Karlsruhe Institute of Technology

Modularity-basiertes Clustern von dynamischen Graphen im Offline-Fall

Diplomarbeit
von

David Lisowski

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuender Mitarbeiter:	Dipl.-Inform. Andrea Schumm
Zweiter betreuender Mitarbeiter:	Dr. rer. nat. Robert Görke

Bearbeitungszeit: 11. Januar 2011 – 10. Juli 2011

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1. Einleitung

Graphen sind sehr weit verbreitet im alltäglichen Leben. So werden beispielsweise soziale Netzwerke, Netzpläne der Bahn, Stadtpläne, Familien-Stammbäume oder Telefon-Netzwerke durch Graphen visualisiert. Im Falle eines sozialen Netzwerkes stehen beispielsweise die Knoten stellvertretend für Personen und eine Kante deutet auf eine Freundschaft zweier Personen hin.

Für solche Graphen existieren unterschiedliche Heuristiken, mit denen eine Partitionierung der Knotenmenge in sogenannte *Cluster* berechnet werden kann. So eine Partitionierung, die auch als *Clusterung* bezeichnet wird, soll eine gewisse Qualität besitzen. Ziel ist es, dass möglichst viele Kanten innerhalb und wenige Kanten zwischen den unterschiedlichen Cluster verlaufen. Hierfür wird in dieser Arbeit die Qualitätsfunktion *Modularity* [NG04] betrachtet. Im Falle eines sozialen Netzwerkes würde man mit Hilfe einer Clusterung Freundeskreise aufdecken: In einem Cluster befinden sich Personen, die alle eng miteinander in Verbindung stehen.

Beobachtet man solche Netzwerke über einen längeren Zeitraum hinweg, so wird deutlich, dass die meisten dieser Netzwerke in der Praxis nicht statisch sondern dynamisch sind, da im Laufe der Zeit neue Knoten und Kanten hinzukommen, aber auch bereits existierende Knoten und Kanten verschwinden. Ein *dynamischer Graph* ist eine Folge von Graphen, die dasselbe Netzwerk zu unterschiedlichen Zeitpunkten darstellen. In dieser Arbeit werden nur dynamische Graphen betrachtet, bei denen die komplette Zeitspanne bekannt ist. Dies bedeutet, dass alle zeitlichen Ausprägungen eines Netzwerkes bereits vor dem Suchen der Clusterung bekannt sind. Dieses Wissen wird eingesetzt, um nicht nur qualitativ geeignete Clusterungen zu finden, sondern auch eine Ähnlichkeit zwischen den Clusterungen der einzelnen Zeitpunkte anzustreben, damit die Entwicklung des Netzwerkes und der zugehörigen Clusterung über die Zeit verfolgt werden kann. Clustert man nämlich so einen dynamischen Graphen nur mittels Modularity-basierter Algorithmen, erhält man zwar qualitativ gute Ergebnisse, erfahrungsgemäß unterscheiden sich aber die Clusterungen verschiedener Zeitpunkte deutlich voneinander. Dies liegt nicht notwendigerweise daran, dass sich die Struktur des Graphen deutlich verändert, sondern kann auch daran liegen, dass der betrachtete Graph mehrere Clusterungen hoher Qualität besitzt und der Clusteralgorithmus irgendeine davon auswählt. In diesem Fall ist es wünschenswert, dass eine Clusterung gewählt wird, die eine Ähnlichkeit zu den Clusterungen vergangener bzw. zukünftiger Zeitpunkte hat.

Für die Bewertung der Ähnlichkeit zweier Clusterungen existieren diverse Vergleichsfunktionen in der Literatur. Eine dieser Funktionen, auf der das Hauptaugenmerk dieser Arbeit

liegt, ist der *Graph-theoretische Rand-Index* [Gör10], der nur Clusterungen von Knotenpaaren berücksichtigt, die mit einer Kante in Verbindung stehen.

1.1. Verwandte Arbeiten

In [GMS+11] werden einige Algorithmen zum Clustern von dynamischen Graphen vorgestellt, die implizit auch Modularity und Graph-theoretischen Rand-Index optimieren. Sie sind speziell für ungewichtete Graphen gedacht, bei denen sich nur vergleichsweise wenige Kanten zwischen benachbarten zeitlichen Ausprägungen des dynamischen Graphen ändern. Einer dieser Algorithmen ist die Grundlage für den im Kapitel 5.2 vorgestellten *Timestep Greedy Algorithmus*.

Evolutionary Clustering [CKT06] von Chakrabarti et al. ist ein Ansatz, der im Kontext des verwandten Forschungsfeldes des Clusterns von Daten entstanden ist. Man erhält jeden Tag neue Daten, die in die Clusterung mit eingebaut werden sollen. Für jeden Zeitschritt existiert eine Clusterung der vorliegenden Daten. Diese Clusterung soll nicht nur von hoher Qualität sein, sondern auch ähnlich, falls die Daten des vorhergehenden Zeitpunktes sich kaum von den aktuell betrachtenden Daten unterscheiden. Sind die Daten jedoch sehr verschieden, soll dies durch die Clusterungen widergespiegelt werden. Chakrabarti et al. stellen ein Framework vor, bei dem all dies berücksichtigt wird. Bestehende Algorithmen wurden entsprechend dem Framework angepasst und mittels Experimente wurde gezeigt, dass diese Algorithmen die gewünschten Eigenschaften besitzen. Dies ähnelt sehr stark der Problemstellung, die wir in dieser Arbeit zum Clustern von Graphen behandeln.

In [GHW09] wird ein Ansatz zum Clustern von dynamischen Graphen vorgestellt. Der dort vorgestellte Algorithmus ist Schnitt-basiert und hat ebenfalls zum Ziel, stabile Clusterungen für dynamische Graphen zu finden. Dieser Algorithmus unterscheidet sich von den in dieser Arbeit betrachteten Algorithmen dadurch, dass er nicht auf Modularity basiert.

Das in [Gla08] vorgestellte *zeitexpandierte Clustern* baut aus allen zeitlichen Ausprägungen eines dynamischen Graphen einen einzigen aggregierten Graphen, der dann mittels Modularity geclustert wird. Anhand dieser Clusterung werden dann Clusterungen für die einzelnen Zeitpunkte bestimmt. Diese Idee wird in Kapitel 4.1 detailliert beschrieben und bei den Versuchen zum Vergleich mit den anderen in dieser Arbeit vorgestellten Heuristiken genutzt.

1.2. Zielsetzung der Arbeit

Modularity und Graph-theoretischer Rand-Index werden zu einer gemeinsamen Zielfunktion kombiniert, bei welcher mit Hilfe eines Parameters α die beiden Kriterien gegeneinander gewichtet werden. Das Ziel dieser Arbeit ist es, für diese Zielfunktion gute Clusterungen für dynamische Graphen zu finden. Hierfür werden diverse dynamische Graphen mittels bereits existierender und neuer Heuristiken experimentell untersucht und miteinander verglichen.

1.3. Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen beschrieben, die zum Verständnis dieser Arbeit gebraucht werden. In Kapitel 3 werden Heuristiken vorgestellt, die eine Modularity-basierte Clusterung eines statischen Graphen vornehmen. Kapitel 4 beschreibt die Problemstellung und Zielsetzung dieser dieser Arbeit. Es erfolgt zudem eine formale Definition der Zielfunktion. Außerdem werden bereits in der Literatur bekannte Heuristiken vorgestellt, um gute und ähnliche Clusterungen zu finden. Darauf aufbauend werden in Kapitel 5 neue Heuristiken zur Findung einer Clusterung eines dynamischen Graphen mit komplett bekannter

Zeitspanne vorgestellt. In Kapitel 6 erfolgt eine experimentelle Untersuchung dieser unterschiedlichen Algorithmen und Kapitel 7 fasst die gewonnenen Erkenntnisse dieser Arbeit zusammen.

2. Grundlagen

In diesem Kapitel werden alle Begriffe und Methoden vorgestellt, die für das Verständnis dieser Arbeit von Bedeutung sind.

2.1. Begriffe

In dieser Arbeit wird durchgängig folgende Notation verwendet.

2.1.1. Graphen

Sei $G = (V, E, \omega)$ ein ungerichteter, gewichteter Graph, der Schleifen, aber keine parallelen Kanten enthalten kann. G besteht aus einer Knotenmenge V, einer Kantenmenge E und einer Kantengewichtsfunktion $\omega : E \to \mathbb{R}_+$.

Sei $n := |V|$ die Anzahl der Knoten und $m := |E|$ die Anzahl aller Kanten, wobei eine Kante e einem ungeordneten Paar $e = \{u, v\}$ zweier Knoten $u, v \in V$ entspricht. Zwei Knoten u und v heißen *benachbart* bzw. *adjazent*, wenn eine Kante der Form $\{u, v\}$ existiert. Eine Kante e, die zwei Knoten u und v verbindet, heißt jeweils *inzident* zu u und zu v bzw. u und v heißen inzident zu e.

Der *Grad eines Knotens* v entspricht dessen Anzahl inzidenter Kanten und wird als $deg(v)$ bezeichnet. Sollte eine dieser inzidenten Kanten von v eine Schleife $\{v, v\}$ sein, so wird diese Kante in $deg(v)$ doppelt gezählt.

Die Kantengewichtsfunktion ω ordnet jeder Kante $e \in E$ ein Gewicht $\omega(e) \in \mathbb{R}_+$ zu. Der *gewichtete Grad* $deg_\omega(v)$ *eines Knotens* $v \in V$ entspricht der Summe der Gewichte aller zu diesem Knoten inzidenten Kanten, wobei das Gewicht von Schleifen doppelt in die Berechnung einfließt.

2.1.2. Clustering

Mit *Clustering* bzw. *Clustern* wird der Vorgang bezeichnet, bei dem für einen gegebenen Graphen G unter Berücksichtigung verschiedener Kriterien eine Partitionierung $\mathcal{C} = \{C_1, \ldots, C_k\}$ der Knotenmenge berechnet werden soll. Diese Partitionierung nennt man auch *Clusterung*. Die Elemente dieser Clusterung heißen *Cluster*.

Die Knoten eines Clusters C induzieren einen Subgraphen $G[C] := (C, E(C), w_{|E(C)})$. Die Kantenmenge $E(C) := \{\{u, v\} \in E : u, v \in C\}$ dieses Subgraphen ist die Menge aller Kanten, die innerhalb des Clusters C verlaufen. Diese Art von Kanten werden auch *Intracluster-Kanten* genannt.

Der *Grad eines Clusters* C wird als $deg(C) := \sum_{v \in C} deg(v)$ festgelegt. $C(u)$ ist dabei der Cluster aus der Clusterung $\mathcal{C}$, in welchem der Knoten $u \in V$ enthalten ist.

Die Menge aller Intracluster-Kanten einer Clusterung $\mathcal{C}$ eines Graphen $G = (V, E, \omega)$ ist definiert als $E(\mathcal{C}) := \bigcup_{C \in \mathcal{C}} E(C)$. Die Menge der *Intercluster-Kanten* ist $E \backslash E(\mathcal{C})$ und $E(C_i, C_j)$ entspricht der Menge der Kanten, die Knoten aus C_i mit Knoten aus C_j verbinden. Cluster müssen nicht zwingend zusammenhängend sein.

$\omega(\mathcal{C})$ ist die Summe der Gewichte aller Intracluster-Kanten eines Graphen G. Dementsprechend ist $\overline{\omega}(\mathcal{C})$ die Summe der Gewichte aller Intercluster-Kanten. Daraus lässt sich das Gesamtgewicht $W := \omega(\mathcal{C}) + \overline{\omega}(\mathcal{C})$ ableiten.

2.2. Bewertung von Clusterungen

Das Ziel des Clusterings ist es, für einen gegebenen Graphen eine Clusterung zu finden, die eine hohe Intracluster-Dichte mit einer niedrigen Intercluster-Dichte vereint. Mit Hilfe sogenannter *Qualitätsfunktionen* ist es möglich, solch eine Clusterung zu bewerten. Dabei gilt für alle im Folgenden vorgestellten Qualitätsfunktionen: Je höher der Funktionswert ist, umso besser ist die Clusterung. Ein einfacher Vertreter solcher Funktionen ist *Coverage*.

2.2.1. Coverage

$$cov_\omega(\mathcal{C}) = \frac{\omega(\mathcal{C})}{W}$$

Coverage ist das Verhältnis des Gesamtgewichtes aller Intracluster-Kanten zu dem Gesamtgewicht aller Kanten des Graphen. Im Falle, dass alle Kanten ein Gewicht von 1 besitzen, bedeutet dies, dass sich möglichst viele Kanten des Graphen innerhalb eines Clusters befinden sollen und möglichst wenig Kanten zwischen den unterschiedlichen Cluster verlaufen sollen. Der Wertebereich von Coverage liegt in $[0, 1]$. Da hier nur das Verhältnis von Intracluster-Gewicht zum Gesamtgewicht betrachtet wird, erhält man für den trivialen Fall, dass sich alle Knoten in einem einzigen Cluster befinden, eine maximale Coverage von 1. Deshalb wird Coverage im Normalfall nicht als alleiniges Maß benutzt, um Clusterungen zu bewerten.

2.2.2. Modularity

Eine andere Qualitätsfunktion, die auf Coverage aufbaut, ist *Modularity* (siehe Newman und Girvan [NG04]):

$$mod_\omega(\mathcal{C}) = \underbrace{\frac{\omega(\mathcal{C})}{W}}_{=cov_\omega(\mathcal{C})} - \underbrace{\frac{1}{4W^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} deg_\omega(v) \right)^2}_{=\mathbb{E}(cov_\omega(\mathcal{C}))}$$

Modularity berechnet die Coverage einer gegebenen Clusterung und zieht davon den Erwartungswert der Coverage derselben Clusterung in einem *Zufallsgraphen* mit ähnlichen Eigenschaften ab. Diesen Zufallsgraphen bekommt man, indem man alle Kanten aus einem Graphen entfernt und sie erneut zufällig einfügt. Das Zufallsmodell ist dabei so gewählt, dass jeder Knoten im Erwartungswert denselben gewichteten Grad erhält wie im Ursprungsgraphen.

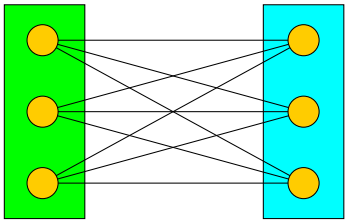


Abbildung 2.1.: Beispiel für eine Clusterung mit minimaler Modularity

Der Wertebereich von Modularity liegt in $[-0.5, 1]$. Im Falle, dass ein Graph keine Kanten hat, ist Modularity mit dem Maximum von 1 definiert, unabhängig davon, welche Clusterung vorliegt. Um zu zeigen, wie der minimale Wert von -0.5 erreicht wird, benötigt man zunächst die Definition eines *vollständigen, bipartiten Graphen* K_{n_1,n_2}. Für solch einen Graphen K_{n_1,n_2} ist die Knotenmenge $V := U_1 \cup U_2$ mit $|U_1| = n_1$, $|U_2| = n_2$ und $U_1 \cap U_2 = \varnothing$. Die Kantenmenge ist als $E := \{\{u, v\} | u \in U_1, v \in U_2\}$ definiert. Der minimale Wert wird bei einer Clusterung $\mathcal{C} = \{U_1, U_2\}$ erreicht. Ein Beispiel für so eine Clusterung ist in Abbildung 2.1 zu erkennen. Hierbei handelt es sich um einen $K_{3,3}$. Für diese Art von Clusterungen $\mathcal{C}$ gilt immer: $\omega(\mathcal{C}) = 0$, da es keine Intracluster-Kanten gibt und $\sum_{v \in C} deg_\omega(v) = W$, da jeweils m Kanten zwischen den Clustern verlaufen. Somit ergibt sich für die Modularity einer Clusterung $\mathcal{C} := \{U_1, U_2\}$ eines vollständigen, bipartiten Graphen K_{n_1,n_2}:

$$\begin{aligned} mod_\omega(\mathcal{C}) &= \frac{\omega(\mathcal{C})}{W} - \frac{1}{4W^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} deg_\omega(v) \right)^2 \\ &= \frac{0}{W} - \frac{1}{4W^2}(W^2 + W^2) \\ &= 0 - \frac{1}{2} \\ &= -\frac{1}{2} \end{aligned}$$

2.3. Vergleich von Clusterungen

Für den Fall, dass man einen Graphen $G = (V, E, \omega)$ auf verschiedene Arten geclustert hat und diese Clusterungen untereinander vergleichen will, existieren unterschiedliche Vergleichstechniken, die die sogenannte *Distanz* von zwei Clusterungen messen. Die Distanz ist ein Maß für die Ähnlichkeit zweier Clusterungen.

Natürlich besteht auch die Möglichkeit, zwei Clusterungen $\mathcal{C}$ und $\mathcal{C}'$ anhand einer Qualitätsfunktion wie Modularity zu vergleichen anstatt eine Distanz-basierte Vergleichstechnik einzusetzen. Allerdings ist dabei Vorsicht geboten, denn es könnte durchaus der Fall eintreten, dass zwei Clusterungen eines identischen Graphen denselben Modularity Wert besitzen, die Clusterungen selbst aber unterschiedlich sind. Dieser Sachverhalt wird in Abbildung 2.2 veranschaulicht. In der Clusterung links sind die Knoten 1,2 und 3 in einem Cluster und die Knoten 4 und 5 im anderen Cluster. Die rechte Clusterung ist sehr ähnlich, nur dass Knoten 3 jetzt im Cluster von Knoten 4 und 5 ist. Beide Clusterungen besitzen zwar den gleichen Modularity Wert, sind aber dennoch unterschiedlich. Daher ist es zu empfehlen, einen Distanz-basierten Vergleich zweier Clusterungen $\mathcal{C}$ und $\mathcal{C}'$ eines Graphen G vorzunehmen. Eine Möglichkeit eines Distanz-basierten Vergleichs ist durch das Zählen von Knotenpaaren gegeben.

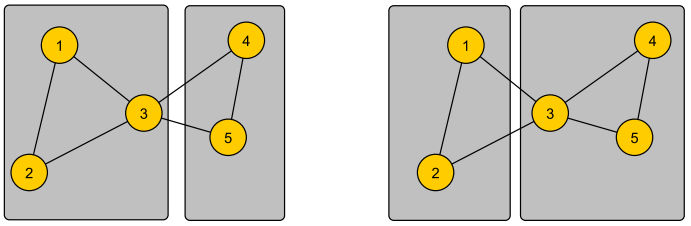


Abbildung 2.2.: Selbe Modularity - unterschiedliche Clusterungen

2.3.1. Distanz-basierter Vergleich: Zählen von Knotenpaaren

In [WW07] werden diverse Vergleichstechniken vorgestellt, die sich mit dem Zählen von Knotenpaaren beschäftigen. Es wird zunächst für jedes ungeordnete Knotenpaar $\{u, v\}$ mit $u \neq v$ betrachtet, ob sich die Knoten im selben Cluster befinden. Dies geschieht in beiden Clusterungen $\mathcal{C}$ und $\mathcal{C}'$ unabhängig voneinander. Sind zwei Knoten in einer Clusterung im selben Cluster, wird dies mit einer 1 gekennzeichnet, ansonsten mit einer 0. Darauf basierend werden vier Mengen S_{ij} mit $i, j \in \{0, 1\}$ definiert. Der Index i steht für die erste und der Index j für die zweite Clusterung. S_{00} entspricht der Menge aller Knotenpaare, die sowohl in Clusterung $\mathcal{C}$ als auch in $\mathcal{C}'$ in unterschiedlichen Clustern sind. S_{11} ist die Menge aller Knotenpaare die in beiden Clusterungen im selben Cluster sind. S_{10} enthält alle Knotenpaare die in Clusterung $\mathcal{C}$ im selben Cluster sind, sich dafür aber in Clusterung $\mathcal{C}'$ in unterschiedlichen Clustern befinden. Genauso gilt für S_{01}, dass alle Knotenpaare dieser Menge in unterschiedlichen Clustern in $\mathcal{C}$ und in demselben Cluster in $\mathcal{C}'$ sind. Der sogenannte *Rand-Index* $\mathcal{R}$ nutzt diese Technik. Es gilt:

$$\mathcal{R}(\mathcal{C}, \mathcal{C}') := \frac{|S_{11}| + |S_{00}|}{|S_{11}| + |S_{00}| + |S_{01}| + |S_{10}|} = \frac{|S_{11}| + |S_{00}|}{\binom{n}{2}}$$

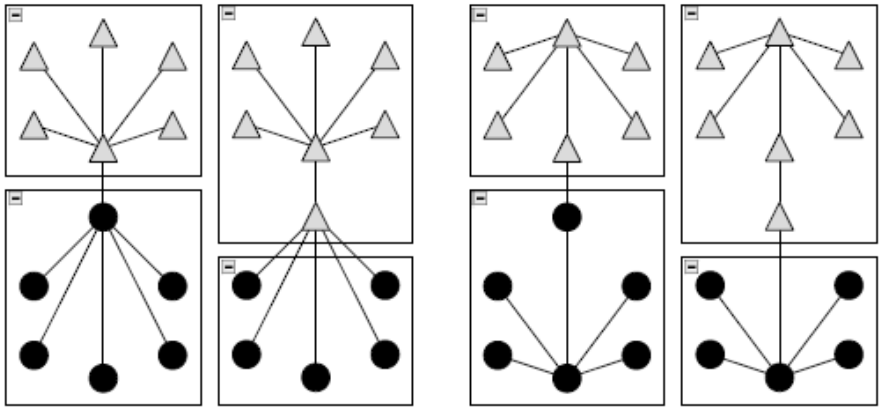
Der Zähler des Rand-Index entspricht den Knotenpaaren, die jeweils in beiden Clusterungen $\mathcal{C}$ und $\mathcal{C}'$ eine Übereinstimmung haben. Dies bedeutet, wenn ein Knotenpaar in $\mathcal{C}$ zum Beispiel im selben Cluster ist, dann ist dieses Paar auch in $\mathcal{C}'$ in einem gemeinsamen Cluster und umgekehrt. Der Nenner dieser Formel entspricht der Anzahl aller möglichen Paare. Der Rand-Index umfasst einen Wertebereich von $[0, 1]$, wobei ein hoher Wert für eine bessere Übereinstimmung der beiden Clusterungen steht. Der große Nachteil des Rand-Index liegt darin, dass die Struktur des Graphen nicht berücksichtigt wird.

Dieser Umstand wird mittels des Beispiels aus der Abbildung 2.3 erläutert. Auf der linken Seite ist Graph G_1 mit zwei Clusterungen $\mathcal{C}_1$ und $\mathcal{C}'_1$ abgebildet. Rechts befindet sich Graph G_2 mit zwei Clusterungen $\mathcal{C}_2$ und $\mathcal{C}'_2$. Beide Graphen haben dieselbe Knotenanzahl, nur die Kanten sind anders angeordnet. Für den Rand-Index gilt:

$$\mathcal{R}(\mathcal{C}_1, \mathcal{C}'_1) = \mathcal{R}(\mathcal{C}_2, \mathcal{C}'_2)$$

Betrachtet man die Qualität der Clusterungen, so ist diese bei $\mathcal{C}_1$ und $\mathcal{C}_2$ identisch, jedoch ist die Qualität von $\mathcal{C}'_1$ schlechter als die von $\mathcal{C}'_2$, da bei $\mathcal{C}'_1$ mehr Kanten zwischen den zwei Clustern verlaufen. Daher wäre es wünschenswert einen Distanz-basierten Vergleich zu erhalten, für den gilt:

$$Distanz(\mathcal{C}_1, \mathcal{C}'_1) < Distanz(\mathcal{C}_2, \mathcal{C}'_2)$$



(a) $\mathcal{C}_1$ (b) $\mathcal{C}'_1$ (c) $\mathcal{C}_2$ (d) $\mathcal{C}'_2$

Abbildung 2.3.: Nachteil des Rand-Index, Quelle: [Gör10]

Damit die Struktur des Graphen mitberücksichtigt wird, wird mit dem *Graph-theoretischen Rand-Index* in [Gör10] eine Alternative zum normalen Rand-Index vorgestellt. Der Graph-theoretische Rand-Index berücksichtigt nur Knotenpaare, die mittels einer Kante in Beziehung stehen. Er besteht aus vier Mengen E_{ij} mit $i, j \in \{0, 1\}$, wobei $E_{ij} := S_{ij} \cap E$ und $e_{ij} := |E_{ij}|$. Formal ergibt sich:

$$\mathcal{R}_g(\mathcal{C}, \mathcal{C}') := \frac{e_{11} + e_{00}}{e_{11} + e_{00} + e_{01} + e_{10}} = \frac{e_{11} + e_{00}}{m}$$

2.4. Dynamische Graphen

Bisher wurden nur statische Graphen betrachtet. In der Praxis jedoch sind Netzwerke nicht notwendigerweise statisch, da sie sich über die Zeit hinweg entwickeln können.

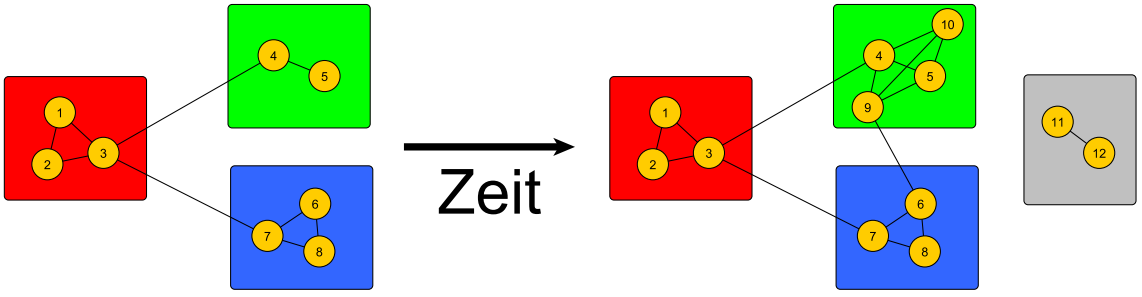


Abbildung 2.4.: Beispiel für einen dynamischen Graphen

Im Beispiel der Abbildung 2.4 ist ein dynamischer Graph zu erkennen. Man nehme an, dieser Graph sei ein kleiner Ausschnitt aus einem sozialen Netzwerk. Die Knotennummern stehen stellvertretend für Personen. Eine Kante zwischen zwei Personen deutet auf eine Freundschaft hin. Stark verbundene Teilgraphen (Freundeskreise) werden dann zu einem Cluster zusammengefasst. Im Beispiel gibt es zunächst drei solcher Cluster, allerdings sind mit der Zeit vier neue Knoten dazugekommen, die neuen Leute im sozialen Netzwerk entsprechen. Knoten 9 und 10 haben schon Freunde in diesem sozialen Netzwerk und kommen somit in den entsprechenden grünen Cluster dieser Freunde. Knoten 9 hat zwar mit Knoten 6 einen Freund im blauen Cluster, allerdings ist die Bindung zum grünen

Cluster stärker, da hier mehr Freunde anwesend sind. Knoten 11 und 12 sind zwei weitere neue Mitglieder im Netzwerk, die allerdings noch keine Freundschaften außer zu sich selbst geknüpft haben. Dieses Beispiel soll die Dynamik von Netzwerken verdeutlichen.

2.4.1. Definition

Ein *dynamischer Graph* $\mathcal{G} = \{G^0, \ldots, G^{t_{max}}\}$ entspricht einer Folge statischer Graphen $G^t = (V^t, E^t, \omega^t)$ (sogenannte *Snapshots*) zu einem jeweiligen Zeitpunkt t mit $0 \leq t \leq t_{max}$. Der Graph G^{t+1} entsteht aus dem Vorgänger G^t, indem eine bestimmte Anzahl an Änderungen an G^t durchgeführt werden. Folgende Änderungen sind möglich:

1. Hinzufügen eines Knotens.
2. Löschen eines Knotens: Alle zu diesem Knoten inzidenten Kanten werden ebenfalls entfernt.
3. Hinzufügen einer Kante mit Gewicht ω.
4. Entfernen einer Kante.
5. Erhöhen bzw. Senken des Kantengewichts.

Mehrere solcher Änderungen werden in Δ zusammengefasst. $\Delta(G^t) = G^{t+1}$, bedeutet dass der Graph G^{t+1} zum Zeitpunkt $t + 1$ dem durch Δ veränderten Graphen G^t aus dem vorherigen Zeitpunkt entspricht. Natürlich kann auch der Fall eintreten, dass sich ein oder mehrere aufeinander folgende Graphen von $\mathcal{G}$ nicht unterscheiden, da keine Änderungen vorgenommen wurden.

2.4.2. Kenntnis der Zeitspanne

Um einen dynamischen Graphen zu clustern, muss zunächst die Ausgangslage bekannt sein. Hierbei gibt es zwei Möglichkeiten:

- *Offline-Fall*: Beim Offline-Fall ist die komplette Zeitspanne des dynamischen Graphen bekannt. Die Lösung der aktuellen Clusterung kann auf vergangenen und zukünftigen Clusterungen und Zuständen des Graphen basieren. Dadurch können globale Ziele festgelegt werden.
- *Online-Fall*: Beim Online-Fall hat man im Gegensatz zum Offline-Fall keine Kenntnisse über zukünftige Ereignisse. Der aktuelle Graph kann nur mit Hilfe vergangener Lösungen geclustert werden.

Der Schwerpunkt dieser Arbeit liegt auf dem Clustern von dynamischen Graphen im Offline-Fall. Dieses Wissen über den kompletten Zeitraum soll genutzt werden, um einen dynamischen Graphen so zu clustern, dass beim Clustern vergangene bzw. zukünftige Ergebnisse mitberücksichtigt werden, damit am Ende die Entwicklung der jeweiligen Cluster deutlich wird. Das Ziel ist es, gute, aber auch ähnliche Clusterungen zu finden.

