Network Science und Algorithm Engineering

Manuel Penschuck

25. April 2024

Einen ganz herzlichen Dank an alle, die durch ihr Feedback, Anregungen und Korrekturen zu diesem Dokument betrugen. Spezieller Dank geht an (in alphabetischer Reihenfolge): Daniel Allendorf, Lukas Geis, Ulrich Meyer, Hung Tran, Zeno Weil.

Inhaltsverzeichnis

Kapitel 1

Einleitung

„Network Science“ und „Algorithm Engineering“ sind zwei aktive und interdisziplinäre Forschungsfelder. Beide Felder sind relativ jung (etwa 20 bis 30 Jahre alt) und haben sich in den letzten Jahren stark weiterentwickelt. Sie könnten beide unabhängig von einander ganze Studiengänge füllen (und tun es auch). In dieser Veranstaltung möchten wir also nicht die Vereinigung beider Felder beleuchten, sondern uns einige Themen in ihrem Schnitt ansehen. Dennoch sollten wir grob die Felder charakterisieren.

Network Science beschäftigt sich mit der Analyse und Modellierung von komplexen Netzwerken. Beispiele sind soziale, Kommunikations- und Verkehrsnetze, biologische Netzwerke und viele mehr. Sie zeichnen sich durch sogenanntes *emergentes Verhalten* aus: Ohne dass einzelnen Knoten oder Kanten bewusst darauf hinarbeiten, entsteht ein komplexes Verhalten des Gesamtsystems. In sozialen Netzen bilden sich z. B. lokal stark vernetzte Gruppen (Communities), es gibt einige zentrale Teilnehmer, die übermäßig stark vernetzt sind (Celebrities), und Netze haben überraschend geringe Distanzen zwischen Nutzern. Network Science versucht, diese Eigenschaften zu finden und zu verstehen. Hierzu vereinigt es Methoden aus Mathematik, Informatik, und Physik.

Ziel des *Algorithm Engineering* ist es, die Spaltung zwischen Theorie und Praxis im Algorithmenentwurf zu überbrücken. Es geht also darum, theoretisch effiziente als auch praktisch schnelle und implementierbare Algorithmen zu entwickeln. Hierbei kommt oft die in Abb. 1.1 dargestellte zyklische Entwurfsmethode zum Einsatz:

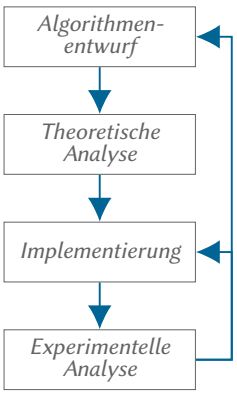


Abbildung 1.1: Algorithm Engineering Zyklus

Wir entwerfen meist zuerst einen Algorithmus und versuchen, dessen Performance theoretisch vorherzusagen; diese Hypothesen werden dann experimentell überprüft und – je nach Ergebnis – der Algorithmus oder die Analyse (z. B. Average Case statt Worst Case) angepasst. Dieser Zyklus wird wiederholt, bis ein Algorithmus gefunden ist, der sowohl praktisch gut funktioniert als auch theoretisch verstanden ist.

Um praktisch nützliche Vorhersagen treffen zu können, verwendet man im Algorithm Engineering möglichst realistische Modelle für die Maschine und die Eingabedaten. Zur Analyse von Graphalgorithmen benötigen wir also geeignete Modelle für Netzwerke. Zur Analyse von großen Netzwerken benötigen wir wiederum effiziente Algorithmen. So ergänzen sich die beiden Felder.

1.1 Über diese Veranstaltung

Lernziele dieser Veranstaltung beinhalten:

- Wichtige Eigenschaften von beobachteten Netzwerken (z. B. Dichte, Gradverteilung, Zusammenhang, Durchmesser, lokales Clustering, globales Clustering)
- Methoden, diese Eigenschaften nachzubilden und zu erklären
- Zufallsgraphen und ihre Eigenschaften
- Effiziente Algorithmen für und auf Zufallsgraphen
- Effiziente, randomisierte Algorithmen (wir betrachten viele Graphgeneratoren – die Techniken eignen sich aber viel allgemeiner für diverse Simulationsaufgaben)

1.2 Begrifflichkeiten

Ein *Netzwerk* ist ein System, in dem Personen/Agenten/Objekte (etc.) miteinander in Relation stehen (z. B. durch eine Freundschaft, ein Gespräch usw.). *Graphen* sind ein abstraktes Modell dieses Systems: Wir identifizieren die Akteure mit den Knoten und die Beziehungen mit den Kanten. Wenn wir ein Netzwerk aus der echten Welt in einen Graphen übersetzen, sprechen wir oft von einem *beobachteten Netzwerk*. Diese Abbildung geht oft mit dem Verlust von Informationen einher (wir wollen für den Anwendungsfall Unwesentliches ausblenden) und ist oft nicht eindeutig.

Beispiel 1.1. Die akademische Arbeit ist von Kollaboration geprägt – Wissenschaftler arbeiten zusammen und publizieren schließlich gemeinsam ihre Ergebnisse. Es handelt sich also ganz klar um ein Netzwerk. Wie modellieren wir aber den zugrunde liegenden Graphen? Hier ein paar Ansätze:

- Wir können Autoren als Knoten auffassen und eine Kante zwischen zwei Autoren ziehen, wenn sie zusammen publiziert haben. Hierbei geht die Information darüber verloren, wie oft zwei Wissenschaftler kollaborierten. Dies könnte man etwa durch Kantengewichte ergänzen, die die Anzahl der gemeinsamen Publikationen kodieren.
- Wir können die Autoren als Knoten und die Publikationen als Kanten auffassen. Da im Allgemeinen die Anzahl der Autoren pro Publikation unbeschränkt ist, ergibt sich ein Hypergraph (ein Graph mit Kanten zwischen mehr als zwei Knoten).
- Wir können die Autoren *und* Publikationen als Knoten auffassen und je eine Publikation mit allen Urhebern verbinden. Dieser Graph ist bipartit[1], da weder Autoren noch Publikationen untereinander verbunden sind. Kanten könnten jetzt sogar noch weitere Informationen tragen, z. B. den Hauptautor. ◀

Keines dieser Modelle ist *richtig* oder *falsch* – die Wahl hängt davon ab, was wir mit dem Graphen bezwecken.

[1] Wir können jeden Hypergraph analog in einen bipartiten Graphen übersetzen.

Kapitel 2

Zufallsgraphen

Ein Zufallsgraph ist ein Modell einer Graphfamilie – etwa wie eine Zufallsvariable ein Modell einer Zufallsgröße ist. Der Zufallsgraph beschreibt also ein Ensemble von Graphen, das so entworfen sein kann, dass bestimmte Eigenschaften in den betrachteten Graphen verstärkt auftreten. Solche maßgeschneiderte Zufallsgraphen werden oft als *Netzwerkmodell* bezeichnet.

Das Forschungsfeld der Zufallsgraphen nahm mit den Arbeiten von Gilbert [8], sowie Erdős und Rényi [7] Anfang der 1960er Jahre an Fahrt auf. Die beiden Arbeiten definieren Zufallsgraphen, die zunächst für abstrakte Untersuchungen (z. B. die probabilistische Methode) verwendet wurden. Der Fokus auf die Modellierung von beobachteten Netzwerken kam erst später hinzu. Dennoch spielen die Modelle in der Netzwerkforschung bis heute eine wichtige Rolle. Wir werden sie daher bald genauer betrachten, fangen jedoch mit einem noch einfacheren Modell an.

2.1 Der Zufallsgraph $\mathcal{G}(n)$

Ein Zufallsgraph $(\mathbb{G}, f)$ ist eine Wahrscheinlichkeitsverteilung $f\colon \mathbb{G} \to [0,1]$ über einer Menge von Graphen $\mathbb{G}$. Oftmals wird der Grundraum $\mathbb{G}$ durch eine Parametrisierung eingeschränkt; auch f kann parametrisiert sein.

$\mathcal{G}(n)$-Graphen

Das einfachste Modell ist $\mathcal{G}(n)$, welches die Gleichverteilung über alle Graphen mit n Knoten beschreibt. Hierbei ist zu beachten, dass wir mit *alle Graphen* in der Regel entweder alle *gerichteten* oder alle *ungerichteten* Graphen meinen. Die Details der Analysen in den folgenden Kapiteln hängen von dieser Entscheidung ab – allerdings ergeben sich meist keine qualitativen Unterschiede. Daher werden wir oft den für uns einfacheren Fall wählen. Je nach Wahl ist der Grundraum von $\mathcal{G}(n)$

$$\mathbb{G}_{\text{ger}}(n) = \{G = (V,E) \mid |V| = n \wedge E \subseteq V \times V\} \tag{2.1}$$

$$\mathbb{G}_{\text{unge}}(n) = \{G = (V,E) \mid |V| = n \wedge E \subseteq \{\{u,v\} \mid u,v \in V \text{ mit } u \neq v\}\}\,. \tag{2.2}$$

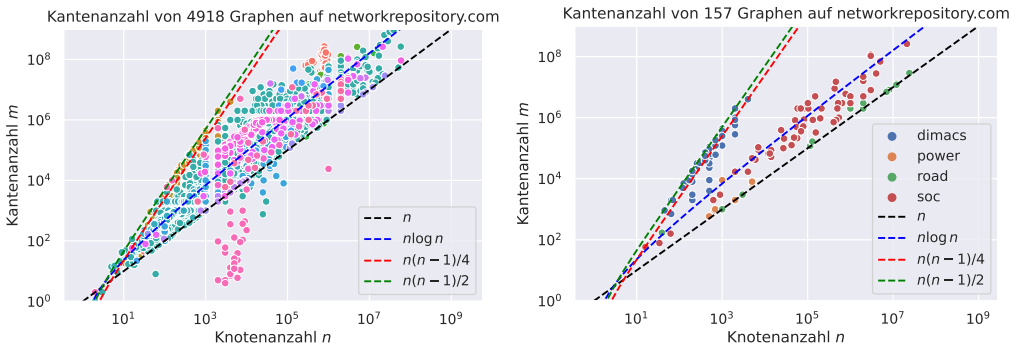


Abbildung 2.1: Die Verteilung der Kantenanzahl in verschiedenen Netzwerken auf [18]. Die Farben der einzelnen Punkte geben an, aus welchem Bereich die Netzwerke stammen.

Die Wahrscheinlichkeitsverteilung f folgt dann als $f_{\mathbb{G}(n)}(G) = 1/|\mathbb{G}(n)|$ oder konkret:

$$f_{\mathbb{G}_{\text{ger}}(n)}(G) = \frac{1}{|\mathbb{G}_{\text{ger}}(n)|} = 2^{-n^2} \tag{2.3}$$

$$f_{\mathbb{G}_{\text{unge}}(n)}(G) = \frac{1}{|\mathbb{G}_{\text{unge}}(n)|} = 2^{-\binom{n}{2}} = 2^{-\frac{n(n-1)}{2}}. \tag{2.4}$$

Die geschlossene Form $|\mathbb{G}_{\text{ger}}(n)| = 2^{n^2}$ in Gleichung (2.3) ergibt sich daraus, dass ein gerichteter Graph n^2 potentielle Kanten hat (die Adjazenzmatrix hat $n \times n$ Einträge). Für jede dieser Kanten haben wir unabhängig genau zwei Möglichkeiten: Sie existiert oder eben nicht. Analog sieht es für die $\binom{n}{2}$ möglichen Kanten in einem ungerichteten Graphen aus.

Beachte aber auch, dass formal Gleichung (2.2) und Gleichung (2.3) widersprüchlich scheinen. In Gleichung (2.2) fordern wir nur, dass die Knotenmenge V die Kardinalität $|V| = n$ hat. Offensichtlich gibt es unbeschränkt viele dieser Wahlen; für $n = 3$ z. B. $\{1, 2, 3\}, \{2, 3, 4\}, \ldots, \{k, k+1, k+2\}$ für alle $k \in \mathbb{N}$. Die Wahl der Knotenbezeichner hat jedoch keinen Einfluss auf die Netzwerkeigenschaften. Daher werden wir in dieser Veranstaltung grundsätzlich annehmen, dass die Knotenmengen in irgendeiner Form fixiert sind, z. B. als $V = \{1, \ldots, n\} = [n]$ oder $V = \{v_1, \ldots, v_n\}$.

Ist das $\mathcal{G}(n)$-Modell aber nun realistisch? Das hängt davon ab, was wir mit ihm bezwecken und was wir mit *realistisch* meinen. Es ist aber sicherlich nicht geeignet, um gängige Netzwerke zu beschreiben. Dies liegt unter anderem daran, dass $\mathcal{G}(n)$ oft Graphen mit vielen Kanten erzeugt; intuitiv hat „jeder zweite Graph“ mindestens die Hälfte aller möglichen Kanten.

Beobachtung 2.1. Sei $G = (V, E)$ ein Graph, der zufällig aus $\mathcal{G}(n)$ mit $n > 1$ gezogen wurde. Dann gilt für gerichtete Graphen $\mathbb{P}\big[|E| \geq n^2/2\big] \geq 1/2$ und für ungerichtete Graphen $\mathbb{P}\big[|E| \geq \binom{n}{2}/2\big] \geq 1/2$. ◀

Beweis. Im Folgenden betrachten wir nur gerichtete Graphen; der Beweis läuft analog für ungerichtete Graphen. Stellen wir uns einen beliebigen Graphen $G = (V, E)$ vor.

Dann sei $\bar{G} = (V, \bar{E})$ sein Komplement, d. h. für alle möglichen Kanten gilt:

$$\forall e \in V \times V: \qquad e \in \bar{E} \Leftrightarrow e \notin E \tag{2.5}$$

Beobachte, dass es eine Bijektion zwischen allen Graphen in $\mathbb{G}$ und ihren Komplementen gibt: Jeder Graph hat ein eineindeutiges Komplement. Per Konstruktion gilt außerdem:

$$\begin{aligned} E \cup \bar{E} &= V \times V && (2.6)\\ \Rightarrow |E| + |\bar{E}| &\geq n^2 && (2.7)\\ \Rightarrow \max(|E|, |\bar{E}|) &\geq n^2/2. && (2.8) \end{aligned}$$

Für jeden Graph G gilt also, dass entweder G selbst oder sein Komplement $\bar{G}$ mindestens $n^2/2$ Kanten hat. Da wir G und $\bar{G}$ je mit gleicher Wahrscheinlichkeit ziehen, gilt $\mathbb{P}\big[|E| \geq n^2/2\big] \geq 1/2$. □

2.2 Kantenanzahl in beobachteten Netzwerken

Wie viele Kanten haben echte Netzwerke? Hierzu führen wir ein Experiment durch: Wir nutzen die Datenbank https://networkrepository.com/, die über 5000 Netzwerke aus unterschiedlichen Bereichen enthält [18]. In Abb. 2.1 zeichnen wir die Kantenanzahl als Funktion der Knotenanzahl. Zwei Eigenschaften fallen direkt auf:

1. Die Kantendichte hängt vom Netzwerktyp ab.
2. Es sind fast immer deutlich weniger als die Hälfte der Kanten vorhanden.

2.2.1 Netzwerktypen haben unterschiedliche Kantendichten

Betrachten wir die erste Beobachtung genauer, indem wir Straßennetze und Freundschaftsnetze vergleichen. Wir modellieren ein Straßennetz dadurch, dass Adressen (Häuser, Kreuzung usw.) als Knoten und Straßen als Kanten dargestellt werden. Diese Netze sind im wesentlichen ein zweidimensionales Konstrukt. Wenn wir Tunnel, Brücken und dergleichen ignorieren, verlaufen Straßen nicht über einander. Daher erwarten wir, dass die Graphen von Straßennetzen fast planar sind. Nach dem Euler'schen Polyedersatz erfüllen einfache, planare und zusammenhängende Graphen:

Straßennetze

planare Graphen

$$|E| \leq 3|V| - 6 \tag{2.9}$$

Knoten in einem Straßennetz sollten also im Schnitt höchstens sechs Nachbarn haben.

In sozialen Netzwerken ist die Situation anders – stellen wir uns etwa einen Freundschaftsgraphen vor, in dem Knoten die Nutzer eines sozialen Netzwerks sind und Kanten eine Freundschaft anzeigen. Im Jahre 2014 hatten Facebook-Nutzer im Schnitt mehr als 300 Freunde (mehr dazu später). Dies ist offensichtlich deutlich mehr als in planaren Graphen möglich wäre. Ganz ähnlich sieht es mit anderen sozialen Netzen aus: Ein durchschnittlicher Erwachsener kennt deutlich mehr als sechs andere Menschen persönlich (oft werden Zahlen zwischen 100 und 300 genannt).

Freundschaftsnetze

2.2.2 Die meisten Netzwerke sind dünn

In Abb. 2.1 hat nur ein verschwindend geringer Anteil der Netzwerke mindestens die Hälfte aller Kanten (d. h. ist oberhalb der roten Linie). Wie erklärt sich das? In der Regel verursacht eine Kante Kosten: Eine Straße muss gebaut werden, eine Freundschaft muss aufrecht erhalten werden (Zeitinvestment), eine Nachricht muss geschrieben werden etc. Daher gibt es in den meisten Netzwerken einen gewissen Selektionsdruck, der dazu führt, dass jeder Knoten nur ausgewählte Nachbarn besitzt. Wir klassifizieren Netzwerktypen, die auffallend viele oder wenig Kanten haben:

dünne und dichte Graphen

Definition 2.2. Graphen mit n Knoten und m Kanten heißen *dünn* (engl. sparse), wenn $m = \mathcal{O}(n \log n)$ gilt, und *dicht* wenn $m = \Omega(n^2)$. ◀

Bemerkung 2.3. Aufgrund der asymptotischen Definition verwendet man *dünn* und *dicht* eher für Graphfamilien (so auch Zufallsgraphen) als für einzelne Instanzen. Manche abweichende Definitionen fordern für *dünne* Graphen $m = \mathcal{O}(n)$. Wir nutzen hier $\mathcal{O}(n \log n)$, um soziale Netzwerke mit einzubeziehen. ◀

2.3 Die Zufallsgraphen $\mathcal{G}(n, m)$ und $\mathcal{G}(n, p)$

Das $\mathcal{G}(n)$-Modell verfügt über keinen Mechanismus, um Kanten auszudünnen. Wir benötigen also Prozesse, die weniger dichte Graphen erzeugen können – am besten parametrisiert, damit wir unterschiedlichen Netzwerktypen Rechnung tragen können. Im Folgenden betrachten wir zwei solcher Modelle.

Erdős-Rényi-Graphen $\mathcal{G}(n, m)$

Das $\mathcal{G}(n, m)$-Modell von P. Erdős und A. Rényi beschreibt die Gleichverteilung über allgemeinen Graphen mit n Knoten und m Kanten, d. h., wir betrachten die Grundmenge

$$\mathbb{G}(n, m) = \{G \mid G = (V, E) \in \mathbb{G}(n) \text{ und } |E| = m\}\,. \tag{2.10}$$

Da gleichverteilt gewählt wird, gilt für die Wahrscheinlichkeitsverteilung $f : \mathbb{G} \to [0, 1]$

$$f(G) = \frac{1}{|\mathbb{G}(n, m)|} \qquad \text{für alle } G \in \mathbb{G}(n, m). \tag{2.11}$$

Aufgabe 2.4. Berechne $|\mathbb{G}(n, m)|$ für gerichtete und ungerichtete Graphen. ◀

Gilbert-Graphen $\mathcal{G}(n, p)$

E. Gilbert beschreibt ein ähnliches Modell, das $\mathcal{G}(n, p)$-Modell – das oft fälschlicherweise „Erdős-Rényi-Modell“ genannt wird. Um das Modell zu beschreiben, weichen wir von der bisherigen expliziten Definition der Grundmenge und Verteilung ab. Stattdessen spezifizieren wir eine randomisierte Konstruktionsvorschrift:

1. Erzeuge n Knoten $V = \{v_1, \ldots, v_n\}$.
2. Setze $E = \emptyset$.
3. Für jedes Paar von Knoten (v_i, v_j) führe ein unabhängiges Bernoulli Experiment durch. Mit Wahrscheinlichkeit p füge (v_i, v_j) als Kante zu E hinzu.
4. Gebe den Graphen $G = (V, E)$ zurück.

Graphisch kann man sich also $\mathcal{G}(n,p)$ als Adjazenzmatrix vorstellen, in der die Einträge unabhängig voneinander mit Wahrscheinlichkeit p auf 1 gesetzt werden:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{cases} 1 & \text{mit Wahrscheinlichkeit } p \\ 0 & \text{mit Wahrscheinlichkeit } 1-p \end{cases}$$

Aufgabe 2.5. Die genannte Konstruktion erzeugt gerichtete Graphen. Zeige, wie die Konstruktion so angepasst werden kann, dass gerichtete Graphen ohne Eigenschleifen erzeugt werden. Wie verändert sich dann die Adjazenzmatrix? Wie verhält es sich mit ungerichteten Graphen? ◀

Durch ihre einfache Konstruktion sind beide Zufallsgraphen bis heute sehr verbreitete Modelle in der Netzwerkforschung. Wir werden jedoch sehen, dass viele Eigenschaften von echten Netzwerken auch von $\mathcal{G}(n,p)$- oder $\mathcal{G}(n,m)$-Graphen nicht beschrieben werden können.

2.3.1 Anzahl von Kanten in $\mathcal{G}(n,p)$

Während bei $\mathcal{G}(n,m)$-Graphen die Anzahl der Kanten durch den Parameter m fixiert ist, ist $|E|$ bei $\mathcal{G}(n,p)$-Graphen eine Zufallsvariable.

erwartete Kantenanzahl $\mathbb{E}[|E|]$ in $\mathcal{G}(n,p)$

Lemma 2.6. Die erwartete Kantenanzahl m in einem gerichteten $\mathcal{G}(n,p)$-Graphen ist

$$\mathbb{E}[m] = pn^2. \qquad ◀$$

Beweis. Fixiere einen Graphen $G = (V,E)$ aus $\mathcal{G}(n,p)$. Für jede Kante (u,v) definiere die Indikatorvariable $I_{u,v}$, die anzeigt, ob die Kante $(u,v) \in E$ enthalten ist:

$$I_{u,v} = \begin{cases} 1 & \text{falls } (u,v) \in E, \\ 0 & \text{sonst.} \end{cases} \tag{2.12}$$

Somit folgt die Kantenanzahl m in G als Summe über die Indikatorvariablen

$$|E| = \sum_{u,v \in V} I_{u,v} = \sum_{(u,v) \notin E} 0 \quad + \sum_{(u,v) \in E} 1. \tag{2.13}$$

Per Definition von $\mathcal{G}(n,p)$ gilt $\mathbb{P}[I_{u,v}{=}1] = p$ und $\mathbb{P}[I_{u,v}{=}0] = 1-p$. Somit folgt für den Erwartungswert jeder Indikatorvariable

$$\mathbb{E}[I_{u,v}] = 1 \cdot p + 0 \cdot (1-p) = p \quad \text{unabhängig für alle } u,v \in V. \tag{2.14}$$

Durch die Linearität des Erwartungswertes folgt schließlich

$$\mathbb{E}[|E|] = \sum_{u,v \in V} \mathbb{E}[I_{u,v}] = \sum_{u,v \in V} p = pn^2. \tag{2.15}$$

□

Aufgabe 2.7. Zeige, dass die erwartete Anzahl an Kanten in einem ungerichteten $\mathcal{G}(n,p)$-Graphen $\mathbb{E}[|E|] = \binom{n}{2}p = pn(n-1)/2$ beträgt. ◀

Aufgabe 2.8. Zeige, dass $\mathcal{G}(n) = \mathcal{G}(n, 1/2)$. ◀

Wie wir am Beweis von Lemma 2.6 sehen, ergibt sich die Kantenanzahl m als Summe von unabhängigen Bernoulli-Zufallsvariablen; sie ist also selbst eine Zufallsvariable und binomial verteilt. Da uns die Binomialverteilung regelmäßig begegnen wird, wollen wir uns diese kurz in Erinnerung rufen.

Binomialverteilung.

Definition 2.9. Die Binomialverteilung $B_{N,p}(k)$ beschreibt die Wahrscheinlichkeit, dass bei N unabhängigen Bernoulli Experimenten mit Wahrscheinlichkeit p genau k Experimente erfolgreich sind. Es gilt

Binomialkoeffizient $\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$$\begin{aligned}
\mathbb{P}[B_{N,p}{=}k] &= \binom{N}{k} p^k (1-p)^{N-k} \\
\mathbb{E}[B_{N,p}] &= Np \\
\mathrm{Var}[B_{N,p}] &= Np(1-p),
\end{aligned}$$

wobei $\mathbb{E}[B_{N,p}]$ und $\mathrm{Var}[B_{N,p}]$ den Erwartungswert und die Varianz der Binomialverteilung beschreiben. ◀

Die Standardabweichung $\sigma = \sqrt{\mathrm{Var}[B_{N,p}]} \leq \sqrt{\mathbb{E}[B_{N,p}]}$ ist also relativ klein. Wir können daher davon ausgehen, dass für hinreichend großes N die Binomialverteilung recht stark um ihren Erwartungswert Np konzentriert ist. Daher sagen wir, dass $\mathcal{G}(n,m)$ und $\mathcal{G}(n,p)$ mit $p = m/n^2$ asymptotisch (d. h. für $n \to \infty$) äquivalent sind. Häufig ist es jedoch einfacher, $\mathcal{G}(n,p)$-Graphen zu analysieren, da – im Gegensatz zu $\mathcal{G}(n,m)$-Graphen – alle Kanten unabhängig gezogen werden.

2.4 Effizientes Ziehen von $\mathcal{G}(n,p)$-Graphen

Zufallsgraphen sind nicht nur in der theoretischen Analysen von Prozessen und Algorithmen nützlich, sondern auch für empirische Untersuchungen. Nehmen wir etwa an, dass wir einen Algorithmus implementiert haben und dessen Geschwindigkeit oder Qualität vermessen möchten. Dann können Zufallsgraphen unter anderem aus folgenden Gründen nützlich sein:

- Wenn die Generatorsoftware vorhanden ist, können wir synthetische Instanzen in quasi unbeschränkter Menge generieren.
- Parametrisierte Generatoren erlauben, den Einfluss gewisser Eigenschaften detailliert zu analysieren, z. B.. Skalierungsexperimente.
- Beobachtete Graphen haben oft „Rauschen“, d. h. nicht verstandene oder insignifikante Strukturen, die Messungen verfälschen können. Zufallsgraphen hingegen haben i. d. R. eine gut verstandene Struktur, die es uns oft ermöglicht, vor dem Testen Hypothesen aufzustellen.

- Einige Generatoren können anhand eines fixierten Startzustands (Random-Seed) dieselben Graphen wiederholt erzeugen. Wir müssen also die Eingaben nicht speichern/transferieren und können dennoch reproduzierbare Experimente durchführen.

Da in verschiedenen wissenschaftlichen Communities der Begriff „Generator“ unterschiedlich benutzt wird, hier noch eine Definition:

Graphgenerator

Definition 2.10. Ein Graphgenerator ist ein Algorithmus, der einen Graphen aus einem Netzwerkmodell erzeugt. Dieser respektiert die Verteilung des zugrundeliegenden Modells. ◀

2.4.1 Ein naiver Ansatz

Betrachten wir folgenden Graphgenerator für $\mathcal{G}(n,p)$-Graphen, der im Wesentlichen eine Eins-zu-eins-Implementierung der Definition von $\mathcal{G}(n,p)$ ist:

Algorithmus 1 : Naiver Graphgenerator für $\mathcal{G}(n,p)$-Graphen

Input : Anzahl der Knoten n und Verbindungswahrscheinlichkeit p
Output : Adjazenzmatrix eines zufälligen Graphen $G \sim \mathcal{G}(n,p)$

```
Allokiere eine Matrix A[1..n, 1..n]
for 1 ≤ i ≤ n do
    for 1 ≤ j ≤ n do
        Setze A[i,j] ← { 0  mit Wahrscheinlichkeit 1 − p
                       { 1  mit Wahrscheinlichkeit p
Gebe A zurück
```

Annahme: Aus einfachen Verteilungen kann in konstanter Zeit gezogen werden.

Um die Laufzeit dieses Algorithmus bestimmen zu können, treffen wir in dieser Veranstaltung folgende Annahme. Wir können unter anderem aus folgenden Zufallsverteilungen in konstanter Zeit ziehen:

- Ganzzahl uniform aus $[a,b]$ für $a,b \in \mathbb{Z}$
- Gleitkommazahl uniform aus $[a,b]$ für $a,b \in \mathbb{R}$
- Bernoulli (folgt aus vorherigem Punkt)
- Binomial, Geometrisch (siehe Abschnitt 2.4.3), Hypergeometrisch, Poisson

Tatsächlich können die meisten dieser Verteilungen nur in *erwartet* konstanter Zeit gezogen werden, praktisch sind die Fluktuationen jedoch so klein, dass wir sie vernachlässigen können. Das erlaubt es uns, uns auf die Eigenschaften der Graphgeneratoren zu konzentrieren.

Lemma 2.11. Algorithmus 1 erzeugt einen $\mathcal{G}(n,p)$-Graphen in $\Theta(n^2)$ Zeit. ◀

Aufgabe 2.12. Beweise das Lemma. ◀

Nun könnte man argumentieren, dass dieser naive Algorithmus bereits optimal ist: Die Ausgabe hat Größe $\Theta(n^2)$ und somit benötigen wir $\Omega(n^2)$ Zeit, um die Ausgabe

zu erzeugen. Außerdem hat ein Graph mit $p = \Omega(1)$ in Erwartung $\Theta(n^2)$ Kanten, was wiederum die Laufzeit von unten beschränkt.

Diese Worst-Case-Schranken passen jedoch nicht zu unserer Beobachtung in Abschnitt 2.2, dass Graphen in der freien Wildbahn meist dünn sind. Es wäre schön, wenn wir diese Graphen schneller erzeugen könnten, jedoch ist Alg. 1 für alle p gleich „schnell“. Hier helfen *ausgabesensitive Algorithmen* (output-sensitive algorithm), deren Laufzeit maßgeblich durch die Größe der Ausgabe bestimmt wird; die meisten Generatoren, die wir betrachten werden, fallen genau in diese Kategorie.

ausgabesensitive Algorithmen

Kantenlisten

Ausgabesensitive Algorithmen müssen jedoch eine Ausgabedatenstruktur nutzen, die sich an die Größe anpasst. Eine Matrix hat immer dieselbe Größe, unabhängig davon, wie viele Einsen (Kanten) vorhanden sind. Wir werden im Folgenden fast immer eine Kantenliste annehmen, die nur $\mathcal{O}(1)$ Speicherworte pro Kante benötigt.

2.4.2 Ziehen mit zufälligen Sprüngen

Wie können wir nun Alg. 1 verändern, so dass er – analog zu Kantenlisten – keine Arbeit für Nichtkanten verrichtet? Idee: Wir müssen diese einfach überspringen! Aber woher wissen wir, wo „Nullen“ gezogen werden, bzw. wie viele Nullen wir überspringen müssen? Zählen können wir sie nicht, ohne wieder direkt quadratische Arbeit zu verrichten.

effizientes Ziehen durchs Überspringen von Nullen

Ganz einfach: Wir fragen uns, wie viele Nullen ℓ Alg. 1 gesetzt hätte und überspringen diese. Zwischen diese zufälligen Sprünge setzen wir dann die Einsen ein. Wenn wir nun ℓ aus einer geeigneten Verteilung ziehen, haben wir zwei unterschiedliche Zufallsprozesse, deren Ausgabe aber nicht unterscheidbar ist. Generalisierungen dieser Methode werden uns noch häufiger begegnen.

n^2-dimensionaler Vektor statt $n \times n$-Matrix

Zur Vereinfachung stellen wir uns die Adjazenzmatrix $A = (a_{ij})_{ij}$ als einen Vektor $B = (b_k)_k$ vor, der dadurch entsteht, dass wir die Zeilen hinter einander anfügen (row-major layout). Eintrag a_{ij} wird also mit $b_{(i-1)n+j}$ identifiziert. Dann besteht unsere Aufgabe darin, genau alle b_k zu finden, die den Wert 1 haben.

$$\Big(\quad \cdots \overbrace{1}^{b_k} \quad \underbrace{0\ 0\ 0 \cdots 0\ 0\ 0}_{\ell \text{ zusammenhängende Nullen}} \quad \overbrace{1}^{b_{k+1+\ell}} \cdots \quad \Big) \tag{2.16}$$

Anzahl Nullen X geometrisch verteilt

Sei X die Zufallsvariable, welche die Anzahl der Nullen nach einer Eins modelliert.

$$\begin{aligned}
\mathbb{P}[X = 0] &= \mathbb{P}[\text{Nächster Versuch: } 1] = p && (2.17)\\
\mathbb{P}[X = 1] &= \mathbb{P}[\text{Nächster Versuch: } 0\text{, dann } 1] = (1-p)p && (2.18)\\
\mathbb{P}[X = 2] &= \mathbb{P}[\text{Nächster Versuch: } 0\text{, dann } 0\text{, dann } 1] = (1-p)^2 p && (2.19)\\
&\vdots && (2.20)\\
\mathbb{P}[X = \ell] &= (1-p)^\ell p && (2.21)
\end{aligned}$$

Die Zufallsvariable ist also *geometrisch verteilt* mit Parameter p. Es ergibt sich also der folgende einfache Algorithmus:

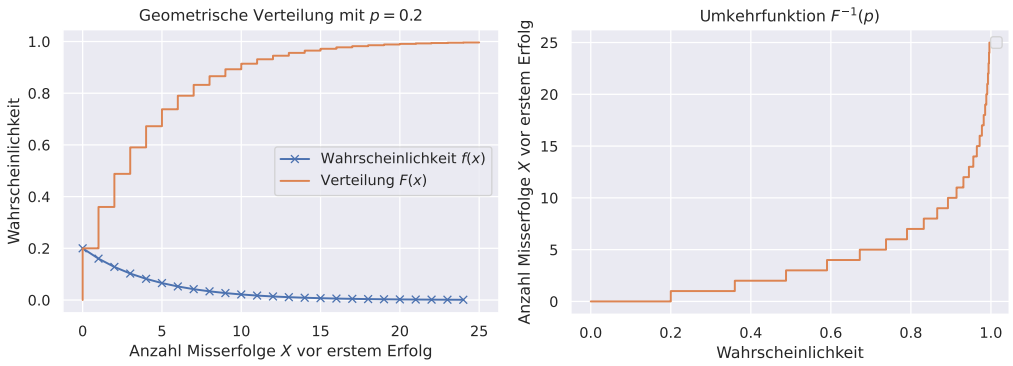


Abbildung 2.2: **Links**: Wahrscheinlichkeitsverteilung und kumulative Verteilungsfunktion einer geometrischen Verteilung mit Parameter $p = 0.2$. **Rechts**: „Umkehrfunktion“ $F^{-1}(p)$ der kumulativen Verteilungsfunktion $F(x)$.

