# AGT Summary

July 3, 2024

# Contents

# 1 Netzwerke und Zentralität

## 1.1 Charakterisierung der wichtigsten Ecke

Hierfür gibt es mehrere Möglichkeiten

- größter Einfluss
- wichtig für Informationsfluss

Die Wichtigkeit wird mit einem Zentralitätsmaß gemessen.

**Definition 1.1.1.** Zentralitätsmaße sind sehr unterschiedlich. Es muss nur erfüllt sein, dass bei einem Sterngraphen das Zentrum das größte Zentralitätsmaß erhält. Möglich sind Bewertungen nach

1. dem Maximalgrad (*degree centrality*)
2. der durchschnittlichen Entfernung zu anderen Ecken (*closeness centrality*) (bzw. der Kehrwert davon)
3. der Anzahl der Komponenten, die mit dieser Ecke verbunden sind (*betweenness centrality*). Dafür sei $\sigma_{s,t}$ die Anzahl der kürzesten $s-t$-Wege. $\sigma_{s,t}(v)$ für $v \neq s,t$ ist dann die Anzahl der kürzesten $s-t$-Wege, die durch $v$ gehen. Damit gilt

$$betweenness(v) = \sum_{s,t \in V()G \setminus \{v\}} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

## 1.2 Berechnung der Zentralitätsmaße

Wir führen nur die Berechnung der betweenness ein. Die anderen beide Maße sind sehr einfach.
Der Algorithmus zur Berechnung von $\sigma_{s,t}$ ist an Dijkstra angelehnt. Beginnend mit $s$ wird die Anzahl der Nachbarn von $s$ bestimmt. Anschließend die Anzahl der Knoten mit Abstand 2 usw. Um die Komplexität der Algorithmen zu bestimmen, werden im Folgenden einige Annahmen getroffen:

1. Kotenadjazenz kann in $\mathcal{O}(1)$ bestimmt werden
2. Kanteninzidenz kann in $\mathcal{O}(1)$ bestimmt werden
3. die Nachbarschaft eines Knoten wird in $\mathcal{O}(1)$ pro Knoten bestimmt
4. die zu einem Knoten inzidenten Kanten können in $\mathcal{O}(1)$ pro Kante bestimmt werden

5. alle elementaren Operationen (z.B. Kante löschen) in $\mathcal{O}(1)$.

Auf diese Weise kann man leicht sehen, dass die Laufzeit zur Berechnung von $\sigma_{s,t}$ für alle $s, t$ in $\mathcal{O}(n \cdot m)$ implementiert werden kann. Wir nehmen nun an, $\sigma_{s,t}$ sei bekannt und wir definieren

$$\rho_s(v) = \sum_{t \neq v} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

Kennt man nun alle $\rho_s(v)$, dann ist

$$betweenness(v) = \frac{1}{2} \sum_{s \neq v} \rho_s(v)$$

**Lemma 1.2.1.** *Sei $v$ ein Knoten mit Distanz mindestens $d \geq 1$ zu $s$ und sei $L$ die Menge der Knoten mit Distanz $d+1$ zu $s$. Dann ist*

$$\rho_s(v) = \sum_{w \in L \cap N(v)} \frac{\sigma_{s,v}}{\sigma_{s,w}} (1 + \rho_s(w)$$

Mit dieser Überlegung lässt sich ein Algorithmus finden, der die *betweenness* jedes Knotens in $\mathcal{O}(mn)$ berechnet.

## 1.3 Random Walks auf Graphen

Wir wählen zunächst einen Startknoten $v_0$ bezüglich einer Wahrscheinlichkeitsverteilung $\pi^{(0)}$. Anschließend wird mit Gleichverteilung ein zufälliger Nachbar $v_1$ von $v_0$ gezogen usw. Wenn man sich nun die Frage stellt, was die Wahrscheinlichkeit ist, dass der erste gezogenen Knoten (d.h. $v_1$) gleich dem Knoten $u$ ist, dann entspricht das der Wahrscheinlichkeit, dass $u$ ein Nachbar von $v_0$ ist mal der Wahrscheinlichkeit, dass anschließend $u$ gezogen wird. Da der zweite Schritt gleichverteilt ist, ergibt sich

$$\pi_u(1) = \sum_{v \in N(u)} \pi_v^{(0)} \cdot \frac{1}{d(v)}$$

Das wird geschrieben als Transition Matrix mit

$$T_{uv} = \begin{cases} \frac{1}{d(v)}, & \text{wenn } uv \in E \\ 0, & \text{sonst} \end{cases}$$

Schreibt man die Wahrscheinlichkeitsverteilung $\pi^{(0)}$ einfach als Vektor, dessen Komponenten zu 1 addieren, ergibt sich

$$\pi^{(n+1)} = T\pi^{(n)}$$

für $n \geq 0$. Um zu überprüfen, ob ein bestimmter Knoten im Random Walk jemals besucht wird, muss die Grenzwertverteilung bestimmt werden

$$\pi^* = \lim_{k\to\infty} T^k \pi^{(0)}$$

Existiert $\pi^*$, dann ist $\pi^k$ eine Cauchy-Folge und man kann leicht sehen, dass $T\pi^* = \pi^*$. Dann ist $\pi^*$ also ein Eigenvektor zum Eigenwert 1 von $T$.