Das Beispiel der Abbildung 2.5 zeigt die zeitliche Entwicklung eines dynamischen Graphen über mehrere Zeitschritte. Nach welcher Methode der Graph geclustert wurde, spielt hierbei keine Rolle. Dieses Beispiel dient nur zur Verdeutlichung der Entwicklung der Cluster über einen bestimmten Zeitraum hinweg. Zum Zeitpunkt $t = 0$ haben wir drei Cluster. Danach finden Änderungen am Graphen statt, die unter anderem zu einem neuen Cluster (grau dargestellt) in $t = 1$ führen. Dieser graue Cluster ist zwar anfangs noch mit drei Knoten sehr klein, aber über die Zeit hinweg sind immer mehr Knoten zu diesem Cluster dazugekommen. Zum Zeitpunkt $t = 10$ sieht man deutlich, wie sich dieser Cluster in seiner

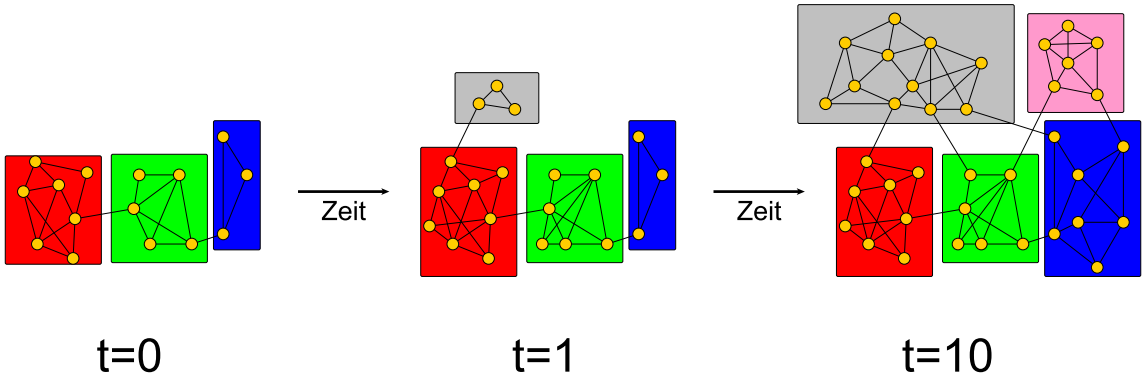


Abbildung 2.5.: Zeitliche Entwicklung eines dynamischen Graphen

Größe entwickelt hat. Genau solche Entwicklungen sollen beim Clustern eines dynamischen Graphen im Offline-Fall verdeutlicht werden.

Im nächsten Kapitel werden Heuristiken zum Clustern statischer Graphen vorgestellt, da diese Heuristiken für die restliche Arbeit und für das Clustern von dynamischen Graphen in Kapitel 5 eine große Rolle spielen.

3. Clustern von statischen Graphen

Die Optimierung von Modularity ist laut [BDH+07] NP-vollständig. Daher wurden einige Heuristiken vorgeschlagen, mit denen man Clusterungen mit hoher Modularity finden kann. Greedy-Heuristiken sind schnell und ermöglichen eine schrittweise Optimierung der Zielfunktion. Eine solche Greedy-Heuristik ist der *Local Greedy Algorithmus*, der im nächsten Abschnitt detailliert beschrieben wird.

3.1. Local Greedy Algorithmus

In [BGLL08] stellen Blondel et al. den Local Greedy Algorithmus vor und zeigen mittels diverser Experimente, dass dieser heuristische Algorithmus schnell und präzise arbeitet. Die Grundidee des Algorithmus ist es, durch lokale Entscheidungen Modularity schrittweise zu verbessern, bis keine Verbesserung mehr erzielt werden kann.

Sei $\mathcal{C}_{u,v}$ die Clusterung, die sich ergibt, wenn Knoten u aus dem Cluster $C(u)$ in den Cluster $C(v)$ verschoben wird. Für die durch diese Verschiebung verursachte Änderung $\Delta mod_\omega(\mathcal{C}, u, v)$ an Modularity ergibt sich dann formal:

$$
\begin{aligned}
&\Delta mod_\omega(\mathcal{C}, u, v) = mod_\omega(\mathcal{C}_{u,v}) - mod_\omega(\mathcal{C}) \\
&= \left[\frac{\omega(\mathcal{C}_{u,v})}{W} - \frac{1}{4W^2}\sum_{C\in\mathcal{C}_{u,v}}\left(\sum_{a\in C} deg_\omega(a)\right)^2\right] \quad - \quad \left[\frac{\omega(\mathcal{C})}{W} - \frac{1}{4W^2}\sum_{C\in\mathcal{C}}\left(\sum_{a\in C} deg_\omega(a)\right)^2\right] \\
&= \sum_{C\in\mathcal{C}_{u,v}}\left[\frac{\omega(C)}{W} - \left(\frac{\sum_{a\in C} deg_\omega(a)}{2W}\right)^2\right] \quad - \quad \sum_{C\in\mathcal{C}}\left[\frac{\omega(C)}{W} - \left(\frac{\sum_{a\in C} deg_\omega(a)}{2W}\right)^2\right]
\end{aligned}
$$

Setzt man für $C_{u,v}$ die Definition ein, ergibt sich für $\Delta mod_\omega(\mathcal{C}, u, v)$:

$$
\begin{aligned}
&= \Bigg[\sum_{C\in\mathcal{C}\setminus\{C(u),C(v)\}}\left[\frac{\omega(C)}{W} - \left(\frac{\sum_{a\in C} deg_\omega(a)}{2W}\right)^2\right] + \frac{\omega(C(u)\setminus\{u\})}{W} \\
&\quad - \left(\frac{\sum_{a\in C(u)\setminus\{u\}} deg_\omega(a)}{2W}\right)^2 + \frac{\omega(C(v)\cup\{u\})}{W} - \left(\frac{\sum_{a\in C(v)\cup\{u\}} deg_\omega(a)}{2W}\right)^2\Bigg]
\end{aligned}
$$

$$
\begin{aligned}
&- \left[\sum_{C \in \mathcal{C} \setminus \{C(u), C(v)\}} \left[\frac{\omega(C)}{W} - \left(\frac{\sum_{a \in C} deg_\omega(a)}{2W} \right)^2 \right] + \frac{\omega(C(u))}{W} \right. \\
&\left. - \left(\frac{\sum_{a \in C(u)} deg_\omega(a)}{2W} \right)^2 + \frac{\omega(C(v))}{W} - \left(\frac{\sum_{a \in C(v)} deg_\omega(a)}{2W} \right)^2 \right] \\
=&\frac{\omega(C(u) \setminus \{u\})}{W} - \left(\frac{\sum_{a \in C(u) \setminus \{u\}} deg_\omega(a)}{2W} \right)^2 + \frac{\omega(C(v) \cup \{u\})}{W} - \left(\frac{\sum_{a \in C(v) \cup \{u\}} deg_\omega(a)}{2W} \right)^2 \\
&- \left[\frac{\omega(C(u))}{W} - \left(\frac{\sum_{a \in C(u)} deg_\omega(a)}{2W} \right)^2 + \frac{\omega(C(v))}{W} - \left(\frac{\sum_{a \in C(v)} deg_\omega(a)}{2W} \right)^2 \right]
\end{aligned}
$$

Dabei ist $\omega(C)$ als die Summe der Gewichte der Intracluster-Kanten des Clusters C definiert.

Algorithm 1 localClustering$(G, \mathcal{C})$

Input: $G = (V, E, \omega), \mathcal{C}$
1: **repeat**
2: **for all** $u \in V$ **do**
3: find neighbor v with maximum $\Delta mod_\omega(\mathcal{C}, u, v)$
4: **if** $\Delta mod_\omega(\mathcal{C}, u, v) > 0$ **then**
5: $\mathcal{C} \leftarrow \text{move}(u, C(v))$
6: **end if**
7: **end for**
8: **until** no real movement
Output: $\mathcal{C}$

Algorithmus 1 betrachtet alle Knoten in zufälliger Reihenfolge und verschiebt diese jeweils in den Cluster des Nachbarknotens, mit dem der höchste positive Zuwachs an Modularity erreicht werden kann. Diese Knotenbewegungen werden so lange durchgeführt, bis es in einer Iteration durch alle Knoten keine einzige Bewegung mehr gibt, da für alle Knoten u und v $\Delta mod_\omega(\mathcal{C}, u, v) \leq 0$ gilt und somit eine Verbesserung an Modularity mit lokalen Verschiebungen nicht mehr möglich ist.

Algorithm 2 localGreedy(G_0)

Input: $G_0 = (V_0, E_0, \omega_0)$
1: $\mathcal{C}_0 \leftarrow \{C_i = \{v_i\} | v_i \in V_0, 1 \leq i \leq n\}$
2: $l \leftarrow 0$
3: **repeat**
4: $\mathcal{C}_l \leftarrow \text{localClustering}(G_l, \mathcal{C}_l)$
5: **if** $\exists C \in \mathcal{C}_l : |C| > 1$ **then**
6: $(G_{l+1}, \mathcal{C}_{l+1}) \leftarrow \text{contract}(G_l, \mathcal{C}_l)$
7: $l \leftarrow l + 1$
8: **end if**
9: **until** no contraction
10: $\mathcal{C}_0 \leftarrow \text{unfurl}(\mathcal{C}_l)$
Output: $\mathcal{C}_0$

Algorithmus 2 stellt den Pseudocode des Local Greedy Algorithmus dar. Zunächst wird die Clusterung $\mathcal{C}_0$ des Ausgangsgraphen G_0 so initialisiert, dass jeder Knoten ein eigenen

Cluster bildet. Nach Abschluss der Initialisierung wird Algorithmus 1 mit dem aktuell betrachteten Graphen und dessen anfänglicher Clusterung aufgerufen. Wenn keine Knotenbewegung mehr möglich ist und somit die erste Phase des Local Greedy Algorithmus beendet ist, werden der aktuelle Graph G_l und dessen Clusterung $\mathcal{C}_l$ in der zweiten Phase von Algorithmus 2 kontrahiert. Die Variable l steht für das *Abstraktionslevel*, welches bei jeder Kontraktion des Graphen erhöht wird.

Eine Kontraktion des Graphen G_l zu einem Graphen G_{l+1} wird dadurch erreicht, dass alle Knoten aus einem Cluster zu einem einzigen Knoten zusammengefasst werden. Jeder durch die Kontraktion entstandene Knoten erhält eine Schleife, die als Gewicht die Summe der Gewichte aller Intracluster-Kanten des entsprechenden Clusters hat. Zudem werden für alle Cluster C_i, C_j mit $i \neq j$ die Kanten aus $E(C_i, C_j)$ zu einer einzigen Kante zusammengefasst. Diese Kante besitzt als Gewicht die Summe der Gewichte der Kanten aus $E(C_i, C_j)$. Die neue Clusterung $\mathcal{C}_{l+1}$ wird dann nach dem selben Muster wie die Clusterung $\mathcal{C}_0$ initialisiert. Das bedeutet, dass jeder Cluster $C \in \mathcal{C}_{l+1}$ aus nur einem einzigen Knoten $v \in V_{l+1}$ besteht.

Nach Abschluss der Kontraktion beginnt der Algorithmus wieder mit der ersten Phase. Diese Vorgehensweise wird so lange wiederholt, bis sich auf einem Level die Clusterung nicht mehr ändert, da eine Verbesserung an Modularity nicht mehr möglich ist und somit keine Kontraktion mehr benötigt wird. Sei p die Anzahl der Kontraktionen. Die Funktion `unfurl` erhält die zuletzt gefundene Clusterung des Graphen G_l mit $l = p$ und überträgt diese Clusterung vom letzten Level p auf das Level $p - 1$, wenn $p > 0$ ist. Dazu muss sich der Local Greedy Algorithmus zuvor gemerkt haben, welche Knoten aus Level p zu welchen Knoten aus Level $p - 1$ gehören. Dies wird so lange Level für Level gemacht, bis man mit Level 0 den Ausgangsgraphen erreicht hat und somit eine Clusterung von diesem erhält.

Nach jeder Kontraktion erhält man eine Vergröberung des Ausgangsgraphen. Ein beispielhafter Ablauf dieser Vergröberung ist in Abbildung 3.1 zu finden. Das Beispiel dient nur zur Verdeutlichung der unterschiedlichen Level, deshalb sind die Kanten des Graphen für eine bessere Übersicht weggelassen worden. In Level 0 wird das Clustering des Ausgangsgraphen G_0 vorgenommen. Hier werden so lange Knotenbewegungen anhand des Local Greedy Algorithmus durchgeführt, bis alle Knoten mit ihrem Cluster zufrieden sind. Das Ergebnis der Clusterung für das jeweilige Level ist in Abbildung 3.1 mit den Kästen gekennzeichnet. Alle Knoten, die im selben Cluster sind, werden zu einem einzigen Knoten zusammengefasst. Diese kontrahierten Knoten bilden dann die Knoten des Graphen G_1 im nächsten Level. Diese Prozedur wird so lange weitergeführt, bis es keiner Kontraktion mehr bedarf. Im Beispiel tritt dieser Fall in Level 3 auf. Hier will jeder Knoten in seinem eigenen Cluster bleiben und da alle Cluster nur aus einem einzigen Knoten bestehen, wird keine Kontraktion mehr durchgeführt. Man hat als Endresultat zwei Cluster, die zur Unterscheidung jeweils rot und grün gefärbt werden. Die Clusterung des höchsten Levels wird dann auf das unterste Level projiziert.

Die Reihenfolge, in der die Knoten in einer Iteration des Local Greedy Algorithmus durchlaufen werden, hat laut [BGLL08] keinen bedeutsamen Einfluss auf die erreichte Modularity, sondern eher auf die Laufzeit. In dieser Arbeit werden, wenn nicht anders erwähnt, die Knoten nach aufsteigender ID betrachtet. Statt die Clusterung auf den höchsten Level einfach auf das unterste Level zu projizieren, können auch noch Verfeinerungsschritte auf den verschiedenen Leveln der Hierarchie erfolgen. Dies nennt man *Multi-Level-Clustering*.

3.2. Multi-Level-Clustering

Multi-Level-Algorithmen werden schon lange beim verwandten Problem der Graphpartitionierung benutzt. Noack und Rotta stellen in [NR09] verschiedene Strategien vor, um

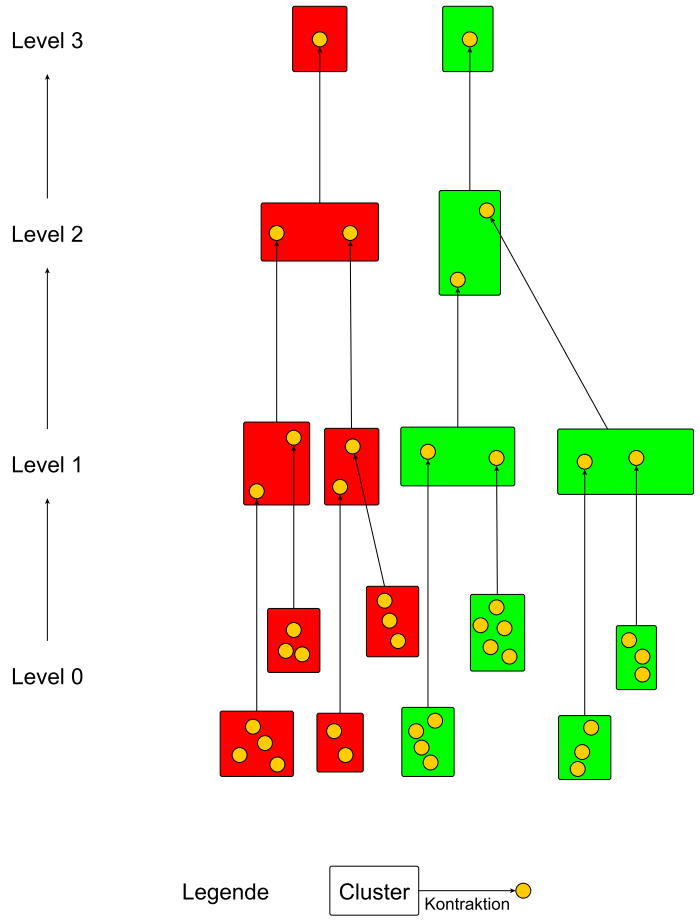


Abbildung 3.1.: Vergröberung eines Graphen G mittels Local Greedy Algorithmus

Graphen Modularity-basiert zu clustern, darunter auch ein Multi-Level-Ansatz. Mittels diverser Versuche wurde festgestellt, dass deren Ansatz mit den bisher besten Algorithmen mithalten kann.

Die Idee des Multi-Level-Clusterings ist es, zunächst mittels Local Greedy Algorithmus eine Clusterung mit einem guten Wert an Modularity zu finden und dann die durch insgesamt p Kontraktionen entstandenen Level des Graphen nochmals in umgekehrter Reihenfolge zu betrachten, um Verfeinerungsschritte vorzunehmen. Dies wird dadurch realisiert, dass in jedem Level durch erneute Knotenbewegungen eine Verbesserung an Modularity angestrebt wird.

Die Verfeinerung startet mit der Clusterung des letzten Levels p. In Level p hat man jetzt immer noch die selbe Ausgangslage wie zuvor, daher sind hier keine Änderungen zu erwarten. Deswegen kann man direkt diese Clusterung auf das Level $p-1$ übertragen. Ab dem Level $p-2$ kann es zu Änderungen der Clusterung kommen. Ab diesem Level wird der Graph auf mögliche Verbesserungen an Modularity mittels Algorithmus 1 untersucht. Sobald keine Knotenbewegung mehr durchführbar ist, wird die aktuelle Clusterung auf das niedrigere Level übertragen. Im Beispiel der Abbildung 3.2 wird so eine Verfeinerung basierend auf den Ergebnissen von Abbildung 3.1 vorgenommen. Die Verfeinerung beginnt in Level 3. Da aber in den obersten zwei Leveln keine Änderungen stattfinden, wird die Clusterung aus Level 3 direkt auf das Level 2 und von dort aus auf das Level 1 übertragen. Erst in diesem Level kann es zu Änderungen kommen. Im Beispiel gibt es einen Knoten, der vom roten in den grünen Cluster verschoben wird. Dies ist auch die einzige Änderung, die an der Clusterung durchgeführt wird. Die durch die Änderung angepasste Clusterung wird

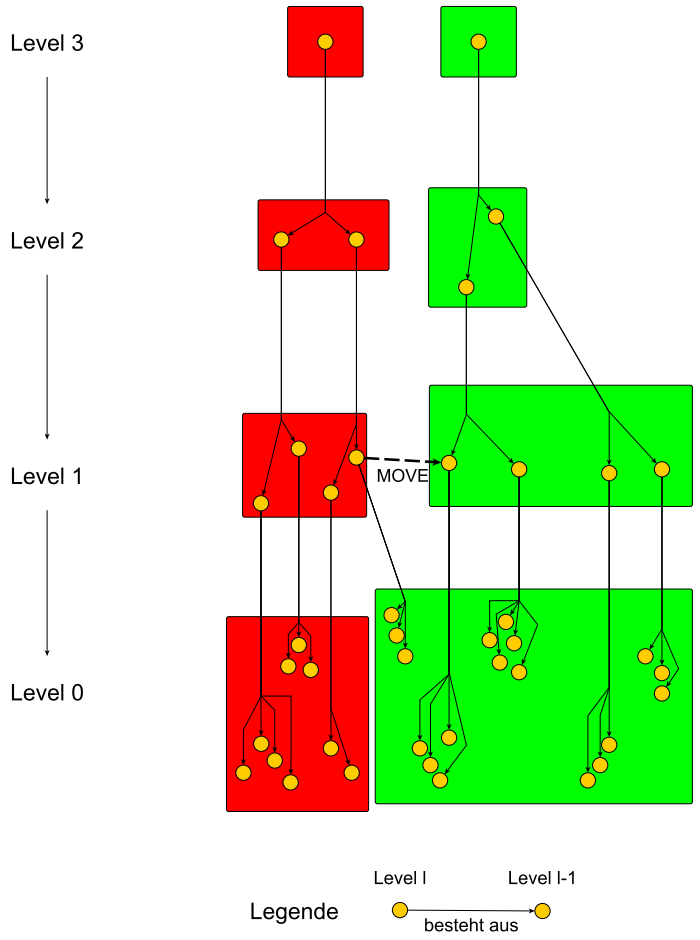


Abbildung 3.2.: Verfeinerung von G aus Abbildung 3.1

dann in das Level 0 übermittelt, wo ein letztes Mal untersucht wird, ob eine Steigerung an Modularity durch Knotenbewegungen möglich ist. Im Beispiel der Abbildung 3.2 ist dies aber nicht der Fall und man erhält somit die endgültige Clusterung des Graphen.

Algorithmus 3 stellt den ganzen Vorgang des Multi-Level-Clusterings in Pseudocode dar. Zeile 3 bis 9 umfasst die Vergröberung und in Zeile 11 bis 17 wird die Verfeinerung durchgeführt.

Da beim Multi-Level-Clustering die Knotenbewegungen in unterschiedlichen Leveln stattfinden, kann die Cluster Zugehörigkeit von mehreren Knoten des Ausgangsgraphen G_0 gleichzeitig geändert werden, da ein Knoten in Level $l \neq 0$ aus einem oder mehreren Knoten aus Level $l - 1$ gebildet wird und alle Änderungen in einem Level indirekt die niedrigeren Level beeinflussen. Eine andere Art, eine Verfeinerung durchzuführen, ist mit den sogenannten Single-Level-Algorithmen (Beispiele in [NR09]) möglich. Diese Algorithmen übertragen die Clusterung des letzten Levels p auf das Level 0. Sie operieren nur in diesem Level 0, weshalb ausschließlich eine Verschiebung einzelner Knoten des Ausgangsgraphen möglich ist.

Algorithm 3 multiLevelClustering(G_0)

Input: $G_0 = (V_0, E_0, \omega_0)$

$\mathcal{C}_0 \leftarrow \{C_i = \{v_i\} | v_i \in V_0, 1 \leq i \leq n\}$
$l \leftarrow 0$
repeat
  $\mathcal{C}_l \leftarrow$ localClustering$(G_l, \mathcal{C}_l)$
  if $\exists C \in \mathcal{C}_l : |C| > 1$ **then**
    $(G_{l+1}, \mathcal{C}_{l+1}) \leftarrow$ contract$(G_l, \mathcal{C}_l)$
    $l \leftarrow l + 1$
  end if
until no contraction
repeat
  $\mathcal{C}_{l-1} \leftarrow$ unfurl$(\mathcal{C}_l)$
  $l \leftarrow l - 1$
  $\mathcal{C}_l \leftarrow$ localClustering$(G_l, \mathcal{C}_l)$
until $l = 0$

Output: $\mathcal{C}_0$

4. Grundlegende Algorithmen und Problemstellung

Für einen dynamischen Graphen $\mathcal{G} = \{G^0, \ldots, G^{t_{max}}\}$ werden im Offline-Fall Clusterungen für jeden Zeitpunkt gesucht. Wird ein dynamischer Graph beispielsweise nur nach Modularity geclustert, so erhält man für jeden Zeitpunkt zwar eine gute Clusterung, jedoch könnte auch der Fall eintreten, dass sich die Clusterungen der unterschiedlichen Snapshots deutlich voneinander unterscheiden. Dies liegt daran, dass Modularity viele lokale Optima besitzt, die ungefähr gleich gut sind. Dies ist natürlich nicht ideal, wenn man die Entwicklung eines Netzwerks verfolgen möchte.

Ziel dieser Arbeit ist es, zu einem gegebenen dynamischen Graphen gute, aber auch ähnliche Clusterungen für jeden Zeitpunkt zu finden. In der Literatur sind für diese Problematik bereits Lösungsansätze vorhanden.

4.1. Zeitexpandiertes Clustern

Ein Lösungsansatz ist das *zeitexpandierte Clustern* [Gla08] von D. Glaser. Beim zeitexpandierten Clustern wird zunächst ein dynamischer Graph so modifiziert, dass Beziehungen zwischen den Zeitpunkten mitberücksichtigt werden. Dies wird durch Einfügen zusätzlicher Kanten realisiert.

Die Abbildung 4.1 zeigt, wie aus einem herkömmlichen dynamischen Graphen ein zeitexpandierter Graph entsteht. In diesem trivialen Beispiel hat der dynamische Graph nur zwei Zeitpunkte. Wenn nun ein Knoten v zu beiden Zeitpunkten vertreten ist, wird eine Kante $\{v^0, v^1\}$ von diesem Knoten zum Zeitpunkt $t = 0$ zum selben Knoten zum Zeitpunkt $t = 1$ hinzugefügt. Diese sogenannte *intertemporäre Kante* verdeutlicht, dass es sich um ein und denselben Knoten handelt, aber zu unterschiedlichen Zeitpunkten. Diese Kanten sollen bewirken, dass derselbe Knoten zu unterschiedlichen Zeitpunkten im selben Cluster sein soll. Das Gewicht dieser Kanten und die Anzahl der Zeitpunkte beeinflussen diesen Effekt. Gäbe es mehr Zeitpunkte als nur zwei wie in diesem Beispiel, so würde diese Prozedur bis zum letzten Zeitpunkt t_{max} auf dieselbe Art und Weise weitergeführt werden (Kanten: $\{v^0, v^1\}, \{v^1, v^2\}, \ldots, \{v^{t_{max}-1}, v^{t_{max}}\}$). Es ist auch möglich mehr als nur einen Zeitschritt zu berücksichtigen. Bei zwei Zeitschritten gäbe es dann maximal zwei zusätzliche Kanten für jeden Knoten v zu einem Zeitpunkt t: $\{v^t, v^{t+1}\}$ und $\{v^t, v^{t+2}\}$. Das Gewicht einer intertemporären Kante wird mit Hilfe der sogenannten *Cosine-Similarity* ermittelt.

Für $G = (V, E, \omega)$ sei $\vec{w^v} = \begin{pmatrix} w_1^v \\ w_2^v \\ \dots \\ w_n^v \end{pmatrix}$ der Vektor mit den Abständen von Knoten v zu allen anderen n Knoten des Graphen. Für den Abstand gilt:

$$w_u^v = \begin{cases} \omega(\{v,u\}) & \text{falls } \{v,u\} \in E \\ 0 & \text{falls } u = v \\ 0 & \text{sonst} \end{cases}$$

Dann gilt für die Cosine-Similarity eines Knotens v zu den Zeitpunkten t und $t+1$:

$$sim(\vec{w^{v^t}}, \vec{w^{v^{t+1}}}) = \frac{\sum_{i=1}^{n}(w_i^{v^t} \cdot w_i^{v^{t+1}})}{\sqrt{\sum_{i=1}^{n}(w_i^{v^t})^2 \cdot \sum_{i=1}^{n}(w_i^{v^{t+1}})^2}}$$

Da es durchaus vorkommen kann, dass sich die Anzahl der Knoten in aufeinander folgenden Zeitpunkten unterscheidet, gilt für $n = maximum\{|V^t|, |V^{t+1}|\}$. Mit Hilfe der Cosine-Similarity lässt sich das Kantengewicht der intertemporären Kanten bestimmen. Dieses beträgt $sim(\vec{w^{v^t}}, \vec{w^{v^{t+1}}}) \cdot x$. Die Variable x steht für das durchschnittliche Kantengewicht der jeweiligen Snapshots. Im Falle eines eines dynamischen Graphen, bei dem jede Kante ein Gewicht von 1 besitzt, ist $x = 1$. Betrachtet man aber einen Graphen mit unterschiedlichen Gewichten, so muss das Gewicht der intertemporären Kanten normiert werden, damit es mit den restlichen Kanten von gleicher Bedeutung ist. Grund für die Normierung ist, dass $sim(\vec{w^{v^t}}, \vec{w^{v^{t+1}}})$ Werte im Bereich $[0, 1]$ liefert. Hätte man nun ein durchschnittliches Kantengewicht in den Snapshots von 50 und man würde keine Normierung der intertemporären Kanten vornehmen, so wären diese beim Clustern des zeitexpandierten Graphen unwichtiger als die normalen Kanten. Nachdem man aus einem dynamischen Graphen einen zeitexpandierten Graphen erstellt hat, kann man diesen wiederum ganz normal mit den Methoden clustern, die für statische Graphen ausgelegt sind.

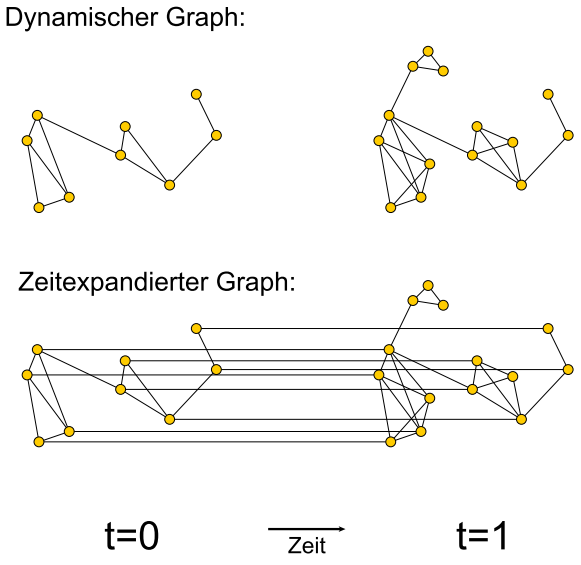


Abbildung 4.1.: Beispiel für einen zeitexpandierten Graphen

In [GMS+11] wird ein anderer Lösungsansatz zum Finden von guten und ähnlichen Clusterungen für dynamische Graphen vorgestellt. Modularity und Graph-theoretischer Rand-Index werden zu einer gemeinsamen Zielfunktion kombiniert. Mit Hilfe des Local Greedy Algorithmus und dieser neuen Zielfunktion werden Clusterungen für dynamische Graphen gesucht. In Kapitel 5.2 wird darauf aufbauend eine Heuristik vorgestellt, die Multi-Level-Clustering mit dieser neuen Zielfunktion kombiniert. Die Kombination von Modularity und Graph-theoretischem Rand-Index wird im Folgenden als *bikriteriell* bezeichnet.

4.2. Kombination von Modularity und Graph-theoretischem Rand-Index

Der Graph-theoretische Rand-Index vergleicht normalerweise zwei unterschiedliche Clusterungen auf Basis desselben Graphen. Bei einem dynamischen Graphen sind in der Regel aber die Knoten- und Kantenmengen seiner Snapshots verschieden. Daher betrachtet hier der Graph-theoretische Rand-Index den Schnitt der Kanten $E'' = E \cap E'$ zweier unterschiedlicher Graphen. Es werden nur Knotenpaare berücksichtigt, die in beiden Graphen G und G' durch eine Kante verbunden sind. Dabei sei e_{00} die Anzahl der Knotenpaare, die in beiden Graphen G und G' in unterschiedlichen Clustern sind und e_{11} steht für die Anzahl der Paare, die in beiden Graphen jeweils in ein einem Cluster sind. Die Formel für die Kombination von Modularity und Graph-theoretischem Rand-Index lautet:

$$\begin{aligned} TD_\alpha(\mathcal{C}') :=& \alpha \cdot mod_\omega(\mathcal{C}') + (1-\alpha) \cdot \mathcal{R}_g(\mathcal{C}, \mathcal{C}') \\ =& \alpha \cdot \left[\frac{\omega(\mathcal{C}')}{W} - \frac{1}{4W^2} \sum_{C \in \mathcal{C}'} \left(\sum_{v \in C} deg_\omega(v) \right)^2 \right] + (1-\alpha) \cdot \frac{e_{11} + e_{00}}{|E''|} \end{aligned}$$

Der Parameter $\alpha \in [0, 1]$ regelt dabei die Stärke der Kriterien. Je höher α ist, desto stärker wird Modularity gewichtet. Die Clusterung $\mathcal{C}'$ ist die Clusterung des aktuell betrachteten Graphen $G' = (V', E', \omega')$ und $\mathcal{C}$ ist die Clusterung des zuvor geclusterten Graphen $G = (V, E, \omega)$.