Algorithmus 2 : Generator für $\mathcal{G}(n,p)$-Graphen mit zufälligen Sprüngen

Input : Anzahl der Knoten n und Verbindungswahrscheinlichkeit p
Output : Kantenliste E eines zufälligen Graphen $G \sim \mathcal{G}(n,p)$

```
Initialisiere eine leere Kantenliste E
k ← −1
while true do
    ℓ ← ziehe die Sprungweite aus einer geometrischen Verteilung mit Parameter p
    k ← k + ℓ + 1
    if k < n^2 then
        (i, j) ← (⌊k/n⌋, k mod n)
        E ← E ∪ {(i + 1, j + 1)}
    else
        Gebe E zurück und beende
```

Lemma 2.13. Algorithmus 2 erzeugt Graph $G = (V, E) \sim \mathcal{G}(n,p)$ in $\Theta(|E|)$ Zeit. ◀

Aufgabe 2.14. Beweise das Lemma. ◀

Aufgabe 2.15. Passe Alg. 2 für ungerichtete Graphen an. ◀

Jetzt müssen wir nur noch herausfinden, wie wir geometrische Sprünge effizient ziehen können. Hierzu kommt meist die Inversionsmethode zum Einsatz.

2.4.3 Inversionsmethode: Ziehen aus der geometrischen Verteilung

Inversionsmethode: effizientes Ziehen, falls Umkehrfunktion bekannt

Die Inversionsmethode (inversion transform method) ist ein allgemeines Verfahren, um eine uniforme Zufallsvariable in eine andere Verteilung zu übersetzen. Wir betrachten die Methode hier am Beispiel der geometrischen Verteilung. In Abb. 2.2 ist die Wahrscheinlichkeitsverteilung $f(\ell)$ und die kumulative Verteilungsfunktion $F(\ell)$ einer geometrischen Verteilung mit Parameter $p = 0.2$ gezeigt. Betrachten wir konkret die ersten drei Werte der Funktionen:

	$\ell = 0$	$\ell = 1$	$\ell = 2$
$f(\ell)$	$p = 0.2$	$(1-p)p = 0.16$	$(1-p)^2 p = 0.128$
$F(\ell)$	$p = 0.2$	$p+(1-p)p = 0.36$	$p+(1-p)p+(1-p)^2 p = 0.488$

Ein Generator sollte also z. B. den Wert 0 mit Wahrscheinlichkeit 20 % ausgeben. Da wir einen deterministischen Algorithmus konstruieren möchten, brauchen wir eine Quelle für den Zufall: Wir erwarten eine uniforme Zufallsvariable $U \in [0, 1)$ als Eingabe. Wir können also z. B. prüfen:

- Wenn $U < 0.2$ ist, dann gebe den Wert 0 zurück. Da U uniform verteilt ist, ist die Wahrscheinlichkeit für $U < 0.2$ genau 0.2.
- Falls nicht, prüfen wir, ob $U < 0.36 = f(0) + f(1) = F(1)$ ist. Falls ja, geben wir den Wert 1 zurück. Da U uniform verteilt ist die Wahrscheinlichkeit $\mathbb{P}[0.2 \leq U < 0.36] = 0.16 = f(1)$.
- Diesen Prozess setzen wir fort, bis wir das passende Intervall gefunden haben.

Die graphische Interpretation ist also die folgende: In Abb. 2.2 (rechts) ist die Umkehrfunktion $F^{-1}(p)$ der kumulativen Verteilungsfunktion $F(x)$ gezeigt. Wir werfen nun mittels der zufälligen Eingabe U einen Punkt auf der x-Achse und schauen, welchen Wert $F^{-1}(U)$ hat – dies ist unsere Ausgabe.

Im folgenden Theorem wird die Inversionsmethode formalisiert, wobei wir vereinfachend einige Annahmen treffen. Die Methode lässt sich aber auch auf andere Ω (inklusive kontinuierliche) und nicht streng monotone $F(x)$ verallgemeinern.

Theorem 2.16. Sei $X \in \Omega$ eine diskrete Zufallsvariable und $f(x)$ und $F(x)$ ihre Wahrscheinlichkeitsverteilung sowie die kumulative Verteilungsfunktion. Vereinfachend sei Ω total geordnet und $F(x)$ streng monoton steigend, sodass es eine Umkehrfunktion $F^{-1}(x)$ mit $F^{-1}(F(x)) = x$ gibt. Sei U eine uniforme Zufallsvariable aus $[0, 1)$. Dann ist $X' := F^{-1}(U)$ eine Zufallsvariable mit der Wahrscheinlichkeitsverteilung $f(x)$. ◀

Beweis. Da F streng monoton ist, existiert die Umkehrfunktion F^{-1} und es gilt:

$$F(x) = p \quad \Leftrightarrow \quad F^{-1}(F(x)) = F^{-1}(p) \quad \Leftrightarrow \quad x = F^{-1}(p) \tag{2.22}$$

Daraus folgt dann direkt:

$$\mathbb{P}\big[X' \leq x\big] \overset{\text{Inv.-Methode}}{=} \mathbb{P}\big[F^{-1}(U) \leq x\big] \tag{2.23}$$

$$\overset{(2.22)}{=} \mathbb{P}[U \leq F(x)] \tag{2.24}$$

$$\overset{\mathbb{P}[U<z]=z \forall z \in [0,1]}{=} F(x) \tag{2.25}$$

$$\overset{\text{Def. } F(X)}{=} \mathbb{P}[X \leq x] \qquad \square \tag{2.26}$$

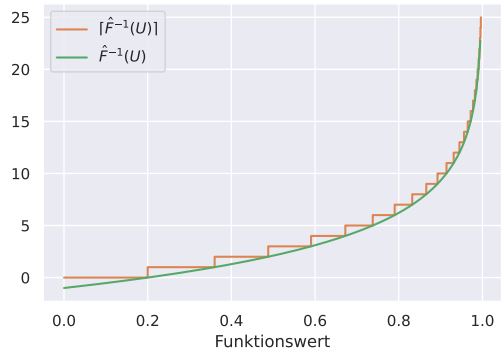


Abbildung 2.3: Die in Abschnitt 2.4.3 berechnet Umkehrfunktion $\hat{F}^{-1}(U)$

Inversionsmethode für geometrische Verteilungen

Zurück zur geometrischen Verteilung mit $f(\ell) = (1-p)^\ell p$. O. B. d. A. sei $0 < p < 1$, da X sonst eine Konstante ist. Die kumulative Verteilungsfunktion lautet

$$F(\ell) \;=\; p\sum_{i=0}^{\ell}(1-p)^i \;\overset{(*)}{=}\; p\underbrace{\frac{1-(1-p)^{\ell+1}}{1-(1-p)}}_{=p} \;=\; 1-(1-p)^{\ell+1}, \tag{2.27}$$

wobei wir in $(*)$ die geschlossene Form der ℓ-ten Partialsumme der geometrischen Reihe nutzen: $\sum_{i=0}^{\ell} x^\ell = (1-x^{\ell+1})/(1-x)$. Nun berechnen wir die Umkehrfunktion $\hat{F}^{-1}(U) = \ell$, indem wir $F(\ell) = U$ setzen und nach U umformen. Hierbei nehmen wir zunächst an, dass die Funktion stetig wäre (daher schreiben wir $\hat{F}^{-1}$ statt F^{-1}):

$$U \;=\; F(\ell) = 1-(1-p)^{\ell+1} \tag{2.28}$$

$$\Leftrightarrow (1-p)^{\ell+1} \;=\; 1-U \tag{2.29}$$

$$\Leftrightarrow (\ell+1)\log(1-p) \;=\; \log(1-U) \tag{2.30}$$

$$\Leftrightarrow \hat{F}^{-1}(U) := \log_{1-p}(1-U)-1 \;=\; \ell \tag{2.31}$$

Schließlich erhalten wir $F^{-1}(U) = \lceil \hat{F}^{-1}(U) \rceil$ durch Aufrunden der gerade berechneten Funktion. Zusammenfassend erzeugt Alg. 3 eine geometrische Zufallsvariable X mit Parameter p in Zeit $\mathcal{O}(1)$:

Algorithmus 3 : Ziehen einer geometrischen Zufallsvariable

```
1 if p = 0 then
2     Gebe ∞ zurück
3 else if p = 1 then
4     Gebe 0 zurück
5 else
6     U ← ziehe uniform aus [0, 1)
7     Gebe ⌈log_{1−p}(1 − U) − 1⌉ zurück
```

2.5 Effizientes Ziehen von $\mathcal{G}(n, m)$-Graphen

Trotz der strukturellen Ähnlichkeiten zwischen $\mathcal{G}(n, p)$- und $\mathcal{G}(n, m)$-Graphen unterscheiden sich ihre Generatoren signifikant. Grund hierfür ist, dass während $\mathcal{G}(n, p)$-Generatoren *in Erwartung* $n^2 p$ Kanten erzeugen, wir für $\mathcal{G}(n, m)$-Generatoren sicherstellen müssen, dass *exakt* m zufällige Kanten produziert werden.

Ähnlich wie zuvor reduzieren wir das Ziehen der richtigen Position von *Einsen* in der Adjazenzmatrix auf den eindimensionalen Fall. Im Folgenden werden wir also folgendes äquivalentes Problem lösen: Ziehe uniform ohne Zurücklegen k Elemente aus einer Menge $S = \{s_1, \ldots, s_N\}$ mit $|S| = N > k$.

2.5.1 Reservoir Sampling

Reservoir Sampling ist eine allgemeine Technik, die es uns erlaubt, k Elemente aus einem Datenstrom zu ziehen. Hierbei müssen wir anfangs *nicht* wissen, wie viele Elemente insgesamt im Strom sind.[1] Im konkreten Fall besteht der Datenstrom also aus allen s_i in beliebiger Reihenfolge. Vereinfachend können wir also annehmen, dass der Strom mindestens k Elemente enthält.

Algorithmus 4 : Reservoir Sampling

Input : Datenstrom S, Stichprobengröße k
Output : Array $R[1..k]$ mit k zufällig ausgewählten Elementen

```
1  Initialisiere ein anfangs leeres Array R mit Kapazität k
2  while |R| < k do
3      x ← nächstes Element aus S
4      Füge x an R an
5  i ← k
6  while S nicht leer do
7      x ← nächstes Element aus S
8      i ← i + 1
9      j ← ziehe uniform aus [1, i]
10     if j ≤ k then
11         R[j] ← x
12 Gebe R zurück
```

In Theorem 2.17 zeigen wir die Korrektheit von Alg. 4 für nur $k = 1$:

Theorem 2.17. Für $k = 1$ liefert Alg. 4 liefert ein uniform gewähltes Element $x \in S$. ◀

Beweis. Sei N die Länge des Stroms (ist dem Algorithmus nicht bekannt). Seien zudem $s_1, \ldots, s_N$ die Elemente in der Reihenfolge, in der sie aus dem Strom gelesen werden; o. B. d. A. gelte $s_i = s_j \Leftrightarrow i = j$ (d. h., alle Elemente sind unterschiedlich; falls nicht, können wir die Tupel (s_i, i) betrachten).

[1] Ein praktischer Anwendungsfall sind z. B. Iteratoren oder Generatoren, die in vielen Programmiersprachen genutzt werden. In Rust ist z. B. `rand::IteratorRandom::choose_multiple` mittels Reservoir Sampling implementiert.

Sei R die Zufallsvariable, welche die Ausgabe des Algorithmus modelliert. Dann ist zu zeigen, dass $\mathbb{P}[R = s_i] = 1/N$ für alle $1 \leq i \leq N$.

$$\begin{aligned} \mathbb{P}[R = s_i] &= \mathbb{P}[s_i \text{ wird in Runde } i \text{ gewählt}] \cdot \mathbb{P}[s_i \text{ wird nicht ersetzt}] && (2.32) \\ &= \frac{1}{i} \cdot \underbrace{\prod_{j=i+1}^{N} \frac{j-1}{j}}_{\text{Teleskop-Produkt}} = \frac{1}{i} \cdot \frac{i}{N} = \frac{1}{N} && (2.33) \end{aligned}$$

Insbesondere gilt also, dass $\mathbb{P}[R = s_i] = 1/N$ unabhängig von i oder s_i ist. □

Aufgabe 2.18. Zeige die Korrektheit von Alg. 4, indem du den Beweis von Theorem 2.17 auf $k \geq 1$ verallgemeinerst. Dies lässt sich etwa über Induktion über die Datenstromlänge $|S|$ erreichen. ◀

Wir können also $\mathcal{G}(n, m)$-Graphen mittels Reservoir Sampling erzeugen. Allerdings entspricht jede potentielle Kante einem Element in $S = V \times V$, für das wir je $\Theta(1)$ Arbeit verrichten müssen; somit folgt eine Gesamtlaufzeit von $\Theta(n^2)$.

2.5.2 Ziehen mit Lookups

Im Folgenden skizzieren wir einen alternativen Ansatz, um k Elemente ohne Zurücklegen aus dem Universum S zu ziehen; dieser basiert auf [4]. Dieser nutzt das Grundgerüst in Alg. 5, das wir um einige Details erweitern müssen, um eine praktisch schnelle Implementierung zu erreichen:

Algorithmus 5 : Basisalgorithmus für Ziehen ohne Zurücklegen

Input : Universum S, Stichprobengröße k
Output : Menge R mit k zufällig ausgewählten Elementen

```
1 Initialisiere R = ∅
2 while |R| < k do
3     x ← zufälliges Element aus S
4     if x ∉ R then
5         R ← R ∪ {x}
6 Gebe R zurück
```

Wir messen die Geschwindigkeit des Algorithmus zunächst nur indirekt, indem wir die Anzahl der Iterationen der `while`-Schleife zählen:

Lemma 2.19. Algorithmus 5 verwendet in Erwartung $N \log(N/(N-k+1)) + o(1)$ Versuche (Iterationen der `while`-Schleife), um aus S mit $|S| = N$ exakt $1 \leq k < N$ Elemente zu ziehen. ◀

Beweis. Der Algorithmus baut R inkrementell auf: Wir beginnen mit $|R| = 0$ und fügen dann schrittweise je ein Element hinzu. Sei X eine Zufallsvariable, die beschriebt, wie viele Iterationen der Algorithmus benötigt um R zu erzeugen. Ferner seien $X_1, \ldots, X_k$ Zufallsvariablen, wobei X_i beschriebt, wie viele Versuche benötigt werden, um das i-te

Elemente zu finden (d. h., um R von $|R| = i-1$ auf $|R| = i$ zu heben). Per Konstruktion gilt $X = \sum_{i=1}^{k} X_i$ und aufgrund der Linearität des Erwartungswertes folgt

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{k} X_i\right] = \sum_{i=1}^{k} \mathbb{E}[X_i]\,. \tag{2.34}$$

Wir können also zunächst die Erwartungswerte $\mathbb{E}[X_i]$ isoliert betrachten. Unmittelbar bevor wir das i-te Element ziehen, haben wir $i-1$ Elemente bereits bestimmt, d. h. $|R| = i-1$. Mit Wahrscheinlichkeit $q_i = (i-1)/N$ erwischen wir also ein bereits gezogenes Element und versuchen es erneut. Mit Wahrscheinlichkeit $p_i = 1 - q_i = (N-(i-1))/N$ ziehen wir aber ein Element, das noch nicht Teil von R ist. Es gilt also

$$\begin{aligned}
\mathbb{P}[X_i = 1] &= p_i \\
\mathbb{P}[X_i = 2] &= (1-p_i)p_i \\
\vdots \quad &= \quad \vdots \\
\mathbb{P}[X_i = 1+\ell] &= (1-p_i)^{\ell} p_i.
\end{aligned}$$

Die Zufallsvariable $X_i - 1$ ist also geometrisch verteilt mit Erfolgswahrscheinlichkeit p_i. Es gilt daher $\mathbb{E}[X_i] = 1/p_i = N/(N-i+1)$. Durch Einsetzen in Gleichung (2.34) erhalten wir:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{k} \frac{N}{N-(i-1)} = \sum_{i=0}^{k-1} \frac{N}{N-i} && (2.35)\\
&= N \sum_{i=N-k+1}^{N} \frac{1}{i} && (2.36)\\
&= N \left[\left(\sum_{i=1}^{N} \frac{1}{i}\right) - \left(\sum_{i=1}^{N-k} \frac{1}{i}\right)\right] && (2.37)\\
&= N\left[H_N - H_{N-k}\right] && (2.38)
\end{aligned}$$

Im letzten Schritt nutzen wir die Definition $H_t = \sum_{i=1}^{t} 1/i$, also die t-te Partialsumme der Harmonischen Reihe. Es gilt, dass $H_t = \ln t + \gamma + 1/(2t) + \varepsilon_t$, wobei $\gamma \approx 0.5772$ die Euler–Mascheroni-Konstante ist und $0 \le \varepsilon_t \le N^{-2}/8$ ein verschwindender Fehlerterm ist. Es folgt also:

$$\begin{aligned}
\mathbb{E}[X] &= N\left[H_N - H_{N-k}\right] && (2.39)\\
&\le N\left[\ln N - \ln(N-k+1) + \mathcal{O}\big(N^{-2}\big)\right] && (2.40)\\
&= N \ln \frac{N}{N-k+1} + o(1) && (2.41)
\end{aligned}$$

□

Beobachte, dass die Analyse im Beweis von Lemma 2.19 sehr eng ist. Wie wir in Abb. 2.4 sehen, gilt für relativ kleine N schon $\mathbb{E}[X] \approx N \ln \frac{N}{N-k+1}$.

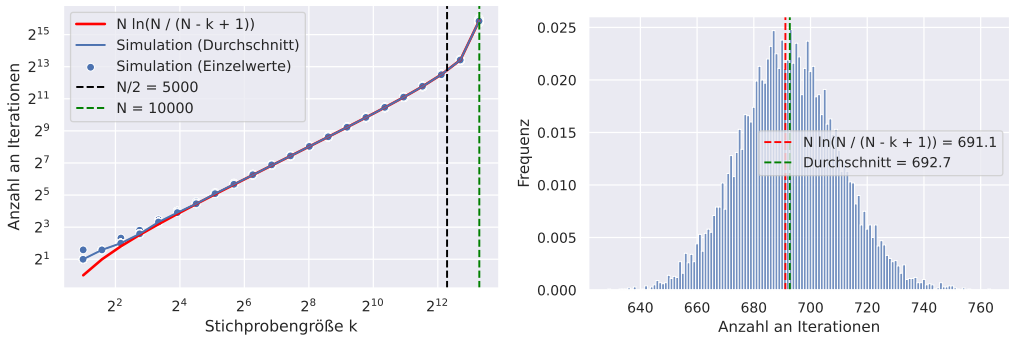


Abbildung 2.4: Simulation von Alg. 5 (ohne Komplementbildung). **Links**: $N = 10\,000$ und $2 \leq k \leq 3N/4$; für jeden Parameter wurden 1 000 unabhängige Simulationen ausgeführt. **Rechts**: $N = 1\,000$, $k = N/2$. Es wurden 10 000 unabhängige Simulationen ausgeführt.

Theorem 2.20. Für $1 \leq k \leq N/2$ verwendet Alg. 5 in Erwartung $\mathcal{O}(k)$ Versuche (Iterationen der `while`-Schleife), um aus S mit $|S| = N$ exakt $1 \leq k < N$ Elemente zu ziehen. ◀

Beweis. Wir wissen aus Lemma 2.19, dass der Algorithmus in Erwartung $T + o(1)$ Iterationen ausführt, wobei

$$T = N \ln \frac{N}{N - k + 1}. \tag{2.42}$$

Wir müssen also nur zeigen, dass $T = \mathcal{O}(k)$ für $1 \leq k \leq N/2$. Beobachte, dass unter diesen Annahmen das Argument des Logarithmus im Intervall $N/(N - k + 1) \in [1, 2]$ liegt. Wir nutzen daher die Taylorreihe von $\ln(x)$ am Punkt $x = 1$:

$$\ln(x) = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{1}{i} (x - 1)^i = (x - 1) - \frac{1}{2}(x - 1)^2 + \frac{1}{3}(x - 1)^3 + \dots \tag{2.43}$$

Da $x \in [1, 2]$ liegt, gilt $(x - 1) \in [0, 1]$ und daher gilt $(x - 1)^i \geq (x - 1)^{i+1}$. Daher können wir uns auf den ersten Term der Entwicklung beschränken und finden

$$\ln(x) \leq x - 1. \tag{2.44}$$

Hiermit können wir nun T abschätzen:

$$\begin{aligned}
T &= N \ln \frac{N}{N - k + 1} && (2.45)\\
&\overset{(2.44)}{\leq} N \left(\frac{N}{N - k + 1} - 1 \right) && (2.46)\\
&\leq N \frac{N - (N - k + 1)}{N - k + 1} && (2.47)\\
&\leq \frac{Nk}{N - k + 1} \leq \frac{Nk}{N - k} && (2.48)\\
&\leq \frac{k}{1 - k/N} \overset{k/N \leq 1/2}{\leq} 2k = \mathcal{O}(k)\,.
\end{aligned}$$

□

Aufgabe 2.21. Zeige einen alternativen Beweis von Theorem 2.20, der die Identität $\exp(x) = \lim_{n\to\infty}(1 + x/n)^n$ nutzt. ◀

Theorem 2.20 besagt, dass Alg. 5 für $k \leq N/2$ erwartet $\mathcal{O}(k)$ Iterationen ausführt und somit (in Erwartung) optimal ist. Nur für $k \to N$ wird der $\log$-Faktor asymptotisch relevant, und wir bekommen eine suboptimale Laufzeit von $\Omega(N \log N)$.

Komplementbildung stellt sicher: $k \geq N/2$

Auf der einen Seite sollte das für *übliche* Graphen kein Problem sein (sie sind dünn und daher $k \ll N$); dennoch lässt sich das Problem einfach vermeiden. Für $k > N/2$ können wir zunächst das Komplement $\bar{R}$ mit $|\bar{R}| = N - k < N/2$ ziehen und dann alle Elemente außer solche in $\bar{R}$ ausgeben. Zwar benötigt die Komplementbildung $R = \bar{\bar{R}}$ während der Ausgabe $\Theta(N)$ Zeit, allerdings hat die Ausgabe ebenfalls Größe $\Omega(N)$, wodurch das asymptotisch optimal bleibt.

2.5.3 Konzentration um den Erwartungswert

Bisher haben wir uns darauf konzentriert, Performance *in Erwartung* zu analysieren. In der Praxis kann das aber manchmal zu wenig sein: Wir stellen uns eine Implementierung vor, die in 99 % der Fälle in 1 s durchläuft, aber in 1 % der Ausführungen 101 s benötigt. Die durchschnittliche („erwartete") Laufzeit wären also 2 s. Wenn wir unsere Pipelines darauf auslegen, kann das zu (im schlimmsten Fall kaskadierenden) Timeouts führen.

Abbildung 2.4 (rechts) scheint zu zeigen, dass Alg. 5 nicht in diese Kategorie fällt. Unter 10 000 Ausführung ist die größte Abweichungen vom Mittelwert nur rund 12 %. Um Beobachtungen wie diese zu formalisieren, nutzt man häufig folgende Definition:

mit hoher Wahrscheinlichkeit

Definition 2.22. Eine Eigenschaft X gilt *mit hoher Wahrscheinlichkeit* (with high probability, whp), wenn für einen hinreichend großen Parameter n die Gegenwahrscheinlichkeit $\mathbb{P}[\neg X] \leq 1/n$ nicht übersteigt. ◀

In anderen Worten: Je „größer" unser Problem wird, desto unwahrscheinlicher sind extreme Ausreißer. In der Regel ist der Parameter aus dem Kontext klar und wird daher in der Literatur oft nicht erwähnt (im folgenden Lemma wäre es k):

Lemma 2.23. Algorithmus 5 verwendet mit hoher Wahrscheinlichkeit $\mathcal{O}(k)$ Iterationen für $k \leq N/2$. ◀

Beweis. Wir verwenden dieselbe Modellierung wie im Beweis von Lemma 2.19: Sei $X = \sum_{i=1}^{k} X_i$ die Gesamtanzahl an Iterationen und $X_i - 1$ geometrisch verteilt mit Erfolgswahrscheinlichkeit p_i.

Da $k \leq N/2$, ziehen wir zu jedem Zeitpunkt mindestens mit Wahrscheinlichkeit $1/2$ ein neues Element, d. h. $p_i \geq 1/2$ für alle $i \geq 1$. Um die harmonische Reihe zu vermeiden, nutzen wir eine schlechtere Abschätzung, bei der wir alle $p_i = 1/2$ auf den minimalen Erfolgswert setzen. Sei $X' = \sum_{i=1}^{k} X'_i$ diese Verteilung und $X'_i - 1$ geometrisch mit Erfolgswahrscheinlichkeit $1/2$.
Für einen beliebigen Grenzwert t gilt also

$$\mathbb{P}[X > t] \leq \mathbb{P}\big[X' > t\big] \,. \tag{2.49}$$

Wir können nun unsere Worst-Case-Abschätzung X' auch alternativ interpretieren: Wir werfen solange Münzen, bis wir k mal „Kopf“ gesehen haben. Die Anzahl der Versuche entspricht dann X'.

Wir formalisieren diese alternative Idee: Sei $Y^{(t)} = \sum_{i=1}^{t} Y_i$, wobei Y_i unabhängige Zufallsvariablen mit $\mathbb{P}[Y_i = 1] = 1/2$ sind. Per Definition gilt

$$\mathbb{P}[X > t] \overset{(2.49)}{\leq} \mathbb{P}\left[X' > t\right] = \mathbb{P}\left[Y^{(t)} < k\right] \leq \mathbb{P}\left[Y^{(t)} \leq k\right] \tag{2.50}$$

Außerdem gilt $\mathbb{E}\left[Y^{(t)}\right] = t/2$. Da Y_i unabhängige Bernoulli-Zufallsvariablen sind, hält die Chernoff-Ungleichung:

$$\mathbb{P}\left[Y^{(t)} \leq (1-\delta)\mathbb{E}[Y_t]\right] \leq \exp\left(-\delta^2 \mathbb{E}\left[Y^{(t)}\right]/2\right) \tag{2.51}$$

$$\overset{\delta:=1/2}{\Rightarrow} \quad \mathbb{P}\left[Y^{(t)} \leq t/4\right] \leq \exp\left(-(1/2)^2(t/2)/2\right) = \exp\left(-t/16\right) \tag{2.52}$$

Im letzten Schritt wählten wir $\delta = 1/2$; nun setzen wir $t = 4k$:

$$\mathbb{P}\left[Y^{(4k)} \leq k\right] \leq \exp(-k/4) \leq 1/k \quad \text{für } k \geq 9 \tag{2.53}$$

$$\Rightarrow \quad \mathbb{P}[X > 4k] \leq 1/k \tag{2.54}$$

Es folgt $\mathbb{P}[X \leq 4k] \geq 1 - 1/k$. Somit gilt $X < 4k$ mit hoher Wahrscheinlichkeit. □

Welche Datenstruktur benötigen wir für R? Sie sollte Existenzanfragen ($x \in R$?) und Einfügen ($R \leftarrow R \cup \{x\}$) effizient durchführen können. Ein Hashset unterstützt beide Operationen in erwartet konstanter Zeit, wodurch sich eine erwartete Laufzeit von $\mathcal{O}(k)$ ergibt; dies ist optimal in Erwartung.

Für praktische Implementierungen kann es sich jedoch auch lohnen, das Hashset durch ein Bitset (d. h. ein Array, in dem jeder Eintrag genau ein Bit groß ist) zu ersetzen. Auf dem Papier benötigt ein Bitset $\Theta(N)$ viele Bits und ist somit in der Initialisierung teuer. In der Praxis benötigt aber jeder Eintrag in einem Hashset mindestens 64 Bit, sodass ein Bitset bereits für $N/k > 64$ weniger Platz benötigt (das trifft etwa auf 30 % der Netzwerke in Abb. 2.1 zu). Zusätzlich sind Existenzanfragen und Einfügeoperationen extrem schnell, wodurch ein Bitset oft einen lohnenswerten Kompromiss zwischen mehr Speicher für schnellere Ausführung darstellen kann.

In Abb. 2.5 stellen wir Laufzeitmessungen für folgendes Experiment dar. Wir variieren die Größe unseres Universums $|S| = N$ von 2^8 bis 2^{32} und betrachten drei Szenarien:

- $k = 10$: Trotz steigender Universumsgröße ziehen wir immer nur 10 Samples.
- $k = \sqrt{N}$: Die Stichprobengröße steigt langsam in der Universumsgröße (dieser Fall entspricht einem $\mathcal{G}(n, m)$-Graphen mit $m = \Theta(n)$).
- $k = N/4$: Jedes vierte Element des Universums wird ausgewählt.

Aufgabe 2.24. Welche Effekte kannst du in Abb. 2.5 beobachten? Wie erklären sich diese? ◀

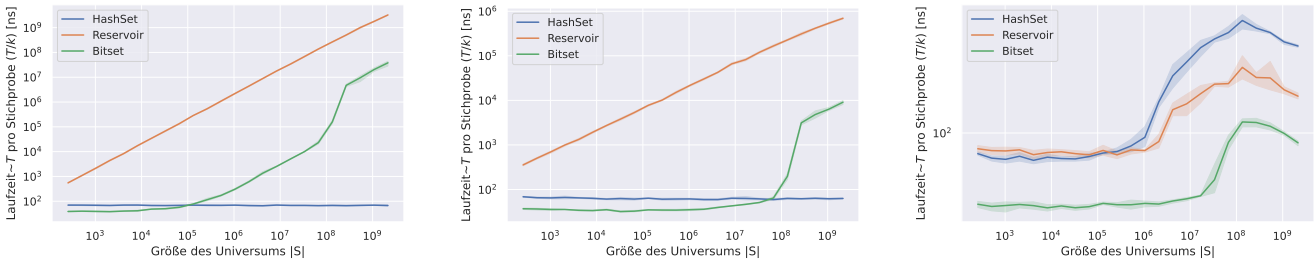


Abbildung 2.5: Laufzeit T pro Sample k für das Ziehen von k Elementen aus $S = \{1, \ldots, N\}$ als Funktion von $|S|$. **Links:** $k = 10$, **Mitte:** $k = \sqrt{N}$, **Rechts:** $k = N/4$.

Wir möchten uns aber nur auf einen zentralen Effekt in den Abbildungen konzentrieren, da dieser bei großen zufälligen Daten oft auftritt.[2] In Abb. 2.5 (rechts) beobachteten wir einen sprunghaften Anstieg der Zeit pro Element – teils um fast einen Faktor 10. Unsere Analysen sagen jedoch eine konstante Laufzeit pro gezogenem Element für $k = N/4$ voraus; einen sprunghaften Anstieg können sie nicht erklären.

Wir sind also im Algorithm-Engineering-Zyklus auf eine Diskrepanz zwischen Theorie und Praxis gestoßen! Ist unsere Analyse falsch? Jaein. In der Bestimmung der Laufzeit nutzen wir eine Unit-Cost-Random-Access-Machine (d. h., wir gingen davon aus, dass jede Operation gleich viel Zeit benötigt). Moderne Computer verfügen jedoch über ausgeklügelte Techniken, die gewisse Aspekte besonders beschleunigen.

Hierzu gehören die sog. Speicherhierarchien: Ein Prozessor kann nur auf Daten operieren, die sich auf dem physischen Chip befinden, genauer: in den Registern. Aus physikalischen und ökonomischen Gründen gibt es aber nur sehr wenig „Register-Speicher“. Die meisten aktiven Daten werden daher im Arbeitsspeicher vorgehalten; dieser ist aber recht langsam. Eine typische CPU kann in der Zeit, die es dauert, ein Datum aus dem Arbeitsspeicher zu beziehen, etwa 100 bis 1000 Operationen ausführen. Um dies aufzufangen, werden mehrere Caches eingefügt: Die CPU „rät“, welche Daten in naher Zukunft benötigt werden und hält diese im schnelleren Cache vor. Dies klappt aber mit Zufallsdaten oft nicht (sie sollen ja gerade *nicht* vorhersehbar sein).

In Abb. 2.5 (rechts) sehen wir also genau den Punkt, ab dem die Datenstrukturen nicht mehr in den Cache passen. Für Hashset und Reservoir-Sampling nutzen diese Datenstrukturen etwa $64k = 16N$ Bits (das Hashset aufgrund des Load-Factors < 1 auch ein bisschen mehr). Das Bitset benötigt N Bits – daher kommt hier der Performanceeinbruch auch später.

Um über diese qualitative Erklärung hinaus das Verhalten analytisch zu erklären, benötigen wir also ein Maschinenmodell, das auch Speicherzugriffe berücksichtigt. Dem widmen wir uns aber erst später. Im Folgenden betrachten wir stattdessen eine Parallelisierung des Algorithmus – dieselbe Strategie führt jedoch auch zu einem Generator, der zufällige Speicherzugriffe auf den Cache beschränken kann. Wir brauchen zunächst ein Maschinenmodell für parallele Berechnungen.

[2] foreshadowing…

2.5.4 Fork-Join-Parallelismus

Das Fork-Join-Modell ist eine Erweiterung der Random-Access-Machine, welche die RAM um die zwei namensgebenden Instruktionen erweitert. Eine Berechnung startet als gewöhnliches sequentielles Programm, einem sog. Task. Angenommen, wir führen gerade Task t_0 aus, dann können wir die Berechnung in zwei „Kindertasks" t_1 und t_2 teilen (*forken*). Beide Tasks werden unabhängig von einander ausgeführt, während t_0 ruht. Sobald t_1 und t_2 fertig sind, kommt es zur Wiedervereinigung (*join*): t_1 und t_2 werden „gelöscht" und t_0 wieder geweckt, der seine Berechnung fortsetzt. Betrachten wir das parallele Summieren $\sum_{i=1}^{n} x_n$ von Zahlen $X = (x_1, \ldots, x_n)$ als einfaches Beispiel:

fork

join

Algorithmus 6 : Parallele Summe im Fork-Join Modell

1 **Function** ParSum($X = (x_1, \ldots, x_n)$)
2   **if** $n = 1$ **then**
3   Gebe x_1 zurück und beende Task
4   Berechne Mitte $m \leftarrow \lfloor n/2 \rfloor$
5   Fork($\underbrace{\text{ParSum}(X_L = (x_1, \ldots, x_m))}_{\textit{Task } t_1}$, $\underbrace{\text{ParSum}(X_R = (x_{m+1}, \ldots, x_n))}_{\textit{Task } t_2}$)
6     $(s_1, s_2) \leftarrow$ Join(t_1, t_2)
7   Gebe $s_1 + s_2$ zurück und beende Task

In Alg. 6 teilen wir die Eingabe rekursiv in zwei gleich große Teile (modulo Rundung) und berechnen die Teilsummen für beide rekursiv. Sobald die Teilergebnisse vorliegen, summieren wir diese in $\mathcal{O}(1)$ Zeit und geben es als Gesamtergebnis zurück. Dieses Schema funktioniert analog für alle assoziativen Operationen (z. B. Produkt, Min, Max).

In der Praxis findet man einige Software-Bibliotheken, die Fork-Join-Parallelismus zur Verfügung stellen; für C++ etwa `Cilk` oder Intels `oneTBB`, für Rust `rayon`. Diese praktischen Lösungen nutzen einen sog. Work-Stealing-Scheduler.

Grob kann man annehmen, dass der Prozessor P_0, der t_0 ausführte, den Task t_2 als verfügbar bekannt gibt und dann selbst t_1 ausführt. Wenn während der Ausführung von t_1 ein anderer Prozessor verfügbar ist, übernimmt dieser t_2. In diesem Fall sind t_1 und t_2 also nebenläufig. Wenn sich kein anderer Prozessor findet, wird P_0 den Task t_2 übernehmen, sobald t_1 beendet wurde. Wir erhalten also ein dynamisches System, das (i) nicht wissen muss, wie viele Prozessoren es gibt, und (ii) auch relativ gut damit umgehen kann, dass t_1 und t_2 evtl. sehr unterschiedliche Laufzeiten haben. Praktisch ist es kein Problem, viel mehr Tasks zu erzeugen als es Prozessoren gibt.