**Satz 1.3.1** (Perron-Frobenius)**.** *Sei $A \in \mathbb{R}^{n\times n}$ sodass $\exists k \in \mathbb{N}$ mit $A^k_{ij} > 0$ für alle $i, j \in [n]$. Dann gibt es einen eindeutigen Eigenwert $\lambda^*$ mit größtem Betrag. Wenn $\lambda^* > 0$ gibt es einen positiven Eigenvektor $v^*$ zu $\lambda^*$ und alle anderen Eigenvektoren zu $\lambda^*$ sind Vielfache von $v^*$. Ist außerdem $\lambda^* = 1$, dann konvergiert $v^{(k+1)} = Av^{(k)}$ gegen ein Vielfaches von $v^*$ für alle positiven Startvektoren $v^{(0)} > 0$.*

Damit kann man sich überzeugen, dass $T$ Eigenwert 1 hat und dass das der größte Eigenwert ist. Außerdem erfüllt $T$ die Eigenschaft aus obigem Theorem, wenn $G$ zusammenhängend und nicht bipartit ist.

### 1.4 Eigenwert Zentralität

Für einen Knoten $v$ verwenden wir wieder die Matrix $T$ und nehmen $\pi^* > 0$ als den Eigenvektor zum Eigenwert 1 mit $||\pi^*|| = 1$. Der Eintrag $\pi^*_v$ ist dann die Eigenwert Zentralität von $v$.
Das Problem dieses Zentralitätsbegriffs ist, dass man den Eigenwert recht leicht erraten kann. Betrachte dazu

$$\overline{\pi}_v = \frac{d(v)}{2\,|E|}\ \forall v \in V$$

Es ist leicht zu sehen, dass dieser Vektor ein Eigenvektor zum Eigenwert 1 ist. Es folgt also, dass wir einen neuen Begriff haben, der aber sehr ähnlich zur *degree centrality* ist. In gerichteten Graphen ist der Begriff ein wenig hilfreicher. Der wichtigste Unterschied ist die modifizierte Matrix $T$ mit

$$T_{vu} = \begin{cases} \frac{1}{d^+(u)}, & \text{wenn es eine Kante von } u \text{ nach } v \text{ gibt} \\ 0, & \text{sonst} \end{cases}$$

Das größere Problem sind Senken (d.h. Knoten $v$ mit ausgehendem Grad $d^+(v) = 0$). Das kann gelöst werden, indem der Prozess neugestartet wird (d.h. eine Kante zu jedem anderen Knoten eingeführt wird).

### 1.5 PageRank

PageRank ist der Suchalgorithmus von Google. Er funktioniert in den folgenden Schritte, die sehr ähnlich zum Eigenwertzentralität sind

1. Wähle unter Gleichverteilung einen Startknoten
2. Mit Wahrscheinlichkeit $1-\alpha$ ($\alpha$ konstant) wähle einen Nachbarn und gehe dorthin.
3. Mit Wahrscheinlichkeit $\alpha$ wähle einen neuen Startknoten.

Die Transitionsmatrix definiert nun eine andere Matrix

$$P = (1-\alpha)T + \frac{\alpha}{n}J$$

wobei $J$ die Matrix mit nur 1 Einträgen ist. Es ergibt sich der Prozess

$$\pi^{(k+1)} = P\pi^{(k)}$$

Da $P$ positiv ist, ergibt Theorem 1.3.1 die Existenz der Grenzwertverteilung $p_v^*$. Es gilt $\text{PageRank}(v) = \pi_v^*$. Da der erste Teil von $P$ *sparse* ist, kann die Iteration relativ effizient durchgeführt werden.

# 2 Clustering

Wie führen zunächst den Begriff des Clustering-Koeffizienten ein. Sei $v \in V$. dann ist

$$C(v) = \frac{|E_G[N(v)]|}{\binom{|N(v)|}{2}}$$

Der durchschnittliche Clustering-Koeffizient ist dann

$$C(G) = \frac{1}{|V|}\sum_{v\in V} C(v)$$

Ein Zufallsgraph mit Kantenwahrscheinlichkeit $p$ hat im Erwartungswert eine Kantendichte $\frac{|E|}{\binom{n}{2}}$ von ungefähr $p$. Der Clustering-Koeffizient ist ebenso ungefähr $p$.

## 2.1 Berechnung des Clustering Koeffizienten

Es ist leicht zu sehen, dass man den Clustering Koeffizienten eines einzelnen Knotens $v$ in $\mathcal{O}(d(v)^2)$ berechnen kann. Um den durchschnittlichen Wert zu bestimmen genügt daher eine Laufzeit von $\mathcal{O}(\sum_{v \in V(G)} d(v)^2)$. Die Summe lässt sich nach oben abschätzen durch $2mn$ wodurch die Laufzeit bei $\mathcal{O}(2mn)$ liegt.
Ist $d(v)$ klein, so ist der Algorithmus sehr effizient, aber ist $d(v) >> \sqrt{m}$ so lässt sich eine Verbesserung erzielen, indem für jede Kante $uw$ überprüft wird, ob $u, v, w$ ein Dreieck bilden. Kombiniert man diese beiden Überlegungen zu einem Algorithmus mittels einer Fallunterscheidung, so erhält man einen Algorithmus zur Berechnung des durchschnittlichen Clustering Koeffizienten mit einer Laufzeit von $\mathcal{O}(m^{\frac{3}{2}})$.
Es gibt außerdem einen randomisierten Ansatz für die Schätzung des durchschnittlichen Clustering Koeffizienten auf Graphen mit Minimalgrad mindestens 2. Hierfür wird zunächst eine Konstante $k \in \mathbb{N}$ festgelegt. Anschließend werden nacheinander $k$ Knoten $v_1, ..., v_k$ zufällig gezogen und aus $N(v_i)$ werden jeweils zwei Nachbarn $u_i, w_i$ zufällig gezogen. Es wird gezählt, wie viele dieser Nachbarn der $k$ Knoten mit $v_i$ ein Dreieck aufspannen und diese Anzahl anschließend durch $k$ geteilt.

