der Arbeit war es nicht mehr möglich weitere Experimente durchzuführen, um geeignetere Parameter sowohl für die Node- als auch für die Link Prediction zu ermitteln. Aus dem gleichen Grund konnten auch die in 2.3 und 2.4.5 vorgestellten Erweiterungen für die Preferential Attachment Theorie und die gruppenbasierten Kennzahl für die Link Prediction nicht mehr getestet werden.

6.1.4. Community Tracking

Die Anzahl der, in einer Partition bestimmten Gruppen, wies teilweise stark Abweichungen auf. Dies führte unter anderem zu den unbefriedigenden Ergebnisse ist auf die Probleme in den beiden Prediction-Schritten begründet.

6.1.5. Überprüfung der Zielerreichung

Die zu Beginn der Arbeit in Kapitel 1.2 aufgestellten Ziele

1. Implementieren eines Community Detection Algorithmus für Apache Giraph
2. Implementierung von Node Prediction Algorithmen für Apache Giraph
3. Implementierung eines Link Prediction Algorithmus für Apache Giraph
4. Erzielen einer höheren Genauigkeit bei der Wiedererkennung von Gruppen, durch das Generieren von Zwischenschritten zwischen den bekannten Datensätzen im Gegensatz zu einem Vergleichen ohne Zwischenschritt.

wurden nur teilweise erreicht.

Der implementierte LPA/COPRA Algorithmus erzielt sowohl in der Erkennung von Gruppen, als auch im Laufzeitvergleich gute Ergebnisse. Für die Node Prediction wurde ein Ansatz implementiert, der zumindest bezüglich der Laufzeit gute Ergebnisse erzielt. Das verwendete Modell der Preferential Attachment Theorie, erzielt bei Jung und Segev (2013) gute Ergebnisse und bedarf einer korrekten Konfiguration. Da die Link Prediction, im Falle der Zitationsnetzwerke, von den Ergebnissen der Node Prediction abhängig ist, ist zu erwarten, dass auch hier bessere Ergebnisse erzielt werden könne. Zudem stehen mit den vorgestellten alternativen Kennzahlen eine Vielzahl an Möglichkeiten der Optimierung zur Verfügung

6.2. Ausblick

Auf Grund der, nur teilweise erreichten Ziele, bietet die Arbeit offene Punkte für eine weitergehende Betrachtung.

6.2.1. Community Detection

Für die verschiedenen Schritte des Community Detection Algorithmus gibt es weitere Ansatzpunkte für Optimierungen und Eigenschaften, die weiter untersucht werden können. Ein Hadoop Cluster, bestehend aus drei virtuellen Maschinen und Datensätze mit $4 \cdot 10^5$ Knoten und $6 \cdot 10^6$ Kanten, ist weit entfernt, von den Dimensionen, für die Giraph ausgelegt ist. Eine Analyse des Algorithmus auf einem, für den produktiven Einsatz dimensionierten Clusters und einer entsprechend großen Datenmengen, ist daher sicherlich interessant.

Label Propagation Phase

In der aktuellen Implementierung versendet ein Knoten nur die Menge, der von ihm Akzeptierten Label. Es bleibt zu untersuchen, wie sich der Algorithmus verhält, wenn zusätzlich die aktuellen Zugehörigkeitsfaktoren versendet und in die Neuberechnung des Labels integriert werden. Zudem kann das Verhalten des Algorithmus bei veränderter Lookup-Rate untersucht werden. Die Lookup-Rate ist die Wahrscheinlichkeit, mit welcher der Algorithmus, bei der Berechnung des aktuellen Labels eines Knotens, auf die vergangenen Labels der Nachbarn zurückgreift.

Finden getrennter Gruppen

Das Finden getrennter Gruppen, mit gleichem Gruppen-Identifikator, skaliert für größere Graphen schlecht. Dies liegt vor allem daran, dass die versendeten Nachrichten in dieser Phase des Community Detection Algorithmus relativ groß werden können. Eine mögliche Optimierung könnte in der Anpassung des *HCC Algorithmus* liegen, welcher zusammenhängende Komponenten in einem Graphen sucht. Die ursprüngliche Implementierung für Hadoop ist in Kang u. a. (2009) beschrieben. Der Algorithmus propagiert das kleinste Label einer zusammenhängenden Komponente entlang der Kanten in dieser Komponente. Es müsste also pro Gruppe immer nur ein einziges Label versendet werden. Entsprechend des aktuell implementierten Broadcast, könnten zusammenhängende Gruppen über die Gruppen-Identifikatoren begrenzt werden, in dem ein Knoten Nachrichten nur dann weiterleitet, wenn er Teil dieser Gruppe ist. Eine Giraph Implementierung des HCC Algorithmus findet sich in den Giraph-Beispielen[4].