Work Stealing

Wir messen die Güte eines Fork-Join-Algorithmus mit zwei Eigenschaften:

1. Die *Arbeit* (engl. work) ist die Summe aller Instruktionen, die das Programm ausführt. Dies entspricht also der Laufzeit, wenn nur ein Prozessor zur Verfügung steht. Für Alg. 6 ergibt sich die Arbeit als

Arbeit / work

$$W(n) = \begin{cases} \mathcal{O}(1) & \text{falls } n = 1 \\ 2 \cdot W(n/2) + \mathcal{O}(1) & \text{falls } n > 1 \end{cases} = \mathcal{O}(n)\,. \tag{2.55}$$

arbeitsoptimal

Der Algorithmus leistet also asymptotisch nicht mehr Arbeit als die Laufzeit der besten sequentiellen Lösung; wir sagen, er ist *arbeitsoptimal*.

Span / Parallel Depth

2. Der Span (engl. span oder parallel depth) misst die Anzahl der Instruktionen auf einem kritischen Pfad (d. h. eine längste Folge abhängiger Schritte). Dies entspricht der Laufzeit, wenn wir unbeschränkt viele Prozessoren zur Verfügung haben. In Alg. 6 müssen wir also analysieren, wie viele Instruktionen vor und nach dem Fork/Join Paar erforderlich sind und wie tief die Rekursion läuft:

$$S(n) = \underbrace{\mathcal{O}(1)}_{\text{Arbeit vor/nach Fork/Join}} + \underbrace{S(n/2)}_{\text{Rekursion}} = \mathcal{O}(\log n)\,. \tag{2.56}$$

Wenn wir ausreichend Prozessoren zur Verfügung haben, können wir also n Zahlen in Zeit $\mathcal{O}(\log n)$ summieren. Zu ähnlichen Ergebnissen kommt man auch in anderen parallelen Maschinenmodellen (z. B. PRAMs).

2.5.5 Rekursives Ziehen

In Alg. 6 haben wir bereits ein gutes Grundgerüst für parallele Algorithmen gesehen: Wenn wir es schaffen, das Problem *schnell* in unabhängige Hälften zu teilen und dann rekursiv zu bearbeiten, sind wir auf einem guten Weg. Bezogen auf unser Problem, k Stichproben aus N Elementen zu ziehen, haben wir zwei Dimensionen:

1. Teile S deterministisch in S_1 und S_2: Wir können das Universum S mit $|S| = N$ in zwei Hälften S_1 und S_2 partitionieren mit $|S_1| = \lfloor N/2 \rfloor$. Dann stellt sich die Frage: Was ist die Anzahl k_1 der Samples aus S_1 bzw. k_2 aus S_2? Klar ist nur $k_1 + k_2 = k$.

2. Teile k deterministisch in k_1 und k_2: Wir können die Stichproben teilen, d. h. $k_1 = \lfloor k/n \rfloor$ wählen; dann müssen wir jedoch eine unbekannte Partitionierung von S finden.

Auf den ersten Blick wirkt die zweite Variante zielführender. Es ist vorteilhaft, wenn beide Teilprobleme etwa die gleiche Stichprobenanzahl haben, da die Arbeit in dieser Anzahl skalieren soll. Tatsächlich liefert aber die erste Variante einfachere Algorithmen, die auch keine signifikanten Probleme mit unbalancierten Teilproblemen haben.

Wie in Abb. 2.6 teilen wir also das Universum S in zwei gleich große Hälften S_1 und S_2. Angenommen, wir täten dies nicht, wie viele Stichproben k_1 würde ein anderer Algorithmus aus S_1 ziehen? Wir wissen es nicht genau, da k_1 eine Zufallsvariable ist – sie ist hypergeometrisch verteilt: aus einer Gesamtpopulation von N mit k Treffern (d. h. die Elemente, die ein anderer Algorithmus gezogen hätte), ziehen wir N_1 Stichproben und erhalten k_1 Treffer. In Erwartung sind es also $\mathbb{E}[k_1] = k|S_1|/|S|$.

Analog zu den zufälligen Sprüngen in Abschnitt 2.4.2 ziehen wir k_1 aus einer hypergeometrischen Verteilung. Dann setzen wir den Prozess bedingt auf k_1 fort (Alg. 7).

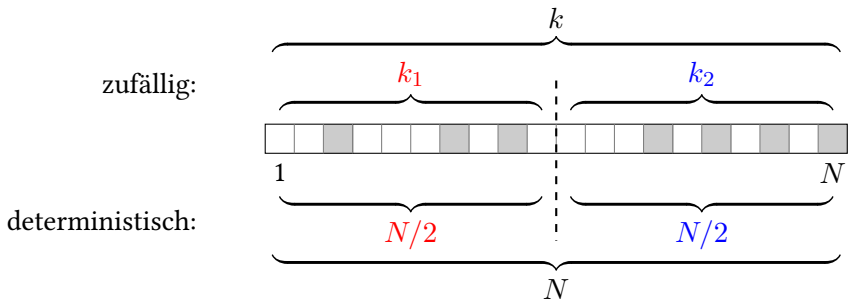


Abbildung 2.6: Ziehen von $k = 7$ Stichproben aus $N = 20$ Elementen ohne Zurücklegen; die grauen Elemente seien ausgewählt. Die linke Hälfte des Universums liefert k_1 Stichproben, die rechte k_2. Im gezeigten Beispiel gilt $k_1 = 3$ und $k_2 = 4$.

Algorithmus 7 : Paralleles Ziehen von k Stichproben aus S ohne Zurücklegen

```
1  Function ParSample(S = {s_1, ..., s_N}, k)
2      if k = 0 then
3          Gebe ∅ zurück und beende Task
4      else if k = 1 then
5          x ← zufällig uniform aus S
6          Gebe {x} zurück und beende Task
7      else if k = N then
8          Gebe S zurück und beende Task
9      N_L ← ⌊N/2⌋
10     k_L ← zufällig hypergeometrisch: ziehe N_L Element aus N mit k Treffern
11     Fork(ParSample(S_1 = {s_1, ..., s_{N_L}}, k_L), ParSample(S_2 = {s_{N_L+1}, ..., s_N}, k − k_L))
       \_____Task t_1_____/  \_____Task t_2_____/
12     (R_1, R_2) ← Join(t_1, t_2)
13     Gebe R_1 ∪ R_2 zurück und beende Task
```

Um die Performance von Alg. 7 zu analysieren, treffen wir zwei Annahmen:

1. Wir können S in $\mathcal{O}(1)$ Zeit in zwei (fast) gleich große Hälften teilen. Das ist in vielen Fällen möglich:
   - Für $\mathcal{G}(n, m)$-Graphen gehen wir davon aus, dass S nur implizit existiert und das Intervall $[1, n^2]$ repräsentiert. Dies können wir durch Verschieben der Intervallgrenzen trivial in $[1, n^2/2]$ und $[n^2/2 + 1, n^2]$ teilen.
   - Wenn S ein Array ist, können in die Rekursion Pointer auf den Anfang und das Ende des Teilproblems gegeben.
2. Die Operationen $R_1 \cup R_2$ läuft in konstanter Zeit. Das ist realistisch: Da die Elemente in der Menge S paarweise verschieden sind, wissen wir, dass $|R_1 \cup R_2| = |R_1| + |R_2| = k$. Es ist also kein Test auf Duplikate o. Ä. notwendig und es reicht

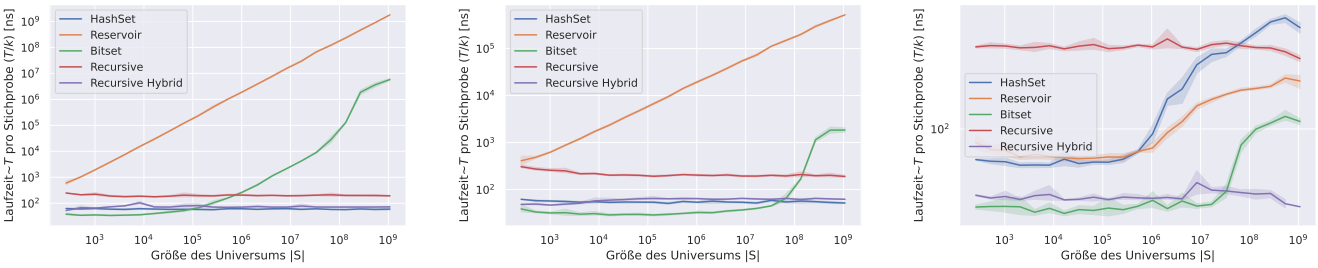


Abbildung 2.7: Laufzeit T pro Sample k für das Ziehen von k Elementen aus $S = \{1, \ldots, N\}$ als Funktion von $|S|$. **Links:** $k = 10$, **Mitte:** $k = \sqrt{N}$, **Rechts:** $k = N/4$.

aus, R_1 und R_2 zu konkatenieren. Das ist mit verketten Listen (inkl. Endpointer) in $\mathcal{O}(1)$ Zeit möglich.

Aufgabe 2.25. Angenommen, wir möchten R als Array bekommen. Zeige, dass es möglich ist R, eingangs mit einer Kapazität von k zu initialisieren und dann in der Rekursion direkt in R zu schreiben. Die Samples sollen dann in derselben relativen Reihenfolge wie in S erscheinen. ◀

Lemma 2.26. Algorithmus 7 hat mit hoher Wahrscheinlichkeit einen Span von $\mathcal{O}(\log k)$ und benötigt in Erwartung $\mathcal{O}(k)$ Arbeit. ◀

Wir überspringen den Beweis; dieser lässt sich aber analog zu [10] führen. Die Hauptideen sind: Ein Span von $\mathcal{O}(\log N)$ folgt trivial wie in unserem Beispiel zum parallelen Summieren dadurch, dass wir in jedem Schritt die Größe von $|S|$ halbieren. Um den Span jedoch auf $\mathcal{O}(\log k)$ zu reduzieren, müssen wir zeigen, dass sich auch die Stichprobengröße ungefähr gleich auf beide Teilprobleme verteilt. Dies folgt aus der relativ kleinen Varianz der hypergeometrischen Verteilung für hinreichend großes k. Nur für sehr kleines k laufen wir Gefahr, dass eines der beiden Teilprobleme alle Samples enthält; das ist aber zum einen unkritisch (weil $|S|$ trotzdem halbiert wird und so die Dichte k/N steigt) und zum anderen, wie wir gleich sehen werden, für praktische Implementierungen unerheblich.

Rekursionsstopper

Wie in Abb. 2.7 zeigt, weist eine (sequentielle) Implementierung von Alg. 7 ein gutes Skalierungsverhalten auf. Die Laufzeit hängt nur von $\mathcal{O}(k)$ ab und steigt nicht für großes N – sie ist jedoch allgemein recht hoch (das Ziehen einer hypergeometrischen Zufallsvariable ist recht langsam). Dies ist ein sehr gängiges Problem in komplexen Algorithmen mit einer einfachen Lösung. Wir verwenden den aufwendigeren Algorithmus nur so lange, wie es von Vorteil ist. Wenn die Teilprobleme hinreichend klein sind und wir alle Prozessoren auslasten, schalten wir als „Rekursionsstopper“ auf spezielle Basisalgorithmen um, die sich besonders gut für kleine Teilprobleme eignen. Der Hybridalgorithmus in Abb. 2.7 nutzt für große N den rekursiven Ansatz. Für kleine Teilprobleme mit $N < 10\,000$ kommt – je nach Dichte k/N – entweder BitSet oder HashSet zum Einsatz.

2.6 Phasenübergänge in $\mathcal{G}(n,p)$ und $\mathcal{G}(n,m)$

Die ersten analytischen Ergebnisse von Zufallsgraphen betrafen das Verhalten von kombinatorischen Eigenschaften für verschiedene Dichten (d. h. m/n bei $\mathcal{G}(n,m)$ bzw. p bei $\mathcal{G}(n,p)$) im Grenzwert für $n \to \infty$. In diesem Kapitel werden wir uns auf ungerichtete Graphen konzentrieren.

Wir nutzen ungerichtete Graphen und setzen $N = \binom{n}{2}$.

Betrachten wir zunächst $\mathcal{G}(n,m)$-Graphen. Für diese zeigten Erdős und Rényi bereits 1960 einige Schwellwerte, an denen sog. Phasenübergänge stattfinden. Für *sehr* dünne Graphen mit $m = o(\sqrt{n})$ bewiesen sie etwa, dass diese wahrscheinlich nur isolierte Knoten und Kanten beinhalten. Wir wollen uns an dieser Aussage ansehen, wie man die Wahrscheinlichkeit eines solchen Ereignisses abschätzen kann.

Lemma 2.27. Sehr dünne $\mathcal{G}(n,m)$-Graphen mit $m = o(\sqrt{n})$ haben für $n \to \infty$ wahrscheinlich keine Knoten mit mehr als einer Kante. ◀

Beweis. Sei $p(n,m)$ die Wahrscheinlichkeit, dass keine zwei Kanten einen gemeinsamen Endpunkt haben. Wir vergleichen nun die Anzahl der möglichen Graphen mit der Anzahl der günstigen Graphen (d. h. ohne zwei Pfade). Zunächst beobachten wir, dass es $|\mathbb{G}(n,m)| = \binom{N}{m} = \binom{\binom{n}{2}}{m}$ verschiedene $\mathcal{G}(n,m)$-Graphen gibt. Wie viele von diesen haben keine Knoten mit mindestens zwei Kanten?

Stellen wir uns dazu vor, dass wir die Kanten $E = \{\{u_1, v_1\}, \ldots, \{u_m, v_m\}\}$ erstellen, indem wir $2m$ Knoten $u_1, v_1, u_2, v_2, \ldots, u_m, v_m$ ziehen. Da keine zwei Kanten einen gemeinsamen Endpunkt haben sollen und Schleifen verboten sind, müssen wir also $2m$ <u>verschiedene</u> Knoten ziehen! Es ist aber egal, in welcher Reihenfolge wir die m Kanten ziehen und ob ein Endpunkt als u_i oder v_i gezogen wird. Die Anzahl ergibt sich also als

$$\frac{\overbrace{\binom{n}{2m}}^{2m \text{ verschiedene Knoten aus } n} \quad \overbrace{(2m)!}^{\text{Mögliche Permutation von } 2m \text{ Elementen}}}{\underbrace{m!}_{\text{Reihenfolge der Kanten egal}} \quad \underbrace{2^m}_{\text{Reihenfolge in einer Kante egal}}}. \tag{2.57}$$

Da wir jeden möglichen Graphen mit derselben Wahrscheinlichkeit ziehen, gilt

$$\bar{p}(n,m) = \frac{\binom{n}{2m}(2m)!}{m!2^m\binom{\binom{n}{2}}{m}}. \tag{2.58}$$

Wir werden im Folgenden eine *untere* Schranke für $\bar{p}(n,m)$ zeigen. Dazu nutzen wir folgende Abschätzung von $\binom{n}{k}$

$$\binom{n}{k} \approx \frac{n^k \exp\left(-\frac{k^2}{2n} - \frac{k^3}{6n^2}\right)}{k!} \qquad \text{für } k = o(n^{3/4}). \tag{2.59}$$

Der Einfachhalt halber verzichten wir auf die Analyse des Fehlers dieser Abschät-

zung. Es folgt

$$
\begin{aligned}
p(n,m) &\geq \frac{\binom{n}{2m}(2m!)}{m!2^m\binom{n^2/2}{m}} && (2.60)\\
&= \frac{\binom{n}{2m}(2m!)}{\binom{n^2/2}{m}m!2^m} && (2.61)\\
&\approx \frac{n^{2m}\exp\left(-2\frac{m^2}{n}-\frac{8m^3}{6n^2}\right)m!(2m!)}{(2m)!\underbrace{(n^2/2)^m}_{=n^{2m}/2^m}\exp\left(-2\frac{m^2}{n^4}-\frac{8m^3}{6n^6}\right)m!2^m} && (2.62)\\
&= \exp\left(-2\frac{m^2}{n}(1-\frac{1}{n^3})-\frac{8m^3}{6n^2}(1-\frac{1}{n^4})\right) && (2.63)\\
&\geq \exp\left(-2\frac{m^2}{n}-\frac{8m^3}{6n^2}\right) && (2.64)\\
&\geq \exp\left(-4\frac{m^2}{n}\right) \quad \text{für hinreichend großes } n && (2.65)
\end{aligned}
$$

Damit gilt also $\lim_{n\to\infty} p(n, m(n)) = 1$, falls $m(n) = o(\sqrt{n})$. □

Beobachte, dass Gleichung (2.58) eine exakte Wahrscheinlichkeit ist, die wir erst im Nachgang von unten abschätzen. Durch eine geeignete obere Schranke lässt sich zeigen, dass für $m \geq c\sqrt{n}$ für ein konstantes $c > 0$ (d. h. nicht von n abhängig) die Wahrscheinlichkeit mindestens eines Knotens mit mindestens zwei Nachbarn gegen 1 geht. Wir sagen daher, dass $\mathcal{G}(n, m)$-Graphen einen Phasenübergang mit Schwellwertfunktion $m(n) = \sqrt{n}$ haben.

Wir wechseln nun zu $\mathcal{G}(n, p)$-Graphen. Hier werden wir einen Beweis sehen, der die Unabhängigkeit aller Kanten ausnutzt. Zuerst schauen wir uns aber einige wohlbekannte Phasenübergänge an. Sei $\varepsilon > 0$ hinreichend klein. Dann gilt:

- Falls $np < 1 - \varepsilon$, haben mit hoher Wahrscheinlichkeit alle Zusammenhangskomponenten Größe $\mathcal{O}(\log n)$.
- Falls $np = 1$, hat die größte Zusammenhangskomponente mit hoher Wahrscheinlichkeit Größe $\Theta(n^{2/3})$.
- Falls $np > 1$, gibt es mit hoher Wahrscheinlichkeit eine einzelne Zusammenhangskomponente mit Größe $\Theta(n)$. Keine andere Komponente hat mehr als $\mathcal{O}(\log n)$ Knoten.
- Falls $np < (1 - \varepsilon)\log n$, gibt es mit hoher Wahrscheinlichkeit einzelne Knoten ohne inzidente Kanten.
- Falls $np > (1 + \varepsilon)\log n$, ist der Graph mit hoher Wahrscheinlichkeit zusammenhängend.

Aufgabe 2.28. Zeige ein möglichst großes $p(n)$, für das $\mathcal{G}(n,p)$-Graphen mit hoher Wahrscheinlichkeit keine Kanten haben. ◀

Wir möchten nun das Verhalten um $np = 1$ besser verstehen und folgen einem algorithmischen Beweis von [11]. Die Idee ist relativ einfach: Wir führen eine Tiefensuche auf einem $\mathcal{G}(n,p)$-Graphen aus, wobei die Nachbarschaft durch Ziehen von Einträgen in der Adjazenzmatrix erkundet wird. Dann zeigen wir, dass für $np < 1$ so wenig Einsen vorhanden sind, dass es sehr unwahrscheinlich ist, eine große Zusammenhangskomponente zu beobachten. Für $np > 1$ hingegen sehen wir ausreichend viele Einsen.

2.6.1 Tiefensuche auf $\mathcal{G}(n,p)$

Im Folgenden definieren wir eine klassische Tiefensuche (DFS) auf eine Weise, die für den folgenden Beweis hilfreich ist. Implementieren sollte man sie so besser nicht... Während einer DFS können wir die Knotenmenge V in drei Teilmengen B, S, U partitionieren:

- Bearbeitete Knoten B sind Knoten, die besucht wurden und keine Kinder in U mehr haben.
- Knoten in S werden gerade erkundet und liegen daher auf dem Stack.
- Unbekannte Knoten wurden noch nicht gefunden.

Bevor die DFS startet, gilt also $B = S = \emptyset$ und $U = V$. Die Suche terminiert sobald $B = V$ und $S = U = \emptyset$. Solange dies noch nicht der Fall ist, wird einer der folgenden Schritte ausführt:

1. Falls $S \neq \emptyset$, sei v der Knoten, der als letztes eingefügt wurde (S ist ein Stack).
   (a) Falls v einen Nachbarn $w \in U$ hat: Entferne w aus U und füge w in S ein.
   (b) Falls nicht, ist v vollständig bearbeitet: Entferne v aus S und füge v in B ein.
2. Falls $S = \emptyset$, entnehme den ersten Knoten $v \in U$ und füge ihn in S ein.

Um Uneindeutigkeiten zu vermeiden, fixieren wir die Knotenordnung. Hierzu nehmen wir o. B. d. A. an, dass $V = \{1, \dots, n\}$ natürliche Zahlen sind. Die Mengen V und U seien aufsteigend geordnet, insbesondere werden immer die kleinsten passenden Elemente entnommen (in Schritten 1a und 2). Fürderhin sei S ein Stack, d. h. das Element, das als letztes eingefügt wurde, wird als erstes entfernt.

Das Besondere an unserer DFS ist, dass wir keine Kantenliste als Eingabe bekommen, sondern uns vorstellen, einen Strom von unabhängig zufälligen Bits $X_1, \dots, X_N$ zu erhalten, wobei $\mathbb{P}[X_i = 1] = p$. Wenn wir nun in Schritt 1a einen Nachbar von v suchen, iterieren wir gleichzeitig über U und X. Dann geben wir das erste w zurück, dessen Bit gesetzt war (falls es existiert). Wenn wir das nächste Mal mit Knoten v zu Schritt 1a kommen, suchen wir nur nach $w' \in U$ mit $w' > w$ – wir machen also an der Stelle weiter, an der wir einen Nachbarn gefunden haben.

Aufgabe 2.29. Zeige, dass jeder Eintrag der Adjazenzmatrix höchstens einmal betrachtet wird und wir mittels der X_i somit ein DFS auf $\mathcal{G}(n,p)$ simulieren. Die zweite Beobachtung (s.u.) folgt hieraus. ◀

Wir werden später folgende Beobachtung ausnutzen:

- In jeder Iteration wird ein Knoten entweder von U nach S (Schritt 1a oder 2) oder von S nach B (Schritt 1b) bewegt. Es gibt also einen stetigen Fortschritt und der Algorithmus terminiert nach $\mathcal{O}(n)$ Schritten.
- Zu jedem Zeitpunkt gilt, dass der Graph keine Kanten zwischen U und B hat; insb. ändern noch nicht entdeckte Kanten hieran nichts mehr.
- Alle Knoten in S gehören zur selben Zusammenhangskomponente.

2.6.2 Zusammenhangskomponenten für $np = (1-\varepsilon)$ sind klein

Wenn wir nun DFS ausführen, werden wir unsere Strukturaussagen zu $\mathcal{G}(n,p)$ auf Eigenschaften Stroms von Zufallsbits reduzieren. Die Hauptarbeit dieser Analyse leisten wir in Lemma 2.30.

union bound

Der Beweis des Lemmas nutzt eine übliche Struktur. Wir möchten zeigen, dass etwas mit hoher Wahrscheinlichkeit nicht passiert, obwohl es viele Möglichkeiten $X_1, \dots, X_n$ hierfür gibt. Man schätzt also zunächst die Einzelwahrscheinlichkeiten $\mathbb{P}[X_i]$ ab (vereinfachend nimmt man oft an, dass die X_i unabhängig sind). Hier sollten sich extrem kleine Erfolgswahrscheinlichkeiten $\mathbb{P}[X_i] \leq 1/n^2$ ergeben. Dann schätzt man die Wahrscheinlichkeit, dass mindestens eins eintritt, einfach als Summe der Einzelwahrscheinlichkeiten nach oben ab. Booles Ungleichung besagt:

Booles Ungleichung

$$\mathbb{P}\left[\bigcup_{i=1}^{n} X_i\right] = \mathbb{P}[\text{mindestens ein } X_i \text{ tritt ein}] \leq \sum_{i=1}^{n} \mathbb{P}[X_i] \tag{2.66}$$

Nun können wir eine wichtige Eigenschaft unserer Zufallsbits zeigen. Wir befinden uns zunächst im subkritischen Bereich, sprich unsere Analyse betrachtet Graphen mit $np \leq (1-\varepsilon)$.

kleines $\varepsilon > 0$
$N = \binom{n}{2}$
$p = (1-\varepsilon)/n$
$k = (10/\varepsilon^2)\ln n$

Lemma 2.30 (basiert auf [11]). Sei $\varepsilon > 0$ eine hinreichend kleine Konstante und seien $X = (X_1, \dots, X_N)$ unabhängige Zufallsvariablen, die Bernoulli-verteilt mit Parameter p sind. Sei $p = (1-\varepsilon)/n$ und $k = (10/\varepsilon^2)\ln n$. Dann gibt es mit hoher Wahrscheinlichkeit keine zusammenhängende Teilfolge (Intervall) mit Länge nk (d. h. $X_i, \dots, X_{i+nk}$), in dem mindestens k Zufallsvariablen den Wert 1 haben. ◀

Beweis. Wir definieren $Y_i = \sum_{j=1}^{nk} X_{i+j-1}$ als die Anzahl an Einsen im Intervall, das mit X_i beginnt. Aus Symmetrie können wir uns zunächst auf die Anzahl Y_1 beschränken. Mittels additiver Chernoff-Ungleichung kann man zeigen, dass

$$\mathbb{P}[Y_1 \geq k] < \exp\left(-\frac{\varepsilon^2(1-\varepsilon)}{3} \cdot \frac{10}{\varepsilon^2}\ln n\right) = \exp\left(-\underbrace{(1-\varepsilon)}_{\to 1}\frac{10}{3}\ln n\right). \tag{2.67}$$

Da die Behauptung nicht nur über ein Intervall Y_1 spricht, sondern über alle $Y_1, \ldots, Y_{N-nk}$, benötigen wir noch einen *union bound*. Hierfür betrachten wir die Einzelereignisse $Y_i \geq k$ für alle i:

$$\begin{aligned}
\mathbb{P}[\exists i:\ Y_i \geq k] &\leq \sum_{i=1}^{N-k} \mathbb{P}[Y_i \geq k] && (2.68)\\
&\leq N \cdot \mathbb{P}[Y_1 \geq k] && (2.69)\\
&\leq \frac{n^2}{2} \cdot \underbrace{\exp\left(-\underbrace{(1-\varepsilon)}_{\to 1}\frac{10}{3}\ln n\right)}_{=o(1/n^3)} && (2.70)\\
&\leq \frac{1}{2n} && (2.71)
\end{aligned}$$

Da die Gegenwahrscheinlichkeit durch $1/2n$ nach oben beschränkt ist, gilt die Aussage mit hoher Wahrscheinlichkeit. □

Wir wissen also, dass für $p \leq (1-\varepsilon)/n$ wir in nk Zufallsbits ziemlich sicher keine k Einsen finden. Die Intuition für $\mathcal{G}(n,p)$ ist also, dass wir in k Zeilen der Adjazenzmatrix keine k Nachbarn finden. Diese k Zeilen können also nicht zusammenhängend sein! Wir formalisieren dieses Bauchgefühl in folgendem Theorem:

$\mathcal{G}(n,p)$ mit $np = (1-\varepsilon)$ hat nur Zusammenhangskomponenten mit $\mathcal{O}(\log n)$ Knoten.

Theorem 2.31 (basiert auf [11]). Sei $\varepsilon > 0$ hinreichend klein. Ferner sei $G \sim \mathcal{G}(n,p)$ ein zufälliger Graph mit $np = (1-\varepsilon)$. Dann haben alle Zusammenhangskomponenten mit hoher Wahrscheinlichkeit eine Größe von höchstens $10\varepsilon^{-2}\ln n$. ◀

Beweis. Beweis durch „Widerspruch“ (tatsächlich kein echter Widerspruch, da die Aussage ja nur mit hoher Wahrscheinlichkeit gelten soll). Nehmen wir an, es gibt eine Zusammenhangskomponente K mit mehr als $k = 10\varepsilon^{-2}\ln n$ Knoten. Betrachten wir die Phasen in der DFS, während K exploriert wird.

Konkret betrachten wir den Moment, zu dem der $(k+1)$-te Knoten entdeckt wird (dieser wird also gleich von U nach S bewegt). Seien $\Delta K = K \cap B$ die Knoten der Zusammenhangskomponente K, die bereits fertig bearbeitet wurden. Dann gilt $|\Delta K \cup S| = |\Delta K| + |S| = k$; sonst hätten wir nicht eben den $k+1$ Knoten entdecken können.

Den ersten dieser k Knoten bekamen wir „kostenlos“, da er in Schritt 2 von U nach S bewegt wurde, um die Suche in der neuen Zusammenhangskomponente zu starten. Jeder weitere Knoten musste über eine Kanten zu einem Knoten in $\Delta K \cup S$ entdeckt wurden sein. Da Kanten zu einer Eins in unseren Zufallsbits korrespondiert, müssen wir als beim Entdecken des $k+1$ Knoten genau k Einsen gesehen haben.

Von jedem Knoten in $\Delta K \cup S$ suchten wir nach höchstens n Nachbarn (grobe Überabschätzung!). In anderen Worten: Wir haben höchstens kn Bits gelesen und dabei k Einsen gefunden. Das passiert gemäß Lemma 2.30 mit hoher Wahrscheinlichkeit nicht. Der „Widerspruch“ zur Annahme folgt. □

Durch geeignete Abschätzung, dass es für $np = (1+\varepsilon)$ „viele“ Einsen gibt, lassen sich folgende Resultate zeigen. Mit hoher Wahrscheinlichkeit enthält $G \sim \mathcal{G}(n,p)$ …

- … einen Pfad mit mindestens $\varepsilon^2 n/5$ Knoten.
- … einen Kreis mit $\Theta(\varepsilon^2)n$ Knoten.
- … eine Zusammenhangskomponente mit mindestens $\varepsilon n/2$ Knoten.

Beobachte, dass die giant component deutlich größer ist als die Pfadlänge. Dies ist inhaltlich richtig (und erwartet – warum??), bedarf aber auch einer deutlich genaueren Abschätzung unserer Zufallsbits.

Weiter ist die Abhängigkeit der ersten beide Aussagen von ε^2 optimal. Tatsächlich ist das Resultat zur Pfadlänge der Grund, weshalb wir uns eine Tiefensuche statt einer Breitensuche angesehen haben. Wenn wir in der Tiefensuche jemals eine Stackgröße von $|S| = k$ erreichen, wissen wir, dass der untersuchte Graph einen Pfad mit mindestens k Knoten enthält; das gilt nicht für die Breitensuche! Der Beweis zeigt also, dass wir ausreichend viele Einsen sehen, um ein hinreichend großen Stack zu bekommen.

2.6.3 Existenz von großen Kreisen

Zuletzt wollen wir uns noch eine Beweistechnik ansehen, bei der wir zwei geeignet parametrisiert $\mathcal{G}(n,p)$-Graphen durch Vereinigung ihrer Kantenmengen „aufaddieren“. Mit dieser Technik lässt sich aus der Existenz (mit hoher Wahrscheinlichkeit) eines Pfads mit αn Knoten $\alpha > 0$ die Existenz (mit hoher Wahrscheinlichkeit) eines Kreises mit $\Theta(\alpha n)$ Knoten folgern. Hierzu ziehen wir einen weiteren $\mathcal{G}(n,p')$-Graphen mit ausreichend großem $p' = o(1/n)$. Diese neuen Kanten fügen wir zu unserem ursprünglichen Graphen G hinzu und erhalten $G' \sim \mathcal{G}(n,p'')$ mit $p'' \lessapprox p + p'$. Da $p' = o(1/n)$, ist $p + p' = (1+\varepsilon)/n + o(1/n)$ für $n \to \infty$ weiterhin durch p dominiert.

Zur einfacheren Illustration nehmen wir $p' = n^{-3/2} = o(1/n)$ an. Weiter betrachten wir die ersten und letzten $\ell = 2n^x$ Knoten des Pfades mit hinreichend großem $x < 1$. Es bleibt ein Mittelteil von M Knoten „übrig“:

$$M = \alpha n - 2\ell = \alpha n - 4n^x = n(\alpha - \underbrace{4n^{x-1}}_{=o(1),\text{ da } x<1}) \tag{2.72}$$

Im Grenzwert von $n \to \infty$ sind also Präfix und Suffix des Pfades relativ zum Mittelteil $2\ell/M = o(1)$ asymptotisch verschwindet klein.

Es gibt aber $\binom{\ell}{2} \geq n^{2x}$ mögliche Kanten zwischen dem Präfix und dem Suffix. Aus G' bekommen wir also erwartet mehr als $n^{2x}p'$ Kanten – eine reicht, um einen Kreis, der länger als M ist, zu schließen. In unserem Beispiel mit $p' = n^{-3/2}$ gilt:

$$n^{2x}p' = n^{2x-3/2} \tag{2.73}$$

Wenn wir also $x = 3/4$ wählen, erwarten wir mindestens eine Kante, die dann den versprochenen Kreis schließt. Für $x > 3/4$ existiert eine solche Kante mit hoher Wahrscheinlichkeit.

2.6.4 Cliquen in $\mathcal{G}(n,p)$

Gemäß der Definition einer Zusammenhangskomponente $K \subset V$ in einem Graphen $G = (V,E)$ finden wir für jedes Knotenpaar $u, v \in K$ einen Pfad von u nach v. Cliquen $C \subseteq V$ sind ein Extremfall dieses Konzeptes, da zwischen allen Konstituenten $u \neq v \in C$ eine direkte Kante existiert. Nach unserer bisherigen Diskussion scheint es eine natürliche Frage zu sein, wie viele Knoten eine größte Clique in einem Graphen $G \sim \mathcal{G}(n,p)$ hat.

Zunächst analysieren wir, wie viele Cliquen der Größe k wir erwarten können. Beachte, dass C genau dann eine Clique ist, wenn der durch $C \subset V$ induzierte Teilgraph genau $\binom{k}{2}$ Kanten mit $k = |C|$ hat. Insbesondere darf der Teilgraph keine *Nichtkanten* enthalten. Da jede Kante existieren muss, ist es besonders einfach, die Wahrscheinlichkeit $\mathbb{P}[C \text{ ist Clique}]$ zu beschreiben:

$$\mathbb{P}[C \text{ ist Clique}] = p^{\binom{|C|}{2}} \tag{2.74}$$

Wir müssen nur noch die Anzahl möglicher Cliquen finden. Es gibt genau $\binom{|V|}{k}$ verschiedene Knotenteilmengen der Größe k. Analog zum Beweis von Lemma 2.6 können wir für jede eine Indikatorvariable I_C annehmen, wobei $\mathbb{P}[I_C = 1] = \mathbb{E}[I_C] = p^{\binom{|C|}{2}}$. Durch die Linearität des Erwartungswertes folgt dann die erwartete Anzahl an Cliquen der Größe k als

$$\binom{|V|}{k} p^{k(k-1)/2}. \tag{2.75}$$

Wenn wir uns den Spezialfall von $\mathcal{G}(n)$ (d. h. $\mathcal{G}(n,p)$ mit $p = 1/2$) ansehen, lässt sich zeigen, dass es eine Funktion $f(n) \approx 2 \ln n$ gibt, sodass eine größte Clique in $G \sim \mathcal{G}(n)$ mit hoher Wahrscheinlichkeit entweder $f(n)$ oder $f(n) + 1$ Knoten hat. Diese können wir aufgrund der kleinen Größe relativ einfach finden! Wenn wir alle möglichen Cliquen explizit enumerieren und prüfen, ergibt sich „nur“ eine quasipolynomielle Laufzeit von $2^{\Theta(\log^2 n)}$. Der Zufall vermeidet also pathologische Strukturen und vereinfacht das eigentlich $\mathcal{NP}$-schwere Clique-Problem auf $\mathcal{G}(n)$ (und auf $\mathcal{G}(n,p)$ mit $p \leq 1/2$) deutlich.