**Satz 2.1.1.** *Sei $\varepsilon > 0, \delta > 0$ und $k = \lceil \ln\left(\frac{2}{\delta}\right) / (2\varepsilon^2) \rceil$. Dann hat der Algorithmus eine Laufzeit von $\mathcal{O}(\ln\left(\frac{1}{\delta}\right) / \varepsilon^2 \cdot \ln n)$ und mit Wahrscheinlichkeit mindestens $1-\delta$ unterscheidet sich der berechnete Wert um maximal $\varepsilon$ vom tatsächlichen Wert.*

Sei $G$ ein Graph mit $V(G) = U \dot{\cup} W$. Wir schreiben

$$\partial_G = \{uw \in E(G) : u \in U, w \in W\}$$

Für $A \subset V$ ist $\partial_G(A)$ die Anzahl der Kanten zwischen $A$ und $v \setminus A := B$. Ist $w$ eine Funktion, die jeder Kante ein Gewicht zuordnet, definiere

$$w(F) = \sum_{e \in F} w(e)$$

für alle $F \subset E$.

**Definition 2.1.2.** Die *expansion* von $A$ ist

$$\frac{w(\partial_G(A))}{\min\{|A|, |B|\}}$$

Der *ratio cut* von $A$ ist

$$\frac{w(\partial_G(A))}{|A|\,|B|}$$

## 2.2 Gomory-Hu Clustering

**Definition 2.2.1.** Sei $G$ ein Graph und $w$ die Kantengewichte. Der Gomory-Hu-Baum $T$ für $G$ ist ein Graph mit

- $V(T) = V(G)$
- beim Löschen einer Kante $uv$ aus dem Baum entstehen zwei Komponenten $T_{uv}(u)$ und $T_{uv}(v)$. Für alle $uv \in E(T)$ soll gelten

$$w(\partial_G(T_{uv}(u))) = \min_{X \subseteq V(G): v \notin X \ni u} w(\partial_G(X))$$

Wir definieren darauf aufbauend

$$\lambda(s,t) = \min_{X \subseteq V(G): t \notin X \ni s} w(\partial_G(X))$$

**Lemma 2.2.2.** *Sei $G$ ein Graph mit Kantengewichten $w$ und $T$ ein Gomory-Hu-Baum für $G, w$. Seien $s, t \in V(G), s \neq t$, sei $P$ der st-Weg in $T$ und $uv \in E(P)$ mit*

$$\min_{ab \in E(P)} w(\partial_G(T_{ab}(a))) = w(\partial_G(T_{uv}(u)))$$

*Dann ist $w(\partial_G T_{uv}(u)) = \lambda(s,t)$.*

**Satz 2.2.3.** *Für alle $G, w$ existiert ein Gomory-Hu-Baum. Ein solcher Baum kann in $\mathcal{O}(n\tau)$ berechnet werden. $\tau$ ist dabei die Laufzeit um einen gewichtsminimalen $s-t$ Schnitt für beliebige $s, t$ zu finden.*

Mit diesem Konzept kann nun ein Clustering in den folgenden Schritten gefunden werden

1. füge einen universellen Knoten $t$ mit Kantengewichten $\alpha$ zurück
2. berechne den G-H-Baum $T$
3. gib die Komponenten von $T - t$ als Cluster zurück

**Lemma 2.2.4.** *Wir arbeiten auf einem Graphen $G$ mit Gewichten $w$. Mit dem G-H-clustering erreichen wir ein Cluster $C \subseteq V(G)$. Dann gilt*

$$\frac{w(\partial_G C)}{|V \setminus C|} \leq \alpha$$

**Lemma 2.2.5.** *Teilt man in der Situation von oben das Cluster $C$ noch weiter in $Q, P$, dann gilt*

$$\frac{w(\partial_G(P,Q))}{\min\{|P|,|Q|\}}$$

## 2.3 Berechnung des Gomory-Hu Baums

**Lemma 2.3.1.** *Die Gewichte eines Cuts sind submodular. Für alle $U, W \subseteq V$ gilt*

$$w(\partial U) + w(\partial W) \geq w(\partial(W \cap U)) + w(\partial(U \cup W))$$

**Lemma 2.3.2.** *Seien $s, t \in V(G)$ und sei für $X \subseteq V$ $\partial_G X$ ein minimaler $s-t$-Cut. Nun wird $X$ zu einem neuen Knoten $v_x$ kontrahiert. Wobei mehrfache Kanten als eine Kante mit der Summe der Gewichte eingeführt wird. Sei $p, q \in V \setminus X$ und $U \subseteq V \setminus X$ sodass $\partial_{G/X}(U \cup v_x)$ ein minimaler $p-q$-Cut om $G/X$ ist. Dann ist $\partial_G(U \cup X)$ ein minimaler $p-q$-Cut in $G$.*

**Definition 2.3.3** (Teil GH Baum)**.** Sei $R \subseteq V(G)$. Dann ist ein Baum $T = (R, F)$ mit einer Partition $(C_r)_{r \in R}$ von $R$ ein GH Baum für $R$, wenn

$$\forall uv \in F : \partial_G \left( \bigcup_{r \in V(T_{uv}(u))} C_r \right)$$

Ist $R = V(G)$, dann entspricht der GH-Baum für $R$ dem GH-Baum für $G$.

Mit dieser Überlegung lässt sich der GH-Baum von $G$ rekursiv aufbauen.

## 2.4 ratio cuts

Die Idee ist, dass für eine gegebenes $k \in \mathbb{N}$ eine Partition $C_1, ..., C_k$ berechnet wird, sodass

$$\sum_{i=1}^{k} w(\partial C_i)$$

minimal ist. Damit nicht nur isolierte Knoten geclustert werden, soll der *ratio cut* minimiert werden:

$$\min_{C_1,...,C_k} \sum_{i=1}^{k} \frac{w(\partial C_i)}{|C_i|}$$

Da dieses Problem aber NP-schwer ist, soll stattdessen eine Annäherung gefunden werden.