In Abbildung 4.2 wird die Berechnung des Graph-theoretischen Rand-Index auf Basis unterschiedlicher Graphen veranschaulicht. Der Ausgangsgraph G besitzt die in der Abbildung oben dargestellte endgültige Clusterung mit zwei Clustern, die jeweils aus drei Knoten bestehen. Der Graph G' unterscheidet sich von seinem Vorgänger G darin, dass der Knoten 7 mit zwei Kanten neu hinzugekommen ist. Diese Änderungen sind grau gekennzeichnet. Dieser Graph hat die in Abbildung 4.2 in der Mitte dargestellte Clusterung. Mit dieser Clusterung findet nun durch die Berechnung von $\mathcal{R}_g(\mathcal{C}, \mathcal{C}')$ ein Vergleich zu der Clusterung von G statt. Beide Graphen haben acht gemeinsame Kanten, daher gilt $|E''| = 8$. Vier durch diese Kanten verbundene Knotenpaare sind in beiden Graphen im selben Cluster ($\{1,3\}$, $\{4,5\}$, $\{4,6\}$ und $\{5,6\}$), weswegen $e_{11} = 4$ ist. Ein Knotenpaar ist in beiden Clusterungen in unterschiedlichen Clustern ($\{3,6\}$), also ist $e_{00} = 1$ und man erhält somit $\mathcal{R}_g(\mathcal{C}, \mathcal{C}') = \frac{4+1}{8} = 0.625$.

Angenommen, man hätte irgendeinen Algorithmus zum Modularity-basierten Clustern, der Knoten 3 vom grünen in den roten Cluster verschiebt, siehe Abbildung 4.2 unten. Dies führt dazu, dass man nur noch insgesamt vier gute Knotenpaare für den Graph-theoretischen Rand-Index besitzt ($\{3,6\}$, $\{1,3\}$ fallen weg, dafür kommt $\{2,3\}$ dazu) und somit einen Graph-theoretischen Rand-Index von 0.5 erreicht. Das Verschieben des Knotens 3 würde sich eventuell, je nachdem welches Kantengewicht verwendet wird, aus Sicht von Modularity lohnen, allerdings nicht aus Sicht des Graph-theoretischen Rand Index, da

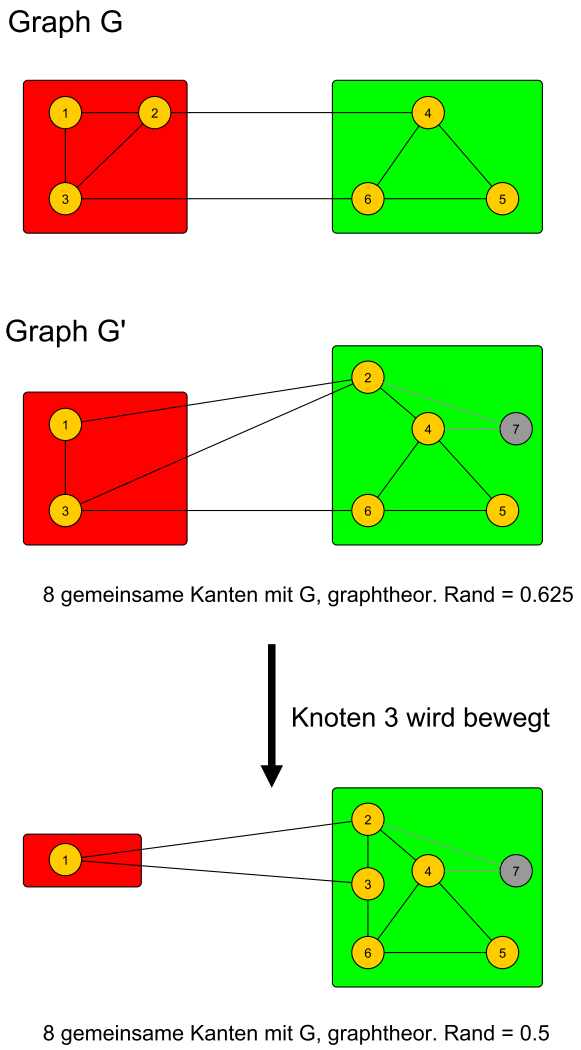


Abbildung 4.2.: Beispiel: Berechnung des Graph-theoretischen Rand-Index

dort eine Verschlechterung von 0.125 zustande kommt. Betrachtet man stattdessen beide Kriterien bei der Verschiebung eines Knotens u in den Cluster von Knoten v, ergibt sich formal eine Änderung an $TD_\alpha(\mathcal{C}')$:

$$\Delta TD_\alpha(\mathcal{C}') = \alpha \cdot \Delta mod_\omega(\mathcal{C}', u, v) + (1 - \alpha) \cdot \Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v)$$

$\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v)$ lässt sich auf ähnliche Art und Weise berechnen wie $\Delta mod_\omega(\mathcal{C}', u, v)$ in Kapitel 3.1. Es gilt bei einer Verschiebung von Knoten u in den Cluster von v in der Clusterung $\mathcal{C}'$ für die Änderung des Graph-theoretischen Rand-Index:

$$\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v) = \mathcal{R}_g(\mathcal{C}, \mathcal{C}'_{u,v}) - \mathcal{R}_g(\mathcal{C}, \mathcal{C}')$$

Wir definieren nun δ_{uv} für G. Es gilt:

$$\delta_{uv} = \begin{cases} 1 & \text{wenn } C(u) = C(v) \\ 0 & \text{sonst} \end{cases}$$

Analog dazu wird δ'_{uv} für G' definiert. In $\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v)$ gilt für u und v, dass beide Knoten zunächst in der Clusterung $\mathcal{C}'$ nicht im selben Cluster sind und somit $\delta'_{uv} = 0$ gilt. Erst nach der Verschiebung sind diese zwei Knoten im gleichen Cluster, was $\delta'_{uv} = 1$ zur Folge hat. Ebenfalls gilt:

- für Clusterung $\mathcal{C}'$: $u \in C(u), u \notin C(v)$ und somit ist $\delta'_{uw} = 0$ für $w \in C(v)$ und $\delta'_{uz} = 1$ für $z \in C(u)$
- für Clusterung $\mathcal{C}'_{u,v}$: $u \notin C(u), u \in C(v)$ und somit ist $\delta'_{uw} = 1$ für $w \in C(v)$ und $\delta'_{uz} = 0$ für $z \in C(u)$

Mit Hilfe von δ_{uv} und δ'_{uv} lässt sich die Formel für den Graph-theoretischen Rand-Index umschreiben:

$$\begin{aligned}\mathcal{R}_g(\mathcal{C}, \mathcal{C}') &= \frac{1}{|E''|} \sum_{\{u,v\} \in E''} (\delta_{uv}\delta'_{uv} + (1 - \delta_{uv})(1 - \delta'_{uv})) \\ &= \frac{1}{|E''|} \sum_{\{u,v\} \in E''} (\delta_{uv}\delta'_{uv} + 1 - \delta_{uv} - \delta'_{uv} + \delta_{uv}\delta'_{uv}) \\ &= \frac{1}{|E''|} \sum_{\{u,v\} \in E''} (1 - \delta_{uv} + \delta'_{uv}(2\delta_{uv} - 1))\end{aligned}$$

Wenn u und v in beiden Clusterungen $\mathcal{C}$ und $\mathcal{C}'$ im selben Cluster sind, also $C(u) = C(v)$ in $\mathcal{C}$ und $\mathcal{C}'$, dann gilt $\delta_{uv}\delta'_{uv} = 1$. Analog dazu ist $(1 - \delta_{uv})(1 - \delta'_{uv}) = 1$, wenn u und v in beiden Clusterungen in unterschiedlichen Clustern sind. Setzt man all diese Erkenntnisse in $\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v)$ ein, erhält man:

$$\begin{aligned}&\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v) = \mathcal{R}_g(\mathcal{C}, \mathcal{C}'_{u,v}) - \mathcal{R}_g(\mathcal{C}, \mathcal{C}') \\ =&\frac{1}{|E''|}\left[\sum_{\substack{\{u,w\} \in E'' \\ w \in C(v)}} \delta_{uw} + \sum_{\substack{\{u,z\} \in E'' \\ z \in C(u)}} (1 - \delta_{uz})\right] - \frac{1}{|E''|}\left[\sum_{\substack{\{u,w\} \in E'' \\ w \in C(v)}} (1 - \delta_{uw}) + \sum_{\substack{\{u,z\} \in E'' \\ z \in C(u)}} \delta_{uz}\right] \\ =&\frac{1}{|E''|}\left[\sum_{\substack{\{u,w\} \in E'' \\ w \in C(v)}} (2\delta_{uw} - 1) - \sum_{\substack{\{u,z\} \in E'' \\ z \in C(u)}} (2\delta_{uz} - 1)\right]\end{aligned}$$

In dieser Formel findet man den Term $2\delta_{uv} - 1$, der eine wichtige Rolle für die Implementierung spielt. Dieser Term wird als zusätzliches zweites Kantengewicht genutzt. Das bedeutet, wenn zwei Knoten u und v im vorherigen Graphen, mit dem man eine Ähnlichkeit der Clusterung erreichen möchte, im selben Cluster sind ($\delta_{uv} = 1$), so erhält man für $2\delta_{uv} - 1$ ein Kantengewicht von 1. Sind u und v zuvor in unterschiedlichen Clustern gewesen, hat dies ein Kantengewicht von -1 zur Folge. Diese Gewichte erhalten alle Kanten aus E'' in G'. Die restlichen Kanten von G' sind für den Graph-theoretischen Rand-Index nicht von Bedeutung und können daher mit einem zweiten Kantengewicht von 0 ignoriert werden.

In Beispiel der Abbildung 4.3 wird der Graph-theoretische Rand-Index als alleiniges Kantengewicht benutzt. Das Beispiel dient zur Verdeutlichung der Clusterauswahl, wenn man

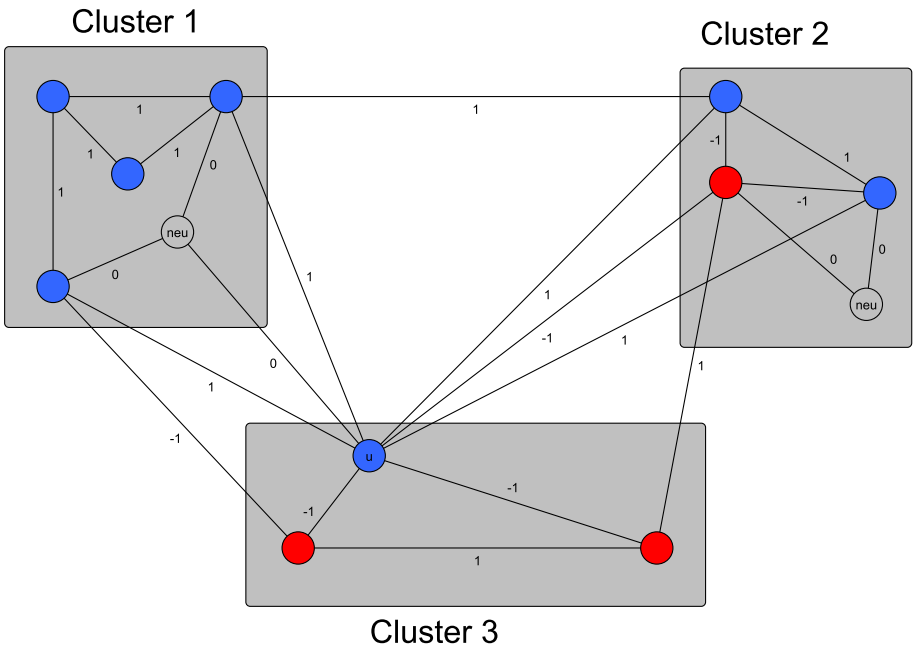


Abbildung 4.3.: Auswahl eines Clusters mittels Graph-theoretischen Rand-Index

einzig allein den Graph-theoretischen Rand-Index als Kriterium einsetzen würde. Die Knoten, die im vorherigen Graphen im selben Cluster gewesen sind, besitzen dieselbe Farbe. Es sind zudem zwei neue Knoten und fünf neue Kanten, die man anhand des Gewichts von 0 erkennt, dazu gekommen.

Folgende Situation trifft nun zu: Man ist im Laufe des Clusteringsvorgangs bei Knoten u angelangt und steht vor der Entscheidung, ob man u in ein vorhandenen Cluster verschiebt oder ob u in seinem eigenen Cluster bleibt, da eine Verschiebung keine Verbesserung einbringt. Eine Verschiebung lohnt sich im Sinne des Graph-theoretischen Rand-Index nur dann, wenn $\Delta\mathcal{R}_g(\mathcal{C}, \mathcal{C}', u, v) > 0$ gilt. Zunächst wird eine Verschiebung zu einem adjazenten Knoten in Cluster 1 betrachtet. Vom Knoten u aus führen insgesamt drei Kanten in den Cluster 1, wovon zwei ein Gewicht von 1 besitzen. Diese zwei Kanten deuten darauf hin, dass der Knoten u im vorherigen Graphen mit den beiden Endpunkten der Kanten im selben Cluster gewesen ist. Die dritte Kante mit dem Gewicht von 0 deutet auf einen neuen Knoten im Cluster 1 hin. Insgesamt erreicht man so bei einer Verschiebung von u eine Verbesserung von $\frac{2}{|E''|}$. Allerdings muss auch berücksichtigt werden, dass der Knoten u den alten Cluster 3 verlässt. Der Knoten u hat insgesamt zwei Kanten mit Endpunkten in Cluster 3. Das Gewicht dieser Kanten geteilt durch $|E''|$ muss von der obigen Verbesserung von $\frac{2}{|E''|}$ abzogen werden. Da beide Kanten ein Gewicht von -1 besitzen, bekommt man insgesamt eine Verbesserung von $\frac{2}{|E''|} - (\frac{-2}{|E''|}) = \frac{4}{|E''|}$, wenn man den Knoten u in den Cluster 1 bewegt. Dieselbe Berechnung wird mit dem Cluster 2 durchgeführt. Dort erreicht man allerdings nur eine Verbesserung von $\frac{3}{|E''|}$, weswegen der Knoten u, wenn man nur nach Graph-theoretischem Rand-Index clustern würde, den Cluster 1 bevorzugen würde.

Mit Hilfe der Kombination von Modularity und Graph-theoretischem Rand-Index lässt sich die Problemstellung dieser Arbeit konkretisieren.

4.3. Formale Problemstellung

Gegeben sei ein dynamischer Graph $\mathcal{G} = \{G^0, \ldots, G^{t_{max}}\}$, für den im Offline-Fall Clusterungen $\mathcal{C}^0, \ldots, \mathcal{C}^{t_{max}}$ für jeden Zeitpunkt $t = 0, \ldots, t_{max}$ gesucht werden. Ein Ziel ist es,

die *Qualität dieser Clusterungen* $Q(\mathcal{C}^0,\ldots,\mathcal{C}^{t_{max}})$ über die komplette Zeitspanne hinweg zu maximieren.

Für die Qualität der gefunden Clusterungen wird im Folgenden der Mittelwert betrachtet. Sei $T \geq 2$ die Anzahl der Zeitpunkte des dynamischen Graphen und $CL := \{\mathcal{C}^0,\ldots,\mathcal{C}^{t_{max}}\}$. Im Falle eines nur nach Modularity geclusterten dynamischen Graphen gilt für die Qualität der Clusterungen:

$$Q_{mod}(T, CL) = \frac{\sum_{t=0}^{T-1} mod_{\omega}(\mathcal{C}^t)}{T}$$

Für die Ähnlichkeit der gefunden Clusterungen wird ebenfalls der Mittelwert betrachtet. Hier finden nur insgesamt $T-1$ solcher Vergleiche statt und daher erhält man für den Mittelwert:

$$Q_{\mathcal{R}_g}(T, CL) = \frac{\sum_{t=1}^{T-1} \mathcal{R}_g(\mathcal{C}^{t-1}, \mathcal{C}^t)}{T-1}$$

Beide Kriterien werden kombiniert zu einer globalen Zielfunktion:

$$Q_{bi}(\alpha, T, CL) = \alpha \cdot Q_{mod}(T, CL) + (1-\alpha) \cdot Q_{\mathcal{R}_g}(T, CL)$$

Bisher wurde der Graph-theoretische Rand-Index eines Snapshots G^t nur mit seinem Vorgänger G^{t-1} berechnet. Natürlich besteht auch die Möglichkeit, den Graph-theoretischen Rand-Index über mehr als einen Zeitschritt hinweg zu berücksichtigen. Dazu wird das sogenannte *Zeitfenster* definiert. Das Zeitfenster drückt aus, über wie viele Zeitschritte hinweg der Graph-theoretische Rand-Index berechnet wird. Im Falle eines Zeitfensters von 1 bedeutet dies, dass nur der Graph zum Zeitpunkt t und dessen Vorgänger zum Zeitpunkt $t-1$ miteinander verglichen werden. Bei einem Zeitfenster von 2 vergleicht man den Graphen zum Zeitpunkt t mit seinem Vorgänger zum Zeitpunkt $t-1$ und mit dem Graphen zum Zeitpunkt $t-2$. Sei $Z \geq 1$ das eingesetzte Zeitfenster, dann gilt für die Qualität der Clusterungen im bikriteriellen Fall:

$$Q_{bi}(\alpha, T, Z, CL) = \alpha \cdot Q_{mod}(T, CL) + (1-\alpha) \cdot \frac{\sum\limits_{\substack{t=1,\ldots,Z \\ x=0,\ldots,t-1}} \mathcal{R}_g(\mathcal{C}^x, \mathcal{C}^t) + \sum\limits_{\substack{t=Z+1,\ldots,T-1 \\ x=T-Z,\ldots,t-1}} \mathcal{R}_g(\mathcal{C}^x, \mathcal{C}^t)}{\sum\limits_{t=1}^{Z} t + \sum\limits_{t=Z+1}^{T-1} Z}$$

Diese Qualitätsfunktion Q_{bi} gilt es zu maximieren. Im nächsten Kapitel werden dafür Heuristiken vorgestellt, die in Kapitel 6 mittels diverser Experimente evaluiert werden.

5. Clustern von dynamischen Graphen im Offline-Fall

In diesem Kapitel werden unterschiedliche Heuristiken zum Clustern eines dynamischen Graphen im Offline-Fall erläutert.

5.1. Statisches Multi-Level-Clustering dynamischer Graphen

Ein dynamischer Graph $\mathcal{G}$ besteht aus mehreren Snapshots $G^0, \ldots, G^{t_{max}}$. Für jeden dieser Snapshots wird das *statische Multi-Level-Clustering* (siehe Algorithmus 3) durchgeführt. Es spielt keine Rolle, mit welchem Snapshot begonnen wird, da jeder Snapshot unabhängig von den anderen Snapshots geclustert wird. Dieser Umstand kann dazu führen, dass sich die jeweiligen Clusterungen deutlich voneinander unterscheiden, da nur nach Modularity geclustert wird. Daher wird im Folgenden eine weitere Heuristik vorgestellt, die die Qualitätsfunktion Modularity und den Distanz-basierten Vergleich durch den Graph-theoretischen Rand-Index kombiniert.

5.2. Timestep Greedy Algorithmus

Beim *Timestep Greedy Algorithmus* wird die in Kapitel 4.1 und 4.2 vorgestellte Idee benutzt, Modularity und den Graph-theoretischen Rand-Index zu kombinieren. Nur der Basisalgorithmus unterscheidet sich darin, dass der Multi-Level Ansatz statt dem reinen Local Greedy Algorithmus benutzt wird.

Gegeben sei ein dynamischer Graph $\mathcal{G}$, der nach beiden Kriterien geclustert werden soll. Begonnen wird mit dem Snapshot zum Zeitpunkt $t = 0$. Dieser erste Snapshot wird nur nach Modularity geclustert, da noch kein Vergleich zu einem Vorgänger möglich ist. Hierfür wird Algorithmus 3 verwendet. Für die Snapshots von $t = 1$ bis $t = t_{max}$ werden dann Clusterungen auf Basis von Modularity und Graph-theoretischem Rand-Index gesucht. Hierfür wird Algorithmus 3 so modifiziert, dass im Algorithmus *localClustering* $\Delta mod_\omega(\mathcal{C}, u, v)$ durch $\Delta TD_\alpha(\mathcal{C})$ ersetzt wird. Dieser veränderte Algorithmus wird als $multiLevelClustering_{TD}$ bezeichnet. Eine Knotenverschiebung findet nur dann statt, wenn $\Delta TD_\alpha(\mathcal{C}') > 0$ gilt.

Algorithmus 4 fasst nochmal den Timestep Greedy Algorithmus zum Clustern eines dynamischen Graphen mit T Zeitpunkten in Pseudocode zusammen. Der Eingabeparameter α regelt die Gewichtung beider Kriterien. Im Falle, dass man den dynamischen Graphen bezüglich der globalen Zielfunktion Q_{bi} clustern möchte, muss eine Normierung von α

Algorithm 4 timestepGreedy($\mathcal{G}, \alpha$)

Input: $\mathcal{G} = \{G_0^t, \ldots, G_0^{t_{max}}\}, \alpha$
1: $t \leftarrow 0$
2: $\mathcal{C}_0^t \leftarrow$ multiLevelClustering(G_0^t)
3: **repeat**
4: $\quad t \leftarrow t + 1$
5: $\quad E'' \leftarrow E_0^t \cap E_0^{t-1}$
6: $\quad G_0^t \leftarrow$ editGraph(G_0^t, G_0^{t-1})
7: $\quad \mathcal{C}_0^t \leftarrow$ multiLevelClustering$_{TD}$(G_0^t, E'', α)
8: **until** $t = t_{max}$
Output: $\{\mathcal{C}_0^0, \ldots, \mathcal{C}_0^{t_{max}}\}$

durchgeführt werden. Die Normierung wird deshalb benötigt, da Modularity T mal berechnet wird, der Graph-theoretische Rand-Index aber nur $T - 1$ mal. Es gilt:

$$\alpha_{norm} = \frac{\alpha \cdot (T-1)}{\alpha \cdot (T-1) + (1-\alpha) \cdot T} = \frac{\alpha \cdot (T-1)}{T - \alpha}$$

Ab dem Zeitpunkt $t = 1$ kommt die Funktion `editGraph` zum Einsatz. Diese Funktion verändert den aktuell betrachteten Graphen so, dass der Graph-theoretische Rand-Index als zusätzliches Kantengewicht verwendet wird. Die Graphen besitzen die Struktur des Beispiels der Abbildung 5.1. Das erste Kantengewicht wird zum Modularity Clustern verwendet. Das zweite Kantengewicht wird für den Graph-theoretischen Rand-Index genutzt. Die Kontraktion wird dann wie gewöhnlich für das erste und zweite Kantengewicht unabhängig voneinander durchgeführt.

Der Timestep Greedy Algorithmus ist auch als Heuristik für den Online-Fall verwendbar. In dieser Arbeit wird es aber für den Offline-Fall benutzt. Die Reihenfolge, mit welchem Snapshot man beim Timestep Greedy Algorithmus anfängt, spielt eine wichtige Rolle, da die Snapshots abhängig voneinander geclustert werden.

5.3. Sequence Greedy Verfeinerung

Mit Hilfe der *Sequence Greedy Verfeinerung* sollen gute Werte für die in Kapitel 4.3 vorgestellte Zielfunktion Q_{bi} für einen dynamischen Graphen gefunden werden. Im Gegensatz zu den bisher vorgestellten Heuristiken wird diese Zielfunktion direkt global optimiert. Durch Änderung an den Clusterungen der unterschiedlichen Zeitpunkte soll Q_{bi} schrittweise verbessert werden. Anstatt eine Maximierung der Zielfunktion zu einem bestimmten Zeitpunkt anzustreben, wird der dynamische Graph bei der Sequence Greedy Verfeinerung als Ganzes betrachtet. Es wird also nicht wie beispielsweise beim Timestep Greedy Algorithmus ein Snapshot nach dem anderen betrachtet, sondern alle Snapshots auf einmal und somit der dynamische Graph in seiner Gesamtheit. Um dies algorithmentechnisch realisieren zu können, müssen alle Snapshots von $\mathcal{G}$ mit ihrer jeweiligen Clusterung bis zum Ende der Sequence Greedy Verfeinerung gespeichert werden. Im Gegensatz zum Timestep Greedy Algorithmus gibt es hier kein zweites Kantengewicht für den Graph-theoretischen Rand-Index. Stattdessen wird für jeden betrachteten Knoten der Graph-theoretische Rand-Index immer neu berechnet. Die Sequence Greedy Verfeinerung arbeitet nur auf dem untersten Abstraktionslevel der jeweiligen Snapshots.

Betrachtet wird nun ein dynamischer Graph mit insgesamt T Zeitpunkten $t_0, \ldots, t_{max}$. Eine Verschiebung des Knotens u in den Cluster von Knoten v zum Zeitpunkt t ergibt für die Verbesserung der Zielfunktion:

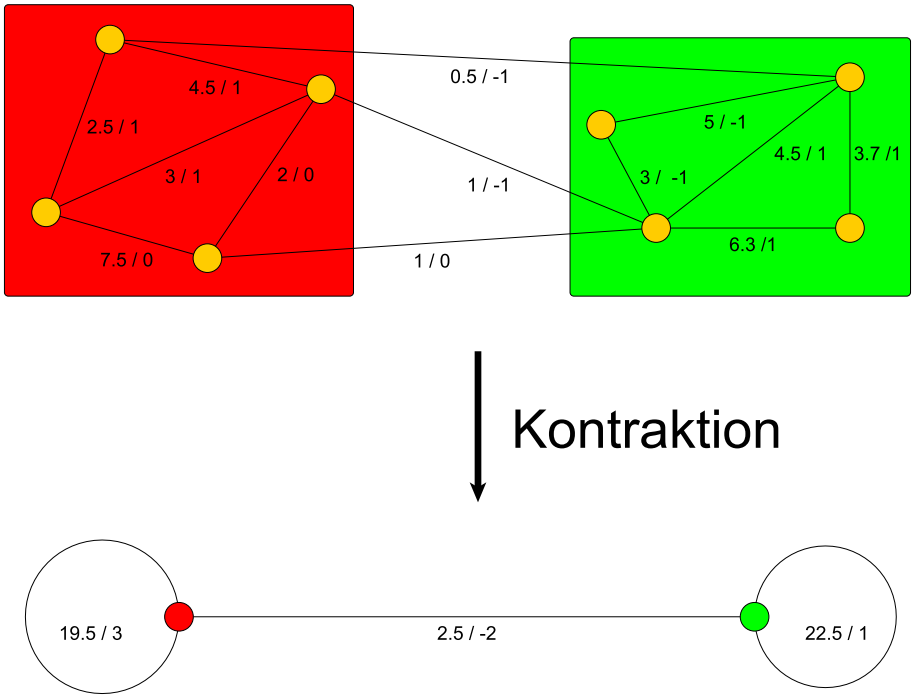


Abbildung 5.1.: Beispiel zur Verdeutlichung der Kontraktion eines Graphen

$$\Delta Q_{bi}(\alpha, T, CL) = \alpha \cdot \frac{\Delta mod_{\omega}(\mathcal{C}^t, u, v)}{T} + (1-\alpha)\frac{\Delta \mathcal{R}_g(\mathcal{C}^{t-1}, \mathcal{C}^t, u, v) + \Delta \mathcal{R}_g(\mathcal{C}^{t+1}, \mathcal{C}^t, u, v)}{T-1}$$

Dabei wird $\Delta \mathcal{R}_g(\mathcal{C}^{t_1}, \mathcal{C}^{t_2}, u, v)$ für $t_{max} < t_1, t_2 < 0$ auf 0 gesetzt. Eine Veränderung an Modularity findet nur am Snapshot G^t statt. Daher wird Δmod_{ω} nur bei diesem Snapshot berechnet. Die Berechnung der Verbesserung des Graph-theoretischen Rand-Index findet, falls vorhanden, mit den Clusterungen des Vorgängers und Nachfolgers von G^t statt.

Algorithmus 5 beschreibt den Vorgang der Sequence Greedy Verfeinerung. Für einen dynamischen Graphen und ein gegebenes α wird zunächst mittels *timestepGreedy* eine Initiallösung der Clusterungen für jeden Zeitschritt gesucht. Als nächstes werden die Knoten aller Snapshots in der Menge A abgelegt. Da ein Knoten zu mehreren Zeitpunkt existieren kann, wird neben der Knoten ID auch der Zeitpunkt t für jeden Knoten der Menge A gespeichert. Diese Menge beinhaltet somit die komplette Knotenmenge des dynamischen Graphen. Der Algorithmus untersucht nun in zufälliger Reihenfolge für alle Knoten von $\mathcal{G}$, ob eine Verbesserung der Zielfunktion erreicht werden kann. Mit Hilfe der Funktion `random_shuffle` [1] wird die zufällige Reihenfolge bestimmt. Diese Reihenfolge ist bei jeder Iteration durch alle Knoten von $\mathcal{G}$ unterschiedlich. Es wird so lange durch alle Knoten iteriert, bis es in einer Iteration keine einzige Bewegung mehr gibt.

5.4. Multi-Level Sequence Greedy Algorithmus

Die Sequence Greedy Verfeinerung arbeitet nur auf dem untersten Abstraktionslevel der jeweiligen Snapshots. Weitere Verbesserungen der Zielfunktion können durchaus möglich sein, wenn man die Sequence Greedy Verfeinerung auf mehrere Abstraktionslevel ausbreitet. Dies wird als *Multi-Level Sequence Greedy Algorithmus* bezeichnet. Die Idee ist es,

[1] `http://www.cplusplus.com/reference/algorithm/random_shuffle/`

Algorithm 5 sequenceGreedy($\mathcal{G}, \alpha$)

Input: $\mathcal{G} = \{G^t, \ldots, G^{t_{max}}\}, \alpha$
1: $A \leftarrow \emptyset$
2: $CL = \{\mathcal{C}^0, \ldots, \mathcal{C}^{t_{max}}\} \leftarrow$ timestepGreedy($\mathcal{G}, \alpha$)
3: **for all** $G^t \in \mathcal{G}$ **do**
4: **for all** $u \in V^t$ **do**
5: $A \leftarrow A \cup \{(u, t)\}$
6: **end for**
7: **end for**
8: **repeat**
9: random_shuffle(A)
10: **for all** $(u, t) \in A$ **do**
11: find neighbor v from u with maximum $\Delta Q_{bi}(\alpha, T, CL)$
12: **if** $\Delta Q_{bi}(\alpha, T, CL) > 0$ **then**
13: $\mathcal{C}^t \leftarrow$ move($u, C(v)$)
14: **end if**
15: **end for**
16: **until** no movement
Output: $\{\mathcal{C}^0, \ldots, \mathcal{C}^{t_{max}}\}$

startend von der Singleton Clusterung für alle Snapshots so lange Änderungen an der Clusterung durchzuführen, bis keine Verbesserung der globalen Zielfunktion Q_{bi} mehr möglich ist. Im Anschluss werden alle Snapshots gleichzeitig kontrahiert und erneut auf dem nächsten Abstraktionslevel auf mögliche Verbesserungen der Zielfunktion durch Knotenverschiebungen untersucht. Dies wird so lange durchgeführt, bis auf einem Level keine Änderung mehr möglich ist. Danach werden für die gefundenen Clusterungen alle Abstrakstionslevel in umgekehrter Reihenfolge betrachtet und eine Verfeinerung durchgeführt.