Kapitel 3

Knotengrade

Die nichtalgorithmischen Aspekte dieses Kapitels orientiert sich stark an [3].

Eine elementare Aufgabe in Network Science ist es, die Wichtigkeit von Knoten zu bewerten – man spricht auch von der *Zentralität* der Knoten. Die Anzahl der Nachbarn (Grad, engl. degree) ist eine sehr einfache Metrik mit direkter Intuition: Wenn ein Knoten überdurchschnittlich viele Nachbarn hat, dann steht die Vermutung im Raum, dass diesem Knoten auch eine überdurchschnittlich wichtige Bedeutung im Netzwerk zukommt. Daher wollen wir die Verteilung von Graden in Graphen in diesem Kapitel genauer betrachten.

Zentralität: Wichtigkeit eines Knotens

Zur Wiederholung: In einem ungerichteten Graphen $G = (V, E)$ beschreibt

$$\deg(u) = |\{e \mid e \in E, \text{s. d. } u \in e\}| \tag{3.1}$$

die Anzahl der Nachbarn von Knoten u. Die Gradsequenz $\mathcal{D}_G$ (falls G klar ist, auch nur $\mathcal{D}$) von $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ ist dann einfach die Auflistung aller Grade im Graphen

Gradsequenz

$$\mathcal{D}_G = (\deg(v_1), \deg(v_2), \ldots, \deg(v_n)). \tag{3.2}$$

Beobachte, dass für jede $\mathcal{D}_G$ von $G = (V, E)$ gilt:

$$\sum_{v \in V} \deg(v) = 2|E| \tag{3.3}$$

3.1 Knotengrade in $\mathcal{G}(n, p)$

Die Gradsequenz $\mathcal{D}_G$ ist ein konkreter „Messwert“ für einen Graph G. Für die meisten Graphen mit vielen Knoten können wir – vereinfachend – davon ausgehen, dass die einzelnen Grade unabhängig aus einer Wahrscheinlichkeitsverteilung gezogen wurden. Wir nennen diese Wahrscheinlichkeitsverteilung dann die *Gradverteilung*. Das ist besonders sinnvoll, wenn G selbst aus einem Zufallsgraph stammt. Dann versuchen wir, die Gradverteilung auf das durch den Zufallsgraphen definierte Ensemble auszuweiten. Im Folgenden betrachten wir etwa die Gradverteilung von $\mathcal{G}(n, p)$-Graphen.

Gradverteilung

Aus Abschnitt 2.3.1 wissen wir bereits, dass die Kantenanzahl eines $\mathcal{G}(n, p)$-Graphen binomialverteilt ist. Die Analyse für Grade läuft analog, allerdings nicht für $\binom{n}{2}$ Einträge

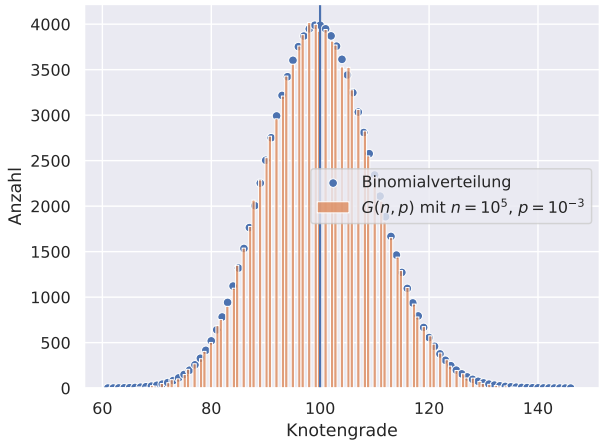


Abbildung 3.1: Histogramm der Gradsequenz $\mathcal{D}_G$ eines Graphen $G \sim \mathcal{G}(n,p)$. Die vertikale Linie zeigt den Durchschnittsgrad $\bar{k}$, die Punkte die Vorhersage mittels Binomialverteilung.

der Adjazenzmatrix, sondern nur für $n-1$ (da Eigenschleifen verboten sind: -1). Der Grad $\deg(u)$ eines Knotens u ist also ebenfalls binomialverteilt:

$$\mathbb{P}[\deg(u) = k] = \underbrace{\binom{n-1}{k}}_{\substack{\text{Anzahl an}\\ \text{möglichen}\\ \text{Nachbarschaften}}} \cdot \underbrace{p^k}_{\substack{\text{Existenz}\\ \text{der } k\\ \text{Kanten}}} \cdot \underbrace{(1-p)^{n-1-k}}_{\substack{\text{Abwesenheit der}\\ \text{restlichen Kanten}}} \tag{3.4}$$

Daher gilt also $\mathbb{E}[\deg(u)] = (n-1)p$.

Wie in Abb. 3.1 dargestellt, approximiert eine konkrete Gradsequenz $\mathcal{D}_G$ die Gradverteilung schon sehr gut. Der Erwartungswert sollte daher gegen den Durchschnittsgrad $\bar{d} = 2m/n$ konvergieren, d. h. $\mathbb{E}[\deg(u)] \approx \bar{d}$. Dann gilt also:

$$\bar{d} \approx \mathbb{E}[\deg(u)] = (n-1)p \tag{3.5}$$

$$p \approx \frac{\bar{d}}{n-1} \tag{3.6}$$

Für tiefere analytische Untersuchen ist die Binomialverteilung ein bisschen unhandlich. Wir wollen daher eine Approximation (siehe [3]) für dünne Netzwerke finden. Aus $m = \mathcal{O}(n \log n)$ folgt direkt $\bar{d} = \mathcal{O}(\log n)$. Somit ist also $\bar{d} \ll n$ eine gute Annahme für dünne Graphen.

Betrachten wir also zunächst den Binomialkoeffizienten in Gleichung (3.4):

$$\binom{n-1}{k} = \frac{1}{k!} \cdot \frac{(n-1)!}{(n-1-k)!} \tag{3.7}$$

$$= \frac{1}{k!} \cdot [(n-1) \cdot (n-1-1) \cdot (n-1-2) \cdot \ldots \cdot (n-1-(k-1))] \tag{3.8}$$

$$\approx \frac{1}{k!} [n-1]^k \tag{3.9}$$

Im letzten Schritt nutzten wir aus, dass $k \ll n$ und somit $n-1 \approx n-k$. Die Wahrscheinlichkeit in Gleichung (3.4), dass $n-1-k$ Nachbarn *nicht* existieren, können wir wie folgt auffassen:

$$(1-p)^{n-1-k} = \exp\left[\ln\left((1-p)^{n-1-k}\right)\right] \tag{3.10}$$

Konzentrieren wir uns nun zunächst auf den Logarithmus:

$$\ln\left((1-p)^{n-1-k}\right) = (n-1-k)\ln(1-p) \approx -(n-1-k)p \tag{3.11}$$

Im letzten Schritt nutzen wir die Abschätzung $\ln(1+x) \approx x$, welche für kleine $x \ll 1$ aus der Taylorreihe von $\ln(1+x) = x - \mathcal{O}(x^2)$ resultiert. Nun können wir Gleichung (3.6) nutzen, um p zu substituieren:

$$-(n-1-k)p \approx -(n-1-k)\frac{\bar{d}}{n-1} \approx -\bar{d} \tag{3.12}$$

Somit folgt letztendlich:

$$(1-p)^{n-1-k} = \exp\left[\ln\left((1-p)^{n-1-k}\right)\right] \approx \exp(-\bar{d}) \tag{3.13}$$

Damit haben wir nun alle Bausteine, um die Binomialverteilung zu approximieren:

$$\begin{aligned}
\mathbb{P}[\deg(u)=k] &= \binom{n-1}{k} \cdot p^k \cdot (1-p)^{n-1-k} && (3.14)\\
&\approx \frac{1}{k!}[n-1]^k p^k \exp(-\bar{d}) && (3.15)\\
&\overset{(3.6)}{\approx} \frac{1}{k!}[n-1]^k \left[\frac{\bar{d}}{n-1}\right]^k \exp(-\bar{d}) && (3.16)\\
&= \frac{\bar{d}^k}{k!}\exp(-\bar{d}) && (3.17)
\end{aligned}$$

Beim letzten Ausdruck handelt es sich um die Poissonverteilung:

Poissonverteilung

Definition 3.1. Für $\lambda > 0$ ist die ganzzahlige und nichtnegative Zufallsvariable X *Poisson-verteilt*, falls

$$\mathbb{P}[X=k] = \frac{\lambda^k}{k!}\exp(-\lambda) \tag{3.18}$$

Dann gilt, dass der Erwartungswert $\mathbb{E}[X] = \lambda$ und die Varianz $\mathrm{Var}(X) = \lambda$ identisch sind. ◀

Für $\bar{k} \ll n$ ist die Gradverteilung nur von $\bar{k}$ abhängig

Aus dieser Approximation mittels Poissonverteilung ergibt sich eine interessante Einsicht. Während die Binomialverteilung von der Knotenanzahl n und Kantenwahrscheinlichkeit p abhängt, hat die Poissonverteilung nur den Parameter $\lambda = p/(n-1) \approx \bar{k}$. Die Interpretation hiervon ist, dass für $\bar{k} \ll n$ die Gradverteilung nur vom Durchschnittsgrad $\bar{k}$, nicht aber von der Knotenanzahl im Graphen abhängen sollte. Abbildung 3.2 untermauert diese Interpretation: Die Binomialverteilungen der zwei größten Netze sind mit dem bloßen Auge nicht mehr von der Poissonverteilung unterscheidbar.

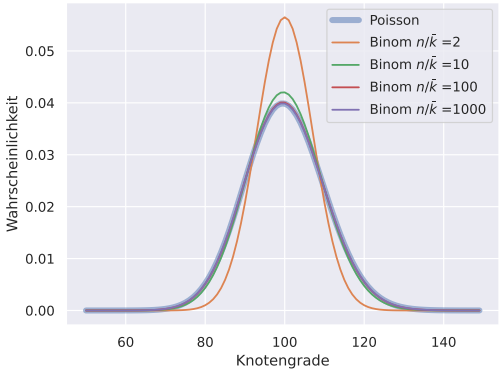


Abbildung 3.2: Vergleich von Binomialverteilung und Poissonverteilung für fixierten Durchschnittsgrad $\bar{k} = 100$. Je größer n, desto eher ist die Approximations-Voraussetzung $\bar{k} \ll n$ erfüllt und desto mehr sind Binomialverteilung und Poissonverteilung deckungsgleich.

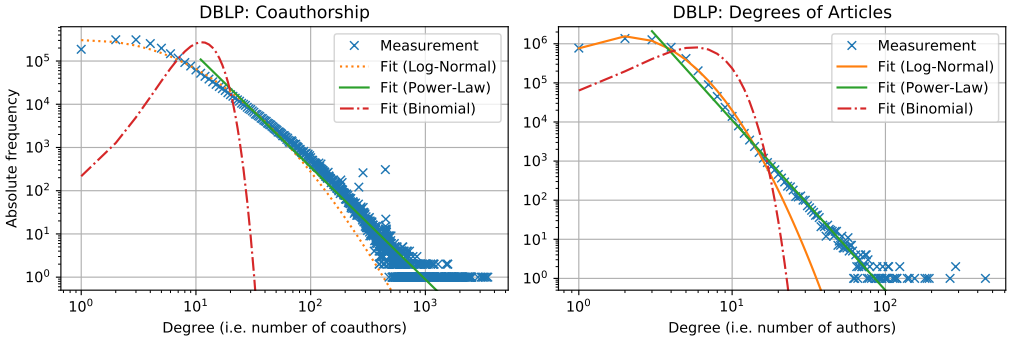


Abbildung 3.3: Gradverteilung von zwei beobachteten Netzwerken [15] verglichen mit drei analytischen Verteilungen.

Diese Einsicht können wir uns wie folgt erklären: Die Poissonverteilung beschreibt die Anzahl von Ereignissen, die innerhalb eines fixierten Intervalls auftreten, wenn die mittlere Rate konstant ist. In diesem Bild ist jede Zeile der Adjazenzmatrix solch ein Intervall und die mittlere Rate entspricht der durchschnittlichen Anzahl an Einsen pro Zeile – also dem Durchschnittsgrad $\bar{d}$.

Beachte auch, dass die Varianz der Poissonverteilung mit $\mathrm{Var}(X) = \bar{d}$ größer ist als die Varianz der Binomialverteilung $\mathrm{Var}(X) = np(1-p) \approx \bar{d}(1 - \bar{d}/n)$, wobei der geklammerte Faktor kleiner als 1 ist.

3.2 Power-Law-Gradverteilung

In Abb. 3.3 stellen wir die Gradverteilung (genauer: das Histogramm der Gradsequenz) zweier beobachteter Netzwerke dar. Die genaue Herkunft der Graphen ist hier nicht wichtig; viele beobachte komplexe Netzwerke zeigen einen qualitativ ähnlichen Verlauf. Was sofort auffällt, ist, dass die rot dargestellte Binomialverteilung keine satisfaktions-

fähige Übereinstimmung mit der beobachteten Verteilung zeigt:

- Die Binomialverteilung sagt vorher, dass die meisten Knoten grob den Durchschnittsgrad haben müssten. Zu beiden Richtungen – sprich für kleinere und größere Grade – sollte es deutlich weniger Knoten geben.
- Der maximale Grad (≈ 30 im linken Schaubild) sollte in etwa dieselbe Größenordnung haben wie der Durchschnittsgrad (≈ 10).

Binomialverteilung und Poissonverteilung haben keine extremen Gerade.

Das beobachtete Netzwerk hat jedoch ein signifikant anderes Verhalten:

- Die meisten Knoten haben sehr geringen Grad – deutlich geringer als der Durchschnittsgrad.
- Der maximal Grad (≈ 4000) ist viel höher als der Durchschnittsgrad (≈ 10). Offensichtlich „ziehen“ diese wenigen Knoten den Durchschnittsgrad also maßgeblich nach oben. Es sollte daher auch nicht überraschend, dass diese sog. Hubs viele Eigenschaften des Netzwerks prägen.

Power Law entspricht einer Geraden im Histogramm.

In Abb. 3.3 fällt ebenfalls auf, dass sich das Histogramm durch eine Gerade (im Folgenden als Linie bezeichnet, um Verwechselung mit Grad auszuschließen) relativ gut approximieren lässt. Dies trifft besonders auf größere Grade zu – aufgrund des doppellogarithmischen Plots stellen diese in absoluten Zahlen einen deutlich größeren Bereich dar, als die Abbildung suggeriert. Wir können also als Arbeitshypothese annehmen, dass die meisten Knotengrade gut durch eine Linie im doppellogarithmischen Plot beschrieben werden. Welche Gesetzmäßigkeit können wir daraus ableiten?

Wir können eine Linie mittels $y = mx + b$ beschreiben. Im doppellogarithmischen Plot sind nun beide Achsen skaliert. Es gilt daher $x = \log d$ und $y = \log h(d)$, wobei d der Grad sei und $h(d)$ die Häufigkeit widerspiegele. Dann gilt:

$$y = mx + b \tag{3.19}$$

$$\log h(d) = m \log d + b \tag{3.20}$$

$$\exp[\log h(d)] = \exp[\log(d^m) + b] \tag{3.21}$$

$$h(d) = \exp(b) \cdot d^m \tag{3.22}$$

Potenzgesetz, Power Law: $p \propto d^{-\gamma}$ für (meist) $2 < \gamma < 3$

Die Häufigkeit skaliert also proportional zu d^m. Wir sprechen daher von einem Potenzgesetz (engl. Power Law). Da die Anzahl der Beobachtungen für größere Grade abnimmt, ist der Exponent immer negativ. Wir schreiben ihn daher als $d^{-\gamma}$, wobei der sog. *Power Law Exponent* γ aus dem Intervall $2 < \gamma < 3$ stammt (falls nicht anders angegeben).

Um eine Wahrscheinlichkeitsverteilung $p(d)$ zu erhalten, müssen wir die Terme noch skalieren:

$$p(d) = C'h(d) = \underbrace{C' \exp(b)}_{:=C} d^{-\gamma} \tag{3.23}$$

Wir absorbieren also den Faktor $\exp(b)$ in die Normierungskonstante C:

$$1 \stackrel{!}{=} \sum_{i=1}^{\infty} C d^{-\gamma} \tag{3.24}$$

$$C = \frac{1}{\sum_{i=1}^{\infty} d^{-\gamma}} = \frac{1}{\zeta(\gamma)} \tag{3.25}$$

$$\Rightarrow p(d) = \frac{d^{-\gamma}}{\zeta(\gamma)} \tag{3.26}$$

Die Funktion $\zeta(\gamma)$ ist die sog. Riemann-Zeta-Funktion (wir werden diese im Folgenden nicht dediziert betrachten).

Beobachte, dass 0^a für $a < 0$ eine Division durch 0 darstellt. Daher ist $p(0)$ nicht definiert und wird explizit in der vorherigen Summierung herausgenommen. Sollte ein Graph $n_0 > 0$ isolierte Knoten beinhalten, betrachtet man diese i. d. R. als Sonderfall. Wir definieren also $p(0) = n_0/n$ und skalieren die Power-Law-Verteilung auf $1 - n_0/n$.

Umgekehrt ist es oft auch sinnvoll, die minimalen und maximalen Grade $d_{\text{min}} < d_{\text{max}}$ explizit zu beschränken. Wir müssen dann nur die Skalierungskonstante auf den kleineren Bereich anpassen:

$$C_{d_{\text{min}}, d_{\text{max}}} = 1 / \sum_{d=d_{\text{min}}}^{d_{\text{max}}} d^{-\gamma} \tag{3.27}$$

Aufgabe 3.2. Entwickele durch ein paar stichprobenartige Rechnungen eine Intuition, welchen Einfluss d_{min} und d_{max} auf $C_{d_{\text{min}}, d_{\text{max}}}$ haben. Erkläre deine Beobachtungen. ◀

3.2.1 Kontinuierlicher Formalismus

In der Herleitung des diskreten Power Law haben wir bereits in der Normierung die transzendente Riemann-Zeta-Funktion $\zeta(\gamma)$ gesehen. Weitere Analysen werden durch die diskrete Natur der Verteilung nicht einfacher. Daher ist es oft bequemer, die Gradverteilung als kontinuierliches Konstrukt anzunehmen. An der Grundform ändert sich dann erst einmal nichts, außer, dass der Grad nun eine strikt positive reelle Zahl sein darf:

$$p(d) = C d^{-\gamma} \qquad d \in \mathbb{R}_{>0} \tag{3.28}$$

Für die Normierung steht uns nun aber das Integral statt der Summe zur Verfügung und führt zu deutlich handlicheren Ergebnissen. Allerdings reicht es nicht mehr, nur $d = 0$ auszuschließen. Wir definieren die Verteilung daher immer für einen expliziten minimalen Grad d_{min}; d_{max} lässt sich analog einführen, wir verzichten aber darauf (siehe auch Aufgabe 3.2).

$$
\begin{aligned}
C_{d_{\text{min}}} &= 1/\int_{d_{\text{min}}}^{\infty} d^{-\gamma}\mathrm{d}d && (3.29)\\
&= 1/\left[\frac{1}{-\gamma+1}d^{-\gamma+1}\right]_{d_{\text{min}}}^{\infty} && (3.30)\\
&= 1/\left(-\frac{1}{-\gamma+1}{d_{\text{min}}}^{1-\gamma}\right) && (3.31)\\
&= \underbrace{(\gamma-1)}_{>0 \text{ für } \gamma>1}{d_{\text{min}}}^{\gamma-1} && (3.32)
\end{aligned}
$$

$$
\Rightarrow \quad p_{\text{cont}}(d) = (\gamma-1){d_{\text{min}}}^{\gamma-1}\cdot d^{-\gamma} \tag{3.33}
$$

Wichtig ist nun allerdings, dass $p_{\text{cont}}(d)$ keine natürliche punktweise Interpretation mehr hat. Vielmehr müssen wir nun immer die Wahrscheinlichkeit, dass ein X aus einem Intervall $[a, b]$ stammt, betrachten:

$$
\mathbb{P}[a \leq X \leq b] = \int_a^b p_{\text{cont}}(d)\mathrm{d}k \tag{3.34}
$$

Aufgabe 3.3. Berechne den Erwartungswert der kontinuierlichen Power-Law-Verteilung mit fixierten Grenzen $0 < d_{\text{min}} < d_{\text{max}} \leq \infty$. ◀

3.2.2 Größe von Hubs

Eine zentrale Motivation, von Poissonverteilungen abzurücken, war es, dass sie keine außergewöhnlich großen Grade zulassen. Daher wollen wir den größten Grad eines Knotens in einer Power-Law-Verteilungen abschätzen.

Konkret suchen wir die Größe d_{nc} (engl. natural cut-off), bei der wir nur einen Knoten mit Grad $\geq d_{\text{nc}}$ erwarten. Es muss also gelten, dass $\mathbb{P}[\deg(u) \geq d_{\text{nc}}] = 1/n$ ist:

$$
\begin{aligned}
1/n &= \int_{d_{\text{nc}}}^{\infty} p(d)\mathrm{d}d && (3.35)\\
&= C\int_{d_{\text{nc}}}^{\infty} d^{-\gamma}\mathrm{d}d && (3.36)\\
&= (\gamma-1){d_{\text{min}}}^{\gamma-1}\left[\frac{d^{-\gamma+1}}{-\gamma+1}\right]_{d_{\text{nc}}}^{\infty} && (3.37)\\
&= (\gamma-1){d_{\text{min}}}^{\gamma-1}\left(0-\frac{{d_{\text{nc}}}^{-\gamma+1}}{-\gamma+1}\right) && (3.38)\\
&= {d_{\text{min}}}^{\gamma-1}{d_{\text{nc}}}^{1-\gamma} && (3.39)
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow \quad {d_{\text{nc}}}^{\gamma-1} &= {d_{\text{min}}}^{\gamma-1}n && (3.40)\\
\Leftrightarrow \quad d_{\text{nc}} &= d_{\text{min}}n^{1/(\gamma-1)} && (3.41)
\end{aligned}
$$

Für $\gamma < 2$ sagt diese Gleichung einen Knoten voraus, der mehr Nachbarn hat, als es Knoten im Graphen gibt. Dies ist nur eines von vielen Indizien, die zu unserer Annahme $\gamma > 2$ führen. Für $\gamma \searrow 2$ (d. h. rechtsseitiger Grenzwert für $\gamma \to 2$) skaliert der maximale Grad als $\Theta(n)$. Für $\gamma = 3$ ist er noch $\Theta(\sqrt{n})$ et cetera.

3.3 Wie entstehen Power-Law-Gradverteilungen?

Das Entstehen von Power-Law-Verteilungen in unterschiedlichsten Netzwerken gibt Anlass zu der Frage, wie solche Verteilungen denn eigentlich entstehen. Barabási und Albert [2] geben eine sehr populäre Erklärung, die besonders durch ihre Simplizität besticht. Ähnliche Techniken wurden jedoch bereits früher verfolgt (z. B. [16]).

Die beiden Autoren schlagen ein neues Modell vor, das wir im Folgenden BA-Modell nennen. Es stellt zwei implizite Annahmen von $\mathcal{G}(n,p)$- und $\mathcal{G}(n,m)$-Graphen infrage:

- In $\mathcal{G}(n,p)$-Graphen fixieren wir die Knotenanzahl initial und ändern diese dann nicht mehr. Netzwerke in der freien Wildbahn unterliegen jedoch in der Regel einer Dynamik: Knoten und Kanten kommen hinzu und verschwinden ggf. wieder.

- Wenn wir einen Knoten im $\mathcal{G}(n,p)$-Graphen als Spieler interpretieren, nimmt das Modell an, dass die Nachbarn rein zufällig ausgesucht werden. Tatsächlich unterliegen aber viele Handlungen im echten Leben einem sog. Bias (d. h. einer verzerrten Wahrscheinlichkeitsverteilung).

  Dies ist besonders auffällig, wenn es zu einer Form von Selektion kommt. So kann kein Mensch alle Webseiten kennen (und dann darauf verlinken); von Facebook, Twitter und Google haben die Meisten aber gehört. Wenn also eine zufällige Person einen Artikel schreibt, ist es deutlich wahrscheinlicher, dass ein Link zu Twitter entsteht als ein Link zu einer bestimmten obskuren Webseite.

3.3.1 Das BA-Modell

BA-Modell beinhaltet Wachstum und Preferential Attachment.

Das BA-Modell trägt beiden Beobachtungen Rechnung. Wir starten zunächst mit einem (meist) kleinen und zusammenhängenden Startgraphen mit n_0 Knoten und m_0 Kanten. Dann fügen wir iterativ N neue Knoten hinzu und erhalten so am Ende $n = n_0 + N$ Knoten. Jeder neue Knoten verbindet sich mit $\nu \leq n_0$ zufällig gewählten Nachbarn. Wir sagen, dass der t-te neue Knoten im t-ten Zeitschritt (für $t = 1, 2, \ldots, N$) hinzufügt wird; unmittelbar nach dem t-ten Zeitschritt hat der Graph also

$$n_t = n_0 + t \text{ Knoten} \quad \text{und} \quad m_t = m_0 + \nu t \text{ Kanten.} \tag{3.42}$$

Analog bezeichnen wir den Grad eines Knotens v nach Zeitschritt t als $\deg_t(v)$. Für $t \ggg n_0$ konvergiert das BA-Modell also gegen einen Durchschnittsgrad von

$$\bar{d}_t = \frac{1}{n_t} \sum_v \deg_t(v) = 2\frac{m_0 + \nu t}{n_0 + t} \to 2\nu. \tag{3.43}$$

Um den zuvor genannten Bias zu implementieren, nehmen wir aber an, dass Knoten mit vielen Nachbarn besonders bekannt sind, und unterstellen einen linearen Zusammenhang. Wenn wir also aus zwei Knoten a und b mit $\deg(a) = 2\deg(b)$ wählen müssen, ziehen wir Knoten a mit doppelter Wahrscheinlichkeit. Sei $p_t(v)$ die Wahrscheinlichkeit, dass wir Knoten v in Zeitschritt $t+1$ ziehen. Dann gilt:

$$p_t(v) = \frac{\deg_t(v)}{\sum_u \deg_t(u)} = \frac{\deg_t(v)}{2(m_0 + \nu t)} \tag{3.44}$$

3.3.2 Dynamik in der Gradverteilung

Der beschriebene Bias wird *Preferential Attachment* bezeichnet. Es bildet sich eine positive Rückkopplung: Ein Knoten mit hohem Grad wird wahrscheinlicher als andere Knoten gezogen. Ein gut vernetzter Knoten akkumuliert also schneller neue Nachbarn und wird dadurch noch berühmter. Der Prozess liegt daher einigen Sprichwörtern zugrunde, z. B. „the rich get richer“ (dt. „Der Teufel … auf den größten Haufen“). Es gibt diverse Methoden zur formalen Analyse des Prozess – eine der einfachsten ist der kontinuierliche Formalismus, bei dem wir Knotengrade und die Zeit durch reelle Zahlen approximieren. Der Einfachheit halber verwenden wir dennoch weiterhin den Begriff Zeitschritt, womit wir ein Zeitintervall von Einheitslänge meinen.

Wir gehen davon aus, dass die ν Kanten in einem Zeitschritt gleichzeitig gezogen werden; es können also Mehrfachkanten entstehen. Die erwartete Anzahl an Nachbarn, die Knoten v in einem Schritt $t+1$ ansammelt, ist dann also $\nu p_t(v)$. Da die Zeit ab jetzt kontinuierlich angenommen wird, *fixieren wir einen Knoten* v_i, der zum Zeitpunkt t_i hinzugefügt wird, und betrachten seinen Grad als zeitabhängige Funktion $d(t)$. Es gilt $d(t) = 0$ für $t < t_i$ und $d(t_i) = \nu$.[1] Analog sei $m(t) = m_0 + \nu t$ die kontinuierliche Kantenanzahl.

Das Wachstum $\mathrm{d}d(t)/\mathrm{d}t$ der Funktion hängt nun vom Grad $d(t)$ selbst ab. Es ergibt sich somit die folgende Differentialgleichung:

$$\frac{\mathrm{d}d(t)}{\mathrm{d}t} = \nu p_t(v) = \nu \frac{d(t)}{2m(t)} = \nu \frac{d(t)}{2(m_0 + \nu t)} \tag{3.45}$$

$$\overset{t \gg 0}{\approx} \nu \frac{d(t)}{2\nu t} = \frac{d(t)}{2t} \tag{3.46}$$

Diese spezielle Differentialgleichung lässt sich relativ einfach lösen. Wir teilen beide Seiten durch $d(t)$ (ab Zeitpunkt $t \geq t_i$ gilt $d(t) > 0$) und integrieren dann von t_i (dem Zeitpunkt, zu dem der Knoten v_i hinzugefügt wurde) bis T:

$$\int_{t_i}^{T} \frac{\frac{\mathrm{d}d(t)}{\mathrm{d}t}}{d(t)} \mathrm{d}t = \int_{t_i}^{T} \frac{1}{2t} \mathrm{d}t \tag{3.47}$$

Die rechte Seite ist ein Standardintegral:

$$\int_{t_i}^{T} \frac{1}{2t} \mathrm{d}t = \left[\frac{1}{2} \log t\right]_{t_i}^{T} = \frac{1}{2} \log\left(\frac{T}{t_i}\right) \tag{3.48}$$

[1] Wir weichen an dieser Stelle von der diskreten Notation ab, bei der erst $d(t_i + 1) = \nu$ wäre. Dies hat aber keinen signifikanten Einfluss auf unsere Analyse und vereinfacht die Formeln.

Die linke Seite lässt sich mittels Substitution von $u = d(t)$ und $\mathrm{d}u = \frac{\mathrm{d}d(t)}{\mathrm{d}t}\mathrm{d}t$ analog zur rechten Seite integrieren. Somit folgt insgesamt:

$$\log\left(\frac{d(T)}{d(t_i)}\right) = \frac{1}{2}\log\left(\frac{T}{t_i}\right) \tag{3.49}$$

$$\frac{d(T)}{d(t_i)} = \sqrt{\frac{T}{t_i}} \tag{3.50}$$

$$d(T) = d(t_i)\sqrt{\frac{T}{t_i}} = \nu\left(\frac{T}{t_i}\right)^{\beta}, \tag{3.51}$$

dynamischer Exponent $\beta = 1/2$

wobei $\beta = 1/2$ als dynamischer Exponent bezeichnet wird.

Diese unscheinbare Gleichung gibt bereits viel über das Verhalten von BA-Netzwerken preis. Zunächst fällt auf, dass der Grad monoton wächst (es gibt schließlich keinen Prozess, um ihn zu reduzieren). Die erwartete Anzahl der Nachbarn von verschiedenen Knoten v_i und v_j unterscheidet sich zudem nur durch die Zeitpunkte t_i und t_j, zu denen sie hinzugefügt wurden. Je früher ein Knoten ins Netz eintrat, desto höher ist sein erwarteter Grad – der sogenannte *first-mover advantage*.

first-mover advantage

Die Situation für später hinzugekommene ist sogar noch schlechter, als man zunächst erwarten würde: Die Startzeit t_i skaliert mit $\sqrt{1/t_i}$ in den Knotengrad. Während ein frühes Mitglied innerhalb einer gewissen Zeitspanne nach seinem Beitritt noch X Nachbarn anhäufen konnte, bekommt ein späterer Knoten in derselben Zeit nur noch einen Bruchteil dieser Knoten. Dies liegt einfach daran, dass es in späteren Zeitpunkten mehr „Konkurrenz“ gibt und somit Nachzügler kaum Aufmerksamkeit bekommen. Dies kann man als Schwäche des Netzwerks interpretieren, da die einzige Chance, im Mittel erfolgreich zu sein, darin besteht, früh beigetreten zu sein.

3.3.3 Gradverteilung des BA-Modells

Im vorherigen Kapitel haben wir herausgefunden, dass der Grad eines Knotens v_i nach dem Beitritt ins Netz zu Zeitpunkt t_i gemäß $d(t) = \nu(t/t_i)^{1/2}$ wächst. Wir müssen dieses Ergebnis noch in eine Gradverteilung zum Zeitpunkt $T \ggg 1$ übersetzen.

Die Intuition hierfür ist folgende: Wir wissen, dass wir zum Zeitpunkt T insgesamt T zufällige Knoten hinzugefügt haben, und können annehmen, dass wir das in einem regelmäßigen Takt getan haben. Dann müssen wir nur noch extrapolieren, welchen Grad wir für jeden dieser Knoten zu Zeitpunkt T erwarten. Konkret fragen wir uns, wann ein Knoten eingefügt werden musste, s. d. er höchsten Grad k hat:

$$d(T) < k \tag{3.52}$$

$$\Leftrightarrow \quad \nu\left(\frac{T}{t_i}\right)^{\beta} < k \tag{3.53}$$

$$\Leftrightarrow \quad \frac{\nu}{k}T^{\beta} < t_i^{\beta} \tag{3.54}$$

$$\Leftrightarrow \left(\frac{\nu}{k}\right)^{1/\beta} T < t_i \tag{3.55}$$

Wir erwarten also, dass Knoten, die ab dem Zeitpunkt $t_i > (\nu/k)^{1/\beta}T$ eingefügt wurden, einen Grad von höchsten k haben. Davon gibt es $T - (\nu/k)^{1/\beta}T$ viele! Sei X nun der Grad eines zum Zeitpunkt T uniform gewählten Knotens und $P(k) = \mathbb{P}[X \leq k]$ die kumulative Verteilungsfunktion. Dann gilt:

$$P(k) = \mathbb{P}[X \leq k] = \frac{T - (\nu/k)^{1/\beta}T}{n_T} = \frac{T - (\nu/k)^{1/\beta}T}{n_0 + T} \tag{3.56}$$

$$\overset{T \gg n_0}{\approx} \frac{T - (\nu/k)^{1/\beta}T}{T} = 1 - (\nu/k)^{1/\beta} \tag{3.57}$$

Für die Gradverteilung benötigen wir nun die Wahrscheinlichkeitsverteilung $p(k)$, also die Ableitung von $P(k)$:

$$p(k) = \frac{\mathrm{d}P(k)}{\mathrm{d}k} = \frac{\mathrm{d}}{\mathrm{d}k}(1 - (\nu/k)^{1/\beta}) \tag{3.58}$$

$$= -\nu^{1/\beta}\frac{\mathrm{d}}{\mathrm{d}k}k^{-1/\beta} \tag{3.59}$$

$$= \frac{1}{\beta}\nu^{1/\beta}k^{-1/\beta-1} \tag{3.60}$$

$$= 2\nu^2 k^{-3} \tag{3.61}$$

Hierbei sind $2\nu^2$ konstante Vorfaktoren. Durch Parametervergleich ergibt sich also eine Power-Law-Verteilung mit Exponent $\gamma = 3$.