**Definition 2.4.1.** Wie immer ist $G = (V, E)$ und $w$ eine Gewichtsfunktion auf den

Kanten. Die Laplace Matrix $L \in \mathbb{R}^{V \times V}$ von $G$ ist

$$L_{u,v} = \begin{cases} -w_{uv}, & \text{if } uv \in E \\ \sum_{e \in \delta(u)} w(e), & \text{if } u = v \\ 0, & \text{sonst} \end{cases}$$

Man kann schreiben $L = B^T \cdot D \cdot B$ wobei $D \in \mathbb{R}^{E \times E}$ im Feld $(e, e)$ das Gewicht $w(e)$ hat und 0 sonst. $B$ ist aus $\mathbb{R}^{E \times V}$. Für $B$ geben wir allen Kanten aus $G$ eine Richtung vor. In der Spalte $e = (u, v)$ steht 1 in Spalte $v$, $-1$ in Spalte $u$, wenn $(v, u)$ eine Kante ist und 0 sonst.

Wir können nun $D^{1/2}$ definieren als jede Matrix die Wurzel genau die Einträge der Matrix $D$ hat. Diese Definition ist daher sinnvoll, da $D$ eine Diagonalmatrix ist. Es folgt $L = (D^{1/2}B)^T(D^{1/2}B)$ und

$$x^T L x = ||D^{1/2}Bx||_2^2 = \sum_{uv \in E} w_{uv}(x_u - x_v)^2$$

**Lemma 2.4.2.** *Seien $G$ und $w$ wie immer. Sei $L$ die Laplace Matrix von $G$. Dann*

1. *$L$ ist symmetrisch und positiv semi-definit*
2. *kleinster Eigenwert ist 0*
3. *ist $\mathcal{C}$ die Menge der Komponenten von $G$. Dann ist $\chi_C$, $C \in \mathcal{C}$ orthogonale Eigenbasis des Eigenwerts 0.*

Zurück zu ratio cuts. Sei $A \subsetneq V$ mit $A \neq \varnothing$. Wähle

$$\mathbb{R}^V \ni z = \sqrt{\frac{|\overline{A}|}{|A|}}\chi_A - \sqrt{\frac{|A|}{|\overline{A}|}}\chi_{\overline{A}}$$

woraus folgt

$$z^T L z = |V| \sum_{uv \in \partial A} w_{uv} \frac{1}{|A|} + \frac{1}{|\overline{A}|} \sum_{j=1, j \neq i}^{k} |\partial(C_i, C_j)|$$

In der Theorie ist dieses Verfahren sehr interessant, allerdings normalerweise nicht praktisch umsetzbar. Deswegen wird das nicht weiter verfolgt.

## 2.5 Cluster-Metrik

Wenn die verschiedenen Algorithmen miteinander verglichen werden, ist ein Test-Graph nötig. Solche Testfälle werden meist so aufgebaut, dass ein objektiv bestes Clustering ex-

istiert (*ground truth*). Dann muss verglichen werden, welcher Algorithmus ein Clustering produziert, dass am nächsten an dieses Clustering heranreicht. Dafür sind Clustering-Metriken nötig.
Rand Distanz: Seien $A, B$ zwei Clusterings. Wir nennen $u, v \in V$ eine Unstimmigkeit, wenn $\exists a \in A : u, v \in a$ aber $\nexists b \in B : u, v \in b$ oder umgekehrt. $d(A, B)$ als die Anzahl der Unstimmigkeiten ist eine Metrik. Diese Metrik ist leicht zu berechnen, denn

$$d(A,B) = \sum_{a \in A} \binom{|a|}{2} + \sum_{b \in B} \binom{|b|}{2} - 2 \sum_{a \in A, b \in B} \binom{|a \cap b|}{2}$$

Entropie: Sei $V$ der Größe $n$ und seien $A, B, C$ drei Clusterings. Diese werden in folgender Weise als Zufallsvariablen modelliert: wähle ein $v \in V$ mit gleichmäßiger Wahrscheinlichkeit. Dann ist

$$X_A : V \to A, v \mapsto a \ni v$$

eine Zufallsvariable wobei

$$\mathbb{P}[X_A = a] = \frac{|a|}{n}$$

Es sei dann weiter

$$H(A) = H(X_A) = -\sum_{a \in A} \frac{|a|}{n} \log_2 \left( \frac{|a|}{n} \right)$$

Die Entropie des Clusterings. Weiter ist

$$H(A,B) = -\sum_{a \in A, b \in B} \frac{|a \cap b|}{n} \log_2 \left( \frac{|a \cap b|}{n} \right)$$

und $VI(A, B) = 2H(A, B) - H(A) - H(B)$ die *variation of information*. Diese ist dann eine Metrik.

# 3 Streaming

## 3.1 HyperLogLog

Es wird eine Reihe an Zahlen eingelesen, die nicht in den Speicher passt. Wie kann die Anzahl der paarweise verschiedenen Zahlen festgestellt werden?
Die Zahlenreihe $a_1, ...a_n$ wird beim Einlesen in Binärdarstellung umgewandelt. Die Binärzahlen werden anschließend mit einer zufälligen aber deterministischen Transformation in eine andere Binärzahl konvertiert. Bei jeder transformierten Zahl werden die letzten Stellen betrachtet und die Anzahl der 0en am Ende gezählt. Die maximale Zahl $R$ von 0en am Ende einer Zahl wird gespeichert. Ausgegeben wird $2^R$.