Auf dem untersten Level entspricht der Multi-Level Sequence Greedy Algorithmus der Sequence Greedy Verfeinerung mit der Ausnahme, dass mit der Singleton Clusterung gestartet wird. Auf den höheren Level ist der Ablauf beim Multi-Level Sequence Greedy Algorithmus anders, da es keine 1-zu-1-Entsprechung der Superknoten gibt. Dies bedeutet, dass ein Superknoten zu unterschiedlichen Zeitpunkten aus unterschiedlichen elementaren Knoten entstanden ist und somit auch die Superkanten adjazenter Snapshots nicht mehr direkt miteinander vergleichbar sind.

Eine Kontraktion bewirkt, dass mehrere elementare Kanten durch eine neue Superkante repräsentiert werden. Dies kann natürlich zu jedem Zeitpunkt sehr unterschiedlich ablaufen. Damit man aber nach der Kontraktion noch weiß, wie viele ursprüngliche elementare Kanten in welcher Superkante im vorherigen bzw. nachfolgenden Snapshot stecken, werden zwei Listen verwendet.

Jede Kante e besitzt eine Liste `e.past` und eine Liste `e.future`. In den Listen stehen die (Super-) Kanten des Vorgängers bzw. des Nachfolgers, welche die elementaren Kanten, die durch e repräsentiert werden, beinhalten. Die Einträge der Listen können als Zeiger auf (Super-) Kanten im Vorgänger bzw. Nachfolger betrachtet werden. Auf dem untersten Abstraktionslevel ist jede Kante selbst nur eine einzige elementare Kante und daher steht in den jeweiligen Listen nur die entsprechende Kante im Vorgänger bzw. Nachfolger mit einer Anzahl von 1, falls sie im vorherigen oder nächsten Snapshot existiert. Existiert die Kante nicht im Vorgänger oder Nachfolger, hat dies zur Folge, dass die entsprechenden Listen leer sind. In Abbildung 5.2 wird dieser Sachverhalt beispielhaft anhand der oberen drei Snapshots dargestellt. Bei den Kanten zum Zeitpunkt $t = 0$ gibt es keine Vorgänger, daher sind die entsprechenden `past` Listen leer (analog: `future` Listen zum Zeitpunkt

$t = 2$). Da hier auf dem Abstraktionslevel 0 gearbeitet wird, haben die Listen maximal einen Eintrag mit dem Vorgänger bzw. Nachfolger. Betrachtet man beispielsweise die Kante $\{1, 3\}$ zum Zeitpunkt $t = 1$, so ist die entsprechende `future` Liste leer, da die Kante im nachfolgenden Snapshot nicht existiert. Da es aber die Kante im vorherigen Snapshot gibt, steht in der `past` Liste das Tupel (1,3,1), wobei die ersten zwei Zahlen für die Endknoten der Kante stehen und die dritte Komponente des Tupels die Anzahl dieser Kante im Vorgänger repräsentiert. Da aber auf Abstraktionslevel 0 nur mit elementaren Kanten gearbeitet wird, gibt es maximal eine Anzahl von 1.

Beim Multi-Level Sequence Greedy Algorithmus wird durch alle Knoten aller Snapshots in zufälliger Reihenfolge iteriert. Steht nun ein Knoten u vor der Entscheidung, ob er in den Cluster C eines anderen Knotens v wechselt, muss zunächst überprüft werden, ob sich die Zielfunktion (ΔQ_{bi} aus Kapitel 5.3) durch eine eventuelle Verschiebung verbessern würde. Die Modularity Berechnung läuft wie immer ab. Für die Berechnung des Graph-theoretischen Rand-Index zum Vorgänger muss u für alle Kanten, die in den Cluster C führen, die `past` Listen durchgehen und für jeden Eintrag der Form $(u, z, 1)$ prüfen, ob die Cluster-ID von u und z im vorherigen Zeitpunkt identisch ist. Im Falle eines positiven Ergebnisses, wird $\frac{1}{|E''|}$ auf den Graph-theoretischen Rand-Index addiert, ansonsten subtrahiert. E'' ist dabei der Schnitt der Kanten der betrachteten benachbarten Snapshots. Diese Prozedur wird analog mit den `future` Listen durchgeführt, um die Änderung des Graph-theoretischen Index zum Nachfolger zu berechnen. Der Inhalt der Listen ändert sich bei einer Verschiebung nicht, sondern nur die Cluster-IDs der betroffenen Knoten. Ausgesucht wird wieder der Nachbarknoten, mit dem der größte positive Zuwachs der Zielfunktion erreicht werden kann.

Nachdem keine Verschiebungen in irgendeinem Snapshot mehr möglich sind, werden alle Snapshots wie gewohnt kontrahiert. Das bedeutet unter anderem, dass mehrere elementare Kanten zu einer Superkante und mehrere Knoten zu einem Superknoten zusammengefasst werden. Die jeweiligen Kantenlisten der zu einer einzigen Superkante zusammengefasste Kanten werden aneinander gehängt und die jeweilige Knoten-ID der Einträge durch die neue Superknoten-ID im Vorgänger oder Nachfolger ersetzt. So entsteht im Beispiel der Abbildung 5.2 beim Snapshot zum Zeitpunkt $t = 1$ aus den Kanten $\{1, 2\}$, $\{1, 3\}$ nach der Kontraktion die Superkante $\{1, 2\}$. Die Einträge der `future` Listen der elementaren Kanten werden zusammengefasst. So entsteht zunächst aus $[(1, 2, 1)]$ und $[(1, 3, 1)]$ die neue `future` Liste $[(1, 2, 1), (1, 3, 1)]$ der Superkante. Da die Kontraktion auch bei den anderen Snapshots durchgeführt wird, stimmen die Knoten-IDs dieser neuen Liste nicht mehr. Aus $[(1, 2, 1), (1, 3, 1)]$ wird daher $[(1, 1, 1), (1, 1, 1)]$, da beim Snapshot zum Zeitpunkt $t = 2$ nur noch ein einziger Superknoten existiert. Mehrfach vorkommende Einträge in einer Liste werden zu einem einzigen Eintrag mit Angabe der Anzahl zusammengefasst. Somit erhält man für dieses Beispiel als Endresultat die Liste $[(1, 1, 2)]$. Für die `past` Liste wird analog vorgegangen.

Steht auf höheren Level ein Knoten erneut vor einer Verschiebung, so läuft dies genauso ab wie auf dem untersten Abstraktionslevel mit der Ausnahme, dass die Listen nun auch Einträge besitzen können, die eine von 1 verschiedene Anzahl beinhalten. Dies muss bei der Berechnung des Graph-theoretischen Rand-Index berücksichtigt werden. Dazu addiert oder subtrahiert man einfach $\frac{Anzahl}{E''}$ zum Graph-theoretischen Rand-Index hinzu. Falls Self-Loop Einträge in den Listen auftauchen, muss eine Clusterprüfung nicht mehr durchgeführt werden, da ein und derselbe Knoten immer den gleichen Cluster besitzt. Zudem können alle unterschiedlichen in der Liste vorkommenden Self-Loop Einträge zu einem einzigen Eintrag zusammengefasst werden.

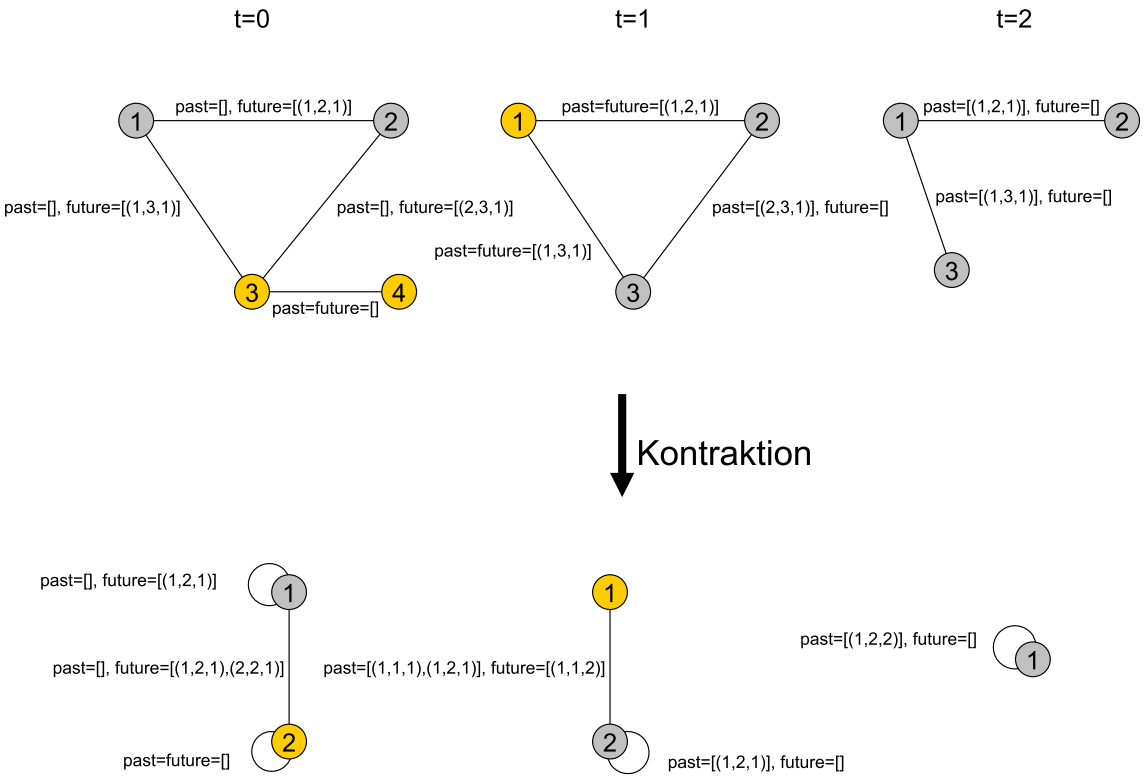


Abbildung 5.2.: Beispiel: Multi-Level Sequence Greedy Algorithmus

5.5. Implementierung und Anwendung der Heuristiken

Die Implementierung des Multi-Level Sequence Greedy Algorithmus war im Rahmen dieser Arbeit leider nicht möglich. Deshalb wird dieser Algorithmus auch bei den Experimenten nicht berücksichtigt. Die anderen in diesem Kapitel vorgestellten Heuristiken sind in C++ in einem einzigen C++ Programm `graph` zusammengefasst. Für die Handhabung der Graphen kommt dabei die *Boost::Graph-Bibliothek*[2] zum Einsatz. Im Folgenden werden alle möglichen Optionen dieser Heuristiken und weitere optionale Optionen des Programms vorgestellt:

- `-s [Wert]`: Diese Option ermöglicht es, das Dateiformat des einzulesenden Graphen zu spezifizieren. Es wird zwischen drei Formaten unterschieden:
  - Wert=0: Kantenliste (Default)
  - Wert=1: Knotenliste mit Cluster-ID und Kantenliste
  - Wert=2: Knoten- und Kantenliste

  Beispiele für diese Formate sind in Anhang A zu finden.
- `-p [Graph-Datei]`: Mit Hilfe dieser Option wird ein gegebener Snapshot nach Modularity geclustert.
- `-q [Graph-Datei]`: Mit Hilfe dieser Option wird ein gegebener Snapshot bikriteriell mittels Timestep Greedy Algorithmus geclustert.
- `-n [Wert]`: 1 (default) gibt an, dass eine Normierung von α erwünscht ist. 0 steht für keine Normierung.

[2] `http://www.boost.org/`

- `-g [Anzahl Snapshots]`: Mit dieser Option kann die Anzahl der Zeitpunkte bzw. Snapshots (definiert als T) festgelegt werden. Dieser Wert wird im Falle einer Normierung von α benötigt. Die Anzahl der Snapshots entspricht der Anzahl der Optionen `p`, `q` oder `r`.
- `-a [Wert]`: Mit dieser Option lässt sich α vom Typ Double im Bereich $[0, 1]$ festlegen. Der Default Wert ist 1.
- `-r [Graph-Datei]`: Hiermit wird ein gegebener Snapshot zufällig geclustert.
- `-m [Wert]`: Mit dieser Option kann die gewünschte maximale Anzahl der Cluster beim zufälligen Clustering festgelegt werden. Der Default Wert entspricht der Anzahl der Knoten des betrachteten Graphen.
- `-t [Graph-Datei]`: Hiermit wird ein von Algorithmus 8 erstellter zeitexpandierter Graph nach Modularity geclustert. Ein Beispiel für das unterstützte Eingabeformat eines zeitexpandierten Graphen befindet sich in Anhang A.1.
- `-o [Wert]`: Diese Option regelt die Sequence Greedy Verfeinerung. Ein Wert von 0 bedeutet, dass die Sequence Greedy Verfeinerung nicht erwünscht ist. Ein Wert von 1 bedeutet aktivierte Sequence Greedy Verfeinerung. Voraussetzung ist, dass die Snapshots zuvor mit der Option `-p` bzw. `-q` geclustert worden sind. Dies ist die Initiallösung. Ein Wert von 2 deutet auch auf aktiviertes Sequence Greedy Clustern hin, allerdings ohne vorheriges Clustern der Snapshots. Jeder Snapshot hat die Singleton Clusterung als Initiallösung.
- `-c [Clustering]`: Mit Hilfe dieser Option wird eine Datei, in der die Clusterung eines einzigen Snapshots steht, eingelesen und der Modularity Wert dieser Clusterung ausgerechnet. (Format: Knotenliste mit Cluster-ID und Kantenliste).

Der erste Parameter umfasst immer die Angabe einer Ausgabedatei, in welche die Ergebnisse des Programms geschrieben werden sollen. Zudem müssen die Snapshots immer in der gewünschten Reihenfolge angegeben werden. Ein beispielhafter Programmaufruf könnte wie folgt aussehen:

```
./graph /log.csv -n 0 -a 0.7 -p /0.txt -q /1.txt -q /2.txt
```

Hiermit wird ein dynamischer Graph mit Hilfe des Timestep Greedy Algorithmus für $\alpha = 0.7$ geclustert. Der erste Snapshots des dynamischen Graphen steht in der Datei `0.txt` und wird nur nach Modularity geclustert. Für den zweiten (`1.txt`) und dritten Snapshot (`2.txt`) wird der Timestep Greedy Algorithmus verwendet. Es wird keine Normierung durchgeführt. Will man zusätzlich noch die Sequence Greedy Verfeinerung aktivieren, wird eine Normierung aufgrund der globalen Zielfunktion Q_{bi} bei den Initiallösungen durchgeführt. Der Aufruf dafür lautet:

```
./graph /log.csv -o 1 -n 1 -g 3 -a 0.7 -p /0.txt -q /1.txt -q /2.txt
```

6. Experimente und Auswertung

In diesem Kapitel werden zunächst diverse dynamische Graphen vorgestellt, die als Versuchsinstanzen genutzt werden. Hierbei wird zwischen Echtweltdaten und eigenen generierten Graphen unterschieden. Im Anschluss werden die zum Clustern verwendeten Algorithmen beschrieben. Diese unterschiedlichen Algorithmen werden experimentell untersucht und miteinander verglichen.

6.1. Verwendete Graphen

Zur Durchführung der Versuche werden unterschiedliche dynamische Graphen verwendet.

6.1.1. Echtweltinstanzen

Ein verwendeter dynamischer Graph baut auf den Patentdaten der Weltorganisation für geistiges Eigentum [Org05] auf. Diese Patente sind in verschiedene Technikbereiche klassifiziert. Für jeden Bereich existiert ein Schlüssel, der aus Buchstaben und Ziffern besteht. Dabei sind viele Patente unter mehreren Schlüsseln angemeldet. Der dynamische Graph, der für die Versuche benutzt wird, beinhaltet die Patentdaten für Deutschland von 1986 bis 2005. Dabei ist jedes Jahr ein Zeitpunkt des dynamischen Graphen. Die Knoten des Graphen sind die Schlüssel (hier: gröbere Klassifizierung auf der dritten Hierarchieebene, da sonst zu viele Knoten existieren). Eine Kante zwischen zwei Knoten bedeutet, dass ein Patent unter beiden Schlüsseln angemeldet ist. Das Kantengewicht wird mittels des Tanimoto-Koeffizienten (mehr dazu in [TSK06] und [Gör10]) bestimmt:

$$tan(\vec{a}, \vec{b}) := \frac{\sum_{i=1}^{n} (\vec{a_i} \cdot \vec{b_i})}{\sum_{i=1}^{n} \vec{a_i}^2 + \sum_{i=1}^{n} \vec{b_i}^2 - \sum_{i=1}^{n} (\vec{a_i} \cdot \vec{b_i})}$$

Die Vektoren $\vec{a}$ und $\vec{a}$ stehen stellvertretend für zwei unterschiedliche Schlüssel. Die Einträge der Vektoren umfassen die für den jeweiligen Schlüssel angemeldeten Patente. Wenn ein Patent i unter zwei unterschiedlichen Schlüssel angemeldet wird, so steht für dieses Patent bei den beiden entsprechenden Vektoren an der selben Stelle i der Eintrag $\frac{1}{q}$, wobei q die Anzahl der Schlüssel ist, unter denen das Patent i angemeldet ist. Falls ein Patent unter einem bestimmten Schlüssel nicht zu finden ist, wird der entsprechende Eintrag mit einer 0 versehen. Der dynamische Graph der Patentdaten für Deutschland enthält insgesamt

130155 Kanten und 11294 Knoten. Das entspricht etwa 6508 Kanten und 565 Knoten pro Zeitschritt.

Ein weiterer Graph, aus dem ein dynamischer Graph für die Versuche erzeugt wird, ist der E-Mail Graph der Fakultät für Informatik [GHH+11]. Dieser Graph beinhaltet anonymisierte Informationen über die E-Mails, die intern versendet wurden. Jede Zeile des Graphen repräsentiert eine versendete E-Mail und ist in sechs Blöcke aufgeteilt:

1. Datum und Uhrzeit der versendeten E-Mail
2. ID des Senders
3. ID des Lehrstuhls des Senders
4. ID des Empfängers
5. ID des Lehrstuhl des Empfängers
6. Einmalige ID für die versendete E-Mail

Der für diese Arbeit verwendete E-Mail Graph der Fakultät umfasst den Zeitraum vom 13. September 2006 bis zum 15. August 2010. In dieser Zeit wurden insgesamt 550000 E-Mails verschickt. Für den daraus erstellten dynamischen Graphen gilt: Ein Zeitschritt umfasst jeweils 30 Tage. Damit erhält man insgesamt 48 Zeitpunkte. Die Knoten des dynamischen Graphen stehen stellvertretend für die Sender- und Empfänger-IDs und als Kantengewicht kommt die Anzahl der verschickten E-Mails im aktuell betrachtenden Zeitraum zum Einsatz. Es gibt insgesamt 94474 Kanten und 23704 Knoten. Dies entspricht einem Durchschnitt von 1968 Kanten und 494 Knoten pro Zeitpunkt.

Weitere Instanzen werden mittels eines Zufallsgenerators erzeugt, welcher im Folgenden erläutert wird.

6.1.2. Generierte Instanzen

C. Staudt und R. Görke stellen in [GS09] einen Zufallsgenerator zum Erstellen von dynamischen Graphen vor, den sogenannten *DCR-Generator*. Dieser kommandozeilenbasierte, in JAVA geschriebene Generator erstellt einen dynamischen, ungewichteten Graphen und speichert diesen in einer Datei ab. Mit Hilfe diverser einstellbarer Parameter ist es Anwendern gestattet, einen dynamischen Graphen nach eigenen Vorlieben zu erstellen. So lässt sich zum Beispiel die Anzahl der Knoten oder die der Cluster festlegen. Durch diese individuellen Anpassungen ist der Generator für eine zielgerichtete Evaluation sehr geeignet. Die optional festlegbaren Parameter sind in der Tabelle 6.1 erläutert.

Der DCR-Generator unterstützt mit GraphML und dem Binärformat zwei mögliche Ausgabeformate. In dieser Arbeit wird nur Letzteres benutzt. In Anhang B werden das Binärformat und zwei Programme, die diese Binärdatei verarbeiten und daraus direkt einen dynamischen sowie einen zeitexpandierten Graphen bilden, vorgestellt. Mit Hilfe des DCR-Generators werden für die Durchführung der Versuche insgesamt 20 dynamische Graphen erstellt. Dabei werden zwei Konfigurationen benutzt. Von jeder Konfiguration werden jeweils 10 dynamischen Graphen generiert.

Für beide Konfigurationen wird die in der Tabelle 6.2 beschriebene Parameterwahl verwendet. Da die Dynamik des Graphen hauptsächlich die Kanten betreffen soll, werden *p_chi* und *eta* entsprechend gewählt. Zudem sollen die Größe eines Clusters und die Gesamtanzahl der Cluster etwa gleich groß sein. Die Anzahl der Zeitpunkte und die Anzahl der Knoten werden so bestimmt, dass im Vergleich zu dem E-Mail Graphen und der Patentdaten nun etwas größere Instanzen erzeugt werden. Es werden zwei Typen von Zufallsgraphen erstellt, die sich bei einer gleichen Anzahl von Knoten deutlich in der Kantenanzahl unterscheiden. Geregelt wird das über den Grad eines Knotens.

Parameter	Bereich	default	Erklärung
n	$\mathbb{N}$	60	anfängliche Knotenanzahl in G^0
p_in	[0,1]	0.02	Wahrscheinlichkeit für die Existenz einer Kante zwischen Knoten mit selber Cluster-ID
p_out	[0,1]	0.01	Wahrscheinlichkeit für die Existenz einer Kante zwischen Knoten mit untersch. Cluster-ID‘s
k	$\mathbb{N}$	2	anfängliche Clusteranzahl
t_max	$\mathbb{N}$	100	Anzahl an Zeitpunkten
p_nu	[0,1]	0.5	geg. Knoten-Ereignis: Wahrscheinlichkeit für das Hinzufügen eines Knotens (1-p_nu ist die Wahrscheinlichkeit für das Löschen)
p_chi	[0,1]	0.5	Wahrscheinlichkeit für ein Kanten-Ereignis (Knoten-Ereignis $\rightarrow$ 1-p_chi)
p_omega	[0,1]	0.02	Wahrscheinlichkeit für ein Cluster-Ereignis
p_mu	[0,1]	0.5	geg. Cluster-Ereignis: Wahrscheinlichkeit für das Vereinigen zweier Cluster (1-p_mu für das Aufteilen eines Clusters)
theta	[0,1]	0.25	Toleranzgrenze für das Akzeptieren einer neuen Clusterung
beta	$\mathbb{R}$	1.0	Exponent der Selection Bias
eta	$\mathbb{N}$	1	Mindestanzahl an Kanten-Ereignissen pro Zeitschritt
p_inList	$[0,1]^k$	-	individuelles p_in für jedes Cluster
D_s	$\mathbb{R}_+^k$	-	Verteilung der Clustergrößen in $\mathcal{C}(G^0)$ (anstelle von beta)
enp	{true,false}	false	neues p_in nach Abschätzung von Gauss (true) oder arithmetisches Mittel
outDir	String	./	Ausgabeordner
fileName	String	dcrGraph _Datum	Dateiname
binary	{true,false}	false	Ausgabe als Binärdatei
graphml	{true,false}	false	Ausgabe als GraphML-Datei

Tabelle 6.1.: Konfigurierbare Parameter des Generators

Parameter	Wert (Konfiguration 1)	Wert (Konfiguration 2)
n	1000	1000
k	32	32
t_max	100	100
p_chi	0.99	0.99
p_omega	0.1	0.1
eta	1000	1000
p_in	$\frac{2}{31}$	$\frac{10}{31}$
p_out	$\frac{1}{968}$	$\frac{10}{968}$

Tabelle 6.2.: Wahl der Parameter für beide Konfigurationen

Bei der ersten Konfiguration (Typ 1) wird ein durchschnittlicher Knotengrad von 3 angestrebt. Dies bedeutet, dass jeder Knoten im Schnitt drei inzidente Kanten besitzt. Zwei dieser Kanten sollen Intracluster-Kanten sein. Bei 1000 Knoten und 32 Clustern besitzt jeder Cluster etwa 32 Knoten. Da von jedem Knoten zwei Intracluster-Kanten ausgehen sollen, gilt für einen Knoten und seine insgesamt 31 möglichen Nachbarn im selben Cluster: $p\_in = \frac{2}{31}$. Bei einem Knotengrad von 3 und einer einzigen Intercluster-Kante, ergibt sich $p\_out = \frac{1}{968}$, da insgesamt 968 Knoten außerhalb des Clusters des aktuell betrachtenden Knoten erreicht werden können.

Bei der zweiten Konfiguration (Typ 2) ist das Ziel, dichtere Graphen zu erstellen. Daher soll jeder Knoten einen durchschnittlichen Knotengrad von 20 mit 10 Intracluster- und 10 Intercluster-Kanten besitzen. Somit gilt: $p\_in = \frac{10}{31}$ und $p\_out = \frac{10}{968}$.

Für alle nicht in der Tabelle 6.2 aufgeführten Parameter werden die Standardwerte benutzt. Von beiden Konfigurationen werden jeweils 10 zufällige Instanzen als Binärdatei generiert, welche mittels des Konverters aus Anhang B zu dynamischen Graphen umgewandelt werden. Dabei wird die gewichtete Version des Konverters benutzt, die jeder hinzugefügten Kante ein Gewicht im Bereich $[1, 100]$ zuweist und nach jedem Zeitschritt über alle Kanten des aktuellen Snapshots iteriert und die Gewichte leicht modifiziert, so dass das durchschnittliche Gewicht fast identisch bleibt. Das neue Kantengewicht entspricht entweder dem alten Gewicht oder dem alten Gewicht multipliziert mit dem Faktor $\frac{5}{6}$ bzw. $\frac{7}{6}$. Diese drei möglichen Fälle treten gleichverteilt auf und sorgen für eine zusätzliche Kantendynamik.

6.2. Experimenteller Aufbau

6.2.1. Zusammenfassung der verwendeten Algorithmen

Für alle vorgestellten Instanzen werden mit Hilfe unterschiedlicher Algorithmen Clusterungen gesucht:

- *Statisches Multi-Level-Clustering* (MLC): Jeder Snapshot wird nur nach Modularity geclustert.
- *Statisches Multi-Level-Clustering mit Sequence Greedy Verfeinerung* (MLC+SG): Alle Snapshots werden nach Modularity geclustert. Im Anschluss wird eine Sequence-Greedy Verfeinerung für Q_{bi} mit entsprechenden Werten für α durchgeführt.
- *Zeitexpandiertes Clustern*: Aus allen Snapshots eines dynamischen Graphen wird ein einziger Graph gebaut, der nur Modularity-basiert geclustert wird. Für das Gewicht der intertemporären Kanten wird die in Kapitel 4.1 beschriebene Cosine-Similarity verwendet, welche mit dem durchschnittlichen Kantengewicht multipliziert wird. Allerdings haben die ersten Versuche gezeigt, dass sich das durchschnittliche Kantengewicht nicht eignet. Damit werden schlechte Resultate für die Modularity erzielt, da die intertemporären Kanten bei Verwendung des Durchschnitts als Faktor der Cosine-Ähnlichkeit keine wichtige Rolle beim Clustering der Instanzen spielen. Jeder ursprüngliche Snapshot wird zu einem eigenen Cluster. Auf Grund dieses Umstandes wird für den E-Mail Graphen und für die Patentdaten das maximale Kantengewicht aus allen Snapshots als Faktor benutzt. Für den E-Mail Graphen ist dies 832 und für die Patentdaten 1. Bei allen zufällig generierten Instanzen wird der Faktor 100 verwendet. Nach Abschluss des Clusterings wird aus dem zeitexpandierten Graphen wieder der ursprüngliche dynamische Graph erzeugt.
- *Timestep Greedy Algorithmus* (TG): Der erste Snapshot wird nach Modularity geclustert, ab dem zweiten dann bikriteriell für die entsprechenden Werte von α.

- *Timestep Greedy Algorithmus mit Sequence Greedy Verfeinerung* (TG+SG).

Ziel aller Algorithmen ist es, möglichst gute Werte für die Zielfunktion Q_{bi} zu erhalten. Für den Timestep Greedy Algorithmus und die Sequence Greedy Verfeinerung werden für α Werte von 0 bis 0.9 (in 0.1 Schritten) verwendet. Beim Clustern der einzelnen Snapshots, wird die in Kapitel 5.2 vorgestellte Normierung von α durchgeführt. Es hat sich durch diverse Versuche herausgestellt, dass es keinen erwähnenswerten Unterschied ausmacht, ob diese Normierung stattfindet oder nicht. Dennoch wird sie in allen folgenden Versuchen verwendet. Bei den generierten Zufallsgraphen wird für jede Konfiguration der Durchschnitt der Ergebnisse aller 10 Instanzen betrachtet.

6.2.2. Verwendete Hardware

Für die Durchführung der Versuche wird die in Kapitel 5.5 vorgestellte Implementierung verwendet. Der Rechner, auf welchem die Versuche laufen, besitzt zwei Xeon E5430 2,66 GHz (Quad-Core) CPU´s und insgesamt 32 GB Arbeitsspeicher, wovon aber nur maximal 8 GB für die Versuche zugewiesen werden. Das Betriebssystem verwendet den Linux Kernel 2.6.34.

6.3. Vergleich: Modularity und Graph-theoretischer Rand-Index

Alle Instanzen werden mittels der oben genannten Algorithmen geclustert. Für die Clusterungen, die für jeden Snapshot gefunden werden, werden Modularity und der Graph-theoretische Rand-Index zum Vorgänger berechnet und daraus Kurvendiagramme erstellt. Der detaillierte zeitliche Verlauf von Modularity und Graph-theoretischem Rand-Index für den Timestep Greedy Algorithmus ohne Sequence Greedy Verfeinerung wird im Folgenden nicht für alle Werte von α aufgeführt, sondern ist wegen großer Ähnlichkeit zum Timestep Greedy Algorithmus mit Sequence Greedy Verfeinerung in Anhang C zu finden.