3.4 Generieren von BA-Graphen

BA-Graphen lassen sich mit einem überraschend einfachen Generator in in der Ausgabegröße linearer Zeit erzeugen. Das Grundgerüst des Algorithmus [4] folgt dem BA-Modell. Wir unterhalten eine Datenstruktur, die den wachsenden Graphen repräsentiert, und fügen dann iterativ Kanten hinzu.

Um zum Zeitpunkt t die m_t Kanten zu speichern, nutzen wir ein Array A mit $2m_t$ Einträgen. Die i-te Kanten $\{u_i, v_i\}$ wird dabei durch $A[2i-1] = u_i$ und $A[2i] = v_i$ gespeichert. Die zentrale Beobachtung ist nun, dass in $A[1..2m_t]$ jeder Knoten u genau $\deg_t(u)$ mal abgespeichert ist. Wir müssen als nur einen Eintrag aus A uniform zufällig ziehen, um die benötigte Verteilung zu erhalten. Der vollständige Generator ist in Alg. 8 zusammengefasst.

Der Generator Alg. 8 hat jedoch eine praktische Schwäche: Die ν Nachbarn eines Knotens werden unabhängig gezogen – es kann also zu Mehrfachkanten kommen. Dies ist in vielen Anwendungen unerwünscht. Daher gehen wir im Folgenden davon aus, dass der Startgraph einfach ist (d. h., keine Mehrfachkanten enthält).

Dann können wir Mehrfachkanten recht einfach erkennen: Wir müssen nur die ν anfänglichen Nachbarn eines jeden neuen Knoten auf Duplikate prüfen. (Warum reicht das?) Es gibt zwei übliche Strategien, wenn wir Duplikate detektiert haben.

1. Wenn wir einen Knoten x mehrfach als Nachbar eines neuen Knotens v gezogen haben, „löschen“ wir alle bis auf eine resultierende Kante. Dies verändert zwar

Algorithmus 8 : Linearzeit-Generator [4] für BA-Graphen

Input : Startgraph als Kantenliste E_0 über Knoten $1, \ldots, n_0$. Anzahl der Zufallsknoten N sowie deren Nachbaranzahl ν.

Output : Kantenliste E

```
1  Allokiere leeres Array E mit Kapazität 2(|E_0| + νN)
2  foreach {u, v} ∈ E_0 do
3      E.pushBack(u)
4      E.pushBack(v)
5      Gebe Kante {u, v} aus und fahre fort
6  for 1 ≤ i ≤ N do
7      for 1 ≤ j ≤ ν do
8          x ← uniform zufällig aus E[1..2(|E_0| + ν(i − 1))]
9          E.pushBack(x)
10         E.pushBack(n_0 + i)
11         Gebe Kante {x, n_0 + i} aus und fahre fort
```

die Ausgabeverteilung, ist aber für große Ausgaben gar nicht so signifikant, da diese relativ selten auftreten (s. u.).

2. Wir stellen sicher, dass wir ν verschiedene Nachbarn ziehen. Eine Technik hierfür haben wir bereits in Abschnitt 2.5 gesehen, um k verschiedene Samples aus N ohne Zurücklegen zu erstellen: Wir ziehen einfach für den neuen Knoten v unabhängig neue Nachbarn; wenn wir einen Nachbarn ziehen, der bereits mit v verbunden ist, verwerfen wir diesen und versuchen es erneut. Mit dem gleichen Argument wie beim Löschen von Nachbarn hat dieses Verfahren keine Auswirkungen auf die erwartete asymptotische Laufzeit.

Aufgabe 3.4. Sei G ein einfacher BA-Graph mit fixierten ν und $N \gg n_0$. Vereinfachend kann angenommen werden, dass G vollständig der erwarteten Gradverteilung unterliegt. Schätze die erwartete Anzahl uniformen Stichproben ab, die benötigt werden, um ν verschiedene Nachbarn zu finden. ◀

Wenn die Ausgabe in den Hauptspeicher des Computers passt, funktioniert der vorgestellte Algorithmus praktisch sehr gut. In **??** werden zwei Algorithmen beschrieben, die über die Grenze hinweg funktionieren.

3.4.1 Paralleles Ziehen von BA-Graphen

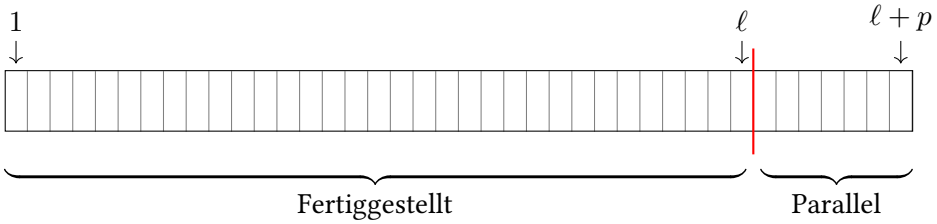
Im Folgenden möchten wir uns zwei parallele Generatoren für BA-Graphen ansehen. Vereinfachend betrachten wir nur $\nu = 1$; die Ideen lassen sich aber auch auf $\nu > 1$ verallgemeinern.

3.4.1.1 Abhängigkeiten vermeiden

Das Generieren von BA-Graphen wurde bis vor wenigen Jahren noch als *inhärent sequentielles* Problem bezeichnet. Jeder gezogene Knoten verändert die Gradverteilung,

s. d. sich zwischen den neu eingefügten Kanten Abhängigkeiten ergeben. Allerdings gibt uns Alg. 8 bereits ein mentales Bild an die Hand, mit dem wir diese Abhängigkeiten in den Griff bekommen können. Um die Beschreibung zu vereinfachen, gehen wir davon aus, dass jeder Eintrag im Array einer Kante entspricht (statt eigentlich zwei Einträgen pro Kante). Dies ist eine asymptotisch nicht relevante Abschätzung zu unseren Ungunsten (siehe auch Abschnitt 3.4.1.2).

Angenommen, wir haben bereits ℓ Kanten berechnet und diese in $E[1..\ell]$ geschrieben. Dann können wir p Kanten parallel bearbeiten, falls alle p Kanten nur auf bereits bekannte Werte zugreifen. In der Skizze korrespondiert dies zu der Situation, dass alle Kanten auf der rechten Seite der roten Linie nur Werte links von der roten Linie lesen.



Leider ist der einzige Wert, für den wir garantieren können, dass es zu keinerlei Abhängigkeiten kommt, $p = 1$; nicht hilfreich. Daher erweitern wir unseren Algorithmus um einen Fallback.

Bevor die p Einträge berechnet werden, schreiben wir an die Stellen $E[\ell+1..\ell+p] \leftarrow \perp$ einen besonderen Wert, der anzeigt, dass diese Zellen keine valide Kante enthält. Dann ziehen wir unabhängig zufällig für jeden der p neuen Einträge den Index s_i, von dem gelesen werden soll. Es gibt zwei Möglichkeiten:

- Falls $s_i \leq \ell$, gibt es keine Abhängigkeiten. Wir können $E[s_i]$ also sicher lesen, die neue Kante berechnen und das Ergebnis in $E[\ell + i]$ schreiben.

- Falls $s_i > \ell$, benötigen wir einen Wert, der gerade erst parallel berechnet wird. Wir lesen nun wiederholt von $E[s_i]$ bis wir einen Wert finden, der nicht $\perp$ ist. Dann berechnen wir unseren eigenen Eintrag und schreiben ihn nach $E[\ell + i]$. Jeden Versuch bezeichnen wir als Runde.

Um die Analyse einfach zu halten, nehmen wir an, dass die Prozessoren alle Standardoperation in konstanter Zeit ausführen können. Weiter können wir am Ende jeder Runde in konstanter Zeit feststellen, ob alle Prozessoren fertig sind. Für diese Annahmen ist das sog. CRCW-PRAM-Modell ein geeignetes Maschinenmodell.

Die schlechte Nachricht ist leider: Der Algorithmus hat ein furchtbares Verhalten im worst case. Für den Fall, dass $s_i = \ell + i - 1$ ist, hängt jede Kante von ihrem Vorgänger ab. Pro „Runde“ des Fallback können wir also nur eine Kante produzieren. Das ist nicht besser als ein sequentieller Algorithmus; wir verschwenden also $\Theta(p^2)$ Arbeit. Die gute Nachricht ist aber, dass dies sehr unwahrscheinlich ist, wenn wir p nicht allzu groß wählen.

Es handelt sich hier um einen Spezialfall des Geburtstags„paradoxon“.

Lemma 3.5. Bei einer vorhandenen Kantenliste E mit $\ell = |E|$ können wir $p = \mathcal{O}\left(\sqrt{\ell}\right)$ Kanten parallel in erwarteter konstanter Zeit einfügen. ◀

Beweis. Wir zeigen, dass die erwartete Anzahl an Abhängigkeiten extrem klein ist. Sei Y_i eine Indikatorvariable mit $Y_i = 1 \Leftrightarrow s_i > \ell$:

$$\mathbb{P}[Y_i = 1] = \mathbb{P}[s_i > \ell] = \frac{i-1}{\ell + i - 1} \tag{3.62}$$

Somit folgt die erwartete Anzahl von Abhängigkeiten als

$$\begin{aligned}
\mathbb{E}\left[\sum_{i=1}^{p} Y_i\right] &= \sum_{i=1}^{p} \frac{i-1}{\ell + i - 1} && (3.63)\\
&= \sum_{i=0}^{p-1} \frac{i}{\ell + i} && (3.64)\\
&\leq p\frac{p}{\ell + p} && (3.65)\\
&\leq \frac{p^2}{\ell} && (3.66)
\end{aligned}$$

Für $p = \mathcal{O}\left(\sqrt{\ell}\right)$ können wir also den Erwartungswert durch $\mathcal{O}(1)$ beschränken. □

Basierend auf Lemma 3.5 können wir nun mehrere Epochen aufbauen. In der ersten Epoche starten wir mit $e_1 = \ell$ Kanten und fügen $\sqrt{\ell}$ neue hinzu. In der nächsten Epoche haben wir dann schon $e_2 = \ell + \sqrt{\ell}$ und können schon $\sqrt{\ell + \sqrt{\ell}}$ neue Ergebnisse erzielen. Es gilt also:

$$e_i = \begin{cases} \ell & \text{falls } i = 1, \\ e_{i-1} + \sqrt{e_{i-1}} & \text{sonst.} \end{cases} \tag{3.67}$$

Da jede Epoche in Erwartung eine Laufzeit von $\mathcal{O}(1)$ hat, hängt also die Laufzeit des Gesamtalgorithmus von der Anzahl notwendiger Epochen ab.

Lemma 3.6. Für $e_1 = \sqrt{m}$ können wir m Kanten in $\mathcal{O}(\sqrt{m})$ Epochen erzeugen. ◀

Beweis. Beginnen wir einfacher: Wie viele Epochen benötigen wir höchstens, um von e_1 Kanten auf $2e_1$ Kanten zu kommen? Wir unterschätzen die Anzahl neuer Kanten pro Epoche durch $\sqrt{e_1}$ (also der Zuwachs aus der ersten Epoche). Dann sind wir nach $\sqrt{e_1}$ Epochen am Ziel $e_1 + \sqrt{e_1}\sqrt{e_1} = 2e_1$.

Dieses Vorgehen wiederholen wir logarithmisch oft. Vorsicht ist aber geboten, da wir, um von $2e_1$ nach $2(2e_1)$ zu kommen, $\sqrt{2e_1}$ (statt wie vorher nur $\sqrt{e_1}$) Epochen benötigen. Sei k die Anzahl an notwendigen Epochen, d. h. $e_k \geq m$. Es folgt:

$$k \leq \sum_{i=0}^{\log_2(m/e_1)} \sqrt{2^i e_1} \tag{3.68}$$

$$= \sqrt{e_1} \sum_{i=0}^{\log_2 e_1} 2^{i/2} \tag{3.69}$$

$$\leq \sqrt{e_1} 2 \sum_{i=1}^{(1+\log_2 e_1)/2} 2^i \tag{3.70}$$

$$= \sqrt{e_1} \mathcal{O}\left(2^{2+(\log_2 e_1)/2}\right) \tag{3.71}$$

$$= \sqrt{e_1} \mathcal{O}(\sqrt{e_1}) = \mathcal{O}(e_1) \tag{3.72}$$

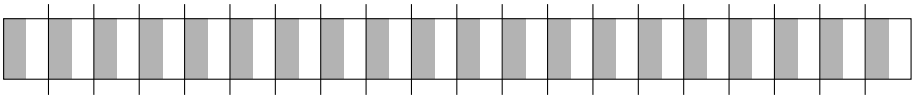
□

Wir können also mit einem sequentiellen Algorithmus starten und in Zeit $\mathcal{O}(\sqrt{m})$ einen Graphen generieren, der hinreichend groß ist, um in den parallelen Betrieb zu wechseln. Ab dann ziehen wir die verbleibenden $m - \sqrt{m}$ Kanten in erwarteter Zeit $\mathcal{O}(\sqrt{m})$.

3.4.1.2 Abhängigkeiten beherrschen

Im vorherigen Kapitel haben wir eine allgemeine Technik gesehen, um sequentielle Algorithmen mit geeigneten zufälligen Zugriffsmustern zu parallelisieren. Wir konstruieren kleine Epochen, um Abhängigkeiten möglichst zu vermeiden und behandeln dennoch auftretende Abhängigkeiten als Sonderfall.

Für BA-Graphen im Speziellen zeigen [19] jedoch, dass wir sogar $\Theta(m)$ Prozessoren sinnvoll Arbeit zukommen lassen und einen BA-Graphen mit hoher Wahrscheinlichkeit in $\mathcal{O}(\log m)$ Zeit erzeugen können. Betrachten wir noch einmal die „echte“ Kantenliste, in der jede Kante aus zwei Einträgen besteht:



Jede Kante besteht aus einem trivialen „deterministischen“ Eintrag, nämlich dem Index des neuen Knotens, der sich aus dem Kantenindex ableiten lässt, und einem zufälligen Wert (hier weiß). Wir nehmen nun an, dass jeder Kante i ein eigener Prozessor p_i zugeordnet ist. Kanten aus dem Startgraph kopieren einfach den entsprechenden Wert. Danach führen alle anderen Prozessoren folgendes Programm parallel aus:

```
1 E[2i − 1] ← Knoten-ID des neuen Knotens
2 E[2i] ← ⊥
3 s_i ← uniform zufällig aus {1 . . . 2(i − 1)}
4 while E[s_i] = ⊥ do
5     Warte auf nächste Runde
6 E[2i] ← E[s_i]
```

Dieser Algorithmus funktioniert so gut, weil sich in jeder Runde die erwartete Anzahl an nicht fertiggestellten Kanten halbiert. Nach der ersten Runde sind alle Kanten fertig, deren s_i ungerade war; in diesem Fall konnten wir nämlich einfach die „deterministische" Hälfte einer Kante lesen. Da s_i uniform gezogen wurde, ist s_i mit Wahrscheinlichkeit $1/2$ ungerade.

In der zweiten Runde können wir von den verbleibenden Kanten all jene fertigstellen, die auf eine Kante zeigen, die in der ersten Runde fertig gestellt wurde. In anderen Worten: Solche Kanten haben ein gerades s_i, das auf ein ungerades s_{s_i} zeigt – dies betrifft in Erwartung $1/4$ aller Kanten.

In der i-ten Runde setzt sich die Kette analog fort. Insgesamt benötigen wir $i - 1$ Verweise von gerade auf gerade, gefolgt von einem einzelnen auf ungerade.

Die erwartete Anzahl der Runden der letzten Kante ist also geometrisch mit Erfolgswahrscheinlichkeit $1/2$ verteilt. In Erwartung sind wir für diese Kante nach 2 Runden fertig. Mittels Chernoff-Ungleichung können wir die Wahrscheinlichkeit, mehr als $\beta \log \nu N$ Runden zu benötigen, für geeignetes β auf $(\nu N)^{-2}$ beschränken. Durch Union-Bound folgt dann, dass der Algorithmus mit hoher Wahrscheinlichkeit eine Laufzeit von $\mathcal{O}(\log \nu N)$ hat.

3.4.1.3 Abhängigkeiten ausschließen

Der eben vorgestellte Algorithmus kann auch so abgeändert werden, dass es zu keinerlei expliziten Abhängigkeiten kommt. Dazu vermeiden wir es einfach, vollständig zufällige Vorgänger zu lesen. Dies ist vor allem im Kontext von verteilten Multicomputern von Vorteil, bei denen Zugriffe auf den Speicher von anderen Computern typischerweise zu erheblicheren Verlangsamung führen.

Die Idee ist simpel: Statt von einer Position der Kantenliste zu lesen, ziehen wir einfach die Berechnung, die dort stattgefunden haben muss, selbst nach.

Angenommen wir befinden uns an Position i und ziehen s_i. Falls s_i ungerade ist, befindet sich dort der Index eines neuen Knotens, den wir einfach nachrechnen können. Falls s_i gerade ist, müssen wir „nur" denselben zufälligen Vorgänger s_{s_i} ziehen, der am Index s_i gezogen wurde.

Dieses Oxymoron können wir lösen, indem wir die deterministische Natur unserer Pseudo-Zufallsgeneratoren ausnutzen. Wir erzeugen an jeder Position der Kantenliste einen Pseudo-Zufallsgenerator, dessen Zustand deterministisch von der Position in der Kantenliste abgeleitet werden kann. Man könnte beispielsweise einfach den Index als Seedwert für den Generator nutzen.

Hierbei müssen wir allerdings vorsichtig sein, da Pseudo-Zufallsgeneratoren in der Regel eine sogenannte Burn-in-Phase benötigen, da die ersten Werte ungewünschte Eigenschaften besitzen können. Außerdem ist das Erzeugen eines Zufallszahlengenerators oft sehr viel teurer als das Ziehen eines Zufallswortes. Daher nutzen die Autoren [19] statt eines dedizierten Pseudo-Zufallsgenerator eine zufällige Hashfunktion.

Dies ist eine pragmatische Entscheidung und liefert – je nach Hashfunktion – ein vergleichsweise kleines Ensemble an möglichen Ausgaben. Empirisch scheinen aber die

erzeugten Graphen ähnlich zu BA-Graphen zu sein. Daher kann der Generator – gerade bei verteilten Berechnungen – das Mittel der Wahl sein.

3.4.2 Skalenfreie Netze

Skalenfreie Netz (scale-free network) sind Graphen mit Power-Law-Gradverteilungen (Gegendarstellung: [1]); hierbei handelt es sich um einen sehr gebräuchlichen Begriff in der Network-Science-Literatur. Der Begriff selbst hat zwei (verwandte) Erklärungen, die uns noch einen Einblick in die Eigenschaften von Power-Law-Verteilungen erlauben.

3.4.2.1 Skaleninvariant

Wir können „skalenfrei“ als „skaleninvariant“ auffassen. Damit wird auf die Selbstähnlichkeit der Power-Law-Verteilung hingewiesen: eine perfekte Power-Law-Verteilung ist in einem doppellogarithmischen Plot immer eine Gerade mit Steigung $-\gamma$. Dabei ist unerheblich, wie groß das zugrunde liegende Objekt (hier: Knotenmenge) ist.

Wir können also die Verteilung umskalieren, also z. B. die Knotenanzahl um einen Faktor a vergrößern. Dann gilt für entsprechende Zufallsvariable X und X':

$$\mathbb{P}[X = k] \propto k^{-\gamma} \tag{3.73}$$

$$\mathbb{P}\left[X' = ak\right] \propto (ak)^{-\gamma} = \underbrace{a^{-\gamma}}_{\substack{\text{von } k \\ \text{unabhängig}}} \cdot k^{-\gamma} \propto k^{-\gamma} \tag{3.74}$$

Beobachte, dass die Proportionalität in den Gleichungen (3.73) und (3.74) jeweils andere Normierungen verbirgt.

3.4.2.2 Unbegrenztes Wachstum

Der Begriff „scale-free network“ wird regelmäßig Barabási und Albert, den Autoren des BA-Modells, zugeschrieben. Barabási selbst erklärt den Begriff über das unbeschränkte Wachstum des k-ten Moments der Power-Law-Verteilung für $n \to \infty$.

Das k-te Moment einer kontinuierlichen (Grad-)Verteilung $p(d)$ ist definiert als:

$$\langle d^k \rangle = \mathbb{E}\left[X^k\right] = \int_{d_{\min}}^{d_{\max}} d^k p(d) \mathrm{d}d, \tag{3.75}$$

wobei $X \sim p$ eine nach p-verteilte Zufallsvariable ist.

Für $k = 1$ erhalten wir also den Erwartungswert, $\langle d^2 \rangle$ ist essentiell in der Berechnung der Varianz $\langle d^2 \rangle - \langle d \rangle^2$, $\langle d^3 \rangle$ beeinflusst die Schiefe (skewness) et cetera. Bezogen auf Power-Law-Verteilungen ergibt sich für $\gamma \neq k + 1$

$$\langle d^k \rangle = \int_{d_{\min}}^{d_{\max}} d^k C d^{-\gamma} \mathrm{d}d = C \left[\frac{1}{k+1-\gamma} \cdot d^{k+1-\gamma} \right]_{d_{\min}}^{d_{\max}} \tag{3.76}$$

$$= C \cdot \frac{{d_{\max}}^{k+1-\gamma} - {d_{\min}}^{k+1-\gamma}}{k+1-\gamma} \tag{3.77}$$

und für $\gamma = k + 1$

$$\langle d^k \rangle = \int_{d_{\text{min}}}^{d_{\text{max}}} d^k C d^{-\gamma} \mathrm{d}d = \int_{d_{\text{min}}}^{d_{\text{max}}} \frac{C}{d} \mathrm{d}d = C \left(\ln \frac{d_{\text{max}}}{d_{\text{min}}} \right). \tag{3.78}$$

Das Verhalten des k-ten Moments für $n \to \infty$ (und dadurch $d_{\text{max}} \to \infty$) wird also maßgeblich durch $k + 1 - \gamma$ beschrieben:

- Für $k + 1 < \gamma$ konvergiert $\lim_{d_{\text{max}} \to \infty} {d_{\text{max}}}^{k+1-\gamma} = 0$. Somit ist $\langle d^k \rangle$ endlich.
- Für $k + 1 = \gamma$ ist das k-te Moment konstant.
- Für $k + 1 > \gamma$ divergiert das k-te Moment und wächst grenzenlos.

Bisher nahmen wir $2 < \gamma \leq 3$ an. In diesem Regime konvergiert $\langle d \rangle$, somit erwarten wir, dass der Durchschnittsgrad in einem wachsenden Netz irgendwann stagniert. Dies widerspricht aber scheinbar unserer Analyse in Abschnitt 3.2.2, wo wir zeigten, dass der Grad von Hubs polynomiell in der Knotenanzahl wächst. Dem wird aber schlicht dadurch Rechnung getragen, dass die Varianz über das divergierende zweite Moment $\langle d^2 \rangle$ wächst. Polemisch gesagt, werden also Hubs als Ausreißer verbucht. Ein zufällig gezogener Knoten kann kleinen oder großen Grad haben mit einer Streuung über viele Größenordnungen – es gibt also kein Konzept von Größenverhältnissen (engl. scale).

Kapitel 4

Gradsequenzen

Während wir im vorherigen Kapitel Gradsequenzen als Stichproben einer Gradverteilung analysiert haben, möchten wir in diesem Kapitel konkrete Gradsequenzen als Charakterisierung eines Graphen verstehen. Genauer gesagt möchten wir für eine gegebene Gradsequenz $\mathcal{D}$ einen Graphen G erzeugen, der folgende Eigenschaften erfüllt:

1. Graph G soll die Gradsequenz $\mathcal{D}$ haben.

2. Graph G soll einfach sein, d. h., keine Mehrfachkanten oder Schleifen haben.

3. Graph G soll eine uniforme Stichprobe aus allen passenden Graphen sein.

Im Folgenden werden wir eine Reihe von Ansätzen untersuchen. Dabei wird sich zeigen, dass es einfach ist, wenn wir nur zwei Eigenschaften (d. h.. 1 & 2, 2 & 3 oder 1 & 3) erfüllen müssen. Für eine allgemeine Gradsequenz $\mathcal{D}$ ist es hingegen schwer allen drei Anforderungen gerecht zu werden.

Bevor wir aber anfangen, sollten wir das Warum klären. Es gibt zwei zentrale Motivationen, Graphen mit allgemeinen Gradsequenzen erzeugen zu wollen:

- Es handelt sich um einen Baustein für komplexere Generatoren. Ein bekanntes Beispiel ist das LFR-Benchmark [12], ein Zufallsgraph, um die Qualität von Community-Detection-Algorithmen zu messen. Hierfür werden Graphen erzeugt, von denen wir wissen, welche Knoten zu welchen Communities gehören. Der Generator erzeugt dafür parametrisierte Communities in Isolation und fügt diese dann zu seinem größeren Graphen zusammen. In beiden Schritten müssen Zufallsgraphen mit gegebenen Gradsequenzen erzeugt werden.

- In datengetriebenen Wissenschaften müssen wir regelmäßig die statistische Signifikanz von Beobachtungen messen. Oft bewerkstelligen wir dies durchs Widerlegen einer Nullhypothese.

  Ein Beispiel in Network Science haben wir bereits gesehen: Wir beobachteten Hub-Knoten, deren Grad wir nicht mit $\mathcal{G}(n, p)$-Graphen erklären konnten. Daraufhin schlugen wir das BA-Modell vor.

Wenig überraschend haben das $\mathcal{G}(n,p)$- und das BA-Modell deutliche Unterschiede (z. B. sind selbst sehr dünne BA-Graphen zusammenhängend). Egal welches Modell wir wählen, es führt immer zu einer Verzerrung in der Bewertung unserer Beobachtungen.

Daher kommen nicht selten uniform zufällige und einfache Graphen, die dieselbe Gradsequenz wie das beobachtete Netz ausweisen, zum Einsatz.

4.1 Configuration-Model

Ein Graph des Configuration-Models lässt sich wie folgt erzeugen: Sei eine Gradsequenz $\mathcal{D} = (d_1, \ldots, d_n)$ mit $\sum_i d_i$ gerade gegeben. Dann legen wir für jedes i genaue d_i Bälle mit Aufdruck i in eine Urne. Solange die Urne nicht leer ist, ziehen wir je zwei uniform zufällige Bälle ohne Zurücklegen. Für jedes Paar mit Aufschrift i und j fügen wir die Kante $\{i, j\}$ in den Graph ein. Per Konstruktion erhalten wir eine Ausgabe mit $m = (\sum_i d_i)/2$ Kanten und n Knoten. Wir bezeichnen die Bälle als *Halbkanten* oder *stubs*, und mit $\mathcal{CM}(\mathcal{D})$ die vom Configuration-Model erzeugte Verteilung.

Prüfen wir kurz unser Lastenheft:

- *Graph G soll die Gradsequenz $\mathcal{D}$ haben.* Das stimmt: Per Konstruktion ist jeder Knoten i genau d_i mal vertreten.
- *Graph G soll einfach sein, d. h., keine Mehrfachkanten oder Schleifen haben.* Das können wir nicht versprechen: Niemand stoppt uns, ein Paar $i = j$ zu ziehen, wenn $d_i > 1$. Analog kann es auch passieren, dass wir dasselbe Paar mehrfach ziehen.
- *Graph G soll eine uniforme Stichprobe aus allen passenden Graphen sein.* Wenn wir zufällig einen einfachen Graphen produzieren, dann ist dieser uniform unter allen einfachen Graphen (siehe Übung).

4.1.1 Effizientes Ziehen aus dem Configuration-Model

Auf den ersten Blick benötigen wir erneut dynamisches gewichtetes Ziehen, um Knoten gewichtete nach ihrem Grad samplen zu können. Derselbe Trick, den wir schon für BA-Graphen gesehen haben, kann aber erneut angewendet werden. Statt die n veränderlichen Gewichte $d_1, \ldots, d_n$ abzuspeichern und daraus gewichtet zu ziehen, erzeugen wir ein Array mit $2m$ Einträgen (einen pro Halbkante) und ziehen daraus uniform. Der entsprechende Generator ist in Alg. 9 skizziert.

Tatsächlich kennen wir aber aus den Übungen bereits einen sehr ähnlichen Algorithmus, der sogar noch besser funktioniert. Es handelt sich um den Fisher-Yates-Shuffle: Statt einer Urne und einer Kantenliste benötigen wir nur ein Array, das für jedes entnommene Element der Urne eine Halbkante fixiert. Wir können also das Configuration-Model implementieren, in dem wir die Urne in eine zufällige Permutation bringen und dann je zwei benachbarte Einträge als Kante interpretieren. Das hat den Vorteil, dass zufällige

Algorithmus 9 : Linearzeit-Generator für das Configuration-Model

Input : Gradsequenz $\mathcal{D} = (d_1, \ldots, d_n)$
Output : Kantenarray E

```
1  if ∑_i d_i ungerade then
2      Breche mit Fehler ab
3  Allokiere leeres Array U für Urne mit Kapazität ∑_i d_i
4  for 1 ≤ i ≤ n do
5      for 1 ≤ j ≤ d_i do
6          U.pushBack(i)
7  Allokiere leeres Array E für Kantenliste mit Kapazität ∑_i d_i
8  while U ≠ ∅ do
9      i ← uniformer Index aus {1, ..., |U|}
10     Vertausche U[i] und U[|U|]
11     x ← U.popBack()
12     y analog zu x gezogen
13     E.pushBack({x, y})
```

Permutationen ein gut verstandenes Konzept sind und es auf diversen Plattformen effiziente Implementierungen gibt.

Aufgabe 4.1. In der Praxis lässt sich dieser Ansatz sogar noch beschleunigen (besonders im parallelen Kontext). Wir partitionieren zunächst die Urne $U[1..2m]$ anhand von Zufallsbits: Am Anfang stehen alle Elemente, für die wir eine 0 warfen, ans Ende kommen alle mit einer 1. Wir permutieren nun nur die größere der beiden Gruppen. Da partitionieren extrem schnell ist, ist der resultierende Algorithmus oft schneller. Zeige, dass wir einen Graphen des Configuration-Models erhalten, wenn wir $U[i]$ und $U[2m - i + 1]$ als i-te Kante interpretieren. ◀

4.1.2 Graphische Gradsequenzen

Wie wir im letzten Kapitel gesehen haben, kann das Configuration-Model Mehrfachkanten und Eigenschleifen erzeugen. Tatsächlich gibt es sogar Eingaben, für die sich das gar nicht vermeiden lässt. Zum Beispiel: $\mathcal{D} = (2)$, $\mathcal{D} = (2, 2)$, $\mathcal{D} = (3, 3)$ oder $\mathcal{D} = (4, 1, 1)$. Wir treffen daher folgende Definitionen:

Definition 4.2. Sei $\mathbb{G}(\mathcal{D})$ die Menge aller einfache Graphen mit Gradsequenz $\mathcal{D}$. Dann beschreibt $\mathcal{G}(\mathcal{D})$ die uniforme Verteilung auf $\mathbb{G}(\mathcal{D})$. ◀

Definition 4.3. Sei $\mathcal{D}$ eine Gradsequenz. Wir nennen $\mathcal{D}$ genau dann *graphisch*, wenn $\mathbb{G}(\mathcal{D}) \neq \emptyset$ (d. h., wenn es mindestens einen einfachen Graphen gibt, der $\mathcal{D}$ erfüllt). ◀

Wir werden später zwei Charakterisierungen von *graphischen* Gradsequenzen sehen.

4.1.3 Erwartete Anzahl an Schleifen und Mehrfachkanten

Angenommen, $\mathcal{D}$ ist graphisch. Wie erhalten wir dann aus dem Configuration-Model einen einfachen Graph? Leider ist dies nicht immer exakt und effizient möglich. Im Folgenden leiten wir aber her, dass dies manchmal gar nicht so schlimm ist.

Hierzu wechseln wir erneut von Gradsequenzen $\mathcal{D}$ zu Gradverteilungen. Sei $p_d = n_d/n$ die Wahrscheinlichkeit, dass ein zufälliger Knoten Grad d hat, wobei n_d die Anzahl der Knoten in $\mathcal{D}$ mit Grad d beschreibt. Die Wahl einer Gradverteilung erlaubt es uns nun, beliebig große Graphen zu betrachten. Wir verdeutlichen dies mit der Notation $\mathcal{CM}(n, \mathcal{D})$. In diesem Setting analysieren wir für große Knotenanzahl n die erwartete Anzahl von Mehrfachkanten $\mathbb{E}[M_n]$ und Eigenschleifen $\mathbb{E}[S_n]$.

Lemma 4.4. Sei $G \sim \mathcal{CM}(n, \mathcal{D})$ ein vom Configuration-Model erzeugter Graph und X ein zufälliger Grad mit $\mathbb{P}[X = d] = p_d$. Wir bezeichnen mit S_n die Anzahl der Eigenschleifen und mit M_n die Anzahl der Mehrfachkanten. Dann gilt

$$\mathbb{E}[S_n] \approx c_n/2 \qquad \text{und} \qquad \mathbb{E}[M_n] \leq c_n^2/4, \tag{4.1}$$

wobei $c_n = (\mathbb{E}[X^2] - \mathbb{E}[X])/\mathbb{E}[X]$. ◀

Aufgabe 4.5. Seien G_1 und G_2 zwei Graphen mit gleicher Knoten- und Kantenanzahl. Beschreibe (qualitativ) zwei Gradverteilungen, s. d. wir für G_1 möglichst wenig kritische Kanten erwarten, während G_2 möglichst viele haben soll. Erkläre (qualitativ), weshalb $\mathbb{E}[M_n] \propto \mathbb{E}[S_n^2]$ plausibel wirkt. ◀

Bemerkung 4.6. Wie bereits in Abschnitt 3.4.2.1 diskutiert, sind $\mathbb{E}[X]$ und $\mathbb{E}[X^2]$ das erste bzw. zweite Moment der Gradverteilung. Wir definieren also c_n über bekannte Größen von Wahrscheinlichkeitsverteilungen, um es möglichst einfach anwenden zu können. Für den folgenden Beweis ist aber hilfreich, c_n nachzurechnen. Für den Durchschnittsgrad wissen wir bereits, dass

$$\mathbb{E}[X] = \sum_{v_i \in V} \frac{d_i}{n} = \frac{2m}{n}. \tag{4.2}$$

Für das zweite Moment gilt gemäß seiner Definition

$$\mathbb{E}[X^2] = \sum_{d=1}^{n-1} d^2 p_d = \sum_{d=1}^{n-1} d^2 (n_d/n) = \frac{1}{n} \underbrace{\sum_{d=1}^{n-1} n_d \cdot d^2}_{n_d\text{-mal Grad } d} = \frac{1}{n} \sum_{i=1}^{n} d_i^2. \tag{4.3}$$

Somit folgt c_n als

$$c_n = \frac{\mathbb{E}[X^2] - \mathbb{E}[X]}{\mathbb{E}[X]} = \frac{\frac{1}{n}\left(\sum_{i=1}^n d_i^2\right) - \frac{1}{n}\left(\sum_{i=1}^n d_i\right)}{\frac{1}{n}\sum_{i=1}^n d_i} = \sum_{i=1}^{n} \frac{d_i(d_i - 1)}{2m}. \tag{4.4}$$

◀

Beweis von Lemma 4.4. Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$. Dann bringt jeder Knoten $v_i \in V$ exakt d_i Halbkanten $s_1^{(i)}, \ldots, s_{d_i}^{(i)}$ in die Urne ein. Wir nehmen an, dass diese unterscheidbar sind.