**Satz 3.1.1.** *Sei $F_0$ die Anzahl der paarweise verschiedenen Elemente eines Streams von $m$ Zahlen und sei $Y$ der Output des Algorithmus ($Y = 2^R$). Jedes $a$ des Streams liegt in $\{1, ..., n\}$. Es wird $\mathcal{O}(\log n)$ Platz gebraucht und für alle Integer $c > 2$ ist*

$$\mathbb{P}\left[\frac{1}{c} \leq \frac{Y}{F_0} \leq c\right] \geq 1 - \frac{3}{c}$$

## 3.2 Streaming mit Graphen

Die Knoten des Graphens sind initial im Speicher. Dieser ist nur $\mathcal{O}(n \cdot \log(n)^p)$ für ein $p$. Das heißt die Kanten (ungefähr $n^2$) können nicht gespeichert werden. Der Stream besteht nun aus den Kanten, also $e_1, e_2, ..., e_m$.

## 3.3 Spanners

Wir wollen für einen sehr großen Graphen $G$ eine komprimierte Version dieses Graphens konstruieren, die die jeweiligen Abstände innerhalb des Graphens bestmöglich erhält.

**Definition 3.3.1.** Gegeben ein Graph $G$, ein $\alpha - \textit{spanner}$ ist ein Subgraph $H$ s.d.

$$d_H(u, v) \leq \alpha d_G(u, v) \; \forall u, v$$

Wir wollen einen $(2k+1) - \textit{spanner}$ berechnen mit $\mathcal{O}(m \log n)$ Zeit. Der Algorithmus wird randomisiert laufen, wobei die erwartete Größe des Graphen $\mathcal{O}(kn^{1+\frac{1}{k}})$ ist. Der folgende Algorithmus dient als Setup.

1. initialisiere $V_0 = V$, $V_1, ..., V_{k-1} = \emptyset$
2. for $i = 1$ to $k - 1$
3. for $x \in V_{i-1}$
4. mit Wahrscheinlichkeit $n^{-\frac{1}{k}}$ ergänze $x$ zu $V_i$ und setze $B_{i,x} = \{x\}$ und $h(x) = i$
5. endfor
6. endfor

$B_{i,x}$ sind *bags* um das Zentrum $x$ und $h(x)$ ist das größte $i$, in dem $x$ noch vorkommt.

1. verwende das Setup von oben
2. setze $H = (V, \emptyset)$ und $l(v) = h(v)$ für alle $v \in V$

3. for uv = next(S)
4. stelle sicher, dass $l(u) \leq l(v)$ durch Tauschen
5. Finde das Zentrum $x$ des bags mit $v \in B_{l(u),x}$
6. if $u \notin B_{l(u),x}$ und es keine Kante $uw$ in $H$ gibt mit $w \in B_{l(u),x}$ ergänze $uv$ zu $H$
7. if $l(u) < l(v)$
8. ergänze $u$ zu $B_{l(u)+1,x}, ..., B_{h(x),x}$
9. setze $l(u) = h(x)$
10. endif
11. endfor

**Lemma 3.3.2.** *Die erwartete Anzahl an bags $B_{k-1,x}$ im Level $k-1$ ist $n^{\frac{1}{k}}$.*

**Lemma 3.3.3.** *Ist $u \in B_{i,x}$ dann gibt es einen $u, x$-Pfad der Länge höchstens $i$.*

**Lemma 3.3.4.** *$H$ ist ein $(2k-1)-$spanner von $G$.*

**Lemma 3.3.5.** $\mathbb{E}[|E(H)|] \leq kn^{1+\frac{1}{k}}$

Aus einer Vermutung von Erdös folgt, dass ein $(2k-1)-$*spanner* $\Omega(n^{1+\frac{1}{k}})$ Kanten braucht.

**Satz 3.3.6.** *Für alle Primpotenzen $q$ und $k \in \{2,3\}$ gibt es einen bipartiten Graphen $D_k(q)$ mit $n = 2q^k$ Knoten, $(\frac{n}{2})^{1+\frac{1}{k}}$ Kanten und Umfang mindestens $2k+2$.*

### 3.4 sparsifier

Sei $G$ ein Graph, $\varepsilon > 0$. Suche $H \subset G$, $w : E(H) \to \mathbb{R}_+$ sodass $\forall \emptyset \neq A \subsetneq V(G)$ gilt

$$(1-\varepsilon)\,|\partial_G(A)| \leq w(\partial_H(A)) \leq (1+\varepsilon)\,|\partial_G(A)|$$

Ohne das Streaming Setting ist das Problem sehr einfach:

1. Berechne kleinsten cut $\lambda$
2. sei $H = (V, \emptyset)$ und
$$p = 9\ln(\frac{n}{\varepsilon^2\lambda})$$
3. $\forall e \in E(G)$ ist $e$ in $E(H)$ mit Wahrscheinlichkeit $p$

4. Gewichte überall $\frac{1}{p}$

**Satz 3.4.1.** *Mit Wahrscheinlichkeit* $1 - \mathcal{O}(\frac{1}{n})$ *ist* $H$ *ein sparsifier.*

**Lemma 3.4.2.** *Sei* $\lambda \geq 1$ *die kleinste Schnittgröße und* $\alpha > 0$*. Dann gilt: die Anzahl der Schnitte der Kardinalität* $\leq \alpha\lambda$ *ist begrenzt durch* $n^{2\alpha}$*.*

**Satz 3.4.3** (Chernoff-Ungleichung)**.**

$$\mathbb{P}[|X - \mathbb{E}[X]| > \varepsilon\mathbb{E}[X]] \leq 2e^{-\frac{\varepsilon^2|\mathbb{E}[X]|}{3}}$$

**Definition 3.4.4.** Sei $G$ ein Graph. Ein Teilgraph $H$ ist ein $k$-Skelett, wenn $\forall C \subseteq E(G)$ mit $|C| < k$ gilt, dass $C$ ein Schnitt in $G$ ist genau dann, wenn $C$ ein Schnitt in $H$ ist.