6.3.1. Ergebnisse für die Zufallsgraphen vom Typ 1

In Abbildung 6.1 ist der zeitliche Verlauf von Modularity und Graph-theoretischem Rand-Index der Zufallsgraphen vom Typ 1, die mittels Timestep Greedy Algorithmus und Sequence Greedy Verfeinerung geclustert werden, dargestellt. Jede Kurve repräsentiert ein anderes α. Clustert man die Zufallsgraphen komplett nach Graph-theoretischem Rand-Index ($\alpha = 0$), so erhält man nur am Anfang hohe Modularity Werte. Die Modularity nimmt jedoch schnell ab und ist bereits ab Zeitpunkt 16 dauerhaft im Bereich von 0. Anfangs ist die Modularity deshalb höher, da der erste Snapshot nur nach Modularity geclustert wird. Diese Clusterung bildet die Basis für alle darauf folgenden Snapshots. Der zweite Snapshot übernimmt komplett diese Clusterung für die Knotenpaare der gemeinsamen Kantenmenge. Der dritte Snapshot übernimmt wiederum die Clusterung des zweiten Snapshots und nach diesem Muster wird bis zum letzten Snapshot voran geschritten. Somit erhält man für den Graph-theoretischen Rand-Index einen Wert von 1 für die komplette Zeitspanne des dynamischen Graphen. Die Knoten, die nicht in einer Kante der gemeinsamen Kantenmenge zweier aufeinander folgender Snapshots vertreten sind, werden Singleton Cluster. Die späteren Snapshots unterscheiden sich bezüglich ihrer Struktur immer mehr vom ersten Snapshot. Dies führt dazu, dass es immer mehr Singleton Cluster gibt und die Modularity dadurch immer geringer wird. Die Sequence Greedy Verfeinerung bringt für diesen Fall nichts mehr, da die Zielfunktion vor der Verfeinerung für $\alpha = 0$ mit einem Wert von 1 ihr Maximum bereits erreicht hat. Ein steigendes α sorgt dafür, dass die Abnahme der Modularity geringer ausfällt. Schon mit $\alpha = 0.1$ bleibt die Modularity bis zum Ende hin immer über 0.6.

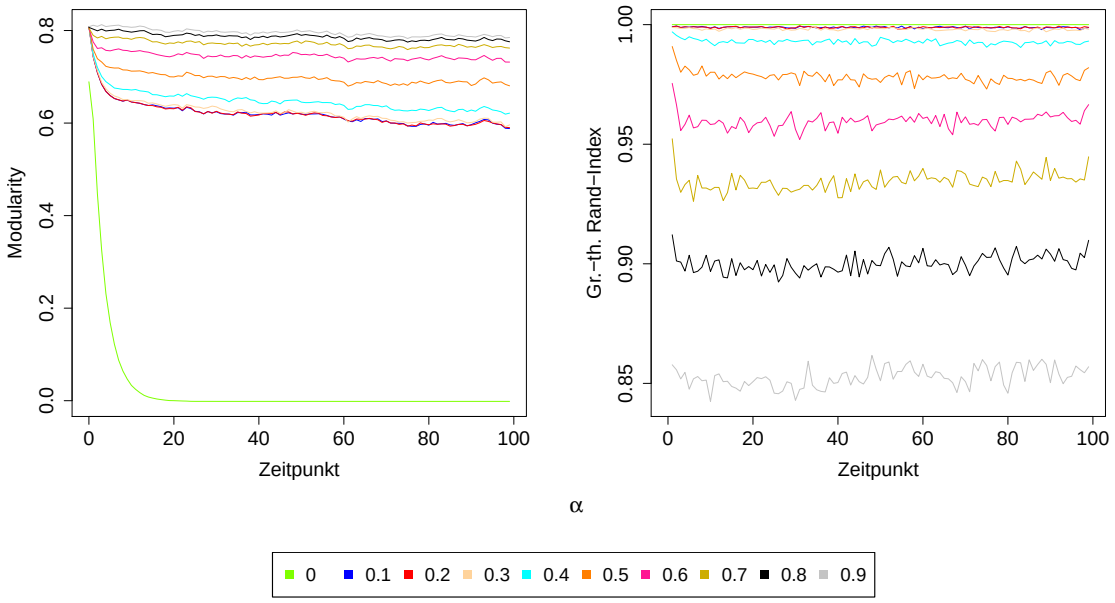


Abbildung 6.1.: Timestep Greedy Algorithmus und Sequence Greedy Verfeinerung der Zufallsgraphen vom Typ 1 (Mittelwerte)

Der Graph-theoretische Rand-Index bewegt sich durchgehend im Bereich von 0.99. In beiden Diagrammen erkennt man, dass die Kurven gestaffelt verlaufen. Das bedeutet, dass ein höheres α für eine durchgehend bessere Modularity und einen durchgehend schlechteren Graph-theoretischen Rand-Index sorgt. In beiden Diagrammen gibt es keine Überschneidungen zweier Kurven. Der Graph-theoretische Rand-Index bleibt selbst mit $\alpha = 0.9$ im Bereich von 0.85 auf einem hohen Niveau.

Clustert man nun alle Snapshots der Zufallsgraphen vom Typ 1 nur nach Modularity mittels des statischen Multi-Level-Clusterings und führt danach eine Sequence Greedy Verfeinerung für unterschiedliche Werte von α durch, so erhält man die in Abbildung 6.2 dargestellten zeitlichen Verläufe von Modularity und Graph-theoretischem Rand-Index. Auffällig ist, dass man für ein kleines α sehr schlechte Modularity Werte für den ersten und letzten Snapshot erhält. Dafür ist der Graph-theoretische Rand-Index am Anfang und am Ende deutlich besser als während der restlichen Zeit. Ein kleines α bedeutet, dass man eher eine Ähnlichkeit der Clusterungen möchte und die Qualität der Clusterungen weniger von Bedeutung ist. Somit hat die Sequence Greedy Verfeinerung stark in Richtung des Graph-theoretischen Rand-Index gearbeitet. Diese Verfeinerung betrachtet für den Graph-theoretischen Rand-Index immer den Vorgänger und Nachfolger des gerade betrachtenden Snapshots. Der erste Snapshot hat allerdings keinen Vorgänger und der letzte keinen Nachfolger. Daher wird hier nur eine Ähnlichkeit zum Nachfolger des ersten Snapshots und zum Vorgänger des letzten Snapshots angestrebt. Da man nur eine Richtung betrachtet, ist es leichter bzgl. des Graph-theoretischen Rand-Index zu optimieren und daher erhält man für diesen gute Werte. Da das Kurvendiagramm nur den Graph-theoretischen Rand-Index zum jeweiligen Vorgänger darstellt, haben somit der zweite und der letzte Snapshot hier eine hohe Ähnlichkeit. Diese hohe Ähnlichkeit sorgt aber für die schlechten Modularity Werte, da die Strukturen der jeweiligen Snapshots sich deutlich voneinander unterscheiden.

Die Kurven verlaufen wie schon zuvor beim Timestep Greedy Algorithmus mit Sequence Greedy Verfeinerung in beiden Diagrammen gestaffelt. Überschneidungen gibt es nur ein paar wenige beim Verlauf des Graph-theoretischen Rand-Index mit $\alpha = 0$. Im Vergleich zu

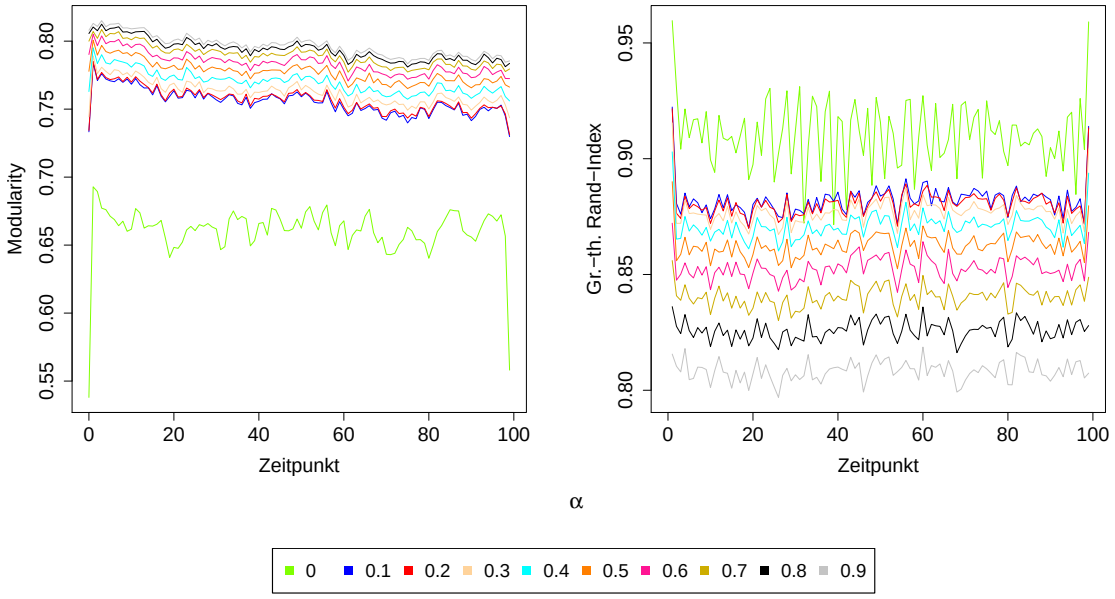


Abbildung 6.2.: Modularity-basiertes Clustern und anschließende Sequence Greedy Verfeinerung der Zufallsgraphen vom Typ 1 (Mittelwerte)

den vorherigen Ergebnissen des Timestep Greedy Algorithmus hat man hier die besseren Modularity Resultate, dafür fallen die Ergebnisse für den Graph-theoretischen Rand-Index schlechter aus. Für $\alpha = 0$ gibt es keinen Modularity Einbruch wie zuvor und daher eignet sich dieser Algorithmus im Gegensatz zum Timestep Greedy auch für dieses α. Es kommt hier auf Grund der Tatsache, dass alle Snapshots zunächst nur nach Modularity geclustert werden, zu keinem Modularity Fall. Es wird zwar mit der Sequence Greedy Verfeinerung in Richtung des Graph-theoretischen Rand-Index optimiert, da aber nur auf dem untersten Abstraktionslevel der Snapshots agiert wird, können nicht die guten Ergebnisse für den Graph-theoretischen Rand-Index für ein kleines α des Timestep Greedy Algorithmus mit Sequence Greedy Verfeinerung erreicht werden.

Vom Timestep Greedy Algorithmus mit und ohne Sequence Greedy Verfeinerung und vom Multi-Level-Clustering mit Sequence Greedy Verfeinerung werden mit $\alpha = 0.5$ jeweils ein Vertreter ausgesucht und mit den Verläufen von zeitexpandiertem Clustern, Multi-Level-Clustering und der Referenz Clusterung des Zufallsgenerators verglichen. Die jeweiligen Kurvenverläufe sind in Abbildung 6.3 zu finden. Wenn man nur nach Modularity clustert und keine Sequence Greedy Verfeinerung durchführt, erhält man die besten Modularity Ergebnisse unter allen Algorithmen. Der Modularity Verlauf bewegt sich durchgehend im Bereich von 0.81. Allerdings sind bei reinem Modularity-basierten Clustern die Ergebnisse für den Graph-theoretischen Rand-Index auch die deutlich schlechtesten unter allen Algorithmen. Die Kurven für den Timestep Greedy Algorithmus mit und ohne Sequence Greedy Verfeinerung sind nahezu identisch. Nur beim Graph-theoretischen Rand-Index sind minimale Abweichungen erkennbar. Interessant sind die Ergebnisse des Multi-Level-Clusterings mit Sequence Greedy Verfeinerung für $\alpha = 0.5$. Vergleicht man diese mit denen des Multi-Level-Clusterings ohne Verfeinerung, so erhält man über die komplette Zeitspanne hinweg betrachtet nur eine etwa 0.02 schlechtere Modularity, aber dafür ist der Graph-theoretische Rand-Index ca. um 0.08 besser. Das zeitexpandierte Clustern hat zu Beginn und am Ende deutlich schlechtere Modularity Werte. Im Gegenzug dazu ist der Graph-theoretische Rand-Index gerade am Anfang und am Ende deutlich besser als während der restlichen

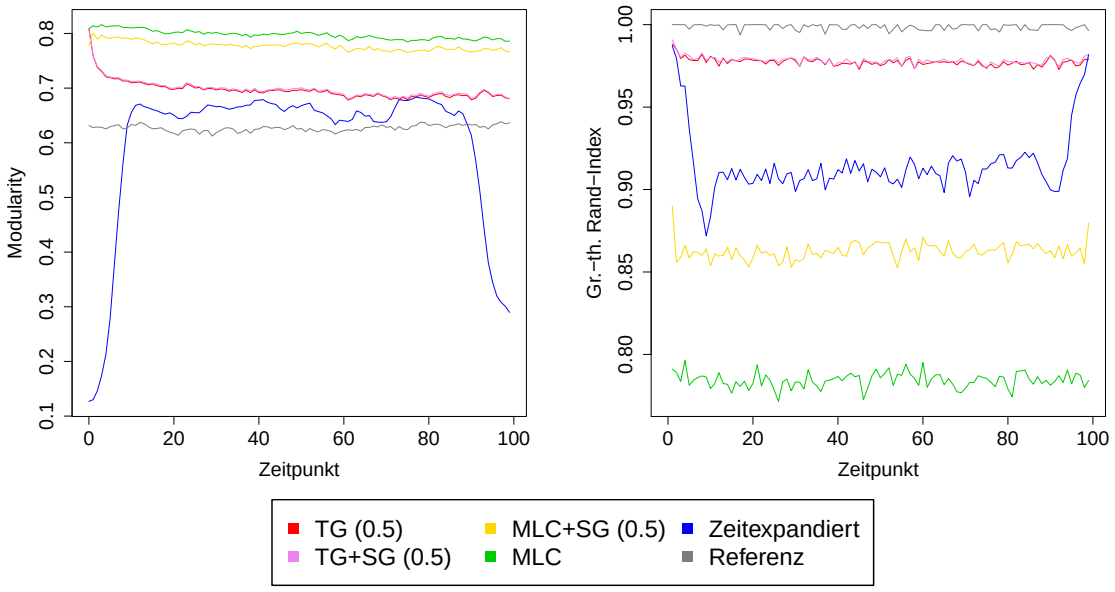


Abbildung 6.3.: Zufallsgraphen vom Typ 1: Vergleich der verschiedenen Algorithmen (Mittelwerte)

Dauer.

Die Referenz Clusterung des Generators hat die schlechtesten Modularity Ergebnisse, dafür ist der Graph-theoretische Rand-Index fortlaufend im Bereich von 0.99 bis 1, anders als erwartet, da mit einem Einbruch des Graph-theoretischen Rand-Index bei jedem Cluster-Ereignis gerechnet wurde. Die Wahrscheinlichkeit für ein Cluster-Ereignis liegt bei 0.1, d.h. für einen dynamischen Graphen mit 100 Snapshots sind dies 10 Cluster-Ereignisse. Der Grund, weshalb der Graph-theoretische Rand-Index sich bei der Referenz Clusterung dennoch auf so einem hohen Niveau bewegt, liegt an den 1000 Kanten-Ereignissen pro Zeitschritt. Der Graph-theoretische Rand-Index betrachtet nur Knotenpaare basierend auf der gemeinsamen Kantenmenge zweier Snapshots. Wenn es nun so ein Cluster-Ereignis, z.B. das Verschmelzen zweier Cluster, gibt, dann sind wegen der vielen Kantenänderungen pro Zeitschritt, nicht viele Knotenpaare betroffen, bei denen sich der Graph-theoretische Rand-Index ändern könnte. Am besten schneiden die beiden Timestep Greedy Algorithmen ab. Beide besitzen einen hohen Graph-theoretischen Rand-Index und sind dennoch von der Modularity her im Bereich von 0.7.

6.3.2. Ergebnisse für die Zufallsgraphen vom Typ 2

Die an den Zufallsgraphen vom Typ 1 durchgeführten Versuche finden ebenfalls an den Zufallsgraphen vom Typ 2 statt und werden auf die selbe Art und Weise protokolliert.

In Abbildung 6.4 sind die Ergebnisse für den Timestep Greedy Algorithmus mit anschließender Sequence Greedy Verfeinerung dargestellt. Auch hier erkennt man wie bei den Zufallsgraphen vom Typ 1 die Staffelung der Kurven für unterschiedliche Werte von α. Das bedeutet, dass ein höheres α eine höhere Modularity, dafür aber einen niedrigeren Graph-theoretischen Rand-Index bewirkt. Es gibt nur ein paar wenige Überschneidungen für höhere Werte von α beim Graph-theoretischen Rand-Index. Im Vergleich zu den Zufallsgraphen vom Typ 1 sinkt hier die Modularity für $\alpha = 0$ nicht mehr so stark und bleibt mit etwa 0.16 für den letzten Snapshot noch über 0. Ursache dafür ist, dass die Zufallsgraphen vom Typ 2 bei etwa gleicher Knotenanzahl wie die Zufallsgraphen vom Typ 1

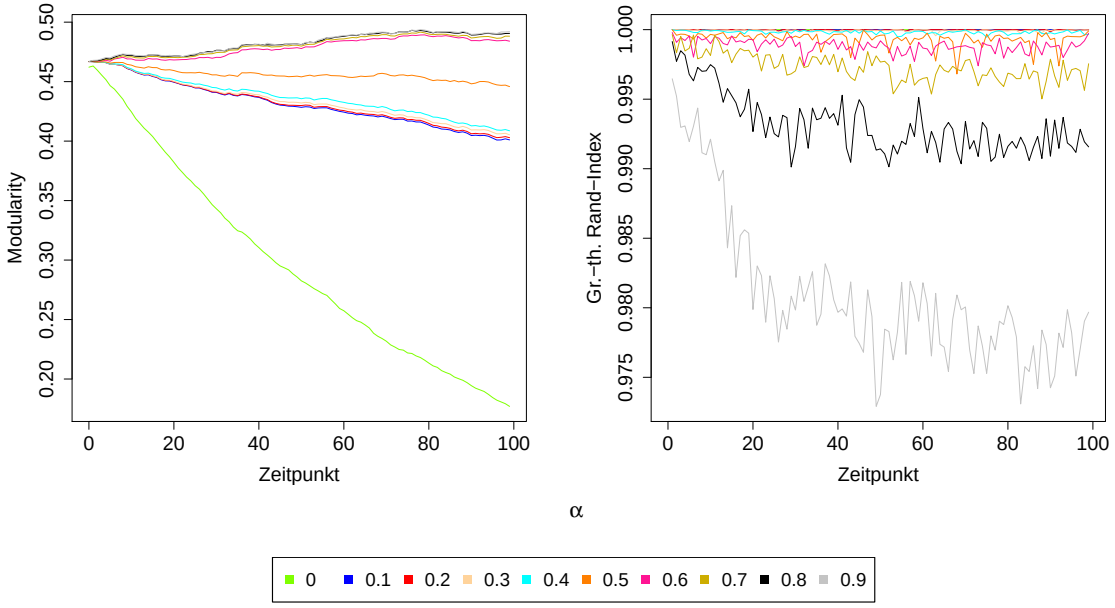


Abbildung 6.4.: Timestep Greedy Algorithmus und Sequence Greedy Verfeinerung der Zufallgraphen vom Typ 2 (Mittelwerte)

viel mehr Kanten besitzen, da diese einen durchschnittliche Knotengrad von 20 haben. Die 1000 Kanten-Ereignisse pro Zeitschritt beeinflussen die Struktur des jeweiligen Snapshots nicht so sehr wie bei den Zufallsgraphen vom Typ 1. Durch die höhere Kantenanzahl pro Snapshot, ist die Wahrscheinlichkeit höher, dass die gemeinsame Kantenmenge zweier aufeinander folgender Snapshots größer ist und es deshalb in den Clusterungen der späteren Snapshots weniger Singleton Cluster gibt. Die Resultate für den Graph-theoretischen Rand-Index sind für alle Werte von α in einem engen Bereich. Die Werte unterscheiden sich um maximal 0.03. Bei den Verläufen von Modularity ist zu erkennen, dass für $\alpha > 0.5$ die Kurven mit zunehmender Zeit steigen. Für $\alpha \leq 0.5$ ist ein Fall der Kurven zu erkennen. Für diese dichten Zufallsgraphen eignet sich die Wahl eines hohen α Wertes, da es bei den Modularity Werten zu größeren Differenzen beim Vergleich der Kurven zweier unterschiedlicher α Werte kommt als beim Graph-theoretischen Rand-Index.

Clustert man die Zufallsgraphen nur nach Modularity und führt im Anschluss eine Sequence Greedy Verfeinerung durch, so erhält man wiederum für alle Werte von α wie schon bei den Zufallsgraphen vom Typ 1 bessere Modularity Resultate, dafür aber einen schlechteren Graph-theoretischen Rand-Index als der Timestep Greedy Algorithmus mit Verfeinerung (siehe Abbildung 6.5). Der Modularity Verlauf aller dargestellten Kurven ist fast identisch. Trotz dessen, dass hier die Skala beim Graph-theoretischen Rand-Index einen größeren Wertebereich als beim Timestep Greedy Algorithmus umfasst, sind die Kurvenverläufe schwankender. Bei Modularity ist dies zwar auch der Fall, aber hier ist die Skalierung feiner als beim Timestep Greedy Algorithmus. Dennoch ist eine Staffelung der Kurven in beiden Diagrammen sichtbar. Jedoch kommt es zu vielen Überschneidungen. Auffallend bei den Zufallsgraphen vom Typ 1 ist die niedrige Modularity bzw. der hohe Graph-theoretische Rand-Index für kleine α Werte am Anfang und am Ende der Zeitspanne. Hier ist das eigentlich nicht mehr der Fall. Zwar erkennt man dies noch beim Graph-theoretischen Rand-Index, aber es muss die feinere Skalierung im Vergleich zum Diagramm der Zufallsgraphen vom Typ 1 beachtet werden. Ein Grund dafür, dass dieser Umstand hier nicht mehr so stark auftritt, ist, dass 1000 Kanten-Ereignisse pro Zeitschritt

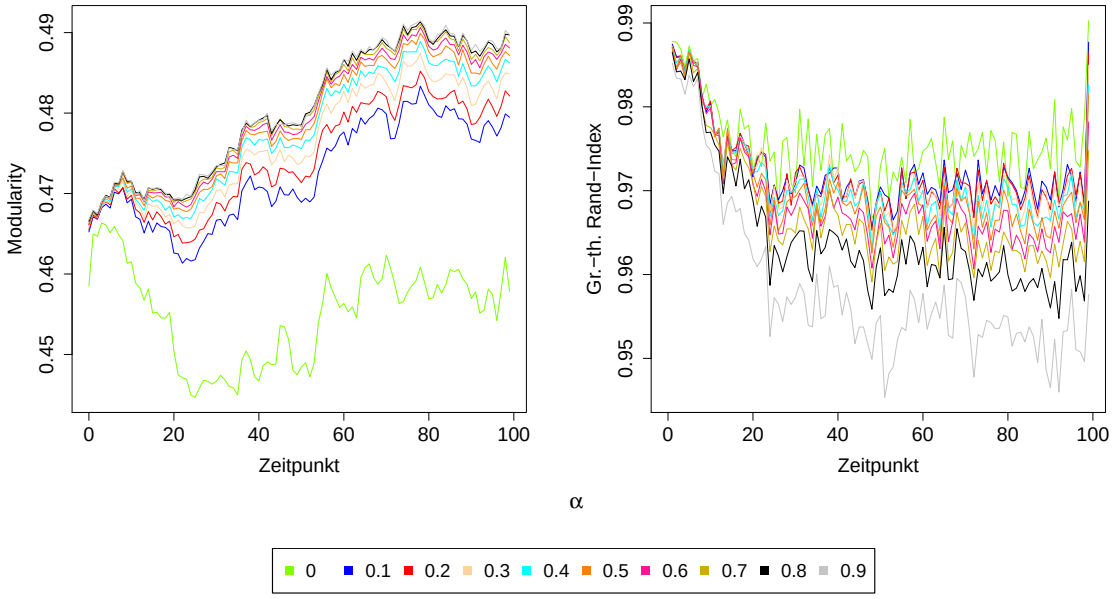


Abbildung 6.5.: Modularity-basiertes Clustern und anschließende Sequence Greedy Verfeinerung der Zufallsgraphen vom Typ 2 (Mittelwerte)

bei dichteren Snapshots, eine kleinere Änderung der Struktur des Snapshots verursachen als bei einem dünneren Snapshot.

In Abbildung 6.6 werden erneut alle Algorithmen, die zum Clustern der Zufallsgraphen vom Typ 2 verwendet werden, miteinander verglichen. Die Modularity Resultate für das zeitexpandierte Clustern werden auf Grund einer durchgehend schlechten Modularity von 0 hier nicht berücksichtigt, damit die Kurven der anderen Algorithmen besser zu erkennen sind. Zwar ist die Modularity beim zeitexpandierten Ansatz sehr schlecht, dafür erreicht man für den Graph-theoretischen Rand-Index einen Wert von 1 für die komplette Zeitspanne. Die Zufallsgraphen vom Typ 2 sind mit einem durchschnittlichen Knotengrad von 20 sehr dicht. Zudem besitzen diese dynamische Graphen 100 zeitliche Ausprägungen. Diese zwei Umstände sorgen dafür, dass das zeitexpandierte Clustern hier nicht mehr richtig funktioniert, da jeder Snapshot ein einziger Cluster ist. Die Kurven der beiden Timestep Greedy und die der beiden Multi-Level-Clustering Algorithmen (alle mit $\alpha = 0.7$) bewegen sich im Modularity Bereich sehr eng beieinander. Beim Graph-theoretischen Rand-Index schneiden die Timestep Greedy Algorithmen allerdings besser ab und sind daher zu bevorzugen. Das Multi-Level-Clustering mit Verfeinerung ist vom Graph-theoretischen Rand-Index auch deutlich besser als ohne Verfeinerung. Die Referenz Clusterung hat einen fortlaufend hohen Graph-theoretischen Rand-Index im Bereich von 1, ist aber zum Beispiel aus Modularity Sicht schlechter als die Timestep Greedy Vertreter, die ebenfalls beim Graph-theoretischen Rand-Index nahe am Verlauf der Referenz Clusterung liegen. Deswegen sind erneut die beiden Timestep Greedy Vertreter die beste Wahl zum Clustern dieses dynamischen Graphen.

6.3.3. Ergebnisse für den E-Mail Graphen der Fakultät

Der nächste dynamische Graph, der mittels unterschiedlicher Algorithmen geclustert wird, ist der E-Mail Graph der Fakultät für Informatik. Die Ergebnisse werden im Folgenden erläutert.