6.2.2. Node- und Link Prediction

Der nächste Schritt im Bereich der Prediction ist die korrekte Konfiguration der Node Prediction, um die Ergebnisse aus (Jung und Segev, 2013) zu reproduzieren. Darauf aufbauend können dann alternative Modelle für die Node Prediction erarbeitet werden. Für die Link Prediction können, aufbauend auf einer funktionierenden Node Prediction für Zitationsnetzwerke, die in der Arbeit vorgestellten Kennzahlen getestet werden. Ein interessanter Aspekt ist hier die Verbesserung der Prediction im Verhältnis zu den eventuell zusätzlich benötigten Iterationen. Für die Berechnung neuer Kennzahlen, die über die Grenze der direkten Nachbarschaft hinausgehen, muss geprüft werden, welche Vor- beziehungsweise Nachteile ein Broadcast gegenüber dem Speichern quasi-lokaler Daten besitzt. Durch einen Broadcast kann die Menge an Informationen, die in einem Knoten gespeichert werden, muss klein gehalten werden. Durch das Speichern quasi-lokaler Informationen (Nachbarn dritten oder vierten Grades) kann der Speicherbedarf für einen einzelnen Knoten stark ansteigen, bietet jedoch die Möglichkeit, einem weiter entfernten Nachbarn, direkt eine Nachricht zu senden und diese nicht über eine Kante pro Iteration durch den Graph wandern zu lassen.

[4] org.apache.giraph.examples.ConnectedComponentsComputation

A. Inhalt der DVD

Dieser Arbeit liegen eine DVD bei, die die folgende Daten beinhalten.

- diese Arbeit als PDF
- den Ursprungsdaten des Zitationsnetzwerks HepPh
- die Skripte für das Ausführen der Experimente
- die C++ Bibliotheken des Louvain Algorithmus
- die C++ Bibliotheken für den Graph-Generator
- die C++ Bibliotheken zur Berechnung der NMI-Kennzahlen

Das Abbild eines Single Node Clusters zur Ausführung der Experimente wird separat über einen Download bereitgestellt. Das Abbild enthält neben dem Framework Apache Giraph, den Quellcode der Arbeit und sämtliche, auch auf der DVD befindlichen Skripte und übersteigt daher die Kapazität einer DVD.

Literaturverzeichnis

[EncyclopediaParallelComputing 2011] Padua, David (Hrsg.): *Encyclopedia of Parallel Computing*. 2011. – URL http://www.springer.com/computer/swe/book/978-0-387-09765-7?wt_mc=Google-_-Book%20Search-_-Springer-_-EN&token=gbgen

[Asur u. a. 2007] Asur, Sitaram ; Parthasarathy, Srinivasan ; Ucar, Duygu: An event-based framework for characterizing the evolutionary behavior of interaction graphs. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA : ACM, 2007 (KDD '07), S. 913–921. – URL http://doi.acm.org/10.1145/1281192.1281290. – ISBN 978-1-59593-609-7

[Barber und Clark 2009] Barber, Michael J. ; Clark, John W.: Towards real-time community detection in large networks. In: *Phys. Rev. E* 80 (2009), Jun, S. 026129. – URL http://arxiv.org/pdf/0808.2633.pdf

[Blondel u. a. 2008] Blondel, Vincent D. ; Guillaume, Jean-Loup ; Lambiotte, Renaud ; Lefebvre, Etienne: Fast unfolding of communities in large networks. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008 (2008), Nr. 10, S. P10008. – URL http://stacks.iop.org/1742-5468/2008/i=10/a=P10008

[Brandes u. a. 2006] Brandes, Ulrik ; Delling, Daniel ; Gaertler, Marco ; Görke, Robert ; Hoefer, Martin ; Nikoloski, Zoran ; Wagner, Dorothea: *On Modularity – NP-Completeness and Beyond*. 2006