Ohne Streaming kann ein $k$-Skelett mit folgendem Algorithmus gefunden werden:

1. for $i = 1, ..., k$
2. Berechne $F_i$ einen aufspannenden Wald von $G - \bigcup_{j=1}^{i-1} E(F_j)$
3. endfor
4. $H = \bigcup_{i=1}^{k} F_i$

Als Streaming-Algorithmus funktioniert das Konzept genau gleich.

1. for $e = next(S)$:
2. sei $i$ der minimale Index, sodass $F_i \cup e$ keinen Kreis enthält. Setze $F_i = F_i \cup e$.
3. endfor

## 3.5 Minimaler Schnitt als Streaming Algorithmus

Wir wollen den minimalen Schnitt approximieren. Mittels Karger kann ein $\varepsilon$-sparsifier bestimmt werden, aber dafür ist a priori der Wert $\lambda$, also der minimale Schnitt nötig. Es wird also stattdessen $p = \frac{1}{2^r}$ für $r \in \{1, ..., l\}$ sodass $2^l \approx \log n$. Es werden aus dem Stream die gewichteten Graphen $G, 1/p$ bestimmt. Diese sind aber immer noch zu groß für den Speicher. Deshalb wird aus $G, 2^r$ direkt mittels Streaming ein $k$-Skelett $H_r$ bestimmt (für $k \approx \log n$). Anschließend wird jenes $i$ bestimmt, sodass $mincut(H_i) < k$ und geben dann $2^i \cdot mincut(H_i)$ zurück.

**Satz 3.5.1.** *Mit hoher Wahrscheinlichkeit gibt der Algorithmus einen Wert* $\alpha$ *zurück, sodass*

$$(1 - \varepsilon)mincut(G) \leq \alpha \leq (1 + \varepsilon)mincut(G)$$

Das Problem, den Maximalgrad eines Graphen mit Streaming zu bestimmen ist mit Speicherplatz $\mathcal{O}(n)$ leicht. Das ist aber nicht nötig, wie wir im Folgenden sehen werden.

**Beispiel 3.6.1** (Zählen mit Hashing)**.** *Wir wollen in einem allgemeinen Stream die häufigsten Elemente bestimmen.*
*Die Idee ist, die Elemente des Streams zu hashen und auf einem kleine Hash-Array die Häufigkeiten zu zählen. Das Problem ist nun, die Kollisionen zu minimieren. Eine Möglichkeit ist, das Hash-Array als Hash-Matrix aufzubauen, wobei jede Zeile einem Hash-Array entspricht. Exakte Kollisionen sind dann wesentlich seltener und um die Häufigkeiten aus der Hash-Matrix zu bestimmen wird schließlich der minimale Wert der Hash-Stellen eines Elements bestimmt.*
*Die Hashfunktionen $h_i$ sollten der Form*

$$h : a \mapsto ((\alpha_i a + \beta_i) \mod p) \mod l$$

*wobei $\alpha_i \beta_i$ gleichverteilt in $\{0, 1, ..., p-1\}$ für eine Primzahl $p \leq n$ ist und $l$ Die Größe der Hashtabelle.*

**Satz 3.6.2.** *Sei $k$ die Anzahl der Hashfunktionen. Sei $\hat{f}_a$ die bestimmte Anzahl an Vorkommen des Elements $a$ in einem Stream der Länge $m$. Es gilt $\hat{f}_a \geq f_a$ die tatsächliche Anzahl an Vorkommen. Außerdem gilt,*

$$\mathbb{P}[\hat{f}_a \leq f_a + \varepsilon m] \geq 1 - \left(\frac{1}{\varepsilon l}\right)^k$$

**Beispiel 3.6.3** (heavy hitters)**.** *Für eine Grenze $T$ wollen wir nun bestimmen, welche Elemente mindestens $T$ mal vorkommen. Mit der Vorüberlegung ist das recht leicht, denn wir können den Stream durchgehen und überprüfen ob $\hat{f}_a \geq T$ ist. Wenn schon ergänzen wir $a$ zu unserer Liste. Auf genau diese Weise können die Knoten mit größtem Grad eines Graphen bestimmt werden.*

Wenn wir nun einen Stream $A = (a_1, ...)$ mit $a_i \in \{0, 1, .., n-1\}$ haben, so können wir $a$ schreiben als $e_{a+1} \in \mathbb{R}^n$. Sei dann $f = \sum_{a \in A} e_{a+1}$. Sind dann $c_r$ die Zeilen der Hash-Matrix $C$, so gilt der folgende

**Satz 3.6.4.** *$\exists L$ sodass*

$$\begin{pmatrix} c_1^T \\ c_2^T \\ \vdots \\ c_k^T \end{pmatrix} = Lf$$

**Lemma 3.6.5** (Johnson-Lindenstrauss)**.** *(Das soll nur eine sehr grobe Darstellung sein). Es gibt für Daten einer sehr hohen Dimension eine passende Zufallstransformation $L$, die mit hoher Wahrscheinlichkeit die Abstände zwischen Originaldaten beibehält, wenn es die Daten auf eine wesentlich niedriger dimensionale Bildmenge abbildet.*

## 3.7 Dynamisches Streaming

In diesem Fall kann sich der Graph innerhalb des Streams verändern. D.h. der Stream $S$ besteht aus Elementen

$$S = ((e_1, \sigma_1 = \pm 1), (e_2, \sigma_2 = \pm 1), ..., )$$

wobei $+1$ bedeutet, dass die Kante hinzugefügt wird und $-1$, dass die Kante gelöscht werden soll. Anstelle von $\pm 1$ kann auch ein Gewicht $w_i$ übergeben werden, wobei $w_i = 0$ dann bedeuten soll, dass die Kante entfernt wird. Gegeben eine Position $t$ im Stream sei $G_t = (V, \{e_i : i \leq t \ \wedge \ \sum_{e_j = e_i} \sigma_j = 1\})$ im ungewichteten Fall.