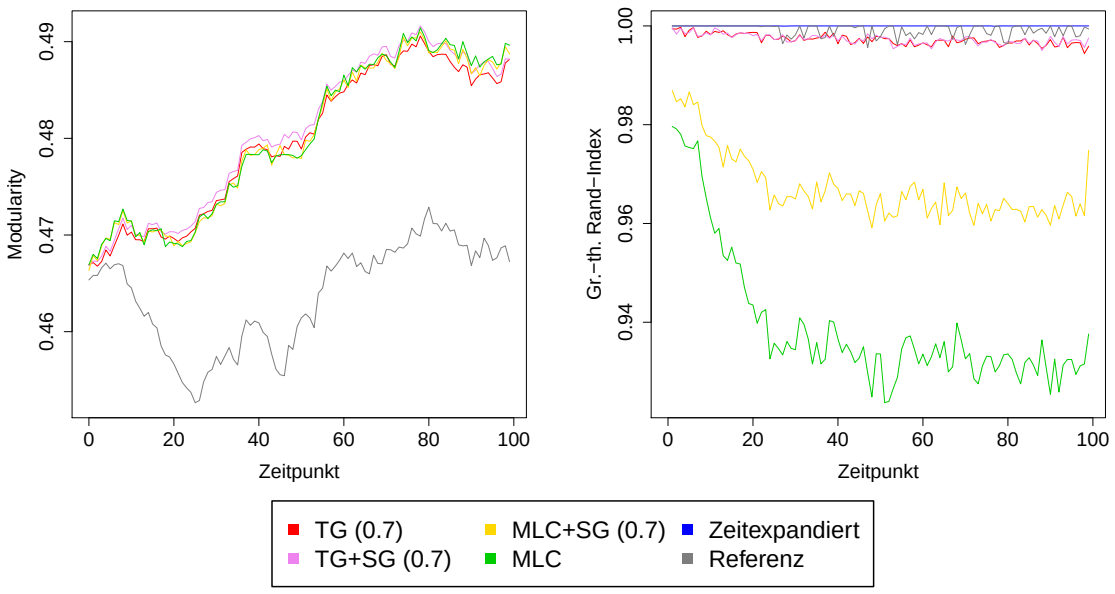


Abbildung 6.6.: Zufallsgraphen vom Typ 2: Vergleich der verschiedenen Algorithmen (Mittelwerte)

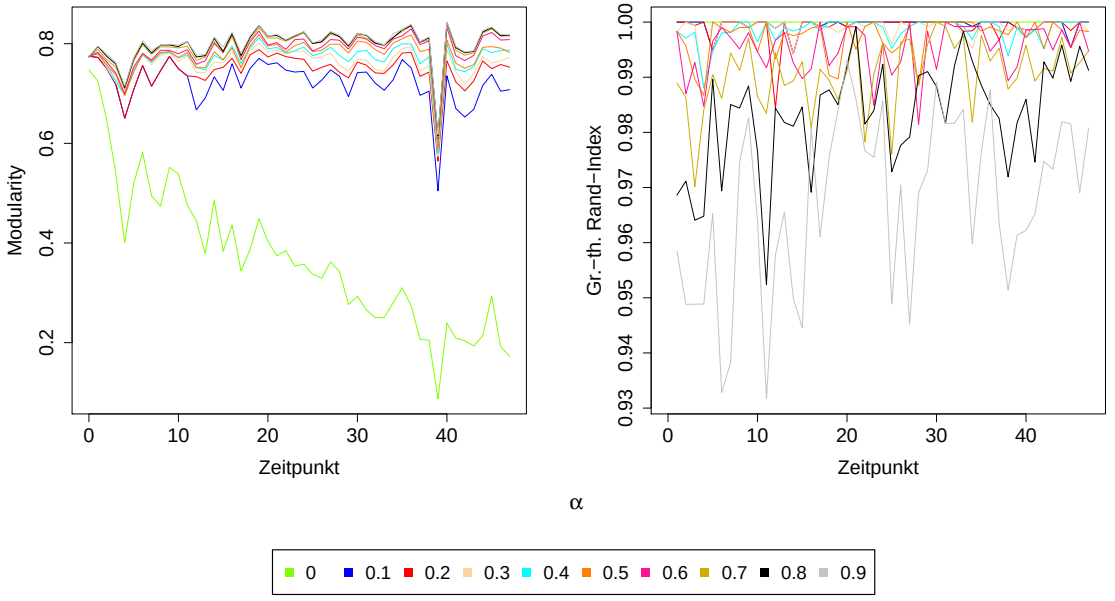


Abbildung 6.7.: Timestep Greedy Algorithmus und Sequence Greedy Verfeinerung des E-Mail Graphen

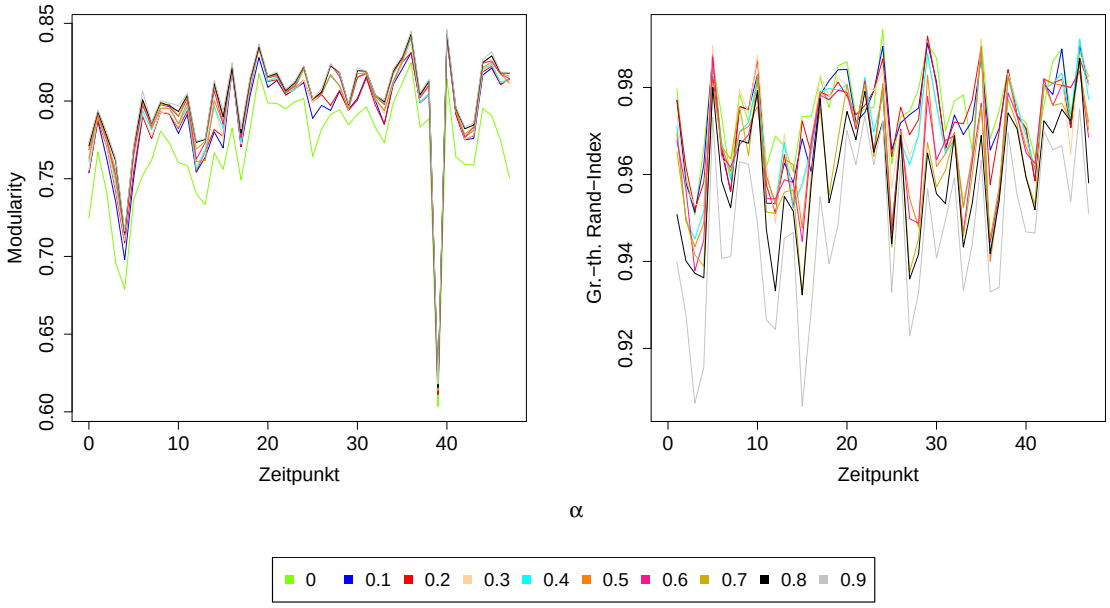


Abbildung 6.8.: Modularity-basiertes Clustern und anschließende Sequence Greedy Verfeinerung des E-Mail Graphen

In Abbildung 6.7 sind die Verläufe von Modularity und Graph-theoretischem Rand-Index, die durch das Clustern des E-Mail Graphen mittels Timestep Greedy Algorithmus mit Sequence Greedy Verfeinerung entstehen, dargestellt. Die Kurven für Modularity verlaufen gestaffelt. Bei den Kurven für den Graph-theoretischen Rand-Index ist eine Staffelung nicht mehr zu erkennen, da es dort vor allem für $\alpha \geq 0.6$ zu großen Schwankungen und vielen Überschneidungen der Kurven kommt. Eigentlich erwartet man, dass man mit einem niedrigeren α einen höheren Graph-theoretischen Rand-Index erhält. Jede Kurve beim Graph-theoretischen Rand-Index basiert auf unterschiedlichen Clusterungen und durch große strukturelle Unterschiede der Snapshots, kommt es eben nicht mehr zu einem gestaffelten Verlauf der Kurven des Graph-theoretischen Rand-Index. Würde man statt 30 Tage einen größeren Zeitraum pro Snapshot betrachten, wäre dieser Umstand nicht mehr so stark ausgeprägt. Bei den Modularity Verläufen fällt besonders der Zeitpunkt 39 auf. Dort ist für alle Werte von α die Modularity deutlich geringer als während der restlichen Zeitspanne. Ursache dafür ist wiederum die Struktur dieses Snapshots. Zum Zeitpunkt $t = 38$ gibt es 2174 Kanten, während es zum Zeitpunkt $t = 39$ nur noch 1646 Kanten gibt. Die Knotenanzahl ist ebenfalls von 548 auf 486 zurück gegangen.

Wendet man das Multi-Level-Clustering und die Sequence Greedy Verfeinerung auf dem E-Mail Graphen an, so erkennt man in Abbildung 6.8 ebenfalls bei $t = 39$ diesen Modularity Fall. Die Überschneidungen beim Graph-theoretischen Rand-Index sind nicht mehr so drastisch wie beim Timestep Greedy Algorithmus mit Verfeinerung. Sonst kommt man zu den selben Schlussfolgerungen für diesen Algorithmus wie schon zuvor bei den beiden Typen von Zufallsgraphen.

Der Vergleich aller angewendeten Algorithmen beim E-Mail Graphen ist in Abbildung 6.9 zu finden. Für die von α abhängigen Algorithmen werden mit $\alpha = 0.5$ jeweils ein Vertreter ausgesucht. Der Timestep Greedy Algorithmus ist mit und ohne Sequence Greedy Verfeinerung bei den Modularity Verläufen fast identisch, nur beim Graph-theoretischen Rand-Index gibt es minimale Unterschiede. Daran erkennt man, dass die Sequence Greedy Verfeinerung mehr in Richtung Ähnlichkeit optimiert als in Richtung Modularity. Das zeit-

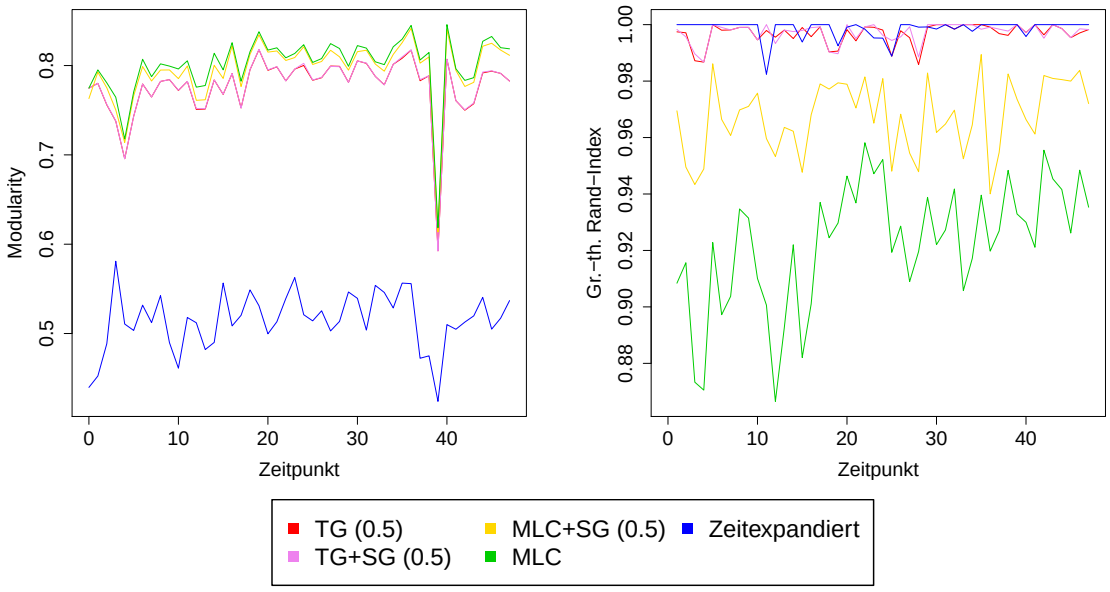


Abbildung 6.9.: E-Mail Graph: Vergleich der verschiedenen Algorithmen

expandierte Clustern liefert die besten Resultate für den Graph-theoretischen Rand-Index, jedoch sind diese nicht viel besser als die Ergebnisse der beiden Timestep Greedy Vertreter. Dafür haben Letztere die deutlich besseren Modularity Resultate und sind daher dem zeitexpandierten Clustern gegenüber klar vorzuziehen. Der Verlauf von Modularity beim statischen Multi-Level-Clustering mit Sequence Greedy Verfeinerung ist nahezu identisch zum Verlauf vom Multi-Level-Clustering ohne Verfeinerung. Mit Verfeinerung erhält man durchschnittlich eine Modularity von 0.801 und ohne Verfeinerung liegt der Durchschnittswert bei 0.795. Dafür sind die Unterschiede beim Graph-theoretischen Rand-Index etwas größer. Die Differenz der Durchschnittswerte liegt bei 0.05 zugunsten des Multi-Level-Clusterings mit Sequence Greedy Verfeinerung. Daher lohnt sich die Verfeinerung beim Modularity-basierten Multi-Level-Clustering.

Die Kurvendiagramme für die Patentdaten sind in Anhang C zu finden, da diese keine neuen Erkenntnisse liefern. Bei den Patentdaten sind die Verläufe ähnlich wie bei den Zufallsgraphen vom Typ 1 sehr gestaffelt mit ganz wenigen Überschneidungen. Es wird nur ein anderer Wertebereich umfasst.

6.4. Vergleich: Zielfunktion Q_{bi} bzgl. des Parameters α

In diesem Abschnitt wird die Zielfunktion Q_{bi} für unterschiedliche Werte von α auf Basis der gefundenen Clusterungen der eingesetzten Algorithmen berechnet und in den Abbildungen 6.10, 6.11, 6.12 und 6.13 als Balkendiagramm dargestellt. Jedes α umfasst 5 bzw. 6 Balken, falls eine Referenz Clusterung existiert. Jeder Balken steht stellvertretend für den eingesetzten Algorithmus, mit denen die Clusterungen erzeugt werden. Mit Hilfe der gefundenen Clusterung, der Modularity Werte und des Graph-theoretischen Rand-Index wird dann die Zielfunktion Q_{bi} für das jeweils betrachtete α berechnet. Die Versuchsergebnisse werden stellvertretend für alle verwendeten dynamischen Graphen erläutert.

Beim Timestep Greedy Algorithmus erreicht man mit der Sequence Greedy Verfeinerung, vor allem für ein α im mittleren Bereich, kleine Verbesserungen. Dies erkennt man am

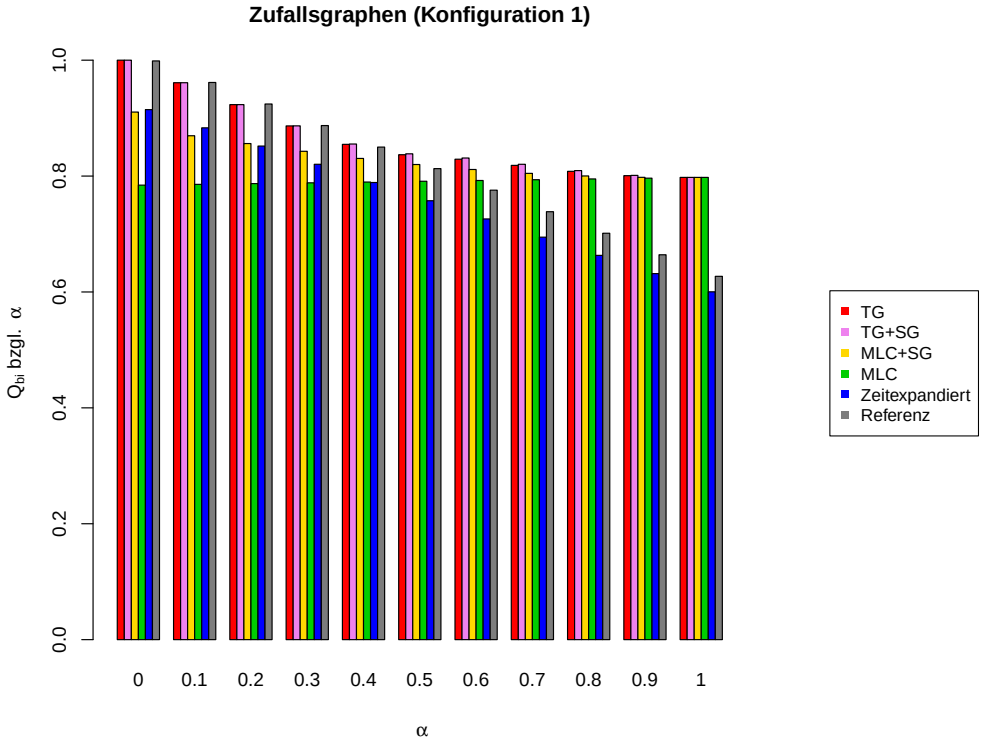


Abbildung 6.10.: Vergleich der Zielfunktionswerte Q_{bi} der zur Evaluierung eingesetzten Algorithmen für die Zufallsgraphen vom Typ 1 (Mittelwerte)

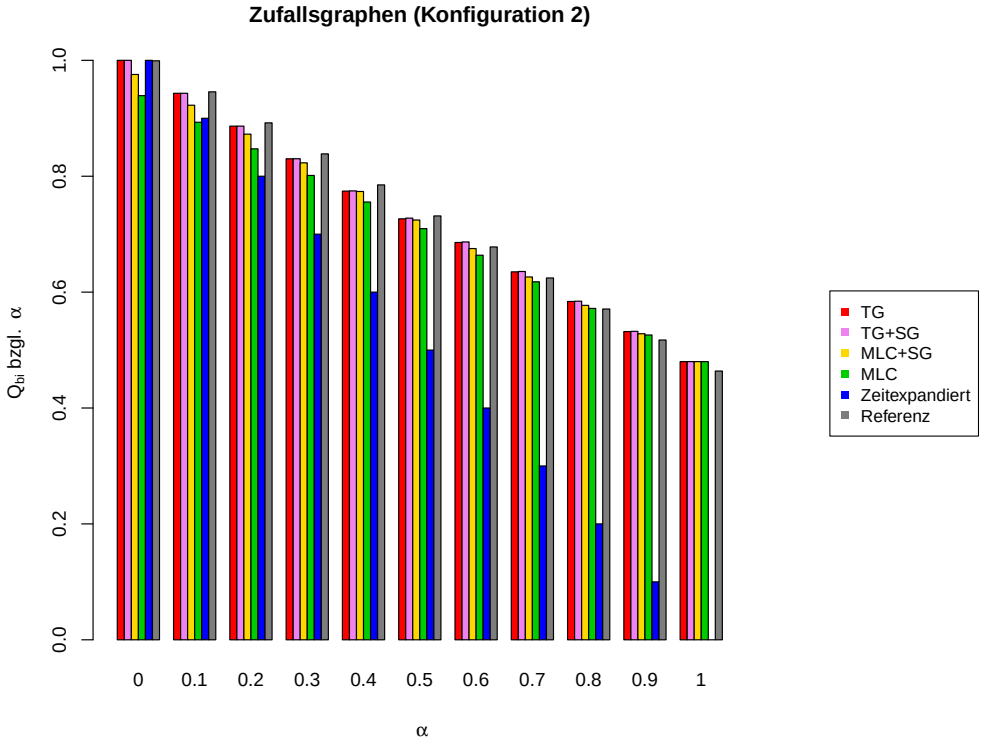


Abbildung 6.11.: Vergleich der Zielfunktionswerte Q_{bi} der zur Evaluierung eingesetzten Algorithmen für die Zufallsgraphen vom Typ 2 (Mittelwerte)

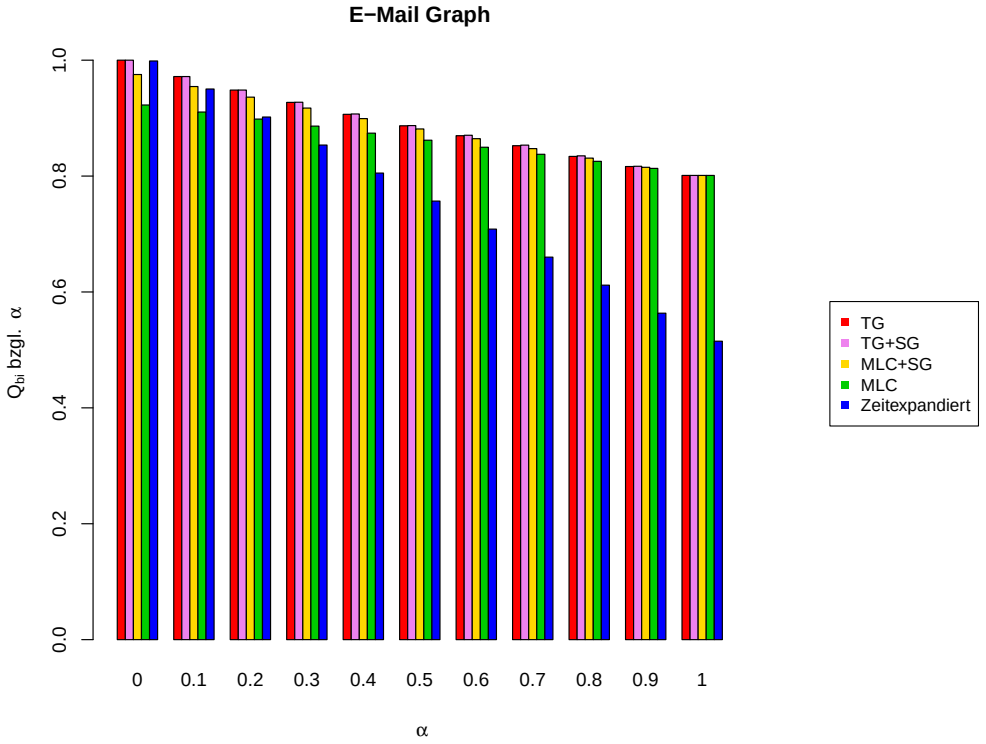


Abbildung 6.12.: Vergleich der Zielfunktionswerte Q_{bi} der zur Evaluierung eingesetzten Algorithmen für den E-Mail Graphen der Fakultät

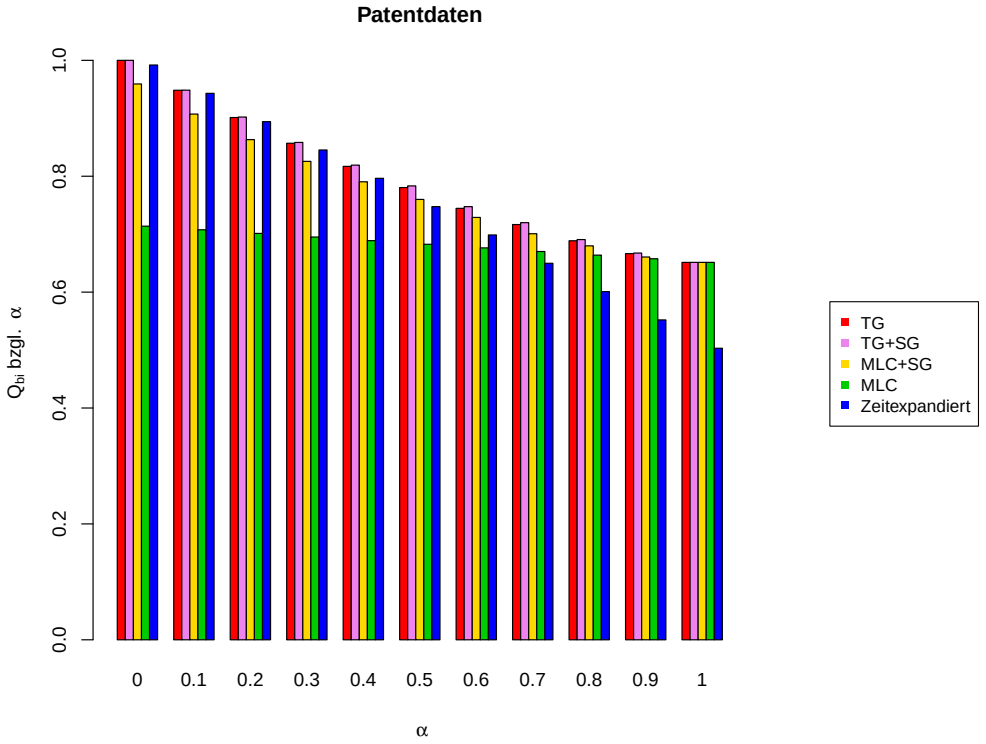


Abbildung 6.13.: Vergleich der Zielfunktionswerte Q_{bi} der zur Evaluierung eingesetzten Algorithmen für die Patentdaten

höheren Zielfunktionswert gegenüber dem Timestep Greedy ohne Verfeinerung. Beide Timestep Greedy Vertreter sind mit Ausnahme von $\alpha = 1$ vom Zielfunktionswert immer besser als die beiden Multi-Level-Clustering Algorithmen.

Für $\alpha = 1$ gilt bei allen dynamischen Graphen, dass die beiden Timestep Greedy Vertreter und die beiden Multi-Level-Clustering Vertreter die selben Zielfunktionswerte besitzen. Dies ist logisch, da in allen Fällen nur nach Modularity geclustert wird. Falls eine Verfeinerung durchgeführt wird, ist damit keine Verbesserung mehr möglich, da der Graph-theoretische Rand-Index gar nicht betrachtet wird und Modularity Snapshot unabhängig ist. Daher hat man bereits vor einer möglichen Verfeinerung die beste Lösung gefunden.

Am meisten bringt die Sequence Greedy Verfeinerung beim Multi-Level-Clustering. Dort erhält man mit Ausnahme $\alpha = 1$ immer bessere Zielfunktionsresultate als das reine Multi-Level-Clustering. Für ein kleineres α sind die Unterschiede am deutlichsten zu erkennen (siehe bei den Patentdaten in Abbildung 6.13). Multi-Level-Clustering mit Sequence Greedy Verfeinerung ist bzgl. Q_{bi} dem puren Modularity-basierten Clustern deutlich vorzuziehen, wenn man Wert auf die Stabilität der Clusterung legt.

Das zeitexpandierte Clustern liefert für alle vier Instanzen im direkten Vergleich zu den beiden Timestep Greedy Vertreter die schlechteren Zielfunktionswerte. Vergleicht man das zeitexpandierte Clustern mit den beiden Multi-Level-Clustering Algorithmen, so ist das zeitexpandierte Clustern nur für ein sehr kleines α konkurrenzfähig. Ansonsten erhält man mit dem Multi-Level-Clustering immer bessere Resultate als mit zeitexpandierten Clustern. Das zeitexpandierte Clustern geht stark in Richtung des Graph-theoretischen Rand-Index und eignet sich daher nur für ein sehr kleines α. Für sehr dichte Graphen wie zum Beispiel die Zufallsgraphen vom Typ 2 ist das zeitexpandierte Clustern nicht geeignet, da dort jeder Snapshot ein eigener Cluster wird. Diesen Sachverhalt erkennt man sehr gut an den Zielfunktionswerten von den Zufallsgraphen vom Typ 2. Mit einem steigenden α sinkt die Zielfunktion und für $\alpha = 1$, wo nur Wert auf Modularity gelegt wird, versagt das zeitexpandierte Clustern total.

Die Referenz Clusterung des Zufallsgenerators existiert nur für die beiden Typen von Zufallsgraphen. Bei den Zufallsgraphen vom Typ 1 erzielt diese bis $\alpha = 0.4$ gute Ergebnisse. Bis dahin ist die Referenz Clusterung sogar auf einem Niveau mit den beiden Timestep Greedy Algorithmen. Bei den Zufallsgraphen vom Typ 2 lohnt sich die Referenz Clusterung für alle α Werte und ist sogar bis $\alpha = 0.5$ besser als die Timestep Greedy Vertreter.

6.5. Vergleich: Clusteranzahl, Knotenbewegungen und Laufzeit

Für die gefundenen Clusterungen der verschiedenen Instanzen werden nicht nur Modularity und der Graph-theoretische Rand-Index protokolliert, sondern auch diverse andere Ergebnisse festgehalten.

Für jeden Algorithmus und jede verwendete Instanz wird die Anzahl der Cluster pro Zeitpunkt protokolliert und daraus für jede Instanz die durchschnittliche Clusteranzahl pro Zeitschritt berechnet. Diese Ergebnisse sind in Abbildung 6.14 dargestellt. Da das zeitexpandierte Clustern und das Multi-Level-Clustering nicht von α abhängig sind, sind diese als waagrechte Linien in die Diagramme integriert. Das Gleiche gilt auch für die Ergebnisse der Referenz Clusterung bei den beiden Typen von Zufallsgraphen.

Bei allen Instanzen haben die drei von α abhängigen Algorithmen für $\alpha = 0$ im Vergleich zu den restlichen α Werten eine deutlich höhere durchschnittliche Clusteranzahl. Dies liegt daran, dass man nur nach dem Graph-theoretischen Rand-Index clustert und man für die Knoten der gemeinsamen Kantenmenge eines Snapshots und seines Vorgängers die

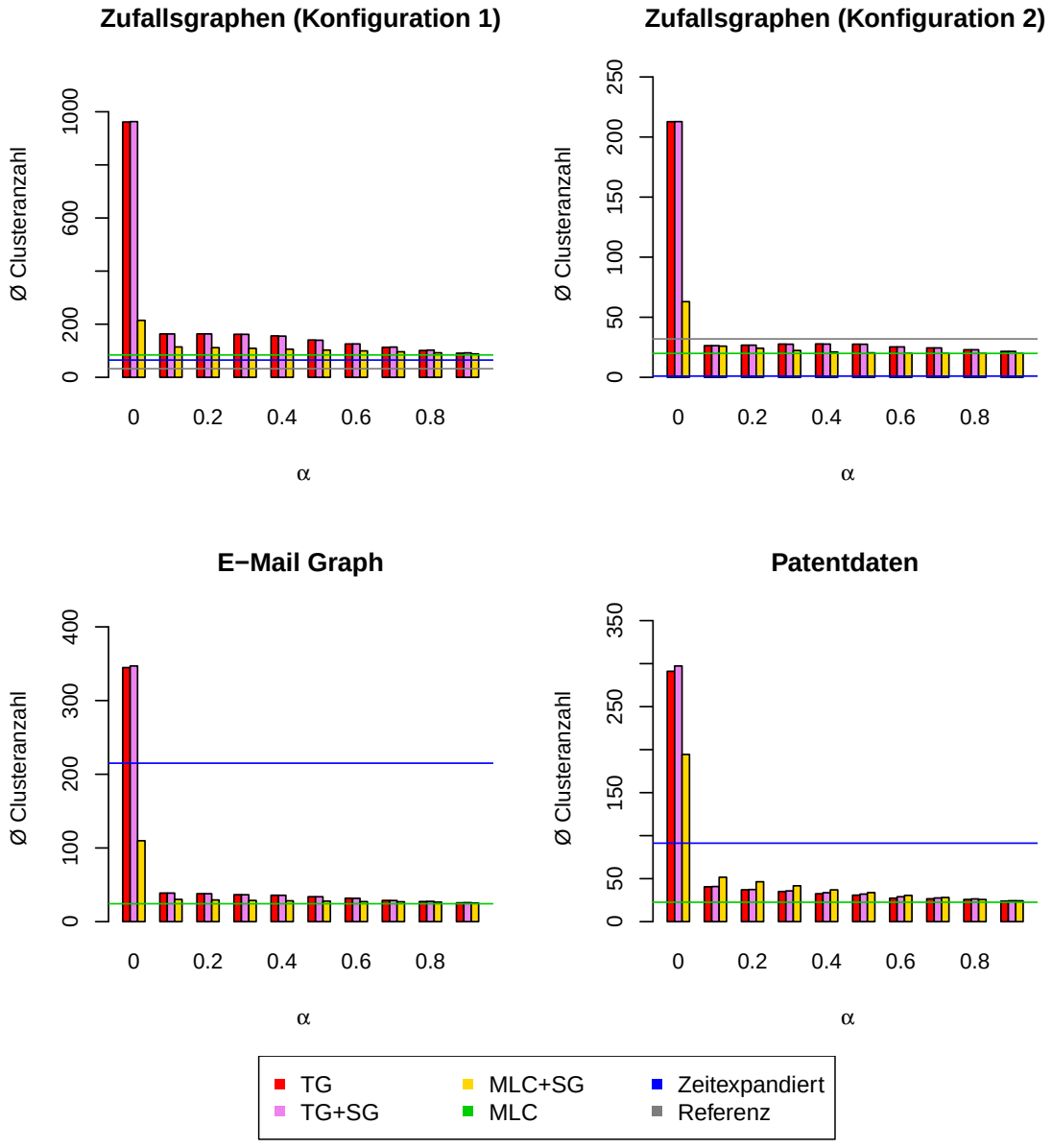


Abbildung 6.14.: Vergleich der durchschnittlichen Clusteranzahl

Clusterungen vom Vorgänger übernimmt. Dies bedeutet, dass alle neu hinzugekommenen Knoten bei den späteren Snapshots Singleton Cluster werden. Für die restlichen α Werte ist die durchschnittliche Clusteranzahl wieder niedriger und bewegt sich für die von α abhängigen Algorithmen auf einem ähnlichen Niveau für die restliche Zeitspanne.

Clustert man nur nach dem Graph-theoretischen Rand-Index, so hat man bekanntlich für den Timestep Greedy Algorithmus einen Zielfunktionswert von 1, da jeder Snapshot einen Graph-theoretischen Rand-Index von 1 mit seinem Vorgänger besitzt. Wie man aber in Abbildung 6.14 erkennt, ist für den Timestep Greedy mit Sequence Greedy Verfeinerung für $\alpha = 0$ die Clusteranzahl höher als für den Timestep Greedy ohne Verfeinerung, trotz dessen, dass sich am Zielfunktionswert nichts ändern kann, da 1 das Maximum darstellt. Schuld daran ist der implementierte Ablauf des Algorithmus. Die zusätzlichen Cluster, die entstehen und somit den Durchschnittswert ansteigen lassen, sind im ersten Snapshot zu finden. Für $\alpha = 0$ wird beim Timestep Greedy Algorithmus nur der erste Snapshot nach Modularity geclustert, der darauf folgende hat bezüglich der gemeinsamen Kantenmenge die selbe Clusterung. Die Knoten der restlichen Kantenmenge sind Singleton Cluster. Der darauf folgende Snapshot wird wieder anhand der gemeinsamen Kantenmenge des Vorgängers geclustert. Jeder betrachtete Knoten des Algorithmus wird immer aus seinem Cluster genommen und erst dann wird überprüft, ob eine Verschiebung zu einem Nachbarn Sinn ergibt, d.h. $\Delta Q_{bi} > 0$. Wenn man nun die Sequence Greedy Verfeinerung anwendet, kommt es zum selben Ablauf. Wird jetzt aber ein Knoten betrachtet, der im nächsten oder vorherigen Zeitpunkt auf keiner der gemeinsamen Kanten zu finden ist, dann gibt es für $\alpha = 0$ keine Verbesserung der Zielfunktion, wenn der Knoten zum Cluster eines Nachbarn wechseln würde. Der Knoten bleibt somit ein Singleton Cluster. Genau das passiert beim ersten Snapshot für alle Knoten, die in einem Cluster mit anderen Knoten sind, aber in keiner Kante der gemeinsamen Kantenmenge von erstem und zweitem Snapshot auftauchen und somit keinen Nachbarknoten besitzen, in dessen Cluster sich ein Wechsel aus Sicht der Zielfunktion lohnen würde. Dies erklärt die höhere Clusteranzahl beim Timestep Greedy mit Verfeinerung gegenüber dem Timestep Greedy ohne Verfeinerung.