Schleifen. Sei $I_{a,b}^{(i)}$ eine Indikatorvariable, die anzeigt, dass von Knoten v_i die a-te und b-te Halbkante eine gemeinsame Kante (Eigenschleife!) bilden; hierzu betrachten wir nur $1 \leq a < b \leq d_i$. Da die Halbkanten uniform zufällig gezogen werden und die Kantenreihenfolge irrelevant sind, gilt

$$\mathbb{P}\left[I_{a,b}^{(i)}\right] = \underbrace{\mathbb{P}\left[I_{1,2}^{(i)}\right]}_{\text{falls } d_i \geq 2} = \frac{1}{2m-1}. \tag{4.5}$$

Dann folgt die Schleifenanzahl als Summe der Erwartungswerte der Indikatorvariablen:

$$\begin{aligned}
\mathbb{E}[S_n] &= \mathbb{E}\left[\sum_{v_i \in V}\left(\sum_{1 \leq a < b \leq d_i} I_{a,b}^{(i)}\right)\right] && (4.6)\\
&= \sum_{v_i \in V}\left(\sum_{1 \leq a < b \leq d_i} \mathbb{E}\left[I_{a,b}^{(i)}\right]\right) && (4.7)\\
&= \sum_{v_i \in V}\binom{d_i}{2}\mathbb{P}\left[I_{1,2}^{(i)}\right] && (4.8)\\
&= \frac{1}{2}\sum_{v_i \in V} d_i(d_i-1)\frac{1}{2m-1} && (4.9)\\
&\approx \frac{1}{2}\sum_{v_i \in V}\frac{d_i(d_i-1)}{2m} = \frac{c_n}{2} && (4.10)
\end{aligned}$$

Beobachte, dass $I_{1,2}^{(i)}$ für Knoten v_i mit Grad $d_i < 2$ nicht definiert ist. In diesem Fall ist aber auch $\binom{d_i}{2} = 0$; somit verzichten wir auf eine Ausnahmebehandlung.

Mehrfachkanten. Die Beweisidee ist analog zur Schleifenanzahl, mit dem Unterschied, dass wir deutlich komplexere Indikatorvariablen benötigen. Wir nutzen $I_{a_1,b_1;a_2,b_2}^{(i,j)}$, um anzuzeigen, dass zwischen den Knoten v_i und v_j zwei Kanten existieren – und zwar mit den a_1-ten und a_2-ten Halbkanten von v_i und den b_1-ten und b_2-ten Halbkanten von v_j. Es folgt:

$$\mathbb{P}\left[I_{a_1,b_1;a_2,b_2}^{(i,j)}\right] = \underbrace{\frac{1}{2m-1}}_{\substack{\text{erste Kante}\\ \text{wie zuvor}}} \cdot \underbrace{\frac{1}{2m-3}}_{\substack{\text{erste Kante}\\ \text{fixiert zwei}\\ \text{Halbkanten}}} \tag{4.11}$$

Dies funktioniert gut für Doppelkanten; wenn im Graphen eine Mehrfachkante mit höherer Multiplizität vorhanden ist, überschätzen wir diese aber (warum?). Daher leiten wir nur eine obere Schranke her. Beachte, dass die zwei Approximationen die Schranke

eigentlich reduzieren, jedoch asymptotisch irrelevant sind:

$$\mathbb{E}[M_n] \leq \sum_{v_i,v_j \in V\,:\, i \leq j} \left[\sum_{1 \leq a_1 < a_2 \leq d_i} \left(\sum_{1 \leq b_1 \neq b_2 \leq d_j} \mathbb{P}\left[I^{(i,j)}_{a_1,b_1;a_2,b_2} \right] \right) \right] \tag{4.12}$$

$$\approx \underbrace{\frac{1}{2}}_{\text{Approx.}} \sum_{v_i,v_j \in V} \left[\binom{d_i}{2} 2 \binom{d_j}{2} \frac{1}{(2m-1)(2m-3)} \right] \tag{4.13}$$

$$\approx \sum_{v_i,v_j \in V} \left[\binom{d_i}{2} \binom{d_j}{2} \frac{1}{(2m)(2m)} \right] \tag{4.14}$$

$$= \left[\frac{1}{2} \sum_{v_i \in V} \frac{d_i(d_i-1)}{2m} \right] \left[\frac{1}{2} \sum_{v_j \in V} \frac{d_j(d_j-1)}{2m} \right] = \frac{c_n^2}{4} \qquad \square$$

Erased-Configuration-Model

Beobachte, dass für Gradverteilung mit endlichem ersten und zweiten Moment c_n für $n \to \infty$ konvergiert. Ein Beispiel hierfür sind etwa reguläre Graphen. In diesem Fall ergibt sich eine erwartete konstante Anzahl an illegalen Kanten, die für $n \to \infty$ eine immer kleinere Rolle spielen. Ein probates Mittel ist es daher, Eigenschleifen zu entfernen und von jeder k-fachen Mehrfachkante $k-1$ Exemplare zu löschen. Man spricht vom sogenannten *Erased-Configuration-Model*.

Während für Gradverteilungen mit $c_n = \mathcal{O}(1)$ das *Erased-Configuration-Model* kaum Einfluss auf die Gradsequenz des Graphen hat, ist das für Graphen mit $c_n = \omega(1)$ nicht immer der Fall. Insbesondere skalenfreie Power-Law-Verteilungen sind notorisch schwierig. Dies liegt unter anderem daran, dass illegale Kanten häufiger auftreten und sich zudem um Hubs herum konzentrieren. Das *Erased-Configuration-Model* hat daher einen Bias und senkt den Grad von Knoten mit vielen Nachbarn besonders stark ab; dies wiederum hat signifikante Auswirkungen auf die Graphstruktur (siehe z. B. [20]). Wir werden später Techniken diskutieren, die illegale Kanten entfernen können, ohne dabei die Gradsequenz zu verändern.

4.1.4 Ans Gute glauben und das Schlechte ablehnen

In Abschnitt 4.1.3 zeigten wir, dass der Anteil illegaler Kanten bei gutmütigen Verteilungen zu vernachlässigen ist. Wir können aber noch einen Schritt weiter gehen, und uns fragen, mit welcher Wahrscheinlichkeit wir *gar keine* nichteinfachen Kanten sehen.

Mit einer sorgsamen Analyse kann man zeigen, dass für $c_n = \mathcal{O}(1)$ und $n \to \infty$ die Größen S_n und M_n gegen unabhängige Poisson-verteilte Zufallsvariablen mit $\lambda = c_n/2$ bzw. $\lambda = c_n^2/4$ konvergieren. Das wiederum führt direkt zu folgendem Satz.

Theorem 4.7. Sei $G \sim \mathcal{CM}(n, \mathcal{D})$. Dann ist die Wahrscheinlichkeit, dass G einfach ist,

$$\mathbb{P}[G \text{ ist einfach}] = \exp\left(-\frac{c_n}{2}\right) \exp\left(-\frac{c_n^2}{4}\right)(1 + o(1)). \qquad \blacktriangleleft$$

Beachte, dass wir in den vorherigen Kapiteln typischerweise exponentielle Schranken für die Fehlerwahrscheinlichkeit gezeigt haben. Theorem 4.7 hingegen beschränkt die Erfolgswahrscheinlichkeit. Für $c_n \ggg 0$ ist das also ein katastrophales Resultat.

Selbst für eigentlich gutmütige reguläre Graphen kommen wir schnell an unsere Grenzen. Wenn wir aber einen sehr kleinen Grad, etwa $r = \alpha 2\sqrt{\log n}$, wählen, ergibt sich aber wenigstens noch eine erwartete polynomielle Laufzeit:

$$c_n = \frac{\mathbb{E}\left[X^2\right] - \mathbb{E}[X]}{\mathbb{E}[X]} = \frac{r^2 - r}{r} \approx r = \alpha\sqrt{\log n} \tag{4.15}$$

Somit ergibt sich eine Erfolgswahrscheinlichkeit von

$$\mathbb{P}[G \text{ ist einfach}] = \exp\left(-\frac{c_n}{2}\right)\exp\left(-\frac{c_n^2}{4}\right)(1 + o(1)) = \Omega\left(\frac{1}{n^{2\alpha}}\right). \tag{4.16}$$

Wenn wir so lange Graphen produzieren, bis wir zufällig einen einfachen Graph erhalten haben, reichen also in Erwartung $\mathcal{O}\left(n^{2\alpha}\right)$ Versuche. Tatsächlich handelt es sich hierbei dann auch um ein uniformes zufälliges Sample aus $\mathbb{G}(\mathcal{D})$ (siehe Rejection-Sampling). Da jeder Graph in Zeit $\mathcal{O}(rn)$ erzeugt werden kann, ergibt sich also ein Generator mit polynomieller Laufzeit.

4.1.5 Einzelne Kanten zurückweisen

Obwohl uns Lemma 4.4 verspricht, dass für viele Gradverteilungen der Anteil der illegalen Kanten asymptotisch insignifikant ist, zeigt uns die Diskussion, dass es oft relativ unwahrscheinlich ist, einen vollständig einfachen Graph zu erhalten. Das liegt schlicht daran, dass eine einzige illegale Kante unter $2m$ Kanten zum Misserfolg führt.

Die Idee liegt auf der Hand: Während wir die Kanten iterativ aus der Urne entnehmen, prüfen wir, ob die gezogene Kante legal ist. Falls nicht, legen wir die beiden Bälle zurück in die Urne und versuchen es erneut. Leider können wir hierbei „stecken bleiben“. Als Extrembeispiel betrachten wir für großes $k \ggg 0$ die folgende Gradsequenz über $n = k + 1$ Knoten:

$$\mathcal{D} = \left(\underbrace{k-1, k-1, \ldots, k-1}_{(k-1)\text{-mal}}, \underbrace{k}_{\text{Knoten } u}, \underbrace{1}_{\text{Knoten } v}\right) \tag{4.17}$$

Hierbei handelt es sich um eine sog. Threshold-Sequenz – eine Gradsequenz, bei der alle einfache Graph isomorph sind (d. h., wenn wir die Knotenindizes löschen, sind alle Graphen identisch). Die hier beschriebene Sequenz erzeugt einen Graph mit einer Clique aus k Knoten, wobei einer der Knoten u noch zu Knoten v verbunden ist.

Das Problem besteht nun darin, dass das Blatt v genau zu Knoten u verbunden sein muss. Die Wahrscheinlichkeit, dass just diese Kante erzeugt wird, skaliert aber grob mit $1/n$. Sobald wir das Blatt v mit einem anderen Knoten $u' \neq u$ verbunden haben, gibt es keine Möglichkeit, die verbleibenden Grade nur mit einfachen Kanten zu verteilen. Es bleibt also grob ein Versuch aus $\mathcal{O}(n)$ nicht stecken.

Wir betrachten daher einen anderen Zufallsgraphen.

4.2 Chung-Lu-Graphen

Das Kernproblem, einzelne Kanten im Configuration-Model zurückzuweisen, besteht darin, dass jede gezogene Kante einen Effekt auf folgende Kanten hat – einfach, weil wir ohne Zurücklegen ziehen und jede eingefügte Kante zwei Bälle aus der Urne entfernt. Das Chung-Lu-Modell hat strukturelle Ähnlichkeiten zum Configuration-Model, umgeht aber genaue diese Schwäche.

Die Eingabe des Chung-Lu-Modells besteht aus einem Vektor $w = (w_1, \ldots, w_n) \in \mathbb{R}^n_{>0}$, der die Funktion einer Gradverteilung übernimmt und nicht normiert sein muss. Daher ist auch die Wahl $w = \mathcal{D}$ nicht unüblich. Es wird dann ein Graph mit n Knoten erzeugt und jede Kante $\{v_i, v_j\}$ unabhängig mit Wahrscheinlichkeit $p_{i,j}$ eingefügt. Es gilt

$$\mathbb{P}[\{v_i, v_j\} \in E] = p_{i,j} := \frac{w_i w_j}{\sigma} \qquad \text{mit} \qquad \sigma = \sum_{v \in V} w_v \tag{4.18}$$

Per Definition kann es also keine Mehrfachkanten geben. In der Analyse des Chung-Lu-Modells werden in der Regel Eigenschleifen erlaubt, um Sonderfälle zu vermeiden. In praktischen Implementierung ist es aber trivial, diese zu verbieten.

Betrachten wir zunächst den erwarteten Grad $\mathbb{E}[\deg v_i]$:

$$\mathbb{E}[\deg v_i] = \sum_{v_j \in V} p_{i,j} = \sum_{v_j \in V} \frac{w_i w_j}{\sigma} \tag{4.19}$$

$$= w_i \cdot \frac{1}{\sigma} \underbrace{\sum_{v_j \in V} w_j}_{=\sigma} \tag{4.20}$$

$$= w_i \tag{4.21}$$

Das Gewicht w_i eines Knotens v_i entspricht also seinem erwarteten Grad. Bei der Wahl von w müssen wir jedoch dafür sorgen, dass für alle i, j der Parameter $p_{i,j} \leq 1$ sich als Wahrscheinlichkeit eignet. Beachte, dass das gilt (weshalb?):

$$\max_{i,j} \{p_{i,j}\} \leq \max_i \{p_{i,i}\} \tag{4.22}$$

Daher können wir durch Beschränkung der „Diagonaleinträge“ gleichzeitig alle $p_{i,j}$ nach oben beschränken. Um nun $p_{i,j} \leq 1$ sicherzustellen, wird oft

$$\max_i \left(w_i^2\right) \leq \sigma \qquad \Leftrightarrow \qquad \max_i \left(\frac{w_i w_i}{\sigma}\right) \leq 1 \qquad \Leftrightarrow \qquad \max_i \left(p_{ii}\right) \leq 1 \tag{4.23}$$

gefordert. Eine Alternative besteht darin, die Definition von $p_{i,j}$ um eine explizite obere Schranke zu erweitern:

$$p'_{i,j} = \min\left(1, \frac{w_i w_j}{\sigma}\right) \tag{4.24}$$

Von Anwenderseite ist $p'_{i,j}$ bequemer, da w weniger Restriktionen unterliegt; die Analyse des Modells wird aber unhandlicher. Daher nutzen wir im Folgenden die ursprüngliche Definition und gehen davon aus, dass w korrekt beschränkt ist.

Bevor wir uns effiziente Generatoren für das Chung-Lu-Modell ansehen, diskutieren wir uns zunächst einige allgemeine Sampling-Techniken, die hierfür nützlich sein werden.

4.2.1 Rejection-Sampling

Im Folgenden verallgemeinern wird die sog. Verwerfungsmethode (eng. rejection sampling), die wir bereits in einer sehr einfachen Ausprägung beim Ziehen ohne Zurücklegen und Generieren einfacher Graphen genutzt haben. Damals war die Kernidee jedes Mal etwa: Wir nutzen einen stochastischen Prozess, der uns Vorschläge unterbreitet (proposal distribution). Dieser Prozess hatte bereits die gewünschte Verteilung, lieferte aber u. U. Vorschläge, die wir kategorisch ablehnen mussten; beim $\mathcal{G}(n, m)$-Modell mussten wir bereits gezogene Kanten zurückweisen. Beim BA-Modell waren es Knoten, zu denen wir bereits verbunden sind, und beim Configuration-Model verwarfen wir nicht einfache Graphen. Nach dem Ablehnen unerwünschter Element erhielten wir dann die Zielverteilung (target distribution) automatisch.

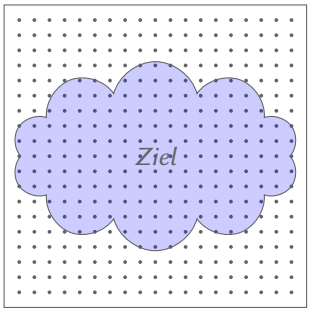


Abbildung 4.1: Rejection-Sampling für Uniforme Verteilung

Abbildung 4.1 veranschaulicht diese Idee. Die proposal distribution entspricht der uniformen Verteilung in der Box (visualisiert durch das reguläre Gitter). Die Zielverteilung ist ebenfalls uniform – allerdings über einer eingeschränkten Grundgesamtheit. Das Gitter ist auch auf dieser uniform verteilt.

Ein weiteres Beispiel ist ein normaler Spielwürfel mit sechs gleich wahrscheinlichen Seiten, die von 1 bis 6 beschriftet sind. Wir können mit diesem Würfel uniform aus $\{1, 2, 3, 4, 5\}$ ziehen: Falls eine Sechs gewürfelt wird, verwerfen wir diese und versuchen es erneut. In Erwartung benötigen wir $6/5$ Versuche, um ein Ergebnis zu produzieren.

Die Verwerfungsmethode ist aber mächtiger – es muss nicht immer ganz oder gar nicht sein. Sie eignet sich auch, um die Wahrscheinlichkeit in einzelnen Bereichen anzuheben bzw. abzusenken.

Angenommen, wir möchten die Zielverteilung T mit $t\colon \Omega \to [0, 1]$ erzeugen. Dann benötigen wir einen Generator für die Verteilung P, der einen Vorschlag mit Wahrscheinlichkeitsverteilung $p\colon \Omega \to [0, 1]$ erzeugt. Wir gehen davon aus, dass mindestens ein $x \in \Omega$ existiert, s. d. $t(x) \neq p(x)$ – sonst sind die Verteilungen identisch und wir fertig. O. B. d. A. nehmen wir an, dass Ω endlich ist und es sich somit um diskrete Verteilungen handelt; die Verwerfungsmethode funktioniert aber auch mit kontinuierlichen Verteilungen.

Als Nächstes benötigen wir einen Skalierungsfaktor s, s. d. $t(x) \leq s \cdot p(x)$ für alle $x \in \Omega$. Beobachte, dass dies impliziert, dass der Support von p den Support von t beinhaltet: Für jedes $x \in \Omega$ gilt $t(x) > 0 \Rightarrow p(x) > 0$. In Abb. 4.1 beinhaltet die Box daher jeden Punkt der Wolke; wären Teile der Wolke außerhalb der Box, könnten dort keine Vorschläge „hinfallen“.

Der Parameter s ist erforderlich, da sowohl p als auch t auf 1 normiert sind, d. h.

$$\sum_{x \in \Omega} p(x) = \sum_{x \in \Omega} t(x) = 1. \tag{4.25}$$

Wenn es nun ein $x \in \Omega$ mit $t(x) < p(x)$ gibt, muss es zwangsläufig ein $x' \in \Omega$ geben, s. d. $t(x') > p(x')$ gilt. Da die Verwerfungsmethode nur ablehnen kann, besteht die einzige Möglichkeit, die relative Frequenz einzelner Elemente zu erhöhen, darin, alle anderen Elemente seltener auftreten zu lassen. Dazu verwenden wir ein $s > 1$ gekoppelt mit „ein bisschen“ Rejection. Die Details sind in Alg. 10 zusammengefasst.

Algorithmus 10 : Generischer Rejection-Sampling-Algorithmus

Input : Generator für P, $p(x)$, $t(x)$ und s, s. d. $\max_x \{t(x)/(s \cdot p(x))\} \leq 1$

```
while für immer do
    Schlage x ~ P vor
    Ziehe u uniform aus [0, 1]
    Berechne a(x) = t(x)/(s · p(x))
    if u ≤ a(x) then
        Akzeptiere: Gebe x zurück und beende
```

Lemma 4.8. In jeder Runde akzeptiert Alg. 10 mit Wahrscheinlichkeit $1/s$. ◀

Beweis. Die Wahrscheinlichkeit, Element $x \in \Omega$ in einer fixierten Runde zu produzieren, folgt als

$$\mathbb{P}[\text{Gebe } x \text{ zurück}] = \mathbb{P}[\text{Schlage } x \text{ vor}]\, \mathbb{P}[\text{Akzeptiere } x \mid \text{Schlage } x \text{ vor}] \tag{4.26}$$

$$= p(x) \cdot \frac{t(x)}{sp(x)} = \frac{1}{s} t(x) \tag{4.27}$$

Beobachte, dass die Ereignisse *Gebe x zurück* und *Gebe y zurück* für $x \neq y$ disjunkt sind, da pro Runde genau ein Element vorgeschlagen wird und es nicht x und y gleichzeitig sein können. Wir können also die Akzeptanzwahrscheinlichkeit als Summe der Wahrscheinlichkeiten berechnen:

$$\mathbb{P}[\text{Akzeptiere}] = \sum_{x \in \Omega} \mathbb{P}[\text{Gebe } x \text{ zurück}] \tag{4.28}$$

$$= \frac{1}{s} \underbrace{\sum_{x \in \Omega} t(x)}_{=1} = \frac{1}{s}$$

□

Der Algorithmus läuft solange, bis zum ersten Mal akzeptiert wird. Die Fehlschläge sind dabei unabhängig voneinander.

Korollar 4.9. Die Anzahl an Runden in Alg. 10 ist geometrisch mit Erfolgswahrscheinlichkeit $\frac{1}{s}$ verteilt. In Erwartung werden also s Versuche benötigt. ◀

Unser Ziel sollte in der Regel sein, ein möglichst kleines s zu finden, das gerade noch $\max_x \{t(x)/(s \cdot p(x))\} \leq 1$ erfüllt. In manchen Fällen ist aber eine obere Schranke schneller zu berechnen, wodurch ein Algorithmus – trotz höherer Rejection-Rate – unterm Strich schneller sein kann.

Aufgabe 4.10. Unser erklärtes Ziel war es, $t(x)$ zu generieren. In Gleichung (4.27) berechneten wir aber, dass x mit Wahrscheinlichkeit $t(x)/s$ ausgegeben wird. Erkläre, weshalb das in Ordnung ist.

Sei $s' > s$ ein suboptimaler Skalierungsfaktor. Wie beeinflusst die Wahl Korrektheit und Effizienz von Alg. 10? ◀

4.2.1.1 Gezinkte Münzen

Wir haben eine ideale deutsche Euromünze, deren beide Seiten A (`Adler`) und Z (`Zahl`) je mit Wahrscheinlichkeit $1/2$ erscheinen. Diese soll als Vorschlagsgenerator dienen; es gilt also $p(A) = p(Z) = 1/2$. In der Zielverteilung soll Z aber mit doppelter Wahrscheinlichkeit auftreten, d. h. $t(A) = 1/3$ und $t(Z) = 2/3$. Berechnen wir den Skalierungsfaktor s:

$$s \geq \max\left\{\frac{t(A)}{p(A)}, \frac{t(Z)}{p(Z)}\right\} = \max\left\{\frac{2/3}{1/2}, \frac{1/3}{1/2}\right\} = \frac{4}{3} \tag{4.29}$$

Natürlich wählen wir den kleinstmöglichen Wert, es gilt also $s = 4/3$. Dann können wir Alg. 10 spezialisieren:

1. Wir werfen eine Münze und erhalten Seite x.
2. Falls $x = Z$, akzeptieren wir mit Wahrscheinlichkeit

   $$a(Z) = \frac{t(Z)}{s \cdot p(Z)} = \frac{\frac{2}{3}}{\frac{4}{3} \cdot \frac{1}{2}} = 1.$$

   Es gibt also nichts zu tun – eine vorgeschlagene `Zahl` wird immer akzeptiert.
3. Falls $x = A$, akzeptieren wir mit Wahrscheinlichkeit

   $$a(A) = \frac{t(A)}{s \cdot p(A)} = \frac{\frac{1}{3}}{\frac{4}{3} \cdot \frac{1}{2}} = \frac{1}{2}.$$

   Statt nun uniform u aus $[0, 1]$ zu ziehen und dann mit $a(A) = 1/2$ zu vergleichen, können wir abkürzen. Wir werfen eine weitere Münze. Falls diese (arbiträr) A ist, akzeptieren wir, sonst verwerfen wir.

Wir können uns den Algorithmus wie folgt bildlich vorstellen, wobei die grauen Werte irrelevant sind und nicht geworfen werden müssen:

Erster Wurf	Zweiter Wurf	Ergebnis	W'keit	…falls akzeptiert
`Zahl`	`Zahl`	`Zahl`	1/4	1/3
`Zahl`	`Adler`	`Zahl`	1/4	1/3
`Adler`	`Zahl`	*Wiederholen*	1/4	–
`Adler`	`Adler`	`Adler`	1/4	1/3

4.2.1.2 Gewichtetes Ziehen ohne Zurücklegen: Rejection-Sampling

Wir betrachten eine Verallgemeinerung des $\mathcal{G}(n, p)$-Samplings aus Abschnitt 2.4.2. Anstelle eines gemeinsamen Akzeptanzparameters p wie bei Gilbert-Graphen sei $w = (w_1, \ldots, w_n) \in [0, 1]^n$ ein Vektor, der jedem Element eine individuelle Wahrscheinlichkeit zuweist. Wir gehen weiter davon aus, dass w absteigend sortiert ist, d. h. $w_i \geq w_{i+1}$ für alle $1 \leq i < n$. Unser Ziel ist es, eine Ausgabemenge $R \subseteq [n]$ zu generieren, d. h. $\mathbb{P}[i \in R] = w_i$. Das Problem ist also *gewichtetes Ziehen ohne Zurücklegen.*

Ein naiver Ansatz iteriert einfach über alle n Elemente und akzeptiert jedes Element mit Wahrscheinlichkeit w_i. Das ist in Laufzeit $\mathcal{O}(n)$ möglich und in Erwartung optimal, falls mindestens ein konstanter Anteil der Elemente von unten durch eine strikt positive Konstante beschränkt ist.

Aufgabe 4.11. Beschreibe für jedes n einen Vektor $w = (w_1, \ldots, w_n)$ mit strikt positiven Gewichten, der zu einer suboptimalen Laufzeit führt. ◀

Das Problem (und auch einen Lösungsansatz!) kennen wir aber schon von $\mathcal{G}(n, p)$-Graphen. Hier haben wir für kleine p zu geometrischen Sprüngen gegriffen und einfach „Nullen" in der Adjazenzmatrix übersprungen. Die „Nullen" entsprechen nun den nicht-gewählten Einträgen.

Allerdings kennen wir hier nicht die genauen Wahrscheinlichkeiten der übersprungenen Elemente. Die offensichtliche Frage lautet daher, welche Erfolgswahrscheinlichkeit wir den Sprüngen zugrunde legen können. Konservativ müssen wir eine Wahrscheinlichkeit wählen, die mindestens so groß wie jedes Gewicht der übersprungenen Elemente ist (es wäre peinlich, ein Element i mit $w_i = 1$ zu verpassen ...).

Hier trifft es sich gut, dass w absteigend sortiert ist. Angenommen, wir haben gerade Element i gewählt, dann führen wir einen geometrischen Sprung mit Erfolgswahrscheinlichkeit w_{i+1} aus. Sei j der Index, den wir gezogen haben. Wenn $w_{i+1} = w_j$ gilt, hatten wir Glück, falls nicht, haben wir w_j überschätzt und müssen das korrigieren … z. B. mit Rejection-Sampling.

Element j wurde mit Wahrscheinlichkeit w_{i+1} vorgeschlagen, es soll aber nur mit Wahrscheinlichkeit w_j ausgegeben werden. Wir akzeptieren also mit Wahrscheinlichkeit $w_j/(s \cdot w_{i+1})$; als Skalierung wählten wir also implizit $s = 1$ (warum?).

Interessant wird es aber, falls der Vorschlag j verworfen wird. Im klassischen Rejection-Sampling generieren wir Vorschläge immer aus derselben Verteilung; falsch ist das auch hier nicht. Effizienter ist es aber, j als neuen Startpunkt zu wählen und einen neuen Sprung mit Erfolgswahrscheinlichkeit w_{j+1} zu ziehen.

Aufgabe 4.12. Begründe, weshalb wir einen neuen Startwert nehmen können, obwohl wir diesen *nicht* in R aufnehmen. ◀

Die Laufzeit dieses Ansatzes analysieren wir im Kontext von Chung-Lu-Graphen später.

4.2.2 Gewichtetes Ziehen mit Zurücklegen: Die Alias-Tabelle

Die Verwerfungsmethode ist ein recht mächtiges Werkzeug, aber nicht immer optimal. Betrachten wir folgendes Problem: Gegeben ist ein Vektor $w = (w_1, \ldots, w_n) \in \mathbb{R}^n_{\geq 0}$ mit $\sum_i w_i = 1$. Wir nehmen an, dass w fremdbestimmt ist und wir a-priori nichts darüber wissen. Wir möchten ein Element X auswählen, wobei $\mathbb{P}[X = i] = w_i$. Vorab dürfen wir $\mathcal{O}(n)$ Zeit investieren, um eine Datenstruktur zu konstruieren.

Mittels Rejection-Sampling ist das einfach. Als Vorschlagsverteilung nehmen wir die uniforme Verteilung auf $[n]$ an; dies ist beste Wahl, falls wir nichts über w wissen. Es gilt also $p(i) = 1/n$ und somit ist kleinste Skalierungskonstante $s = n \max_i w_i$. Diese müssen wir eingangs einmal berechnen – das geht, wie gefordert, in Zeit $\mathcal{O}(n)$.

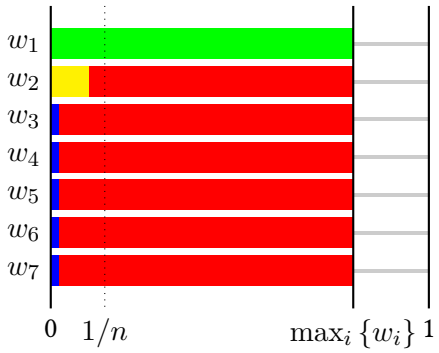


Abbildung 4.2: Rejection-Sampling bei stark verzerrter Verteilung. Wir wählen zuerst eine Zeile zufällig uniform. Dann ziehen wir uniform zufällig einen horizontalen Punkt. Wenn wir dabei die rote Linie treffen, verwerfen wir – bei der gezeigten Verteilung in Zeilen 2 bis n also fast immer.

Beim Blick auf Korollar 4.9 sollten die Alarmglocken läuten. Für ein $\varepsilon > 0$ betrachten wir die Eingabe $w_1 = 1 - \varepsilon$ und $w_i = \varepsilon/(n-1)$ für $1 < i \leq n$. In der Grenze $\varepsilon \to 0$ gilt dann $s \to n$, womit wir eine in der Eingabegröße erwartet linear Laufzeit erhalten. Mit einem Binärbaum hätten wir $\mathcal{O}(\log n)$ Zeit geschafft.

Eine vergleichbar problematische Situation ist in Abb. 4.2 visualisiert. Dem gegenüber wäre eine besonders gute Eingabe ein $w_i = 1/n \pm \varepsilon$ für alle i. In der Abbildung entspräche das der Situation, dass alle Zeilen etwa auf der gestrichelten Linie endeten – es käme zu keiner Rejection.

Tatsächlich können wir durch einen kleinen Kniff eine beliebige Eingabe komplett ohne Verwerfung samplen. Hierzu beobachten wir, dass die Fläche der grünen Box von w_1 rechts des gestrichelten Durchschnitts genau der roten Fläche links des Durchschnitts entspricht. Dies folgt, da die gestrichelte Linie sonst nicht der Durchschnitt wäre.

Die Idee der Alias-Methode besteht darin, den Überschuss auf die rote Fläche aufzuteilen. Hierbei wollen wir aber am *Entweder-oder*-Schema der Verwurfsmethode festhalten; sprich, pro Zeile soll es nur zwei Wahlen geben. Das ist problematisch, da es bis zu $n-1$ überlange Zeilen geben kann, die wird dann nicht alle in die eine unterlange Zeile verteilen dürfen.

Um im Folgenden nicht ständig den Durchschnitt $1/n$ als Faktor schreiben zu müssen, nehmen wir an, dass die Gewichtssumme $\sum_i w'_i$ auf n statt auf 1 normiert ist. Dann folgt das durchschnittliche Gewicht als $\sum_i w'_i/n = n/n = 1$. Die Alias-Tabelle baut auf folgender Einsicht auf:

Theorem 4.13. Seien $w_1, \ldots, w_n$ nichtnegative, reelle Gewichte mit Summe $\sum_i w_i = n$. Dann können wir eine Tabelle mit n Zeilen konstruieren, so dass jede Zeile ein Gewicht von genau 1 hat, und höchstens zwei Einträge pro Zeile vorhanden sind. ◀

Beweis durch Induktion. Für $n = 1$ ist die Aussage trivial wahr, da es nur ein Gewicht $w_1 = 1$ gibt. Nehmen wir also an, dass die Aussage für n gilt und betrachten $n+1$.

Wir haben also Gewichte $w_1, \ldots, w_{n+1}$ mit $\sum_{i=1}^{n+1} w_i = n+1$. Dann existieren

Indizes k und g mit $w_k \leq 1$ und $w_g \geq 1$ (beobachte, dass wir $k = g$ nicht ausschließen). Da $w_k \leq 1$, müssen wir ein Gewicht $c = 1 - w_k$ ergänzen, um eine Zeile mit Gewicht 1 zu erhalten. Dies nehmen wir uns von w_g und produzieren die Zeile.

Übrig bleiben alle unveränderten Gewichte w_i mit $i \notin \{g, k\}$. Falls $g = k$, war $w_k = 1$ und $c = 0$. Falls $g \neq k$, verbleibt noch Eintrag g mit Gewicht $w_g - c > 0$. In jedem Fall sind noch n Gewichte mit einer Summe von $(n + 1) - 1 = n$ im Spiel. Per Induktionsannahme können wir für diese eine entsprechende Tabelle konstruieren. □

Konkret besteht unsere Tabelle aus zwei Arrays $G[1 \ldots n]$ und $A[1 \ldots n]$. In der i-ten Zeile speichern wir als ersten Eintrag Element i mit einem anteiligen Gewicht von $G[i] \leq w_i$ ab. Der Rest der Zeile wird vom sog. Alias $A[i]$ eingenommen, der per Konstruktion ein Gewicht $1 - G[i]$ hat und daher nicht explizit gespeichert werden muss.

Für jede endliche Verteilung $w_1, \ldots, w_n$ können wir eine solche Tabelle in Zeit $\mathcal{O}(n)$ konstruieren – i. A. ist die Tabelle aber nicht eindeutig. Eine mögliche Belegung kann mittels zweier Listen K (klaaner) und D (driwwer) erzeugt werden.

- Falls $w_i < 1$, fügen wir (i, w_i) in K ein.
- Falls $w_i \geq 1$, fügen wir (i, w_j) in D ein.

Solange K und D nicht leer sind, entnehmen wir ein Element (k, w_k) aus K und ein Element (g, w_g) aus D. Wir schreiben $G[i] = w_k$ und $A[i] = g$. Sei $w'_g = w_g - (1 - w_k)$ das verbleibende Gewicht von g. Wenn $w'_g < 1$ fügen wir (g, w'_g) in K ein, sonst in D.

Per Konstruktion kann K niemals vor D leer werden; es kann aber durchaus vorkommen, dass K leer ist, während D noch Werte enthält. Dann haben aber alle Einträge in G Gewicht genau 1; wir können sie in der offensichtlichen Art in G einfügen. Praktische Implementierungen müssen aufgrund der wiederholten Subtraktionen mit Rundungsfehlern der Gewichte umgehen können; dann kann auch D zuerst leer werden; die Elemente in K haben dann aber Gewicht $1 - \varepsilon$.

Das Ziehen aus der Alias-Tabelle läuft analog zu Rejection-Sampling: Wir wählen eine Zeile $1 \leq i \leq n$ und ein reelles $0 \leq u \leq 1$ jeweils rein zufällig aus. Falls $u < G[i]$, geben wir i zurück, sonst $A[i]$. Dies gelingt in Zeit $\mathcal{O}(1)$.