**Beispiel 3.7.1.** *Gegeben sei ein dynamischer Stream $(a_i, w_i)$. Definiere Einheitsvektoren $e_a \in \mathbb{R}^n$. Berechne den Gewichtvektor*

$$w = \sum_{i=1}^{m} w_i e_{a_i}$$

*Das Ziel ist nun, den Support von $w$ zu bestimmen.*

**Beispiel 3.7.2** (Zusammenhang)**.** *Gegeben sei ein dynamischer Graph-Stream $S$. Es soll festgestellt werden, ob der korrespondierende Graph zusammenhängend ist. Wir betrachten nur den ungewichteten Fall.*

1. *for $(e, \sigma) = enxt(S)$*
2. *berechne eine Abbildung wie unter Lemma 3.6.5 $\bar{x}$*
3. *endfor*
4. *wähle zufällig eine Kante unter $\bar{x}$ und kontrahiere diese*

5. *repeat*

*Ist am Ende nur ein Knoten übrig, so sit der Graph mit hoher Wahrscheinlichkeit zusammenhängend.*

### 3.8 Sampling

Ziel ist es, ein Element aus einem gegeben Vektor mittels Gleichverteilung zu samplen.

**Definition 3.8.1.** Sei $\delta > 0$. Eine lineare Funktion $L : \mathbb{R}^n \to \mathbb{R}^l$ heißt *discerning*, wenn es einen effizienten Algorithmus gibt, der für Input $Lv$ mit Wahrscheinlichkeit $1 - \delta$ ein gleichwahrscheinliches Sample von $supp(v)$ ausgibt

**Satz 3.8.2.** *Für $\delta > 0$ gibt es einen Algorithmus und ein $L$, das discerning ist, sodass*

1. *$Lv$ effizient berechnet werden kann*
2. *Speicher ist maximal $\mathcal{O}((\log n)^2 \log(1/\delta))$.*

**Satz 3.8.3.** *Wenn $x$ sparse ist, und $\bar{x} = Ax$, dann kann $x$ wieder bestimmt werden mit compressed sensing.*

Mit genau dieser Idee, kann das Stream Sampling Problem gelöst werden.

### 3.9 Graph-Komponenten bestimmen

Gegeben sei ein ungewichteter dynamischer Stream $S$. Man bestimmt zuerst einen *graph sketch*

1. Initialize $\bar{a}^{(v)} = 0$ für alle $v \in G$
2. Initialize mapping $L$ as in above theorem for $\delta = 0.01$
3. for $(uv, \sigma) = next(S)$ do
4. Swap $u, v$ if necessary s.t. $u < v$
5. Update $\bar{a}^{(u)} = \bar{a}^{(u)} + \sigma Le^{(u)}$ and $\bar{a}^{(v)} = \bar{a}^{(v)} - -\sigma Le^{(u)}$
6. endfor

Damit funktioniert der Algorithmus

1. run *graph sketches* $\lceil 13 \ln n \rceil$ times to compute $\bar{a}^{(v),r}$
2. Set $\mathcal{V} = \{\{v\} : v \in V\}$

3. for $r = 1, ..., \lceil 13 \ln n \rceil$ do

4. for $X \in \mathcal{V}$ do

5. try to sample index $uv$ from $supp(\sum_{v \in X} a^{(v)})$ via $\sum_{v \in X} \bar{a}^{(v),r}$

6. if sampling was successful, store $uv$ in a list $\mathcal{L}$

7. endfor

8. contract each $uv$ in $\mathcal{L}$

9. endfor

**Satz 3.9.1.** *Mit hoher Wahrscheinlichkeit ist $\mathcal{V}$ die Menge der Knotenmengen der Komponenten.*

# 4 Schwere Probleme

## 4.1 Maximum Independent Set

Für einen Baum ist dieses Problem leicht zu lösen.

**Lemma 4.1.1.** *Sei $X$ eine unabhängige Knotenmenge. Dann existiert eine unabhängige $X'$ mit $|X'| \geq |X|$ und die Blätter sind in $X'$.*

Graphen, die sich kaum von einem Baum unterscheiden, lassen sich mit fast dem selben Algorithmus auch lösen. Das motiviert die

**Definition 4.1.2.** Sei $G$ ein beliebiger Graph. Eine *Baumzerlegung* von $G$ ist ein Baum $T$, sodass jeder Knoten $t$ von $T$ mit einer Knotenmenge $X_t \subset V(G)$ korrespondiert. Also

1. $V(G) = \bigcup_{t \in V(T)} X_t$

2. $\forall uv \in E(G)\ \exists t \in V(T)$ mit $u, v \in X_t$

3. $\forall v \in V$ mit $v \in X_s, X_t$, $s, t \in V(T)$ und ist $r$ auf $s-t$-Weg in $T$, dann $v \in X_r$.