Für das zeitexpandierte Clustern und die Zufallsgraphen vom Typ 2 gibt es im Schnitt nur ein Cluster. Dies ist das bereits erwähnte Problem, dass dieser Algorithmus bei so einem dichten Graphen mit vielen Zeitschritten nicht mehr richtig funktioniert.

Bei der Referenz Clusterung erkennt man für beide Typen von Zufallsgraphen, dass sich die durchschnittliche Clusteranzahl im Bereich der eingestellten anfänglichen Clusteranzahl von 32 befindet. Wenn man nur Modularity-basiert clustert, erhält man im Schnitt immer weniger Cluster als die von α abhängigen Algorithmen.

Vergleicht man die durchschnittliche Clusteranzahl der beiden Typen von Zufallsgraphen, erkennt man, dass die Zufallsgraphen vom Typ 2 immer weniger Cluster besitzen als die Zufallsgraphen vom Typ 1. Das liegt wahrscheinlich an der Dichte der Graphen. Die Zufallsgraphen vom Typ 2 besitzen deutlich mehr Kanten pro Zeitpunkt bei einer ähnlicher Knotenanzahl wie die Zufallsgraphen vom Typ 1. Somit existieren in den Zufallsgraphen vom Typ 2 mehr Verbindungen zwischen den Knoten, was bedeutet, dass ein Cluster mehr Knoten im Schnitt besitzt, da viel mehr Knoten durch eine Kante im Zusammenhang stehen als bei den Zufallsgraphen vom Typ 1. Ein anderer Grund für die geringere durchschnittliche Clusteranzahl ist auch, dass durch die vielen Kanten die gemeinsame Kantenmenge aufeinander folgender Snapshots größer ist und daher auch nicht mehr so viele Singleton Cluster wie bei den Zufallsgraphen vom Typ 1 entstehen.

Die Laufzeitmessung der unterschiedlichen Algorithmen ist in Abbildung 6.16 zu finden. Einen deutlichen Einfluss auf die Laufzeit hat die Anzahl der Knotenbewegungen, die die Algorithmen durchführen (siehe Abbildung 6.15). Eine Knotenbewegung bedeutet, dass ein Knoten sein zugehöriges Cluster ändert.

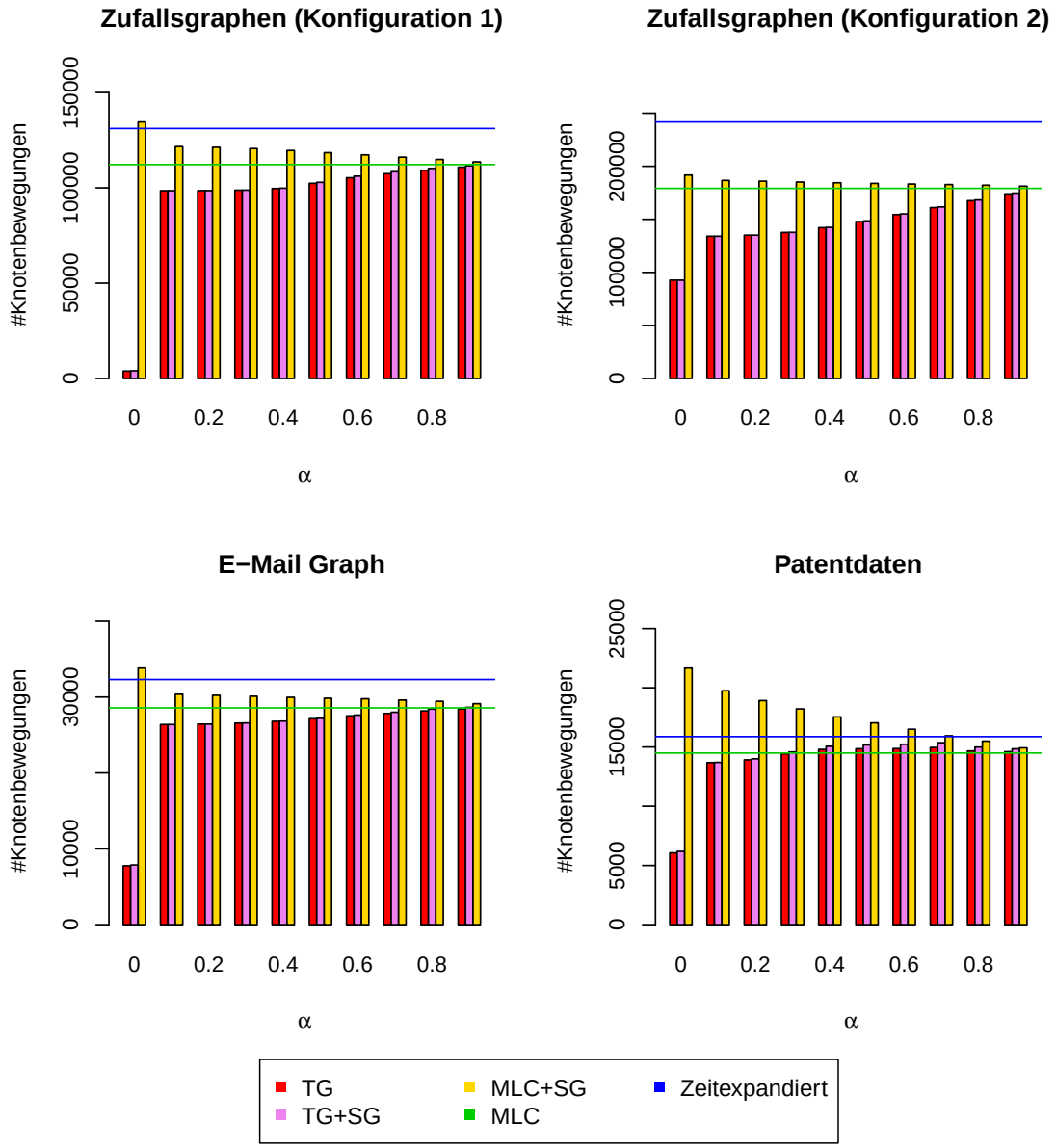


Abbildung 6.15.: Vergleich der Gesamtanzahl an Knotenbewegungen über die gesamte Sequenz

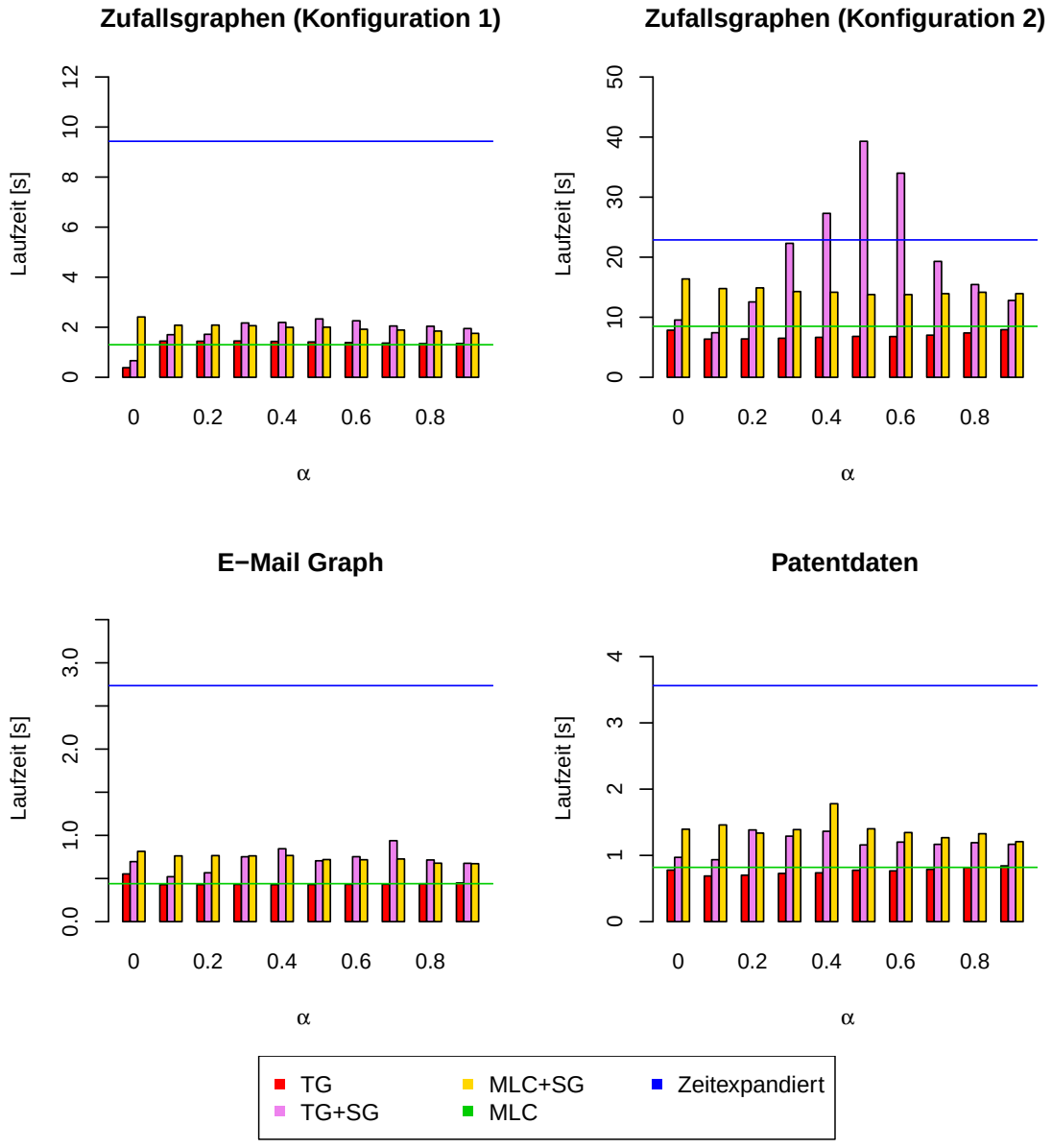


Abbildung 6.16.: Messung der Laufzeit über die gesamte Sequenz

Die beiden Timestep Greedy Algorithmen haben mit Ausnahme der Patentdaten die wenigsten Knotenbewegungen im Vergleich zu den anderen Algorithmen. Ab $\alpha = 0.1$ haben beide Algorithmen deutlich mehr Knotenbewegungen als mit $\alpha = 0$, da man ab diesem α nicht nur die Clusterungen einfach so übernimmt, sondern auch eine gewisse Qualität für die Clusterungen haben möchte. Zudem sind beide Algorithmen immer sehr ähnlich bezüglich ihrer Gesamtanzahl an Knotenbewegungen. Mit der Sequence Greedy Verfeinerung erhält man in manchen Fällen mehr Knotenverschiebungen. Dies erkennt man am besten an den Ergebnissen der Patentdaten aufgrund der feineren Skalierung der y-Achse. Dafür sind die Laufzeitunterschiede der beiden Timestep Greedy Algorithmen auf den Diagrammen deutlicher zu erkennen. Bei den Zufallsgraphen vom Typ 2 fallen diese vor allem im mittleren α-Bereich auf, vor allem aber für $\alpha = 0.5$. Dort hat man mit der Sequence Greedy Verfeinerung fast die 6-fache Laufzeit im Vergleich zum Timestep Greedy Algorithmus ohne Verfeinerung. Die Laufzeitunterschiede machen sich bei größeren und dichteren dynamischen Graphen eher bemerkbar, da eine Knotenverschiebung pro Iteration beim Sequence Greedy ausreicht, um erneut durch alle Knoten und Kanten zu iterieren. Beim E-Mail Graphen mit einem durchschnittlichen Knotengrad von 4 und bei den Zufallsgraphen vom Typ 1 sind diese Laufzeitunterschiede geringer als bei den Zufallsgraphen vom Typ 2 und bei den Patentdaten. Letztere besitzen einen durchschnittlichen Knotengrad von 11.5 und sind damit auch dichter als der E-Mail Graph oder die Zufallsgraphen vom Typ 1.

Beim Multi-Level-Clustering mit Verfeinerung hat man für $\alpha = 0$ die meisten Bewegungen. Mit steigendem α nimmt dann die Anzahl der Knotenbewegungen ab. Dies ist völlig nachvollziehbar, da die Vorberechnung dieses Algorithmus nur auf Modularity basiert und die Verfeinerung für ein kleines α stärker in die andere Richtung, die des Graph-theoretischen Rand-Index, optimiert.

Das zeitexpandierte Clustern ist mit Ausnahme des mittleren α-Bereichs bei den Zufallsgraphen vom Typ 2 von der Laufzeit her der schlechteste Algorithmus. Dabei bezieht sich die Laufzeitmessung hier nur auf das Clustern des zeitexpandierten Graphen. Das Erstellen des zeitexpandierten Graphen wird hier nicht in der Laufzeit berücksichtigt. Dennoch schneidet dieser zeitexpandierte Ansatz von der Laufzeit her sehr schlecht ab, obwohl dieser Algorithmus einfach nur das statische Multi-Level-Clustering eines aggregierten Graphen darstellt.

7. Zusammenfassung und Ausblick

Der Schwerpunkt dieser Arbeit liegt beim Clustern von dynamischen Graphen im Offline-Fall. Die Clusterungen der unterschiedlichen Snapshots des dynamischen Graphen sollen dabei nicht nur von guter Qualität sein, sondern auch eine gewisse Stabilität besitzen. Hierfür wurde die Zielfunktion Q_{bi} definiert, die Modularity und Graph-theoretischen Rand-Index kombiniert. Der Parameter α steuert die Richtung, in die optimiert werden soll. Das Ziel dieser Arbeit ist es, gute Clusterungen bezüglich dieser Zielfunktion zu finden. In [GMS+11] wird diese Zielfunktion in Verbindung mit dem Local Greedy Algorithmus genutzt. Der in dieser Arbeit vorgestellte Timestep Greedy Algorithmus baut auf dieser Idee auf und erweitert diese um eine Verfeinerungsphase. Es besteht zudem die Möglichkeit, nach dem Clustern eines dynamischen Graphen noch nach weiteren Verbesserungen der Zielfunktion zu suchen. Die in dieser Arbeit vorgestellte Sequence Greedy Verfeinerung betrachtet hierfür alle Snapshots zeitgleich. Für diese neuen und alten, bereits bestehenden Heuristiken, wie dem zeitexpandierten Clustern, wurden für unterschiedliche dynamische Graphen Clusterungen gesucht und die Ergebnisse protokolliert. Als verwendete Instanzen kamen neben bereits existierenden Echtweltdaten auch eigene generierte Instanzen zum Einsatz, um eine zielgerichtete Evaluation zu ermöglichen.

Ein Heuristik, mit der Versuche und Vergleiche stattgefunden haben, ist das zeitexpandierte Clustern. Diese Heuristik arbeitet mit der Standardkonfiguration stark in Richtung des Graph-theoretischen Rand-Index. Es hat sich herausgestellt, dass dieser zeitexpandierte Ansatz nicht für sehr dichte Graphen mit vielen Zeitschritten geeignet ist. Deutlich wird dies anhand der Ergebnisse der Zufallsgraphen, bei denen teilweise die Clusterungen der einzelnen Snapshots trivial sind. Um dies zu verhindern, müsste man manuell sehr viel an den verwendeten Parametern verändern.

Multi-Level-Clustering ist ein anderer Algorithmus, der ebenfalls untersucht wurde. Dieser Algorithmus clustert alle Snapshots nur nach Modularity. Wird im Anschluss eine Sequence Greedy Verfeinerung für unterschiedliche Werte von α ausgeführt, so erhält man bemerkenswerte Ergebnisse. Die Sequence Greedy Verfeinerung verbessert den Graph-theoretischen Rand-Index adjazenter Clusterungen deutlich, ohne die Modularity der einzelnen Clusterungen signifikant zu verringern.

Unter allen betrachteten Algorithmen schneidet der Timestep Greedy Algorithmus mit und ohne Sequence Verfeinerung bezüglich Q_{bi} am besten ab, da direkt in Richtung der Zielfunktion gearbeitet wird. Die Sequence Greedy Verfeinerung lohnt sich vor allem für mittlere Werte von α, d.h. Modularity und Graph-theoretischer Rand-Index sind beide von etwa gleicher Bedeutung.

7.1. Ausblick

Die Experimente haben gezeigt, dass die Sequence Greedy Verfeinerung weitere Verbesserungen der Zielfunktion Q_{bi} bewirkt. Diese Verfeinerung arbeitet aber nur auf dem untersten Abstraktionslevel der jeweiligen Snapshots. Bessere Resultate sind mit dem Multi-Level Sequence Greedy Algorithmus durchaus möglich. Dieser Algorithmus wird zwar in Kapitel 5.4 vorgestellt, konnte aber in Rahmen dieser Diplomarbeit nicht mehr evaluiert werden. Die Implementierung und Evaluierung dieses Algorithmus soll Gegenstand zukünftiger Arbeit sein.

Da die Cluster für jeden Zeitpunkt neu aufgebaut werden, können sich deren IDs zu jedem Zeitpunkt ändern. Um die zeitliche Entwicklung eines Clusters zu betrachten ist es daher wünschenswert, möglichst automatisch einen Zusammenhang zwischen alten und neuen IDs herzustellen. Dies kann durch eine individuelle Färbung jedes Clusters bewerkstelligt werden. Ein Cluster besitzt dabei in jedem Snapshot, in dem er existiert, dieselbe Farbe. Somit ist es möglich die Entwicklung der Cluster zu verfolgen. Diese Betrachtung soll Gegenstand zukünftiger Arbeiten sein.

Literaturverzeichnis

[BDH+07] Ulrik Brandes, Daniel Delling, Martin Höfer, Marco Gaertler, Robert Görke, Zoran Nikoloski und Dorothea Wagner: *On Finding Graph Clusterings with Maximum Modularity.* In: Andreas Brandstädt, Dieter Kratsch und Haiko Müller (Herausgeber): *Proceedings of the 33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)*, Band 4769 der Reihe *Lecture Notes in Computer Science*, Seiten 121–132. Springer, October 2007. http://dx.doi.org/10.1007/978-3-540-74839-7_12.

[BGLL08] Vincent Blondel, Jean Loup Guillaume, Renaud Lambiotte und Etienne Lefebvre: *Fast unfolding of communities in large networks.* Journal of Statistical Mechanics: Theory and Experiment, 2008(10), 2008. http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[CKT06] Deepayan Chakrabarti, Ravi Kumar und Andrew S. Tomkins: *Evolutionary Clustering.* In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Seiten 554–560. ACM Press, 2006. http://doi.acm.org/10.1145/1150402.1150467.

[CNM04] Aaron Clauset, Mark E. J. Newman und Cristopher Moore: *Finding community structure in very large networks.* Physical Review E, 70(066111), 2004. http://link.aps.org/abstract/PRE/v70/e066111.

[DGG+] Daniel Delling, Marco Gaertler, Robert Görke, Zoran Nikoloski und Dorothea Wagner: *How to Evaluate Clustering Techniques.* Technischer Bericht, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH).

[GHH+11] Robert Görke, Martin Holzer, Olaf Hopp, Johannes Theuerkorn und Klaus Scheibenberger: *Dynamic network of email communication at the Department of Informatics at Karlsruhe Institute of Technology (KIT)*, 2011. i11www.iti.kit.edu/projects/spp1307/emaildata.

[GHW09] Robert Görke, Tanja Hartmann und Dorothea Wagner: *Dynamic Graph Clustering Using Minimum-Cut Trees.* Technischer Bericht, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2009. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000011476, Informatik, Uni Karlsruhe, TR 2009-10.

[Gla08] Dieter Glaser: *Zeitexpandiertes Graphenclustern - Modellierung und Experimente.* Diplomarbeit, Department of Informatics, February 2008. http://i11www.iti.uni-karlsruhe.de/teaching/theses/finished, Diplomarbeit.

[GMS+11] Robert Görke, Pascal Maillard, Andrea Schumm, Christian Staudt und Dorothea Wagner: *Dynamic Graph Clustering Combining Modularity and Smoothness.* Technischer Bericht, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2011. http://digbib.ubka.uni-karlsruhe.

de/volltexte/1000022451, Karlsruhe Reports in Informatics 2011-11 (and invitational submission to a Special Issue of the ACM Journal on Experimental Algorithmics).

[Gör10] Robert Görke: *An Algorithmic Walk from Static to Dynamic Graph Clustering*. Dissertation, Fakultät für Informatik, 2010. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000018288.

[GS09] Robert Görke und Christian Staudt: *A Generator for Dynamic Clustered Random Graphs*. Technischer Bericht, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2009. http://i11www.iti.uni-karlsruhe.de/projects/spp1307/dyngen, Informatik, Uni Karlsruhe, TR 2009-7.

[Laf10] Geraud Oscar Fofie Lafou: *Engineering von Modularity-basiertem Graphenclustern*, 2010. http://i11www.iti.uni-karlsruhe.de/_media/teaching/theses/sa-fofielafou-10.pdf.

[NG04] Mark E. J. Newman und Michelle Girvan: *Finding and evaluating community structure in networks*. Physical Review E, 69(026113), 2004. http://link.aps.org/abstract/PRE/v69/e026113.

[NR09] Andreas Noack und Randolf Rotta: *Multi-level Algorithms for Modularity Clustering*. In: Jan Vahrenhold (Herausgeber): *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, Band 5526 der Reihe *Lecture Notes in Computer Science*, Seiten 257–268. Springer, June 2009. http://www.springerlink.com/content/qugv7708h3806230/.

[Org05] World Intellectual Property Organization: *International Patent Classification (IPC)*, 1986-2005. http://www.wipo.int/classifications/ipc/en/general/.

[TSK06] Pang Ning Tan, Michael Steinbach und Vipin Kumar: *Introduction to Data Mining*. Addison-Wesley, 2006. http://www.pearsonhighered.com/educator/academic/product/0,1144,0321321367,00.html.

[WL98] Prof. D. Wagner und A. Liebers: *Grundlagen: Begriff zu Graphen*, 1998. http://i11www.iti.uni-karlsruhe.de/information/scripts.

[WW07] Silke Wagner und Dorothea Wagner: *Comparing Clusterings – An Overview*. Technischer Bericht 2006-04, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2007. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000011477.

Anhang

A. Programm `graph`: Unterstützte Formate

Anhand des Beispielgraphen in Abbildung A.1 werden die unterstützen Eingabeformate vorgestellt.

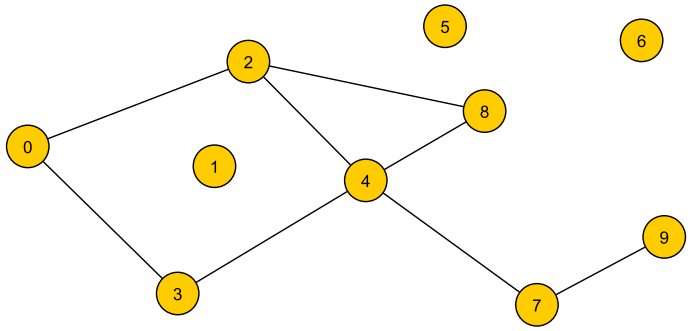


Abbildung A.1.: Beispielgraph zur Verdeutlichung der unterstützen Formate

Das erste Format beinhaltet nur die Kantenliste. In jeder Zeile stehen zunächst die zwei Endknoten, gefolgt vom Kantengewicht. Da der abgebildete Graph ungewichtet ist, wird für jede Kante ein Gewicht von 1 angewendet. Mit diesem Format ist es dennoch möglich, alleinstehende Knoten zu erstellen. Das Programm `graph` wurde so implementiert, dass alle Knoten bis zur höchsten auftretenden Knoten-ID in der Kantenliste hinzugefügt werden. Begonnen wird bei Knoten 0. Leider ist es nicht möglich, einen alleinstehenden Knoten hinzuzufügen, der eine höhere Knoten-ID hat als alle in der Kantenliste vorkommenden Knoten.

Format 1 Kantenliste

```
0 2 1
0 3 1
2 4 1
2 8 1
3 4 1
4 7 1
4 8 1
7 9 1
```

Deswegen werden zwei weitere Eingabeformate unterstützt. Beide Formate unterscheiden sich nur bei den Zeilen mit den Knoten voneinander. Dort ist es möglich, eine anfängliche Cluster-ID für jeden Knoten mit anzugeben. Wird dies nicht getan, so bildet jeder Knoten ein eigenen Cluster. Das Format mit der Cluster-ID Angabe wird hier nicht explizit aufgeführt.

Format 2 Knoten- und Kantenliste

```
0
1
2
3
4
5
6
7
8
9
0 2 1
0 3 1
2 4 1
2 8 1
3 4 1
4 7 1
4 8 1
7 9 1
```

A.1. Format des zeitexpandierten Graphen

Das Einleseformat eines zeitexpandierten Graphen ist so aufgebaut, dass zunächst in jeder Zeile alle Knoten mit Zusatzinformation stehen, gefolgt von der Kantenliste, die auch die intertemporären Kanten enthält. Eine Knotenzeile sieht dann so aus:

```
0 ID 0 t 0
```

An erster Stelle steht die Knoten-ID, die sich auf den kompletten Graphen bezieht. Diese ID gibt es kein zweites Mal. Sie wird benötigt, um ein und denselben Knoten zu unterschiedlichen Zeitpunkten voneinander unterscheiden zu können. Die Kantenliste wird anhand dieser IDs dargestellt. Danach folgt ein String ”ID” und die Knoten-ID, die der Knoten im aktuell betrachtenden Snapshot besitzt. Die letzten zwei Einträge sind ein String ”t” und der Zeitpunkt des aktuell betrachtenden Knotens.

B. DCR-Generator

B.1. Binärdatei als Ausgabeformat

In den ersten vier Bytes der Binärdatei steht die Anzahl der Operationen. Die darauf folgenden vier Bytes beinhalten die Anzahl der Argumente. Danach folgen alle Operationen mit jeweils einem Byte pro Operation und die Argumente mit jeweils vier Bytes.

Der DCR-Generator unterscheidet zwischen zwei Arten von Clusterungen. Die normale Clusterung, auch als *ground-truth* bezeichnet, ist die Clusterung, mit der der Generator arbeitet, also zum Beispiel das Einfügen von Kanten mir einer gewissen Wahrscheinlichkeit. Die *Referenz Clusterung* ist die Clusterung, von der der Generator meint, dass diese ein guter Clustering-Algorithmus finden sollte. Sei $id(u)$ die Knoten-ID von Knoten u, $id(C)$ die Cluster-ID und $id(C_{ref})$ die Referenz Cluster-ID. Mit Hilfe dieser Informationen können alle möglichen Operationen, die in der Binärdatei stehen, beschrieben werden. Es gibt insgesamt sieben Operationen, die in der Tabelle B.1 zu finden sind. Die Knoten-ID beim

Operation	Argument 1	Argument 2
1 - Knoten hinzufügen	$id(C)$	$id(C_{ref})$
2 - Knoten u löschen	$id(u)$	-
3 - Kante $\{u, v\}$ hinzufügen	$id(u)$	$id(v)$
4 - Kante $\{u, v\}$ löschen	$id(u)$	$id(v)$
5 - Cluster von u setzen	$id(u)$	$id(C)$
6 - Referenz-Cluster u setzen	$id(u)$	$id(C_{ref})$
7 - nächster Zeitschritt	-	-

Tabelle B.1.: Mögliche Optionen in der Binärdatei des Generators

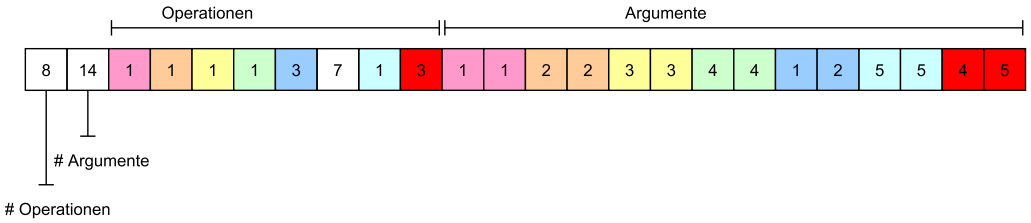


Abbildung B.2.: Beispiel für den Inhalt einer Binärdatei in Dezimalform

Hinzufügen eines Knotens beginnt bei 1 und wird für jeden neuen Knoten um eins erhöht. Die IDs von gelöschten Knoten werden nicht erneut vergeben.

Das Beispiel in Abbildung B.2 zeigt den Inhalt einer solchen Binärdatei in Dezimalform. In diesem Beispiel gibt es insgesamt 8 Operationen und 14 Argumente. Die zueinander gehörenden Argumente und Operationen sind jeweils mit derselben Farbe gekennzeichnet. Es werden zunächst die Knoten 1 bis 4 und die Kante $\{1, 2\}$ in den dynamischen Graphen hinzugefügt. Operation 7 signalisiert einen Zeitschritt. Im Anschluss daran werden der Knoten 5 und die Kante $\{4, 5\}$ in den Graphen eingefügt.

Ziel ist es, diese in der Binärdatei hinterlegten Informationen zu verwerten und daraus den entsprechenden dynamischen Graphen mit seinen einzelnen Snapshots zu erstellen. Im nächsten Abschnitt wird ein Algorithmus vorgestellt, der solche Binärdateien verarbeitet und daraus dynamische Graphen erstellt.