[Chakrabarti u. a. 2006] Chakrabarti, Deepayan ; Kumar, Ravi ; Tomkins, Andrew: Evolutionary clustering. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA : ACM, 2006 (KDD '06), S. 554–560. – URL http://doi.acm.org/10.1145/1150402.1150467. – ISBN 1-59593-339-5

[Chen u. a. 2010] Chen, Zhengzhang ; Wilson, K.A. ; Jin, Ye ; Hendrix, W. ; Samatova, N.F.: Detecting and Tracking Community Dynamics in Evolutionary Networks. In: *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, Dec 2010, S. 318–327

[Ching 2012] Ching, Avery: *Processing Edges on Apache Giraph*. 2012. – URL http://de.slideshare.net/Hadoop_Summit/processing-edges-on-apache-giraph

[Curtiss u. a. 2013] Curtiss, Michael ; Becker, Iain ; Bosman, Tudor ; Doroshenko, Sergey ; Grijincu, Lucian ; Jackson, Tom ; Kunnatur, Sandhya ; Lassen, Soren ; Pronin, Philip ; Sankar, Sriram ; Shen, Guanghao ; Woss, Gintaras ; Yang, Chao ; Zhang, Ning: Unicorn: A System for Searching the Social Graph. In: *Proc. VLDB Endow.* 6 (2013), August, Nr. 11, S. 1150–1161. – URL http://dl.acm.org/citation.cfm?id=2536222.2536239. – ISSN 2150-8097

[Danon u. a. 2005] Danon, Leon ; Díaz-Guilera, Albert ; Duch, Jordi ; Arenas, Alex: Comparing community structure identification. In: *Journal of Statistical Mechanics: Theory and Experiment* 2005 (2005), Nr. 09, S. P09008. – URL http://stacks.iop.org/1742-5468/2005/i=09/a=P09008

[Diestel 2010] Diestel, Reinhard: *Graph theory. 4th ed.* 4th ed. Berlin: Springer, 2010. – xviii, 436 p. S. – ISBN 978-3-642-14278-9

[Euler 1736] Euler, L.: Solutio problematis ad geometriam situs pertinentis. In: *Commentarii Academiae Petropolitanae* 8 (1736), S. 128–140

[Fard u. a. 2012] Fard, Arash ; Abdolrashidi, Amir ; Ramaswamy, Lakshmish ; A., Miller J.: Towards Efficient Query Processing on Massive Time-Evolving Graphs. In: *8th International Conference Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2012, S. 567 – 574

[Fortunato 2010] Fortunato, Santo: Community detection in graphs. In: *Physics Reports* 486 (2010), Nr. 3-5, S. 75–174

[Girvan und Newman 2002] Girvan, M. ; Newman, M. E. J.: Community structure in social and biological networks. In: *Proceedings of the National Academy of Sciences* 99 (2002), Nr. 12, S. 7821–7826

[Greene u. a. 2010] Greene, D. ; Doyle, D. ; Cunningham, P.: Tracking the Evolution of Communities in Dynamic Social Networks. In: *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, 2010, S. 176–183

[Gregory 2010] Gregory, Steve: Finding overlapping communities in networks by label propagation. In: *New Journal of Physics 12, 103018* (2010).

– URL http://arxiv.org/ct?url=http%3A%2F%2Fdx.doi.org%2F10%252E1088%2F1367-2630%2F12%2F10%2F103018&v=d068f10e

[Hopcroft u. a. 2004] Hopcroft, John ; Khan, Omar ; Kulis, Brian ; Selman, Bart: Tracking evolving communities in large linked networks. In: *Proceedings of the National Academy of Sciences* 101 (2004), April, S. 5249–5253. – URL http://www.pnas.org/cgi/content/full/101/suppl_1/5249

[Jung und Segev 2013] Jung, Sukhwan ; Segev, Aviv: Analyzing future communities in growing citation networks. In: *Proceedings of the 2013 international workshop on Mining unstructured big data using natural language processing*. New York, NY, USA : ACM, 2013 (UnstructureNLP '13), S. 15–22. – URL http://doi.acm.org/10.1145/2513549.2513553. – ISBN 978-1-4503-2415-1

[Kang u. a. 2009] Kang, U. ; Tsourakakis, Charalampos E. ; Faloutsos, Christos: PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In: *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*. Washington, DC, USA : IEEE Computer Society, 2009 (ICDM '09), S. 229–238. – URL http://dx.doi.org/10.1109/ICDM.2009.14. – ISBN 978-0-7695-3895-2