4.2.3 Intermezzo: Schnelles Ziehen von uniformen Ganzzahlen

Dieses Kapitel basiert auf [13]. Wir übernehmen auch die Notation $a \div b = \lfloor a/b \rfloor$.

Auf praktischen Computern erhalten wir (Pseudo-)Zufallszahlen in der Regel in der Größe eines Maschinenwortes, oft mit $L = 32$ oder 64 Bit. Wir können dies entweder als L unabhängige Zufallsbits interpretieren, oder z. B. als eine vorzeichenlose Ganzzahl X, die uniform auf $[0, 2^L)$ verteilt ist. In Anwendungen benötigen wir jedoch häufig Zahlen aus einem Intervall $[a, b)$ und müssen daher X transformieren.

Den kleinsten Wert anzupassen ist einfach: Wir ziehen aus $Y' \in [0, b - a)$ und geben $Y = a + Y'$ zurück. Daher reduzieren wir im Folgenden die Diskussion auf

Intervalle $[0, s)$, wobei wir $1 \leq s \leq 2^L$ annehmen (die obere Schranke lässt sich durch Konkatenation mehrerer L-Bit-Wörter umgehen).

In der Praxis wird eine Zahl $X \in [0, 2^L)$ oft auf $[0, s)$ reduziert, indem $Y = X \bmod s$ berechnet wird – alleine auf GitHub findet sich das Pattern „`rand()  %`“ fast eine halbe Million mal. Der einzige Grund: Es ist schnell zu schreiben und garantiert $Y \in [0, s)$. Es ist aber zum einen recht langsam und zum anderen auch nur dann uniform, wenn $2^L \bmod s = 0$ gilt. Dies lässt sich an einem Bild leicht erkennen. Betrachte die Sequenz $0, \ldots, 2^L \bmod s$:

$$\underbrace{\overbrace{0, 1, \ldots, s{-}1}^{s \text{ Werte}}, \overbrace{0, 1, \ldots, s{-}1}^{s \text{ Werte}}, \ \ldots, \ \overbrace{0, 1, \ldots, s{-}1}^{s \text{ Werte}}}_{(2^L \div s) \text{ Blöcke mit insg. } 2^L - (2^L \div s) \text{ Werten}}, \overbrace{0, 1, \ldots, (2^L \bmod s){-}1}^{2^L \bmod s \text{ Werte}}, \tag{4.30}$$

Jeder Wert in $[0, s)$ taucht also exakt $\lfloor 2^L/s \rfloor$ oder $\lceil 2^L/s \rceil$ oft auf. Es handelt sich also genau dann um eine uniforme Verteilung, wenn $\lfloor 2^L/s \rfloor = \lceil 2^L/s \rceil \ \Leftrightarrow \ 2^L \bmod s = 0$. Wir formalisieren dies in folgendem Lemma:

Lemma 4.14. Seien $a < b$ die Grenzen des Intervalls $[a, b)$ und $s \in \mathbb{N}_{>0}$. Dann existieren für jedes $0 \leq c < s$ genau dann $(b - a) \div s$ verschiedene Ganzzahlen $x \in [a, b)$ mit $x \bmod s = c$, wenn s die Intervalllänge $b - a$ teilt. ◀

Beweis durch Bild. □

Für $s \ll 2^L$ kann der Unterschied zwischen $\lfloor 2^L/s \rfloor$ und $\lceil 2^L/s \rceil$ verschmerzbar klein sein, für $s > 2^L/2$ gibt es aber einige Elemente, die mit doppelter Wahrscheinlichkeit von anderen zurück geliefert werden.

4.2.3.1 Rejection Sampling to the help

Es sollte nicht überraschen, dass Rejection-Sampling hier helfen kann. Ein naiver Ansatz verwirft $X \geq s$ und akzeptiert nur $X < s$. Wenn X uniform ist, ist es die Ausgabe dann auch. Allerdings ist für $s \ll 2^L$ die Verwerfungsrate exorbitant hoch; in Erwartung benötigen wir $2^L/s$ Versuche!

Besser ist es natürlich, ein möglichst großes $k \geq 1$ zu wählen, s. d. $ks \leq 2^L$. Dann verwerfen wir nur $X \geq ks$ und liefern sonst X/k zurück. Die beste Wahl ist $k = 2^L \div s$. Dieses Verfahren hat eine Akzeptanzwahrscheinlichkeit von

$$\frac{k \cdot s}{2^L} = 1 - \frac{2^L \bmod s}{2^L} \geq 1/2, \tag{4.31}$$

und benötigt daher in Erwartung höchstens zwei Versuche.

Beobachte, dass es hierbei egal ist, ob wir $X \geq ks$ oder $X < (2^L - ks) = 2^L \bmod s = (2^L - s) \bmod s$ verwerfen. Die letzte Variante ist aber auf echten Maschinen in der Regel schneller. Das führt zu folgendem Schema, das auch als OpenBSD-Algorithmus bekannt ist:

Algorithmus 11 : OpenBSD-Algorithmus zum Ziehen uniformer Ganzzahlen

```
1 t ← (2^L − s) mod s  /* 2^L mod s = (2^L − s) mod s. 2^L ist aber ggf. ein Overflow */
2 x ← zufällige Ganzzahl aus [0, 2^L)
3 while x < t do
4     x ← zufällige Ganzzahl aus [0, 2^L)
5 return x mod s
```

Der OpenBSD-Algorithmus ist einfach und korrekt, hat aber einen enormen Nachteil in der Praxis: Pro Aufruf muss ein Modulo und eine Division (das ist idR dieselbe CPU-Instruktion!) berechnet werden. Eine `div`-Instruktion gehört zu den teuersten skalaren Operationen[1], die CPUs unterstützen. Im Vergleich zu einfacher Arithmetik (z. B. Addition) ist eine Division oft um ein bis zwei Größenordnungen langsamer. Insbesondere geht das Erzeugen einer Zufallszahl bei modernen, nichtkryptographischen Pseudozufallsgeneratoren oft schneller vonstatten als eine einzelne Division!

OpenJDK verwendet ein anderes Verfahren, das die Anzahl der Divisionen an die Anzahl der Runden des Rejection-Samplings knüpft.

Algorithmus 12 : Java-Algorithmus zum Ziehen uniformer Ganzzahlen

```
1 x ← zufällige Ganzzahl aus [0, 2^L)
2 r ← x mod s
3 while x − r > 2^L − s do
4     x ← zufällige Ganzzahl aus [0, 2^L)
5     r ← x mod s
6 return r
```

Wir müssen genau dann verwerfen, wenn $ks \leq x < (k+1)s$ mit $(k+1)s > 2^L$, d. h., wenn x aus einem „s-Intervall“ stammt, das nicht vollständig in $[0, 2^L)$ enthalten ist. Beobachte, dass $x - r = x - (x \bmod s) = (x \div s)s$ dem größten Vielfachen von s entspricht, das x nicht übersteigt; es ist also die linke Intervallgrenze. Die rechte Intervallgrenze ergibt sich dann als $x - r + s$. Allerdings kann $x - r + s$ zu einem Überlauf führen; daher prüft der Algorithmus – äquivalent, aber sicher vor Überlauf – auf $x - r > 2^L - s$.

Der Vorteil des Java-Algorithmus gegenüber der OpenBSD-Methode besteht darin, dass Java im günstigen Fall nur eine Division ausführen muss. Wenn für $s \to 2^L$ die Verwerfungsrate jedoch steigt, nimmt auch die Anzahl der Division zu und ist im Worst-Case gar unbeschränkt (eine mit hoher Wahrscheinlichkeit gültige Schranke lässt sich mittels Chernoff zeigen).

Lemire [13] schlägt daher ein Verfahren vor, das oft komplett ohne (generische) Division auskommt und im Worst-Case nur eine benötigt. Hierzu nutzt er aus, dass eine Division $x \div 2^k$ nicht durch eine aufwendige `div`-Instruktion ausgeführt werden muss, sondern einfach einem Bitshift um k Bits nach rechts entspricht. Konkret nutzen wir, dass

$$0 \leq (x \cdot s) \div 2^L < s \quad \forall x \in [0, 2^L), s \in [0, 2^L). \tag{4.32}$$

Dabei implementieren wir $(x \cdot s) \div 2^L$ als `shift-right`$(x \cdot s, L)$; der Ausdruck

[1] Sehr umfangreiche Benchmarks und Infos: https://www.agner.org/optimize/instruction_tables.pdf

ist also im Wesentlichen eine Multiplikation von zwei L-Bit-Zahlen zu einer $2L$-Bit-Zahl gefolgt von einem Shift um L Bits. Analog können wir die unteren L Bits mittels Verundung der unteren L Bits extrahieren. Warum ist das nützlich?

Durch die Multiplikation von s mit einem zufälligen $x \in [0, 2^L)$ bilden wir auf alle Vielfachen von s in $[0, s \cdot 2^L)$ ab. Durch die Division mit 2^L bilden wir alle Vielfachen von s in $[0, 2^L)$ auf 0, alle in $[2^L, 2 \cdot 2^L)$ auf 1 und alle in $[i2^L, (i+1) \cdot 2^L)$ auf i ab. Im Allgemeinen teilt aber s nicht 2^L (sonst wären wir schon fertig); daher müssen wir $2^L \bmod s$ Elemente aus dem Intervall herausschneiden. Wir verwerfen daher für jedes i die Elemente in $[i2^L, i2^L + (2^L \bmod s))$ und akzeptieren $[i2^L + (2^L \bmod s), (i+1) \cdot 2^L)$. Das führt zu folgendem Algorithmus:

Algorithmus 13 : Uniformes Ziehen ganzer Zahlen fast ohne Division

```
1  x ← zufällige Ganzzahl aus [0, 2^L)
2  m ← x · s
3  ℓ ← m mod 2^L
4  if ℓ < s then                    /* Abkürzung, falls wir akzeptieren können */
5      t ← (2^L − s) mod s          /* 2^L mod s = (2^L − s) mod s */
6      while ℓ < t do
7          x ← zufällige Ganzzahl aus [0, 2^L)
8          m ← x · s
9          ℓ ← m mod 2^L
10 return m ÷ 2^L
```

Der `if`-Block ist dabei nicht funktional wichtig – wenn wir die Bedingung durch `true` ersetzen, arbeitet der Algorithmus weiterhin korrekt und führt die Rejection richtig aus. Er ist aber essentiell in der Vermeidung der Division. Beobachte, dass $t = 2^L \bmod s = (2^L - s) \bmod s < s$ ist und wir nur verwerfen, wenn $\ell < t$. Das bedeutet aber insbesondere auch, dass wir für $\ell \geq s$ wissen, dass ℓ nicht kleiner als t sein kann. Daher müssen wir t erst gar nicht berechnen. Der Algorithmus läuft daher nur mit Wahrscheinlichkeit $s/2^L$ in den `true`-Block; mit der Gegenwahrscheinlichkeit $1 - s/2^L$ ist er also divisionsfrei.

Die Strategie, Abkürzungen für sicheres Verwerfen/Akzeptieren zu konstruieren, um unnötige Berechnungen zu sparen, ist eine gängige Technik in effizienten Implementierungen von Rejection-Sampling.

4.2.4 Alias-Tabelle: Teil 2

In Abschnitt 4.2.3 stellen wir effiziente Methoden vor, um uniforme Ganzzahlen zu ziehen. Hiermit können wir nun schnell eine zufällige Zeile in der Alias-Tabelle bestimmen. Es geht aber noch besser:

1. Der erste Ansatz besteht darin, Alg. 13 zu optimieren. Beobachte, dass wir immer einen Zeilenindex aus $[0, n)$ (oder äquivalent $[1, n]$) ziehen. Es gilt also, dass die Intervallgröße auf $s = n$ fixiert ist. Wir können daher $t \leftarrow (2^L - s) \bmod s$ einmalig bei der Konstruktion der Tabelle berechnen und ab dann echt divisionsfrei

ziehen. Dann können wir sogar auf die getrennte `if`-Abfrage verzichten. Diese Technik funktioniert immer, wenn sich die Intervallgröße nicht (oft) ändert.

2. Wir können Rejection-Sampling sogar komplett vermeiden, indem wir die Anzahl der Elemente n auf die kleinste Zweierpotenz $n' \geq n$ aufrunden. Dann reicht es aus, $\log_2 n'$ Zufallsbits zu entnehmen, um eine zufällige Zeile zu samplen. Für die neuen $n' - n < n$ Elemente setzen wir einfach ein Gewicht von 0 (und renormieren ggf. die restlichen Elemente).

Die Einfachheit der Alias-Tabelle macht sie in praktischen Anwendungen oft komplexeren Datenstrukturen überlegen. Ein Beispiel wird in [5] vorgestellt, in dem eine Abwandlung der Alias-Tabelle für ganzzahlige und dynamische Verteilungen genutzt wird. Die dort betrachtete Anwendung muss eine Urne mit n Bällen verwalten, wobei jeder Ball eine von k Farben hat. Zwei Bälle gleicher Farbe sind ununterscheidbar. Es gilt außerdem, dass $n \ggg k^2$ ist (oft ist $k = o(\log n)$). Die Anwendung benötigt drei Operationen:

- Ziehe einen Ball zufällig uniform und gebe die Farbe zurück.
- Ziehe *und entnehme* einen Ball zufällig uniform und gebe die Farbe zurück.
- Füge einen Ball mit beliebiger Farbe ein.

Um die Datenstruktur speichereffizient zu gestalten, konstruieren wir eine Tabelle mit k Zeilen. Statt die Verteilung auf 1 oder k zu normieren, verwenden wir direkt die Anzahl an Kugeln pro Farbe. Das hat zum Einen den Vorteil, dass wir mit reiner Ganzzahlarithmetik arbeiten können, zum Anderen aber auch nicht renormieren müssen, wenn Kugeln hinzugefügt oder entnommen werden.

Aus unserer Diskussion des letzten Kapitels wissen wir aber bereits, dass es im Allgemeinen nicht möglich sein kann, n Kugeln auf k Zeilen so zu verteilen, dass alle Zeilen identisches Gewicht haben. Sei $T[i]$ das Gewicht der i-ten Zeile. Falls $n \bmod k \neq 0$, gibt es mindestens zwei Zeilen i und j mit $T[i] \leq \lfloor n/k \rfloor \neq \lceil n/k \rceil \geq T[j]$. Mit Rejection-Sampling können wir aber die leichte Verzerrung einfach korrigieren. Hierfür berechnen wir einfach eingangs das Gewicht g_{max} der schwersten Zeile und akzeptieren eine zufällig gezogen Zeile i mit Wahrscheinlichkeit $T[i]/g_{\text{max}} \leq 1$.

In einer klassischen Alias-Tabelle haben alle Zeilen Gewicht genau 1. Somit ist es ausreichend, pro Zeile die Wahrscheinlichkeitsmasse $G[i]$ eines Eintrags zu speichern – die des anderen folgt automatisch als $1 - G[i]$. In der modifizierten Tabelle speichern wir hingegen das ganzzahlige Gewicht $G[i]$ des ersten Eintrags sowie das Gesamtgewicht $T[i]$ der Zeile. Das Gewicht des zweiten Eintrags ist also $T[i] - G[i]$ – eine Größe, die aber niemals explizit benötigt wird.

Wir können also Ziehen *mit Zurücklegen* implementieren. Die Verwerfungsmethode erlaubt es uns aber auch, einfach *ohne Zurücklegen* zu ziehen: Wenn wir einen Eintrag in der i-ten Zeile zufällig gezogen haben, reduzieren wir $T[i]$ um eins; wenn es der erste Eintrag der Zeile war, auch noch $G[i]$. Dieser Zufallsprozess ist recht gutmütig verteilt.

Aus Ball-in-Bins-Spielen wissen wir, dass wenn wir n Bälle auf k Eimer uniform zufällig verteilen, für $n > k \log k$ mit hoher Wahrscheinlichkeit kein Eimer mehr als $n/k + \mathcal{O}\left(\sqrt{(n \log k)/k}\right)$ Bälle hat. Der zuvor genannte Prozess ist sogar noch konzentrierter, da Zeilen mit wenig Gewicht seltener gewählt werden und Zeilen mit höherem Gewicht schneller Bälle verlieren. Wir können also davon ausgehen, dass alle Zeilen mit hoher Wahrscheinlichkeit ähnlich schnell an Masse verlieren und das System so balanciert bleibt. Allerdings sollten wir g_{max} von Zeit zu Zeit aktualisieren, um eine unnötig hohe Verwerfungsrate zu vermeiden (s. u.).

Jedoch sprach unser Lastenheft davon, dass wir auch noch Kugeln mit beliebigen Farben hinzufügen können müssen. Das Einfügen einer Kugel mit Farbe i implementieren wir, indem wir $G[i]$ und $T[i]$ je um eins inkrementieren; wir erhöhen also einfach das Gewicht des ersten Eintrags der i-ten Zeile. Falls $T[i]$ nun g_{max} übersteigt, setzen wir noch $g_{\text{max}} \leftarrow T[i]$. Ein bösartiger Gegenspieler könnte folglich immer dieselbe Farbe hinzufügen und damit erreichen, dass g_{max} linear steigt. Das führt zu einer Situation analog zu Abb. 4.2 – eine Zeile vereinnahmt den Großteil der Wahrscheinlichkeitsmasse, während alle anderen zu einer enorm hohen Verwerfungsrate führen.

Wir lösen das Problem, indem wir nicht nur g_{max}, sondern auch g_{min} speichern und aktualisieren (zeige, dass das auch trivial implizit geht!). Bei jeder Änderung einer der beiden Größen prüfen wir, ob noch die Invariante

$$\alpha \frac{n}{k} < g_{\text{min}} \leq g_{\text{max}} < \beta \frac{n}{k} \tag{4.33}$$

erfüllt ist. Falls nicht, rekonstruieren wir die Datenstruktur. Es gilt also $\alpha \leq 1$ und $\beta \geq 1$; falls $\beta/\alpha = \mathcal{O}(1)$, ergeben sich in Erwartung $\mathcal{O}(1)$ Versuche, bis wir eine Kugel akzeptieren.

Das Neubauen der Datenstruktur ist in Zeit $\mathcal{O}(k)$ möglich. Für $\alpha < 1$ und $\beta > 1$ benötigen wir $\Omega(n/k)$ Zugriffe auf die Datenstruktur, bis die Invariante frühstens verletzt ist. Im schlimmsten Fall müssen wir also $\mathcal{O}(k/(n/k)) = \mathcal{O}\left(k^2/n\right)$ amortisierte Arbeit pro Zugriff leisten. Da wir $k^2 < n$ annehmen, ist dieser Beitrag nur $\mathcal{O}(1)$. Alle Ergebnisse werden von folgendem Theorem zusammengefasst:

Theorem 4.15. Sei U eine Urne, die mittels zuvor beschriebener dynamischer Alias-Tabelle implementiert ist und n Kugel mit k Farben unterstützt. Dann benötigt U genau $\Theta(k \log n)$ Bits Speicherplatz. Falls $n \geq k^2$, unterstützt sie folgende Operationen:

- Ziehen einer Kugel *mit Zurücklegen* in erwarteter Zeit $\mathcal{O}(1)$
- Ziehen einer Kugel *ohne Zurücklegen* in erwartet amortisierter Zeit $\mathcal{O}(1)$
- Einfügen einer Kugel mit beliebiger Farbe in amortisierter Zeit $\mathcal{O}(1)$ ◀

4.2.5 Allgemeines dynamisches gewichtetes Ziehen

Dieses Kapitel basiert auf [14].

Im vorherigen Abschnitt diskutierten wir eine Modifikation der Alias-Tabelle, die dann effizient ist, wenn die ganzzahlige Gewichtssumme W deutlich größer als die Anzahl

der Zeilen n ist (wir nehmen $W > n^2$ an). Im Folgenden betrachten wir eine Datenstruktur, die n beliebige, positive Gewichte unterstützt. Um die Beschreibung einfach zu halten, stellen wir eine Vereinfachung vor, die Ziehen in erwarteter Zeit $\mathcal{O}(\log^*(n))$ und Änderungen eines Wertes in Zeit $\mathcal{O}\big(2^{\log^*(n)}\big)$ unterstützt. Mit ein paar Tricks können beide Schranken auf $\mathcal{O}(1)$ (worst-case) gedrückt werden.

iterierter Logarithmus
$\underbrace{\log_2(\ldots\log_2}_{k\text{-mal}}(x)) \leq 1$

Hierbei ist $\log^*(n)$ der sog. *iterierte Logarithmus*. Dieser ist definiert als das kleinste ganzzahlige k, s. d. $\log^{(k)}(n) \leq 1$, wobei $\log^{(k)}(n)$ die k-fache Anwendung des Logarithmus ($\log_2(\log_2(\ldots\log_2(x))))$ beschreibt. Es ist eine extrem langsam wachsende Funktion. Beispielsweise gilt $\log^*(10^{80}) = 4$, wobei 10^{80} etwa der Anzahl an Atome im Universum entspricht, und $\log^*(2^{65536}) = 5$, wobei 2^{65536} mehr als 20 000 Ziffern in Dezimalschreibweise hat. Für alle praktischen Anwendungen, in denen n linear zu einer Ressource (etwa Speicher oder Laufzeit) korrespondiert, können wir $\log^*(n)$ daher als Konstante von höchstens 4 annehmen.

Als Eingabe erhalten wir n strikt positive[2] Gewichte $(w_1, \ldots, w_n) \in \mathbb{Q}^n_{>0}$. Dann sei $W = \sum_{i=1}^n w_i$ deren Summe. Wir nehmen ein RAM-Modell an, wobei Arithmetik auf Ganzzahlen (oder Festkommadarstellung mit ausreichender Präzision) der Größenordnung $\mathcal{O}(W)$ in konstanter Zeit möglich sei. Unsere Aufgabe ist es, eine Datenstruktur zu konstruieren, die folgende Operationen effizient unterstützt:

- *sample*(): Gibt ein zufälliges $1 \leq X \leq n$ zurück, wobei $\mathbb{P}[X = k] = w_k/W$ ist.
- *update*(i, Δ): Aktualisiert das Gewicht w_i auf $w_i \leftarrow w_i + \Delta$ und passt die Datenstruktur an.

Hierzu erstellen wir Bäume in Bottom-up-Richtung. Die Blätter entsprechen den Eingabeelementen und werden als Level $\ell = 1$ bezeichnet. Die Datenstruktur wird nur Bäume mit Höhe $\mathcal{O}(\log^*(n))$ beinhalten. Da es aber $\mathcal{O}(1)$ Bäume geben *kann*, sollten wir im Schnitt pro Ebene k Kinder auf $\mathcal{O}(\log k)$ Eltern verteilen – dann folgt die Tiefe $\mathcal{O}(\log^*(n))$ automatisch.

$R_j^{(\ell)}$: Intervall $[2^{j-1}, 2^j)$ auf Ebene ℓ

Intervallnummer

rekursive Struktur

Wir gruppieren die Elemente nach ihrem Gewicht in exponentiell wachsende Intervalle: So enthält $R_j^{(1)}$ alle Gewichte in $[2^{j-1}, 2^j)$, wobei wir j als *Intervallnummer* bezeichnen. Da die Gewichte unskaliert sind, kann es sowohl negative als auch positive Intervallnummern geben. Wir bezeichnen die Summe aller Gewichte in Intervall $R_j^{(\ell)}$ als $\mathrm{weight}\big(R_j^{(\ell)}\big)$. Je nachdem, wie viele Elemente m in ein Intervall $R_j^{(\ell)}$ fallen, wählen wir für das Intervall einen von drei Fällen:

- Kein Element ($m = 0$): Das Intervall wird ignoriert und insb. nicht gespeichert.
- Genau ein Element ($m = 1$): Wir bezeichnen $R_j^{(\ell)}$ als Wurzel und speichern sie in der sog. Level-Tabelle T_ℓ ab (mehr dazu später!).
- Mindestens zwei Elemente ($m \geq 2$): Wir bezeichnen $R_j^{(\ell)}$ als internes Intervall. Es entspricht einem einzelnen Element mit Gewicht $\mathrm{weight}\big(R_j^{(\ell)}\big)$ auf der nächsthöheren Ebene $\ell + 1$.

[2] Es ist trivial, auch Gewicht 0 zu unterstützen; wir verbieten es hier nur, um Randfälle zu vermeiden.

In Level $\ell > 1$ fassen wir die internen Knoten aus $\ell - 1$ entsprechend ihrer Gewichtssumme in Intervalle $R_j^{(\ell)}$ zusammen und verfahren rekursiv analog zu Level $\ell = 1$. Ein Baum entsteht also dadurch, dass wir die Gewichte, interne Intervalle und Wurzel als Knoten interpretieren und mittels Kanten die Zugehörigkeit darstellen. Beachte, dass jeder Knoten – mit Ausnahme der Wurzel und der Blätter – mindestens zwei Nachfahren hat. Die Datenstruktur hat daher $\Theta(n)$ Knoten.

Level-Tabellen

Da es im Allgemeinen mehrere Wurzeln gibt, ist die Datenstruktur ein Wald, wobei jeder Baum über die Level-Tabelle T_ℓ seiner Wurzel $R_j^{(\ell)}$ adressierbar ist. Jede Level-Tabelle T_ℓ entspricht einer Hashtabelle, welche die Wurzeln $R_j^{(\ell)}$ mit Schlüssel j speichert (wir können auch alle Level-Tabellen in einer Hashtabelle zusammenfassen, wenn wir die Schlüssel auf (ℓ, j) erweitern). Pro Level-Tabelle speichern wir noch das Gesamtgewicht $\mathrm{weight}(T_j) = \sum_{j \in T_\ell} \mathrm{weight}\left(R_j^{(\ell)}\right)$ des Levels.

Wir verwalten zudem ein Bitset, um die Wurzel in Level ℓ zu markieren. Etwas „hacky“ speichern wir dies als eine Binärzahl $X_\ell = \sum_{j \in T_\ell} 2^j$ in Festkommadarstellung, in der das j-te Bit genau dann gesetzt ist, wenn $j \in T_\ell$. Aufgrund unserer Annahmen bzgl. des Maschinenmodells können wir X_ℓ in einem Wort darstellen und in konstanter Zeit z. B. das höchstwertige Bit finden.

4.2.5.1 Ziehen aus der Datenstruktur

Sampling

Schließlich sei L das größte Level in der Datenstruktur; wir werden zeigen, dass $L = \mathcal{O}(\log^*(n))$ gilt. Jetzt können wir den Sampling-Algorithmus beschreiben:

1. Ziehe ein $1 \leq \ell \leq L$ zufällig gewichtet nach $\mathrm{weight}(T_\ell)$. Da $L = \mathcal{O}(\log^*(n))$, können wir das mittels linearer Suche implementieren: Wir ziehen U uniform aus $[0, W)$ und suchen das kleinste ℓ, s. d. $U < \sum_{1 \leq k \leq \ell} \mathrm{weight}(T_k)$.

2. Ziehe ein $j \in T_\ell$ zufällig gewichtet nach $\mathrm{weight}\left(R_j^{(\ell)}\right)$. Auch hier können wir eine lineare Suche anwenden: Ziehe U uniform aus $[0, \mathrm{weight}(T_j))$ und suche das kleinste j, s. d. $U \leq \sum_{k \leq j} \mathrm{weight}\left(R_k^{(\ell)}\right)$. Zwar haben wir keine nützliche obere Schranke für die Anzahl der Wurzeln pro Level, jedoch können wir bei der schwersten Wurzel (d. h. dem höchstwertigsten gesetzten Bit in X_ℓ) starten und dann j iterativ dekrementieren. Da sich das größtmögliche verbleibende Gewicht jedes Mal grob halbiert, ergibt sich eine erwartet konstante Laufzeit (vgl. geometrisch Verteilung).

3. Innerhalb von $R_j^{(\ell)}$ wählen wir nun ein Element gewichtet nach dessen Gewicht mittels Verwerfungsmethode. Da alle Gewichte in $R_j^{(\ell)}$ aus dem Intervall $[2^{j-1}, 2^j)$ stammen, ergibt sich eine Akzeptanzwahrscheinlichkeit von mindestens $1/2$. Falls $\ell = 1$, geben wir das entsprechende Element aus, anderenfalls steigen wir rekursiv in das gewählte interne Intervall ab und wiederholen 3.). Da $L = \mathcal{O}(\log^*(n))$, ergeben sich höchstens $\mathcal{O}(\log^*(n))$ Rekursionsschritte.

4.2.5.2 Erstellen und Aktualisieren der Datenstruktur

Initial können wir die Datenstruktur in Zeit $\mathcal{O}(n)$ konstruieren. Dafür verarbeiten wir die $\mathcal{O}(n)$ Gewichte der Eingabe und fassen sie in Intervalle $R_j^{(1)}$ zusammen. Alle internen Intervalle (d. h. solche, mit mindestens zwei Elementen) sammeln wir zunächst in einer Queue. Diese verarbeiten wir rekursiv, nachdem das aktuelle Level fertiggestellt wurde. Die Verarbeitungszeit eines Levels ist linear in der Anzahl der Elemente des Levels möglich. Da die Datenstruktur insg. $\Theta(n)$ Knoten hat, folgt also eine Gesamtlaufzeit von $\mathcal{O}(n)$.

Interessanter sind dynamische Updates. Hierzu soll der Anwender ein beliebiges Gewicht w_i um einen beliebigen Betrag Δ anpassen können. Wir beginnen in Level $\ell = 1$ und passen das entsprechende Gewicht an. Es gibt zwei Optionen:

- Die Änderungen sind ausreichend klein, s. d. das Gewicht in seinem ursprünglichen Intervall $R_j^{(1)}$ verbleibt. Dann müssen wir das entsprechende Intervall anpassen (d. h. sein Gesamtgewicht und die Änderungen auch rekursiv in Tabelle T_ℓ und Level $\ell + 1$ widerspiegeln).
- Wenn das Element von $R_j^{(1)}$ in ein neues Intervall $R_{j'}^{(1)}$ wechselt, müssen wir die Änderungen für beide Intervalle propagieren.

Pro Level kann sich also die Anzahl der betroffenen Intervalle im Worst-Case verdoppeln. Da es $L = \mathcal{O}(\log^*(n))$ Level gibt, folgt schlimmstenfalls eine Update-Zeit von amortisiert $\mathcal{O}\big(2^{\log^*(n)}\big)$. Die Amortisierung findet dabei über die Hilfsdatenstrukturen (z. B. Hash-Tabellen) statt.

4.2.5.3 Tiefe des Waldes

Im Folgenden skizzieren wir die Beweisidee, um die Tiefe des Waldes zu beschränken. Die vollständige Analyse der Datenstruktur findet sich in [14], ist aber recht technisch und wird daher hier nur zusammengefasst.

Lemma 4.16. Wenn ein Intervall $R_j^{(\ell)}$ mindestens $m \geq 2$ Kinder hat, ist das Gesamtgewicht $\text{weight}\big(R_j^{(\ell)}\big)$ in $[2^{k-1}, 2^k)$ wobei $\log_2(m) - 1 < k - j < \log_2(m) + 1$. ◀

Beweis. Jedes Kind in $R_j^{(\ell)}$ hat ein Gewicht in $[2^{j-1}, 2^j)$. Daher folgt für das Gesamtgewicht $\text{weight}\big(R_j^{(\ell)}\big) \in [m2^{j-1}, m2^j)$. Wir weisen das Intervall als Kind dem Intervall $R_k^{(\ell+1)}$ zu:

$$2^{k-1} \leq \text{weight}\big(R_j^{(\ell)}\big) < m2^j \tag{4.34}$$

$$\Leftrightarrow \quad \log_2(2^{k-1}) < \log_2(m2^j) \tag{4.35}$$

$$\Leftrightarrow \quad k - 1 < \log_2(m) + j \tag{4.36}$$

$$\Leftrightarrow \quad k - j < \log_2(m) + 1 \tag{4.37}$$

Analog folgt aus $m2^{j-1} \leq \text{weight}\big(R_j^{(\ell)}\big) < 2^k$ die Schranke $\log_2(m) - 1 < k - j$. □

Auf dieser Basis können wir nun die Anzahl der Kinder von höheren internen Intervallen beschränken:

Lemma 4.17. Für $\ell \geq 2$ sei $R_j^{(\ell)}$ ein Intervall mit mindestens $m \geq 2$ Kindern. Dann hat eines dieser Kinder mindestens $2^{m-1} + 1$ Nachfolger; außerdem hat das Intervall mindestens $2^m + m - 1$ Enkelkinder. ◀

Beweis. Seien $R_{j_1}^{(\ell-1)}, \ldots, R_{j_m}^{(\ell-1)}$ die Kinder von $R_j^{(\ell)}$, wobei $j > j_1 > j_2 > \ldots > j_m$. Aufgrund der strikten Monotonie gilt $j_i \leq j - i$. Aus Lemma 4.16 wissen wir außerdem, dass $R_{j_i}^{(\ell-1)}$ mindestens $2^{j-j_i-1} + 1 \geq 2^{i-1} + 1$ Kinder hat (sonst könnte es nicht Kind von $R_j^{(\ell)}$ sein). Für $i = m$ ergibt sich also also der erste Teil der Behauptung. Die Anzahl der Enkelkinder lässt sich als Summe der Kinder von unten beschränken

$$\sum_{i=1}^{m}(2^{i-1} + 1) = m + \sum_{i=0}^{m-1} 2^i = m + 2^m - 1. \tag{4.38}$$

□

Aus diesen beiden Lemmata folgt, dass für ein $\ell \geq k \geq 3$ die Anzahl der Nachfahren auf Level $\ell - k$ eines internen Intervalls $R_j^{(\ell)}$ größer als

$$2^{2^{\cdot^{\cdot^{2^m}}}} \Big\}k\text{-mal} \tag{4.39}$$

ist, d. h. 2^{2^m} für $k = 3$ und $2^{2^{2^m}}$ für $k = 4$. Diese untere Schranke für die Anzahl der Nachfahren übersetzt sich dann wiederum in eine obere Schranke $\mathcal{O}(\log^*(n))$ der Tiefe.

4.3 Charakterisierungen graphischer Sequenzen

Wir können graphische Gradsequenzen anhand mehrerer Eigenschaften charakterisieren, wobei das Erdős-Gallai-Theorem sowie das Havel-Hakimi-Theorem wohl die bekanntesten Ansätze darstellen. Das Erdős-Gallai-Theorem ist nicht konstruktiv und kommt daher in der Praxis besonders zum Testen auf graphische Sequenzen zum Einsatz. Aus dem Havel-Hakimi-Theorem hingegen können wir einen Graphengenerator konstruieren. Die Intuition beider Ansätze ist aber ähnlich.

4.3.1 Erdős-Gallai-Theorem

Theorem 4.18. Sei $\mathcal{D} = (d_1, \ldots, d_n)$ eine nicht wachsende Gradsequenz, d. h.

$$n > d_1 \geq d_2 \geq \ldots \geq d_n \geq 1. \tag{4.40}$$

Dann ist $\mathcal{D}$ genau dann graphisch, wenn folgende beiden Bedingungen erfüllt sind:

1. Die Summe der Grade $\sum_{i=1}^{n} d_i$ ist gerade (durch 2 teilbar).
2. Für alle $1 \leq k \leq n$ gilt $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i)$. ◀

Aufgabe 4.19. Beschreibe einen möglichst effizienten Algorithmus, der das Erdős-Gallai-Theorem nutzt, um zu entscheiden, ob eine Gradsequenz $\mathcal{D}$ graphisch ist. Analysiere dessen Laufzeit. ◀

Im Folgenden diskutieren wir kurz, weshalb das Erdős-Gallai-Theorem eine notwendige Bedingung formuliert (d. h., jede graphisch Sequenz erfüllt das Theorem). Die Gegenrichtung (d.h., jede erfüllende Gradsequenz ist graphisch) ist deutlich komplexer und wird hier nicht behandelt.