B.2. Dynamischen Graphen bauen

Der DCR-Generator erstellt einen dynamischen Graphen und speichert diesen in einer Binärdatei ab. Algorithmus 6 verarbeitet diese Binärdatei. Zunächst werden die ersten zwei Integer aus der Binärdatei eingelesen und in den Variablen *num_ops* (Anzahl der Operationen) und *num_args* (Anzahl der Argumente) gespeichert. Mit Hilfe dieser zwei Variablen werden zwei Arrays erstellt, die mit dem entsprechenden Inhalt der Binärdatei gefüllt werden. Das Array *operations* beinhaltet alle Operationen und im Array *arguments* werden die dazugehörigen Argumente gespeichert. Diese zwei Arrays werden in Algorithmus 7 parallel zueinander durchlaufen und dabei wird stückweise der dynamische Graph erstellt. Die Funktion *Graph*, die einen leeren Graphen erstellt, wird vor dem Durchlaufen der Arrays für den ersten Snapshot angewendet. Der dynamische Graph wird gemäß Tabelle B.1 aufgebaut. In dieser Arbeit ist nur die Referenz Clusterung wichtig, da sie als Grundlage zum Vergleich mit anderen Clustering-Algorithmen dient. Da der Generator nur ungewichtete Graphen erstellt, erhält jede erstellte Kante zusätzlich ein Gewicht von 1. Dies wird im Folgenden als ungewichteter Fall bezeichnet, jedoch lässt sich mittels dieser Kantengewichte die bisher getroffene Notation weiterhin verwenden. Nach jedem Zeitschritt wird

Algorithm 6 BinParser(binFile)

Input: binFile

```
1: binFile.open()
2: num_ops ← binFile.read(Integer)
3: num_args ← binFile.read(Integer)
4: operations ← new Byte[numOps]
5: arguments ← new Integer[numArgs]
6: for i = 0 → (num_ops − 1) do
7:    operations[i] ← binFile.read(Integer)
8:    i ← i + 1
9: end for
10: for i = 0 → (num_args − 1) do
11:    arguments[i] ← binFile.read(Byte)
12:    i ← i + 1
13: end for
14: binFile.close()
```

Output: $num\_ops, operations, arguments$

der aktuelle Snapshot G^t gespeichert und der nächste Snapshot G^{t+1} vorbereitet, der zu Beginn des Zeitpunktes $t + 1$ identisch zu G^t ist. Algorithmus 7 erstellt einen Snapshot nach dem anderen und baut somit den gewünschten dynamischen Graphen auf.

Es existiert ebenfalls noch eine gewichtete Version von Algorithmus 7, die sich nur darin unterscheidet, dass jede neu hinzugefügte Kante anstelle eines Gewichts von 1 ein zufälliges Gewicht vom Typ Double im Bereich $[1, 100]$ erhält. Außerdem wird nach jedem Zeitschritt $t \to t + 1$ durch alle Kanten von G^{t+1} iteriert und das Kantengewicht so modifiziert, dass das neue Kantengewicht dem alten Gewicht oder dem alten Gewicht multipliziert $\frac{5}{6}$ bzw. $\frac{7}{6}$ entspricht. Diese Faktoren wurden so gewählt, dass der neue Mittelwert der Kantengewichte des betrachtenden Snapshots dem alten Mittelwert ähnelt. Diese drei möglichen Fälle treten gleichverteilt auf. Dies sorgt für eine zusätzliche Dynamik bei den Kantengewichten.

Der Konverter in Algorithmus 7 ist in `C++` geschrieben. Es gibt insgesamt drei einstellbare Programmoptionen. Mit der Option `-d [Ausgabeordner]` wird der Ausgabeordner festgelegt, in dem der dynamische Graph gespeichert werden soll. Jeder Snapshot wird als eigene Textdatei gespeichert. Weglassen dieser Option sorgt dafür, dass der Default Ordner verwendet wird. Dieser muss zuvor in der Programmdatei eingetragen werden. Die Option `-r [Binärdatei]` erstellt aus einer Binärdatei einen ungewichteten dynamischen Graphen und die Option `-w [Binärdatei]` macht dies analog für den gewichteten Fall. Eine von diesen beiden Optionen ist verpflichtend. Ein beispielhafter Aufruf könnte wie folgt aussehen:

```
./binconvert -d /home/user/DynGraph/ -r /home/user/binary.graphj
```

B.3. Zeitexpandierten Graphen erstellen

Die Binärdatei des Generators kann auch zum Erstellen von zeitexpandierten Graphen benutzt werden. Algorithmus 8 erhält eine generierte Binärdatei und baut daraus einen zeitexpandierten Graphen. Dieser Algorithmus ähnelt Algortihmus 7 im Ablauf. Anstelle der einzelnen Snapshots $G^0, \ldots, G^{t_{max}}$ werden hier nur insgesamt zwei Graphen G_{now} und G_{TE} verwendet. G_{now} ist der aktuelle vom Generator erstellte Snapshot und G_{TE} ist der zeitexpandierte Graph, der mit Hilfe von G_{now} stückweise aufgebaut wird. Da die Referenz Clusterung des Generators nicht für zeitexpandierte Graphen konzipiert wurde, fällt diese

Algorithm 7 binConverter(binFile)

Input: binFile

1: $(num\_ops, operations, arguments) \leftarrow$ binParser(binFile)
2: $t \leftarrow 0$
3: $G^t \leftarrow$ Graph()
4: $j \leftarrow 0$
5: **for** $i = 0 \rightarrow (num\_ops - 1)$ **do**
6: **if** $operations[i] = 1$ **then**
7: add_vertex(G^t) with Index $arguments[j]$ and Cluster-ID $arguments[j+1]$
8: $j \leftarrow j + 2$
9: **else if** $operations[i] = 2$ **then**
10: remove_vertex($arguments[j], G^t$)
11: $j \leftarrow j + 1$
12: **else if** $operations[i] = 3$ **then**
13: add_edge($arguments[j], arguments[j+1], G^t$)
14: $\omega(arguments[j], arguments[j+1], G^t) = 1$
15: $j \leftarrow j + 2$
16: **else if** $operations[i] = 4$ **then**
17: remove_edge($arguments[j], arguments[j+1], G^t$)
18: $j \leftarrow j + 2$
19: **else if** $operations[i] = 5$ **then**
20: $j \leftarrow j + 2$
21: **else if** $operations[i] = 6$ **then**
22: change_cluster($arguments[j], G^t$) to $arguments[j+1]$
23: $j \leftarrow j + 2$
24: **else if** $operations[i] = 7$ **then**
25: $t \leftarrow t + 1$
26: $G^t \leftarrow G^{t-1}$
27: **end if**
28: $i \leftarrow i + 1$
29: **end for**

Output: $\mathcal{G} = \{G^0, \ldots, G^{t_{max}}\}$

im Algorithmus 8 weg. Operation 1 fügt nur noch Knoten hinzu ohne eine Cluster-ID zu setzen und Operation 5 und 6 spielen hier keine Rolle mehr. Der große Unterschied zu Algorithmus 7 ist aber, dass nach jedem Zeitschritt (Operation 7) und ganz zu Schluss die Funktion *buildTEG* aufgerufen wird, mit deren Hilfe der zeitexpandierte Graph aufgebaut wird.

Die Funktion *buildTEG* wird anhand des Beispiels der Abbildung B.3 erklärt. Auf der linken Seite der Abbildung befindet sich G_{now} und auf der rechten Seite G_{TE}. Zum Zeitpunkt $t = 0$ besteht G_{now} aus vier Knoten und vier Kanten. Es folgt der erste Zeitschritt und somit auch der erste Aufruf von *buildTEG*. Beim ersten Aufruf werden die Knoten und Kanten von G_{now} 1 zu 1 in den zeitexpandierten Graphen G_{TE} kopiert. Zum Zeitpunkt $t = 1$ finden dann minimale Änderungen an G_{now} statt. Knoten 5 und die Kante $\{3, 5\}$ sind neu hinzugekommen. Es erfolgt ein erneuter Zeitschritt, welcher bewirkt, dass die aktuelle Ausprägung von G_{now} zum zeitexpandierten Graphen hinzugefügt wird. Da der zeitexpandierte Graph bereits Knoten besitzt, muss für jeden neu hinzugefügten Knoten in G_{TE} überprüft werden, ob dieser Knoten zu einem vorherigen Zeitpunkt schon in G_{TE} eingefügt wurde. Damit dies ermöglicht werden kann, speichert jeder Knoten des zeitexpandierten Graphen noch zusätzlich intern den Zeitpunkt des Hinzufügens in den dynamischen Graphen ab. Dadurch ist es auch möglich ein und denselben Knoten zu unterschiedlichen

Algorithm 8 binToTEG(binFile)

Input: binFile

1: $(num\_ops, operations, arguments) \leftarrow$ binParser(binFile)
2: $G_{now} \leftarrow$ Graph()
3: $G_{TE} \leftarrow$ Graph()
4: $j \leftarrow 0$
5: **for** $i = 0 \rightarrow (num\_ops - 1)$ **do**
6: **if** $operations[i] = 1$ **then**
7: add_vertex(G_{now}) with Index $arguments[j]$
8: $j \leftarrow j + 2$
9: **else if** $operations[i] = 2$ **then**
10: remove_vertex($arguments[j], G_{now}$)
11: $j \leftarrow j + 1$
12: **else if** $operations[i] = 3$ **then**
13: add_edge($arguments[j], arguments[j + 1], G_{now}$)
14: $\omega(arguments[j], arguments[j + 1], G^t) = 1$
15: $j \leftarrow j + 2$
16: **else if** $operations[i] = 4$ **then**
17: remove_edge($arguments[j], arguments[j + 1], G_{now}$)
18: $j \leftarrow j + 2$
19: **else if** $operations[i] = 5$ or $operations[i] = 6$ **then**
20: $j \leftarrow j + 2$
21: **else if** $operations[i] = 7$ **then**
22: $G_{TE} \leftarrow$ buildTEG(G_{now})
23: **end if**
24: $i \leftarrow i + 1$
25: **end for**
26: $G_{TE} \leftarrow$ buildTEG(G_{now})

Output: G_{TE}

zeitlichen Ausprägungen voneinander zu unterscheiden. Wurde so ein Knoten also schon zuvor in G_{TE} eingefügt, so wird eine intertemporäre Kante zwischen diesen Knoten eingefügt. In dieser Arbeit werden nur intertemporären Kanten über genau einen Zeitschritt hinweg betrachtet. Im Beispiel der Abbildung B.3 wird für die Knoten 1 bis 4 beim zweiten Aufruf von *buildTEG* jeweils eine intertemporäre Kante hinzugefügt. Die aktuell hinzugefügten intertemporären Kanten sind rot dargestellt. Dieser Vorgang wird nach jedem Zeitschritt ausgeführt. Nach dem letzten Zeitschritt kann es noch zu Änderungen am aktuellen Graphen kommen. Sobald keine Änderungen mehr an G_{now} durchgeführt werden, wird die letzte zeitliche Ausprägung dieses Graphen zum zeitexpandierten Graphen hinzugefügt. Im Beispiel der Abbildung B.3 gibt es insgesamt zwei Zeitschritte. Nach dem zweiten Zeitschritt wird der Knoten 3 und alle inzidenten Kanten gelöscht und der dritte und letzte Aufruf von *buildTEG* sorgt für die Fertigstellung des zeitexpandierten Graphen.

Für den gewichteten Fall gilt wiederum, dass jede neu hinzugefügte Kante ein zufälliges gleichverteiltes Gewicht im Bereich von $[0, 100]$ erhält. Zudem wird nach jedem Zeitschritt durch alle Kanten des aktuelle Graphen G_{now} iteriert und die Kantengewichte entsprechend des gewichteten Falles von Algorithmus 7 modifiziert.

Am Gewicht der intertemporären Kanten werden niemals Änderungen durchgeführt. Dieses Gewicht wird gemäß Kapitel 4.1 beim Erstellen dieser Kanten festgelegt. Für den ungewichteten Fall gilt $x = 1$ und für den gewichteten Fall ist $x = 50$.

Algorithmus 8 ist wie Algorithmus 7 in C++ geschrieben und bietet die gleichen Optionen.

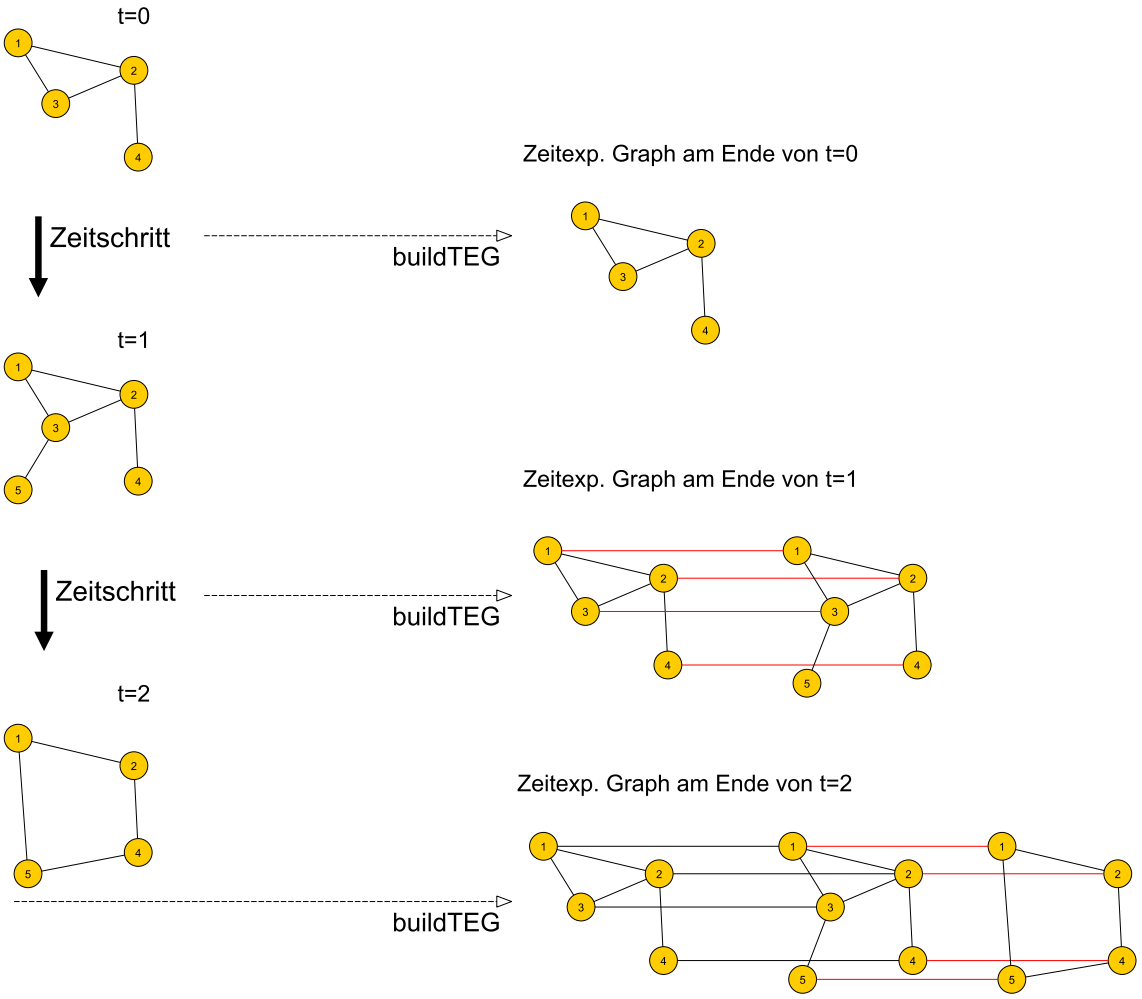


Abbildung B.3.: Beispielhafter Ablauf bei einer Konvertierung einer Binärdatei zu einem zeitexpandierten Graphen

Ein möglicher Aufruf könnte so aussehen:

```
./teconvert -d /home/user/DynGraph/ -r /home/user/binary.graphj
```

C. Weitere Versuchsergebnisse

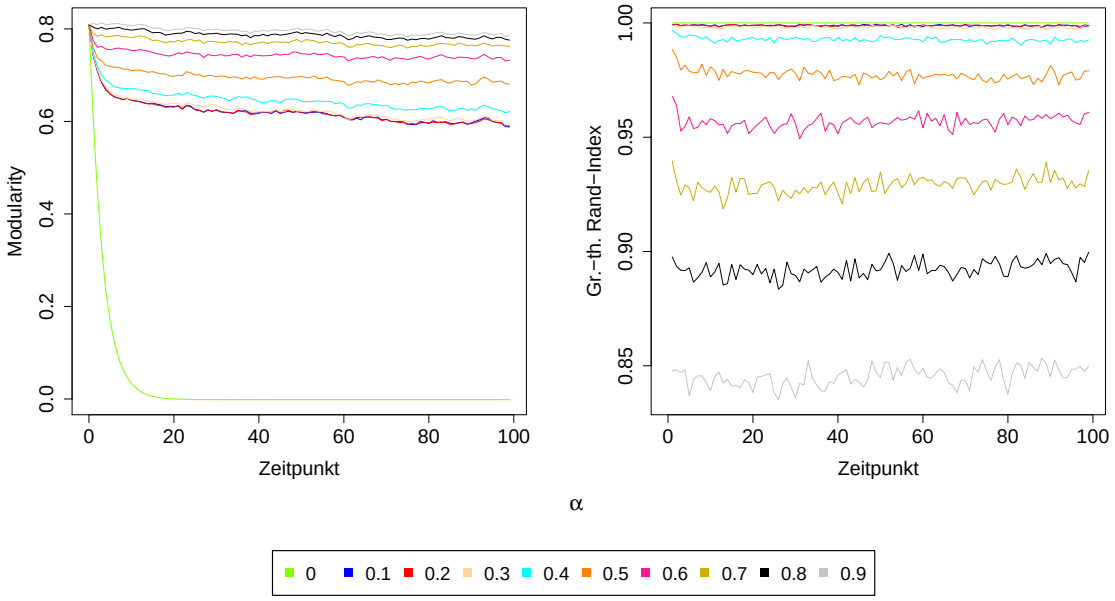


Abbildung C.4.: Clustern der Zufallsgraphen vom Typ 1 mittels Timestep Greedy Algorithmus (Mittelwerte)

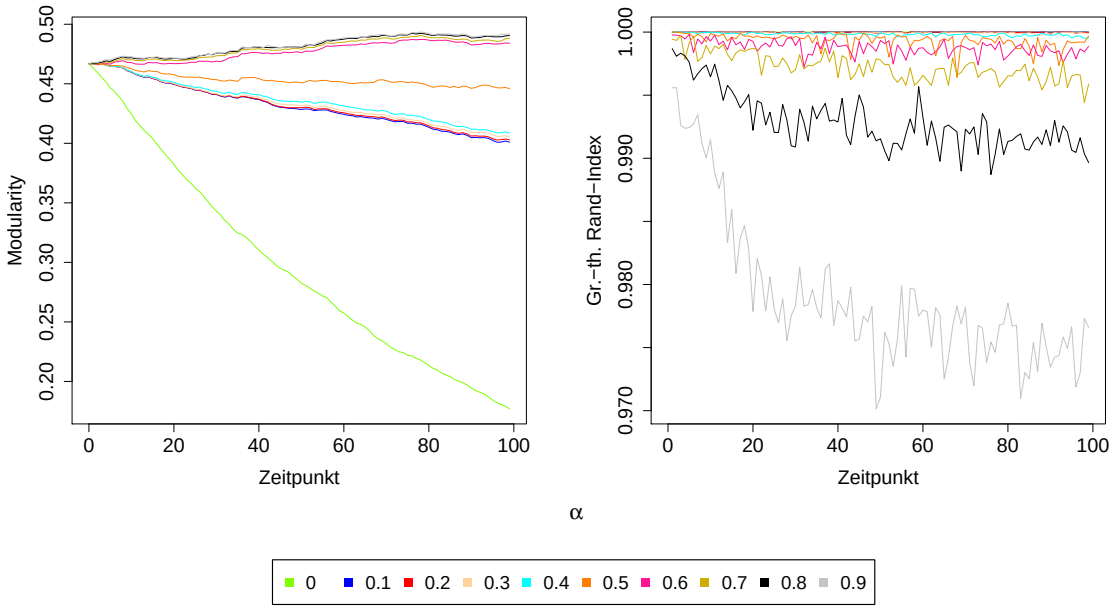


Abbildung C.5.: Clustern der Zufallsgraphen vom Typ 2 mittels Timestep Greedy Algorithmus (Mittelwerte)

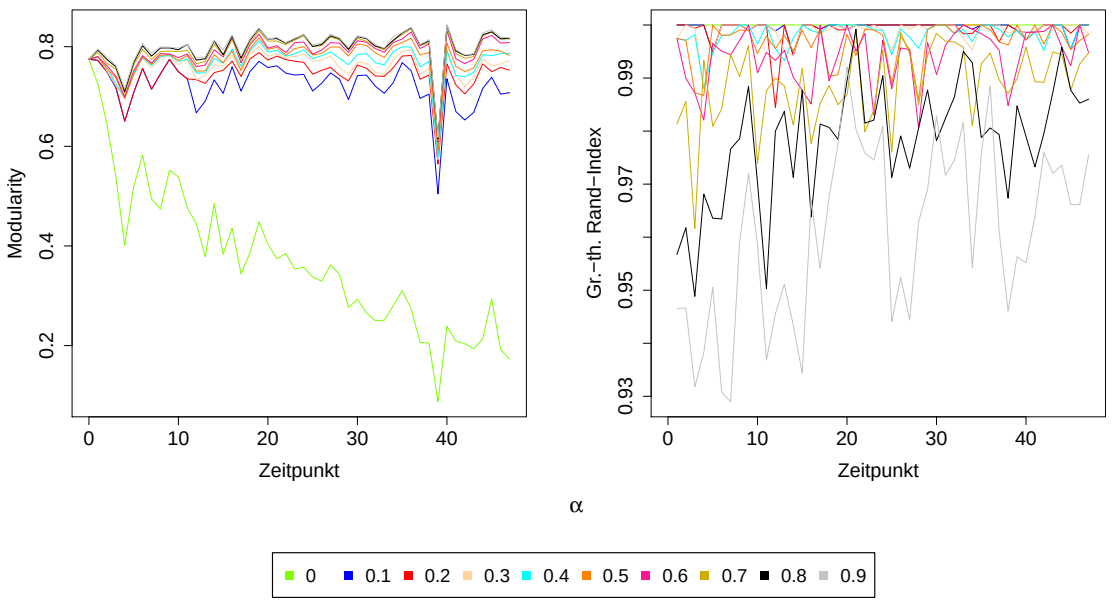


Abbildung C.6.: Clustern des E-Mail Graphen mittels Timestep Greedy Algorithmus

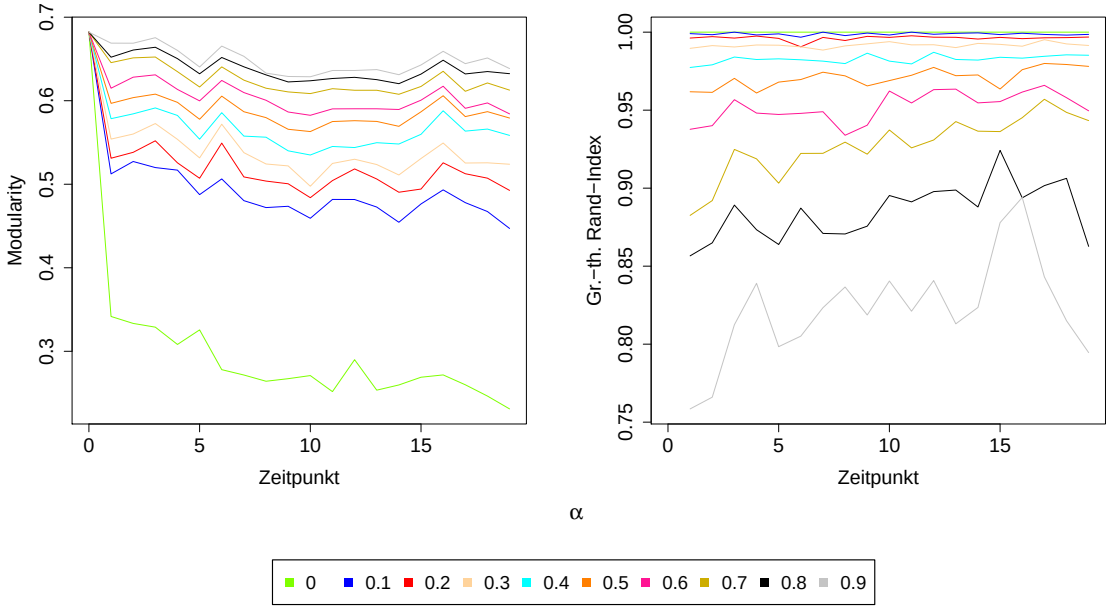


Abbildung C.7.: Clustern der Patentdaten mittels Timestep Greedy Algorithmus

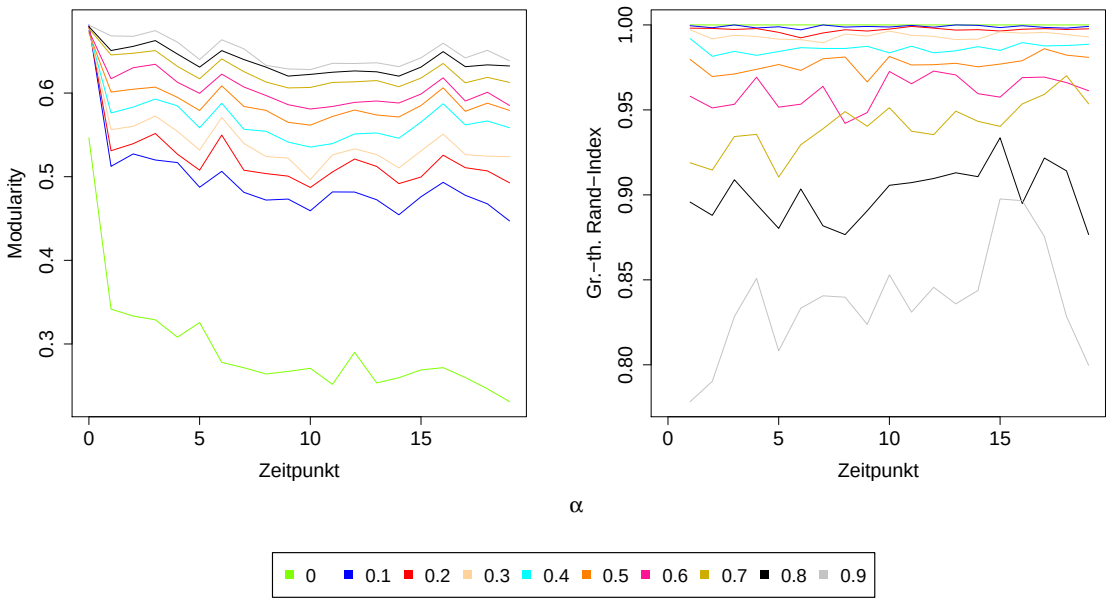


Abbildung C.8.: Timestep Greedy Algorithmus und Sequence Greedy Verfeinerung der Patentdaten

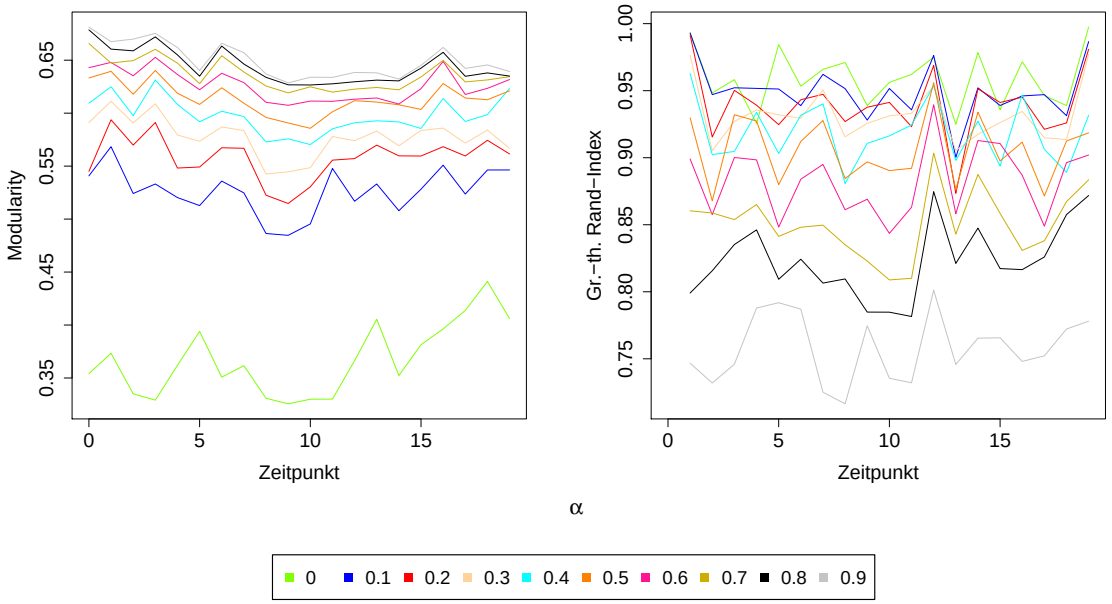


Abbildung C.9.: Modularity-basiertes Clustern und anschließende Sequence Greedy Verfeinerung der Patentdaten

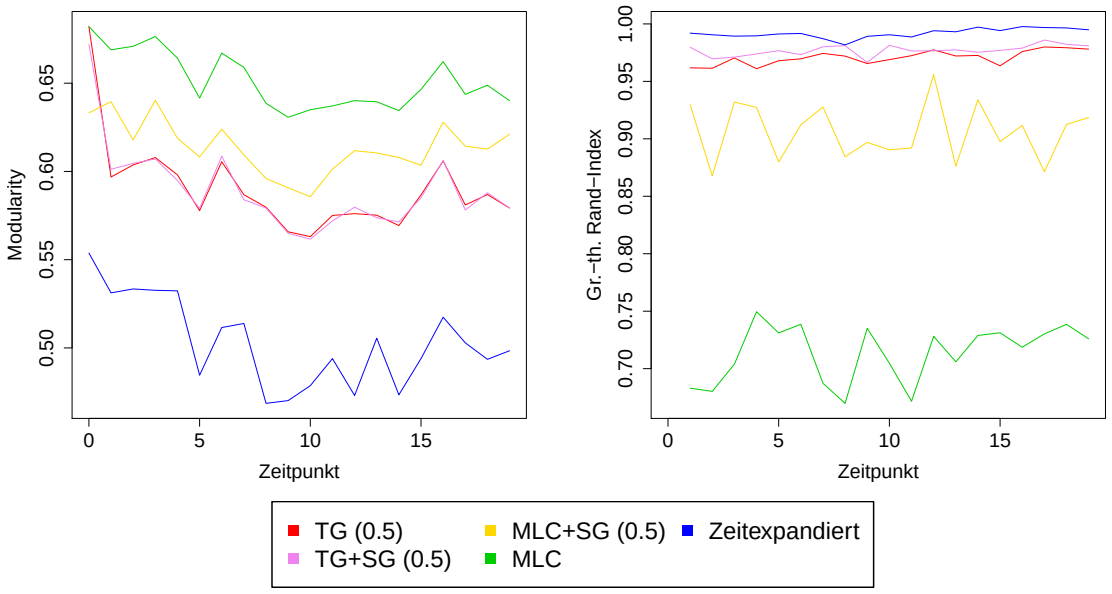


Abbildung C.10.: Patentdaten: Vergleich der verschiedenen Algorithmen