[Kim und Bonneau 2009] Kim, Hyoungshick ; Bonneau, Joseph: Privacy-enhanced Public View for Social Graphs. In: *Proceedings of the 2Nd ACM Workshop on Social Web Search and Mining*. New York, NY, USA : ACM, 2009 (SWSM '09), S. 41–48. – URL http://doi.acm.org/10.1145/1651437.1651445. – ISBN 978-1-60558-806-3

[Lancichinetti und Fortunato 2009] Lancichinetti, Andrea ; Fortunato, Santo: Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. (2009). – URL http://arxiv.org/abs/0904.3940

[Lancichinetti u. a. 2009] Lancichinetti, Andrea ; Fortunato, Santo ; Kertesz, Janos: Detecting the overlapping and hierarchical community structure of complex networks. In: *New Journal of Physics 11, 033015* (2009). – URL http://arxiv.org/abs/0802.1218

[Lancichinetti u. a. 2008] Lancichinetti, Andrea ; Fortunato, Santo ; Radicchi, Filippo: Benchmark graphs for testing community detection algorithms. In: *Phys. Rev. E* 78 (2008), Oct, S. 046110. – URL http://link.aps.org/doi/10.1103/PhysRevE.78.046110

[Leskovec u. a. 2008] Leskovec, Jure ; Backstrom, Lars ; Kumar, Ravi ; Tomkins, Andrew: Microscopic Evolution of Social Networks. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA : ACM,

2008 (KDD '08), S. 462–470. – URL http://doi.acm.org/10.1145/1401890.1401948. – ISBN 978-1-60558-193-4

[Leskovec u. a. 2005] Leskovec, Jure ; Kleinberg, Jon ; Faloutsos, Christos: Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. New York, NY, USA : ACM, 2005 (KDD '05), S. 177–187. – URL http://doi.acm.org/10.1145/1081870.1081893. – ISBN 1-59593-135-X

[Leung u. a. 2009] Leung, Ian X. Y. ; Hui, Pan ; Liò, Pietro ; Crowcroft, Jon: Towards real-time community detection in large networks. In: *Phys. Rev. E* 79 (2009), Jun, S. 066107. – URL http://link.aps.org/doi/10.1103/PhysRevE.79.066107

[Liben-Nowell und Kleinberg 2007] Liben-Nowell, David ; Kleinberg, Jon: The link-prediction problem for social networks. In: *Journal of the American Society for Information Science and Technology* 58 (2007), Nr. 7, S. 1019–1031. – URL http://dx.doi.org/10.1002/asi.20591. – ISSN 1532-2890

[Liu u. a. 2012] Liu, Wei ; Kan, Andrey ; Chan, Jeffrey ; Bailey, James ; Leckie, Christopher ; Pei, Jian ; Kotagiri, Ramamohanarao: On Compressing Weighted Time-evolving Graphs. In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2012 (CIKM '12), S. 2319–2322. – URL http://doi.acm.org/10.1145/2396761.2398630. – ISBN 978-1-4503-1156-4

[Lü u. a. 2009] Lü, L. ; Jin, C.H. ; Zhou, T.: Similarity index based on local paths for link prediction of complex networks. In: *Physical Review E* 80 (2009), Nr. 4, S. 046122

[Lü und Zhou 2011] Lü, Linyuan ; Zhou, Tao: Link prediction in complex networks: A survey. In: *Physica A: Statistical Mechanics and its Applications* 390 (2011), Nr. 6, S. 1150 – 1170. – URL http://www.sciencedirect.com/science/article/pii/S037843711000991X. – ISSN 0378-4371

[MacKay 2002] MacKay, David J. C.: *Information Theory, Inference & Learning Algorithms*. New York, NY, USA : Cambridge University Press, 2002. – ISBN 0521642981

[Malewicz u. a. 2010] Malewicz, Grzegorz ; Austern, Matthew H. ; Bik, Aart J. ; Dehnert James C. ; Horn, Ilan ; Leiser, Naty ; Czajkowski, Grzegorz: Pregel: A System for Large-Scale Graph Processing. In: *SIGMOD'10,* (2010), S. 135–145

[Martella 2012] Martella, Claudio: *Apache Giraph: Distributed Graph Processing in the Cloud.* 2012. – URL http://prezi.com/9ake_klzwrga/apache-giraph-distributed-graph-processing-in-the-cloud/