Die *Weite* der Baumzerlegung ist $\max_{t \in V(T)}\{|X_t| - 1\}$. Die Baumweite von $G$ ist

$$tw(G) = \min \text{Weite einer Baumzerlegung}$$

**Lemma 4.1.3.** *Sei $(T,(V_t))$ die Baumzerlegung von $G$ und $st \in E(T)$. Seien $T_s, T_t$ die Komponenten von $T - st$ die $s$ bzw. $t$ enthalten. Seien*

$$U_s = \bigcup_{r \in V(T_s)} V_r \text{ und } U_t = \bigcup_{r \in V(T_r)} V_r$$

*dann trennt $V_s \cap V_t$ $U_s$ von $U_t$.*

**Lemma 4.1.4.** *Sei $H$ ein Teilgraph von $G$ und $(T,(V_t))$ eine Baumzerlegung von $G$. Setze $V_t' = V_t \cap V(H)$ für alle $t \in V(T)$. Dann ist $(T,(V_t'))$ eine Baumzerlegung von $H$ und $tw(H) \leq tw(G)$.*

**Definition 4.1.5.** Ein Graph $H$ ist ein Minor von $G$, wenn sich $H$ aus Löschen von Ecken und Kanten und Kontraktionen von Kanten in $G$ ergibt. Man schreibt $H \preceq G$.
Alternativ: $\exists B_h \subset V(G)$ mit $h \in V(H)$ sodass

1. $B_h$ pw. disjunkt
2. $G[B_h]$ zsh.
3. $\forall h, h' \in E(H)$ gilt $\exists$ Kante zwischen $B_h$ und $B_h'$

**Lemma 4.1.6.** *Sei $H \preceq G$. Dann ist $tw(H) \leq tw(G)$.*
*Sei $(T,(V_t))$ die Baumzerlegung von $G$. Setze $V_t' = \{h \in V(H) : B_h \cap V_t \neq \varnothing\}$ für alle $t \in V(T)$. Dann ist $(T,(V_t'))$ eine Baumzerlegung von $H$.*

**Lemma 4.1.7.** *Sei $(T,(V_t))$ eine Baumzerlegung von $G$ und $W \subseteq V(G)$. Dann entweder*

1. *$\exists s,t \in V(T)$ und $w_1, w_2 \in W$ mit $w_1, w_2 \notin V_2 \cap V_t$ und $V_s \cap V_t$ trennt $w_1$ und $w_2$ in $G$ oder*
2. *$\exists t \in V(T)$ mit $W \subseteq V_t$.*

**Lemma 4.1.8.** *Es folgt $tw(G) = 1 \Leftrightarrow G$ ist ein Wald.*

**Lemma 4.1.9.** *Jeder Graph $G$ hat Baumzerlegung $(T,(V_t))$ der Weite $tw(G)$, sodass $\forall s,t \in V(T)$, $s \neq t$ gilt $V_s \not\subseteq V_t$ und $V_t \not\subseteq V_s$.*

**Lemma 4.1.10.** *Sei $(T,(V_t))$ Baumzerlegung von $G$, sodass $\forall s \neq t \in V(T)$ gilt $V_s \not\subseteq V_t$ und $V_s \not\subseteq Vs$ dann ist*

$$|V(T)| \leq |V(G)|$$

**Satz 4.1.11.** *Es gilt $tw(G) \leq 2$ genau dann, wenn $K_4 \not\preceq G$.*

**Definition 4.1.12.** Ein Graph ist $k$-degeneriert, wenn jeder Teilgraph von $G$ eine Ecke vom Grad höchstens $k$ enthält.

**Satz 4.1.13.** *Ist $tw(G) \leq k$, dann ist $G$ $k$-degeneriert.*

**Korollar 4.1.14.** *Aus $tw(G) \leq k$ folgt $|E(G)| \leq kn - \binom{k+1}{2}$.*

## 4.2 Allgemeines Modell für Algorithmen der dynamischen Programmierung

Die Idee ist es, mathematische Logik der ersten Stufe zu verwenden. Dabei sind die Variablen jeweils Ecken oder Kanten und es gibt die üblichen logischen Operatoren. Zusätzlich gibt es die beiden Operatoren

1. $adj(u, v) \Leftrightarrow$ wenn $u, v$ benachbart
2. $inc(v, e) \Leftrightarrow$ wenn $e, v$ inzident

Mit diesem Formalismus lassen sich einfache graphentheoretische Eigenschaften (zum Beispiel isolierte Knoten, Existenz von Dreiecken etc.) aufschreiben, aber nicht alle. Zum Beispiel lässt sich Bipartität nicht definieren.
Die nächste Stufe ist *monadic second order logic* (MSO). Hier sind Mengen auch als Variablen erlaubt.

**Satz 4.2.1.** *Für jedes Problem, dass sich in MSO ausdrücken lässt und für jedes $k \in \mathbb{N}$ ein Linear-Zeit-Algorithmus auf Graphen $G$ mit $tw(G) \leq k$.*

## 4.3 Berechnung der Baumzerlegung

**Satz 4.3.1.** *Es existiert ein Algorithmus, sodass, wenn $G$ Baumweite $\leq k$ hat, dann berechnet der Algorithmus eine Baumzerlegung der Weite $\leq k$ in $\mathcal{O}(2^{k^3}n)$ Zeit.*

**Definition 4.3.2.** Sei $G$ ein Graph und $\varnothing \neq W \subset V(G)$, dann ist $S$ ein *balanced $W$-separator*, wenn gilt $|W \cap V(C)| \leq \frac{1}{2}|W|$ für jede Komponente $C$ von $G - S$.

**Lemma 4.3.3.** *Ist $tw(G) \leq k$ existiert ein $S$ balanced $W$-separator mit $|S| \leq k + 1$.*

**Definition 4.3.4.** Eine *separatation* eines Graphen $G$ sind zwei Teilgraphen $A, B$, sodass $A \cup B = G$. Die Ordnung der separation ist $|V(A \cap B)|$.
Eine *$W$-weakly balanced separation* sind zwei Teilgraphen $A, B$, sodass

- $(A, B)$ ist ein separator
- $1 \leq |W \setminus B| \leq \frac{2}{3}|W|$
- $1 \leq |W \setminus A| \leq \frac{2}{3}|W|$

**Lemma 4.3.5.** *Ist $tw(G) \geq 2k + 3$, $k \geq 2$, dann existiert eine $W$-weakly balanced separation der Ordnung $\leq k + 1$.*