[McDaid und Hurley 2010] McDaid, A. ; Hurley, N.: Detecting Highly Overlapping Communities with Model-Based Overlapping Seed Expansion. In: *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, 2010, S. 112–119

[McDaid u. a. 2011] McDaid, Aaron F. ; Greene, Derek ; Hurley, Neil J.: Normalized Mutual Information to evaluate overlapping community finding algorithms. In: *CoRR* abs/1110.2515 (2011). – URL http://arxiv.org/pdf/1110.2515v2.pdf

[Milgram 1967] Milgram, Stanley: The Small World Problem. In: *Psychology Today* 67 (1967), Nr. 1, S. 61–67

[Newman 2004] Newman, M. E. J.: Analysis of weighted networks. In: *Phys. Rev. E* 70 (2004), Nov, S. 056131. – URL http://link.aps.org/doi/10.1103/PhysRevE.70.056131

[Newman 2005] Newman, M. E. J.: Power laws, Pareto distributions and Zipf's law. In: *Contemporary Physics* 46 (2005), December, S. 323–351. – URL http://arxiv.org/abs/cond-mat/0412004

[Palla u. a. 2007] Palla, Gergely ; Barabási, Albert lászló ; Vicsek, Tamás ; Hungary, Budapest: Quantifying social group evolution. In: *Nature* 446 (2007), S. 2007

[Raghavan u. a. 2007] Raghavan, Usha N. ; Albert, Reka ; Kumara, Soundar: *Near linear time algorithm to detect community structures in large-scale networks.* September 2007. – URL http://arxiv.org/abs/0709.2938

[Scott 2000] Scott, John: *Social Network Analysis: A Handbook.* Second. Sage Publications, 2000. – URL http://www.amazon.com/Social-Network-Analysis-Professor-Scott/dp/0761963383/ref=sr_1_1?ie=UTF8&s=books&qid=1256622319&sr=1-1. – ISBN 0761963391

[Soundarajan und Hopcroft 2012] Soundarajan, Sucheta ; Hopcroft, John: Using community information to improve the precision of link prediction methods. In: *Proceedings of the 21st international conference companion on World Wide Web.* New York, NY, USA : ACM, 2012 (WWW '12 Companion), S. 607–608. – URL http://doi.acm.org/10.1145/2187980.2188150. – ISBN 978-1-4503-1230-1

[Wasserman und Faust 1994] Wasserman, S. ; Faust, K. ; Granovetter, Mark (Hrsg.): *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994

[Watts und Strogatz 1998] Watts, D.J. ; Strogatz, S.H.: Collective dynamics of 'small-world' networks. In: *Nature* (1998), Nr. 393, S. 440–442

[White 2012] White, Tom: *Hadoop: The Definitive Guide*. Yahoo Press, 2012. – 688 S

[Wilson u. a. 2012] Wilson, Christo ; Sala, Alessandra ; Puttaswamy, Krishna P. N. ; Zhao, Ben Y.: Beyond Social Graphs: User Interactions in Online Social Networks and Their Implications. In: *ACM Trans. Web* 6 (2012), November, Nr. 4, S. 17:1–17:31. – URL http://doi.acm.org/10.1145/2382616.2382620. – ISSN 1559-1131

[Xie u. a. 2011] Xie, Jierui ; Kelley, Stephen ; Szymanski, Boleslaw K.: Overlapping Community Detection in Networks: the State of the Art and Comparative Study. In: *CoRR* abs/1110.5813 (2011)

[Xie und Szymanski 2011] Xie, Jierui ; Szymanski, Boleslaw K.: Community Detection Using A Neighborhood Strength Driven Label Propagation Algorithm. In: *CoRR* abs/1105.3264 (2011)

[Zheleva u. a. 2010] Zheleva, Elena ; Getoor, Lise ; Golbeck, Jennifer ; Kuter, Ugur: Using friendship ties and family circles for link prediction. In: *Proceedings of the Second international conference on Advances in social network mining and analysis*. Berlin, Heidelberg : Springer-Verlag, 2010 (SNAKDD'08), S. 97–113. – URL http://dl.acm.org/citation.cfm?id=1883692.1883698. – ISBN 3-642-14928-6, 978-3-642-14928-3

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Mai 2014 Pascal Jäger