Die Anforderung, dass die Gradsumme gerade sein muss, folgt trivial aus dem Handshaking-Lemma: jede Kante hat zwei Endpunkte und erhöht somit die Gradsumme um zwei – eine Ausnahme wäre es lediglich, wenn wir Eigenschleifen nur als Grad 1 zählten; das ist aber egal, da einfache Graphen sowieso keine Schleifen besitzen.

Für die zweite Bedingung konzentrieren wir uns zunächst auf die Ungleichung für $k = n$. Diese lautet

$$\sum_{i=1}^{n} d_i \leq n(n-1) + 0 \tag{4.41}$$

Dies ist trivial richtig, da nur der vollständige Graph K_n mit $\mathcal{D} = (n-1, \ldots, n-1)$ die Ungleichung mit Gleichheit erfüllt. Für jeden anderen Graphen ist die linke Seite der Ungleichung echt kleiner.

Für $k < n$ beschreibt also der Term $k(k-1)$ auf der rechten Seite die Grade, die durch Kanten innerhalb der ersten k Knoten erklärt werden können. Existiert ein k mit $(\sum_{i=1}^{k} d_i) - k(k-1) = \Delta > 0$, müssen Δ Kanten die ersten k Knoten mit den verbleibenden $n-k$ verbinden. Es muss also $\sum_{i=k+1}^{k} d_i \geq \Delta$ gelten; tatsächlich muss die Grenze aber noch schärfer gewählt werden. Da wir keine Mehrfachkanten erlauben, darf jeder der $n-k$ verbleibenden Knoten zu jedem der k ersten Knoten höchstens einmal verbunden werden. Falls $d_i > k$ für ein $i > k$, können wir diesen Grad nicht vollständig nutzen – sondern nur k „Endpunkte“ zählen. Daher fordert das Erdős-Gallai-Theorem $\Delta \leq \sum_{i=k+1}^{n} \min(k, d_i)$.

4.3.2 Havel-Hakimi-Theorem

Theorem 4.20. Sei $\mathcal{D} = (s, t_1, \ldots, t_s, d_{s+1}, \ldots, d_n)$ eine nicht wachsende Gradsequenz auf n Knoten, d. h.

$$n > s \geq t_1 \geq \ldots t_s \geq d_{s+1} \geq \ldots \geq d_n \geq 1. \tag{4.42}$$

Dann definiere $\mathcal{D}' = \text{sort}((t_1-1, \ldots, t_s-1, d_{s+1}, \ldots, d_n))$ als die sortierte Gradsequenz auf $n-1$ Knoten, die dadurch entsteht, dass wir den ersten (= größten) Eintrag entfernen und die verbleibenden s größten Einträge um eins reduzieren. Dann ist $\mathcal{D}$ genau dann graphisch, wenn $\mathcal{D}'$ graphisch ist. ◀

$\mathcal{D}'$ ist graphisch. $\Rightarrow$ $\mathcal{D}$ ist graphisch.

Beweis. Wir zeigen zunächst die Implikation „$\mathcal{D}'$ ist graphisch. $\Rightarrow$ $\mathcal{D}$ ist graphisch.“ Angenommen, $\mathcal{D}'$ ist graphisch, dann existiert ein einfacher Graph G', der $\mathcal{D}'$ erfüllt. Dann können wir einen neuen einfachen Graphen G erzeugen, in dem wir einen neuen Knoten u hinzufügen und ihn mit den Knoten mit Graden $t_1-1, \ldots, t_s-1$ verbinden. Der neue Knoten u hat offensichtlich Grad s und erhöht die Grade seiner Nachbarn auf $t_1, \ldots, t_s$. Daher erfüllt der erzeugte Graph G die Gradsequenz $\mathcal{D}$. Da jede neue Kante zu Knoten u inzident ist, können sie nicht mit alten Kanten in G' kollidieren. Weiterhin

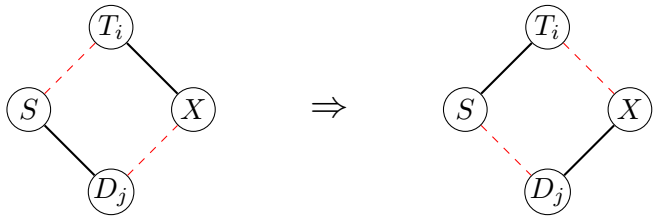


Abbildung 4.3: Edge Switch im induzierten Teilgraph auf Knoten S, T_i, D_j und X (rot gestrichelt zeigt an, dass hier keine Kante existiert). Beachte, dass sich die Grade der Knoten nicht ändern.

sind die neuen Kanten zu paarweise verschiedenen existierenden Knoten inzident – auch hier können sich weder Schleifen noch Mehrfachkanten bilden.

$\mathcal{D}$ ist graphisch. $\Rightarrow$ $\mathcal{D}'$ ist graphisch.

Nun zur Gegenrichtung „$\mathcal{D}$ ist graphisch. $\Rightarrow$ $\mathcal{D}'$ ist graphisch.“ Angenommen $\mathcal{D} = (s, t_1, \ldots, t_s, d_{s+1}, \ldots, d_n)$ ist graphisch, dann existiert ein einfacher Graph G, dessen Knoten wir $S, T_1, \ldots, T_s, D_{s+1}, \ldots, D_n$ (anhand des offensichtlichen Mappings) nennen. Wenn S zu $T_1, \ldots, T_s$ verbunden ist, sind wir fast fertig: Wir entfernen S und alle inzidenten Kanten und erhalten den Graphen G', der $\mathcal{D}'$ erfüllt.

Im Allgemeinen ist dies aber nicht richtig, d. h., es existiert ein T_i, das *nicht* adjazent zu S ist. Wir konstruieren nun aus G einen einfachen Graphen G_i, in dem die Kante $\{S, T_i\}$ existiert, ohne zuvor ggf. existierende Kanten $\{S, T_j\}$ zu verändern.

Fixiere also ein i, s. d. die Kante $\{S, T_i\}$ nicht existiert, und ein j, s. d. die Kante $\{S, D_j\}$ existiert. Aufgrund der Sortierung von $\mathcal{D}$ gilt $t_i \geq d_j$:

- Falls $t_i = d_j$, können wir einfach die Knoten T_i und D_j tauschen und sind fertig.
- Falls $t_i > d_j$, hat T_i echt mehr Nachbarn als D_j; daher existiert ein Nachbar X, der mit T_i, aber nicht mit D_j verbunden ist. Wie in Abb. 4.3 gezeigt, führen wir einen sog. Edge Switch aus und ersetzen die Kanten $\{S, D_j\}$ und $\{T_i, X\}$ durch $\{S, T_i\}$ und $\{D_j, X\}$. Hierdurch findet keine Gradveränderung statt.

In beiden Fällen erhalten wir also einen Graphen, in dem nun S mit T_i verbunden ist. Durch Mehrfachanwendung dieser Konstruktion erhalten wir schlussendlich einen Graphen, in dem S mit allen T_j verbunden ist, und können wie zuvor verfahren. □

Das Havel-Hakimi-Theorem lässt sich direkt in einen Generator übersetzen. Wir verwalten Knoten mit ihren Graden als Gewicht in einer Max-Priority-Queue. Bis diese leer ist, führen wir die folgende Schritte aus:

- Entnehme einen Knoten S mit größtem Grad s.
- Entnehme s schwerste Knoten T_i mit Grad t_i und speichere sie als (T_i, t_i) in einem Puffer.
- Gebe für alle i die Kanten $\{S, T_i\}$ aus.
- Füge für alle i, falls $t_i > 1$, Knoten T_i mit Gewicht $t_1 - 1$ erneut ein.

Mit klassischen Priority-Queues (z. B. Max-Heap) lässt sich hierdurch eine Laufzeit von $\mathcal{O}((m+n)\log n)$ erzielen; unter der Annahme, dass die Eingabe keine Knoten von Grad 0 erhält, vereinfacht sich der Term zu $\mathcal{O}(m \log n)$.

Da die Schlüssel in der PQ jedoch ganzzahlig sind und aus dem Intervall $[1, n)$ stammen, können wir eine Bucket-Queue verwenden. So kann etwa $\mathcal{D}$ als doppelt verkettete Liste dargestellt werden. Wenn $\mathcal{D}$ sortiert ist, befinden sich alle Einträge mit gleichem Grad in einer zusammenhängenden Gruppe. Wir unterhalten dann eine zweite sortierte Liste, die jeweils auf das erste Element der Gruppe zeigt. Durch diese Konstruktion ist es möglich, alle benötigten Operationen in Zeit $\mathcal{O}(1)$ auszuführen – der Generator läuft also in optimaler Laufzeit $\mathcal{O}(m)$.

Leider sind Listen in praktischen Implementierungen aufgrund der zahlreichen Allokationen und mangelnden Lokalität fast immer ein Performance-Bottleneck. Wenn $\mathcal{D}$ sortiert ist und die Anzahl der *verschiedenen* Grade $N_{\mathcal{D}} \ll n$ deutlich kleiner als die Anzahl der Knoten ist, ergibt sich hier ein Optimierungspotential. Statt jeden Knoten einzeln zu speichern, verzichten wir auf die „lange“ lineare Liste und speichern stattdessen für jede Gradgruppe drei Informationen ab: den Grad, die kleinste Knoten-ID in der Gruppe, die Anzahl der Knoten in der Gruppe.

Dann brauchen wir noch einen kleinen Trick: Wenn wir Knoten T_i auswählen, beginnen wir am Ende einer Gruppe. Das führt dazu, dass $\mathcal{D}$ auch nach Updates monoton bleibt; außerdem kann man zeigen, dass sich während der Ausführung des Algorithmus die Länge der Liste höchstens verdoppelt [9].

Kapitel 5

Graphrandomisierung

Mit dem Havel-Hakimi-Generator können wir nun für jede graphische Sequenz einen Graphen als Zeugen für diese Sequenz erzeugen. Die generierten Graphen sind aber zum einen deterministisch, d. h., für eine fixierte Sequenz $\mathcal{D}$ wird immer derselbe Graph erzeugt, und zum anderen pathologisch (extrem viele Dreiecke, extrem hohe Grad-Assortativität …). Das Fixed-Degree-Sequence-Model (FDSM) schlägt daher vor, dass wir erst den Havel-Hakimi-Algorithmus ausführen und dann den Graphen durch viele zufällige kleine Änderungen lokal perturbieren. Es besteht die begründete Hoffnung, dass wenn der hierdurch implizierte Irrweg lang genug ist, wir dann schlussendlich einen (fast) uniformen Graph aus $\mathcal{G}(\mathcal{D})$ produzieren.

Fixed-Degree-Sequence-Model (FDSM)

Formal beschreiben wir die Perturbation als einen Markov-Chain-Monte-Carlo-Prozess. Der Zustandsraum der Markov-Kette ist $\mathbb{G}(\mathcal{D})$. Für jeden Graphen $G \in \mathbb{G}(\mathcal{D})$ bestimmt der gewählte Prozess, welche anderen Graphen $G' \subseteq \mathbb{G}(\mathcal{D})$ (mit welcher Wahrscheinlichkeit) innerhalb von einem Schritt erreicht werden können. Es entsteht also ein Übergangsgraph (den wir auch Markov-Kette nennen), dessen gerichtete und gewichtete Kanten mögliche Schritte zusammenfassen.

Übergangsgraph

5.1 Edge-Switching

Edge-Switching ist wohl der meist genutzte Prozess zum Perturbieren von Graphen. Das ist sicherlich auch dem geschuldet, dass er sehr einfach zu implementieren ist und in der Praxis relativ gut funktioniert (mehr dazu später).

Anders als bei den meisten Modellen gibt es signifikante funktionale Unterschiede zwischen gerichteten und ungerichteten Graphen. Edge-Switching funktioniert aber mit noch ausgefalleneren Graphklassen (siehe z. B. [6]), z. B.

- ungerichtete, einfache Graphen mit fester Gradsequenz
- ungerichtete, einfache zusammenhängende Graphen mit fester Gradsequenz [22]
- gerichtete Graphen mit fester Gradsequenz
- gerichtete, einfache Graphen (mit einfacher Modifikation) mit fester Gradsequenz

- Graphen mit fester Joint-Degree-Sequence [21] (d. h., die Anzahl der Kanten zwischen Knoten mit Graden d_1 bzw. d_2 wird für alle Paare (d_1, d_2) definiert)
- bipartite Graphen
- weitere Klassen bei Modifikation möglich

allgemeine Edge-Switches

Das allgemeine Framework führt eine fixierte Anzahl an sog. *Edge-Switches* aus. Diese unterscheiden sich vor allem darin, welche Graphen als legal betrachtet werden:

1. Ziehe aus G zwei Kanten $e_1 = (a, b)$ und $e_2 = (u, v)$ zufällig mit Zurücklegen.
2. Falls $e_1 = e_2$: Verwerfe den Switch ersatzlos.
3. Berechne zwei neue Kanten $e_1' = (a, v)$ und $e_2' = (u, b)$.
4. Erzeuge Graph G' aus G durch Ersetzen von e_1 und e_2 durch e_1' und e_2'.
5. Wenn G' durch die Modifikation illegal wurde, verwerfe den Switch ersatzlos.

ungerichtete Switches

In ungerichteten Graphen entscheiden wir durch fairen Münzwurf, ob wir $e_1 = \{a, b\}$ und $e_2 = \{u, v\}$ durch

$$e_1' = \{a, v\} \qquad e_2' = \{b, u\}\,, \tag{5.1}$$

oder

$$e_1' = \{a, u\} \qquad e_2' = \{b, v\}\,, \tag{5.2}$$

zu ersetzen versuchen.

5.2 Edge-Switching auf unbeschränkten Matrizen

Bevor wir analysieren, ob Edge-Switching eine uniforme Verteilung erzeugt, sollten wir uns eine viel grundlegendere Frage stellen: Können wir von einem beliebigen Startpunkt aus überhaupt jeden anderen Graphen erzeugen? Formal fragen wir also, ob der Übergangsgraph stark zusammenhängend ist (analog: Ist die Markov-Kette irreduzibel?). Die Antwort wird ein klares und unmissverständliches „It depends …“ sein.

Es ist hilfreich, über die Adjazenzmatrizen zu argumentieren. Beobachte, dass die Sequenzen der Zeilen- und Spaltensummen der Adjazenzmatrizen von allen Graphen in $\mathbb{G}(\mathcal{D})$ identisch sind. Dies motiviert folgende Abstraktion:

$\mathbb{M}(Z, S)$: Binärmatrizen mit fixierten Zeilen- und Spaltensummen

Seien $Z = (z_1, \ldots, z_n)$ und $S = (s_1, \ldots, s_m)$ Zeilen- und Spaltensummen. Dann sei $\mathbb{M}(Z, S)$ die Menge aller $\{0, 1\}^{n \times m}$ Matrizen, die diese Beschränkungen erfüllen. Konkrete Graphklassen stellen i. d. R. weitere Anforderungen; so müssen ungerichtete einfache Graphen noch erfüllen:

- symmetrische Matrizen und damit $Z = S$

strukturelle Nullen: verbotene Kanten

- nur 0-Einträge auf der Hauptdiagonalen (wir sprechen von *strukturellen Nullen*)

Für den Moment betrachten wir aber solche Einschränkungen nicht.

Aufgabe 5.1. Welche Graphklasse entspricht $\mathbb{M}(Z, S)$ am ehesten? ◀

Einen gerichteten Edge-Switch können wir als sog. *alternierendes Rechteck* modellieren. Für eine Matrix $(m_{ij})_{ij} = M \in \mathbb{M}(Z, S)$ nennen wir $((i_1, j_1), (i_2, j_2))$ alternierendes Rechteck, wenn $m_{i_1 j_1} = m_{i_2 j_2} = 1$ und $m_{i_1 j_2} = m_{i_2 j_1} = 0$; jeweils gegenüberliegende Ecken haben also die gleichen Werte in M und deren „Flip" entspricht dem Effekt eines akzeptierten Edge-Switches.

Theorem 5.2 (nach [17]). Seien A und B zwei Matrizen in $\mathbb{M}(Z, S)$. Sei t das Minimum der Anzahl von Nullen in A und der Anzahl von Einsen in A. Dann können wir A in B mittels einer Folge von höchstens t alternierenden Rechteck-Switches überführen. ◀

Beweis. OBdA sei $(a_{ij})_{ij} = A \neq B = (b_{ij})_{ij}$ (da wir sonst fertig sind). Dann sei d die Hamming-Distanz zwischen A und B, d. h. die Anzahl der Einträge, in denen sich beide unterscheiden. Beobachte, dass d eine gerade positive Zahl sein muss (weshalb?).

Dann existiert ein Index (i_1, j_1) mit $a_{i_1 j_1} \neq b_{i_1 j_1}$ und sei oBdA $a_{i_1 j_1} = 1$. Da die Summe der j_1-ten Spalte fixiert ist, muss es in dieser eine Zeile i_2 geben, s. d. $a_{i_2 j_1} = 0$ und $b_{i_2 j_1} = 1$. Da wiederum die Summe der i_2-ten Zeile fixiert ist, muss es ein j_2 geben, s. d. $a_{i_2 j_2} = 1$ und $b_{i_2 j_2} = 0$. Wir haben also drei Ecken gefunden.

Könnten wir $a_{i_1 j_2} = 0$ garantieren, hätten wir ein alternierendes Rechteck und wären fertig; im Allgemeinen gilt das aber nicht. Daher setzen wir diesen Prozess so lange fort, bis wir einen sog. alternierenden Kreis schließen. Das muss irgendwann passieren, da wir nur $d < \infty$ passende Einträge haben. Sei $i_1 j_1, i_2 j_1, i_2 j_2, i_3 j_2, \ldots, i_k j_k, i_1 j_k$ ein *kürzester* alternierender Pfad. Wir zeigen, dass die ersten vier Einträge sich immer für einen Switch eignen:

- Wenn $a_{i_1 j_2} = 0$, bilden $i_1 j_1$, $i_2 j_1$, $i_2 j_2$ und $i_1 j_2$ ein alternierendes Rechteck.
- Wenn $b_{i_1 j_2} = 1$, können wir denselben Switch in B ausführen.
- Der Fall $a_{i_1 j_2} = 1$ und $b_{i_1 j_2} = 0$ ist ein Widerspruch dazu, dass der betrachtete Kreis minimale Länge hat, da wir die ersten drei Einträge entfernen können.

In jedem Fall können wir also in A oder B einen Switch ausführen, der die Hamming-Distanz zwischen den beiden Matrizen um mindestens zwei reduziert. Da jeder Switch reversibel ist, folgt der starke Zusammenhang der Übergangsmatrix direkt.

Weiter wissen sogar, dass A und B mittels höchstens $d/2$ Switches ineinander überführt werden können. Aufgrund der Symmetrie,

$$d/2 = |\{(i, j) \mid i, j, \text{s. d. } a_{ij} = 1 \wedge b_{ij} = 0\}| \tag{5.3}$$

$$= |\{(i, j) \mid i, j, \text{s. d. } a_{ij} = 0 \wedge b_{ij} = 1\}|, \tag{5.4}$$

folgt direkt

$$d/2 \leq t = \min\Big(|\{(i, j) \mid i, j, \text{s. d. } a_{ij} = 1\}|, \tag{5.5}$$

$$|\{(i, j) \mid i, j, \text{s. d. } a_{ij} = 0\}|\Big). \tag{5.6}$$

□

Wir können also Matrizen in $\mathbb{M}(Z, S)$ ineinander überführen. Das impliziert auch, dass der Übergangsgraph von gerichteten Graphen ohne Mehrfachkanten (aber ggf. mit Eigenschleifen) stark zusammenhängend ist.

Leider gilt das nicht für gerichtete *einfache* Graphen. Als Gegenbeispiel dient ein Graph mit Knoten $V = \{a, b, c\}$ und Kreis $a \to b \to c \to a$. Es ist nicht möglich, die Orientierung dieses Dreiecks auf $a \leftarrow b \leftarrow c \leftarrow a$ umzukehren, obwohl beide Graphen eine identische Gradsequenz aufweisen.

Aufgabe 5.3. Konstruiere eine Sequenz von Edge-Switches, die das Dreieck umkehren kann, und Eigenschleifen verwendet. ◀

Glücklicherweise sind Dreiecke das einzige Problem von Edge-Switching auf einfachen gerichteten Graphen. Es gibt zwei einfache Lösungsansätze:

- Führe neue Switchings, die die Orientierung eines gerichteten 3-Kreises ändern, ein.
- Bevor der MCMC-Prozess startet, führen wir einen Vorverarbeitungsschritt aus, der alle gerichteten 3-Kreise sucht und zufällig orientiert.

Für einfache *un*gerichtete Graphen ist dies alles nicht notwendig – die entsprechende Markov-Kette ist stark zusammenhängend.

5.3 Grenzverteilung von Edge-Switching

Wir wissen nun, dass Edge-Switching auf wichtigen Graphklassen potentiell alle Graphen erzeugen kann. In diesem Kapitel werden wir zeigen, dass für sehr lange Irrwege die Wahrscheinlichkeitsverteilung der Ausgabe zu einer Gleichverteilung über alle möglichen Graphen konvergiert; der Prozess hat also eine uniforme Grenzverteilung. Da wir in der Praxis jedoch immer nur endlich lange Irrwege ausführen können, wird der Edge-Switching-MCMC-Algorithmus oft als *approximativ uniform* bezeichnet.

Zunächst wiederholen wir ein paar Definition aus den Anfangstagen des Informatikstudiums. Hierfür stellen wir eine Markov-Kette $\mathcal{M}$ oft als Tupel (G, P) mit gerichteten Graph $G = (V, E)$ dar, wobei P eine stochastische $|V| \times |V|$-Matrix ist (d. h., jede Zeilensumme ist 1) und $(i, j) \in E \Leftrightarrow P_{ij} > 0$. Gegeben einer Startverteilung $\pi \in [0, 1]^n$ (Zeilenvektor) entspricht das Vektormatrixprodukt πP der Verteilung nach einem Schritt und πP^k der Verteilung nach k Schritten.

Definition 5.4. Sei $\mathcal{M} = (G, P)$ eine Markov-Kette. Dann heißt die Verteilung $\mathcal{G}(\mathcal{M})$ Grenzverteilung der Kette $\mathcal{M}$, wenn für alle Startverteilungen π gilt:

$$\lim_{k \to \infty} \pi P^k = \mathcal{G}(\mathcal{M}) \tag{5.7}$$

◀

Wenn eine Markov-Kette eine Grenzverteilung hat, ist es egal, wo wir starten: Irgendwann „vergessen“ wir die Startverteilung… Jetzt wäre noch wichtig, dass wir überall „raus“ kommen können. Solche Markov-Ketten nennen wir ergodisch:

Definition 5.5. Sei $\mathcal{M} = (G, P)$. Die Markov-Kette $\mathcal{M}$ heißt *ergodisch*, wenn $\mathcal{G}(\mathcal{M})$ existiert und die Grenzwahrscheinlichkeit $(\lim_{k\to\infty} P^k)_j > 0$ für alle j. ◀

Es kann nun gezeigt werden, dass eine Markov-Kette genau dann ergodisch ist, wenn sie (i) irreduzibel und (ii) aperiodisch ist. Ein Kette heißt *aperiodisch*, wenn für jeden Knoten v der größte gemeinsame Teiler der Längen aller Pfade von v nach v den Wert 1 hat.

Theorem 5.6. Edge-Switching impliziert für u. a. (i) einfache, ungerichtete und (ii) gerichtete Graphen mit möglichen Eigenschleifen eine ergodische Markov-Kette. ◀

Beweis. Wir zeigen den Beweis für gerichtete Graphen; er folgt analog für ungerichtete Graphen. Aus Abschnitt 5.2 wissen wir bereits, dass der Übegangsgraph von Edge-Switching stark zusammenhängend ist. Die Markov-Kette ist also irreduzibel.

Wir müssen also nur noch zeigen, dass sie auch aperiodisch ist. Hier wird ein vermeintlicher Bug zum Feature: Im Edge-Switching-Framework nutzen wir Rejection-Sampling und verwerfen illegale Switches ersatzlos. Die Behauptung ist nun, dass für jeden Zustand der Kette mindestens ein illegaler Switch existiert (es würde sogar reichen, wenn es eine einzige Eigenschleife gibt!):

Da wir ohne Zurücklegen ziehen, kann es passieren, dass $e_1 = e_2$; wir verwerfen also mindestens mit Wahrscheinlichkeit $1/m$. Dies entspricht einer Eigenschleife im Übergangsgraph. Angenommen, es existiert ein Pfad v nach v der Länge k (muss existieren, da die Kette irreduzible ist). Dann können wir durch Umwege über Eigenschleifen einen Pfad beliebiger Länge $k' > k$ erzeugen. Der ggT aller Pfade ist 1.

Die Markov-Kette ist also irreduzibel und aperiodisch und somit ergodisch. □

Das ist aufgrund des folgenden Lemmas hilfreich:

Lemma 5.7. Sei $\mathcal{M} = (G, P)$ eine Markov-Kette mit symmetrischer Übergangsmatrix $P_{ij} = P_{ji}$ für alle i, j. Falls die Grenzverteilung σ existiert, ist σ die Gleichverteilung. ◀

Beweis. Angenommen, es existiert eine stationäre Verteilung. Dann gilt für $\sigma = \left(\frac{1}{n}, \ldots, \frac{1}{n}\right)$ und jeden Zustand j:

$$(\sigma P)_j \overset{\text{Vek.-Mat.-Mul.}}{=} \sum_{i=1}^{n} \sigma_i P_{ij} = \frac{1}{n} \sum_{i=1}^{n} P_{ij} \tag{5.8}$$

$$\overset{P \text{ symmetrisch}}{=} \frac{1}{n} \sum_{i=1}^{n} P_{ji} \tag{5.9}$$

$$\overset{P \text{ stochastisch}}{=} 1/n = \sigma_j \tag{5.10}$$

□

Auch hier wird wieder der Bug des Rejection-Samplings zum Feature. Jeder gerichtete Switch wird über genau die zwei betroffenen Kanten e_1 und e_2 definiert und somit mit Wahrscheinlichkeit $1/m^2$ gewählt. Durch wiederholte Wahl dieser beiden Kanten wird der Switch wieder rückgängig gemacht. Die Wahrscheinlichkeiten von Vorwärts- und

Rückwärtsrichtung sind also identisch und die Übergangsmatrix ist daher symmetrisch. Analoges gilt auch für ungerichtete Switches.

5.4 Implementieren von Edge-Switching

Die sequentielle Implementierungen von Edge-Switching für gerichtete und ungerichtete Graphen ist relativ ähnlich. Daher nehmen wir im Folgenden gerichtete Graphen an – für einen ungerichteten Graphen müssen wir nur beachten, dass $\{u, v\} = \{v, u\}$ ist und beide Kanten„richtungen“ diesselbe Kante repräsentieren. Üblicherweise repräsentieren wir beide Richtung durch eine kanonische Repräsentation (u, v), wobei wir z. B. durch $u \leq v$ Uneindeutigkeiten verhindern. Vor jedem Zugriff auf die im Folgenden beschriebenen Datenstrukturen würden wir also prüfen, ob die Invariante gilt, und anderenfalls die Kantenrichtung flippen.

Ein zufälliger Edge-Switch benötigt die folgenden Operationen:

- zufälliges Ziehen einer Kante $(u, v) \in E$ (um die aktiven Kanten zu finden)
- Prüfen, ob eine Kante (u, v) existiert (um Mehrfachkanten zu vermeiden)
- Löschen einer Kante $(u, v) \in E$ (um Updates auszuführen)
- Hinzufügen einer Kante (u, v) (um Updates auszuführen)

Die beiden letzten Operationen können auch in einer zusammengefasst werden.

5.4.1 Auf Adjazenzmatrizen

Adjazenzmatrizen können alle Operationen außer das Ziehen einer zufälligen Kante in konstanter Zeit ausführen. Das Ziehen von zufälligen Kanten können wir aber mit einer dedizierten Datenstruktur lösen – z. B. ein ungeordnetes Array A, in dem jede Kante genau einmal gespeichert wird. Bei der Implementierungen müssen wir uns aber bei Updates minimal geschickt anstellen. Für ein Update der Kante (u, v) benötigen wir den entsprechenden Index i in A mit $A[i] = (u, v)$; diesen müssen wir glücklicherweise nicht suchen, da wir ausnutzen können, dass wir nur Kanten ersetzen, die wir vorher gezogen haben. Wir merken uns den Index einfach … Mit diesen wenigen Tricks können wir jeden Edge-Switch in konstanter Zeit ausführen.

5.4.2 Auf Adjazenzlisten

Adjazenzlisten sind in vielerlei Hinsicht eine Herausforderung für Edge-Switching. Allerdings verwalten viele existierenden Softwarebibliotheken Graphen in Adjazenzlisten; daher kann es lohnenswert sein, direkt auf der nativen Graphrepräsentation zu arbeiten. Dies gilt vor allem dann, wenn wir während jedes Switches zusätzliche Eigenschaften prüfen wollen und z. B. eine Breitensuche ausführen möchten, um den Zusammenhang des Graphs zu prüfen.

Um in einer Adjazenzliste eine Kante uniform zufällig zu ziehen, bietet sich ein zweistufiges Experiment an. Zunächst wählen wir einen Knoten u mit Wahrscheinlichkeit

$\deg(u)/(2m)$ und dann einen Nachbarn von u uniform zufällig. Da Edge-Switching den Grad von Knoten nicht verändert, ist es sinnvoll, vorab in Zeit $\Theta(n)$ eine Alias-Tabelle zu konstruieren, um das gewichtete Ziehen zu beschleunigen.

Um schnelle Updates zu unterstützen, ist es in der Regel hilfreich, eine unsortierte Adjazenzliste vorzuhalten. Dann kosten aber Existenzanfragen für die Kante $\{u, v\}$ Laufzeit $\mathcal{O}(\min(\deg(u), \deg(v)))$. Alternativ könnten wir die Nachbarschaften sortiert halten, dann dauern aber Updates $\mathcal{O}(\deg(u) + \deg(v))$ Zeit. Gerade bei verzerrten Gradverteilungen (z. B. Power-Law) bietet sich daher die erste Variante an.

Es gibt aber noch einen weiteren Ansatz: So nutzt die Implementierung von [22] eine Adjazenzliste, bei denen hinreichend große Nachbarschaften als Hashsets ausgelegt sind. In diesem Setting können wir einen Edge-Switch mit einer erwartet konstanten Laufzeit ausführen.

5.4.3 Mit Hashsets

Wie wir bereits gesehen haben, sind Adjazenzmatrizen für Edge-Switching auf dichten Graphen eine gute Wahl (in diesem Fall können wir sogar auf die Kantenliste verzichten! Warum?) Für dünne Graphen kann aber der Speicherverbrauch von $\Theta(n^2)$ Bits problematisch werden. Wir können daher ein Hashset nutzen, das alle Kanten abspeichert (für ungerichtete Graphen in kanonischer gerichteter Form).

Im Vergleich zu unserer Diskussion von *Adjazenzmatrizen* können wir sogar gänzlich darauf verzichten, die Kantenliste zu samplen, wenn wir ein Hashset mit offener Adressierung nutzen. Da sich die Anzahl der Kanten während Edge-Switching nicht ändert, ist es einfach, einen konstanten Load-Factor des Hashsets zu garantieren. Dieser sollte so gewählt werden, dass nur ein konstanter Anteil der Zellen im Hashset leer ist; dann ziehen wir so lange uniform Zellen aus dem Hashset, bis wir eine Kante gefunden haben, und geben diese dann zurück. Durch den beschränkten Load-Factor ergibt sich eine konstante Akzeptanzwahrscheinlichkeit.

Gerade im Kontext von parallelen Algorithmen ist es hilfreich, auf dedizierte Kantenliste zu verzichten, da wir sonst sicherstellen müssen, dass die Updates in beiden Datenstrukturen von verschiedenen nebenläufigen Prozessen nicht zu Inkonsistenzen führen.

Kapitel 6

Zufällige Permutationen

Wir haben bereits mehrfach zufällige Permutationen benötigt; in der Tat müssen so häufig Sequenzen von Elementen zufällig angeordnet werden, dass fast jede allgemeine Programmiersprache über sog. Shuffle-Funktionen verfügt.

Algorithmus 14 : Fisher-Yates-Shuffle

Input : Eingabearray $A[1 \dots n]$
Output : Array A

```
1 for n ≥ i ≥ 2 do
2     j ← uniform aus [1 ... i]
3     Tausche A[i] und A[j]
4 Gebe A zurück
```

Für kleine Eingaben $A[1 \dots n]$ ist meist Fisher-Yates-Shuffle die beste Wahl. Wie in Alg. 14 dargestellt, partitioniert Fisher-Yates konzeptionell die Eingabe im Urnenpräfix und im Ergebnissuffix. In jeder Iteration wird genau ein Element $A[j]$ aus der Urne gewählt und dem Ergebnis an Position $A[i]$ hinzugefügt. Durch den Tausch von $A[i]$ und $A[j]$ stellen wir sicher, dass hierbei keine Elemente verloren gehen. Dieser Algorithmus hat eine Laufzeit von $\mathcal{O}(n)$ und benötigt nur $\mathcal{O}(\log n)$ Bits zusätzlichen Speicher (u. a. für i und j). Zur Erinnerung eine Definition:

Definition 6.1. Wir bezeichnen einen Algorithmus als *in-place*, wenn dieser auf einer Eingabe der Größe N nur $o(N)$ zusätzliche Speicherworte verwendet. ◀

Beim Fisher-Yates-Shuffle handelt es sich also um einen In-Place-Algorithmus.

6.1 Paralleles Fisher-Yates

Fisher-Yates ist auf den ersten Blick ein inhärent sequentielles Problem. In jedem Schritt wird ein Element aus der Urne entnommen, was den Zustand der Urne verändert. Daher ist das Ergebnis von Shun et al. [**?**] besonders spannend.

Sie zeigten, dass der Algorithmus fast ohne Modifikation in erwartet linearer Arbeit und $\mathcal{O}(\log n \log^* n)$ auf einer Priority-Write-CRCW implementiert werden kann. Die Idee hierbei ist einfach. Statt in jedem Schritt einen zufälligen Index zu ziehen, gehört

das Array $H[1..n]$ zur Eingabe. Dabei entspricht $H[i]$ dem Index, mit dem $A[i]$ getauscht werden soll; es übernimmt also die Rolle von j in Alg. 14.

Das Vorberechnen der Indizes erlaubt es uns, schon Abhängigkeiten zu erkennen, bevor Swaps durchgeführt werden. Weiter können wir die Swaps nebenläufig und out-of-order ausführen, wenn hierbei die erkannten Abhängigkeiten beachtet werden. Hierzu nutzen die Autoren ihr *Reserve-and-Commit*-Framework: Wir suchen für jeden Index j in A den ersten Swap, der dieses Element in der sequentiellen Ausführung verändern würde (d. h. das größte i mit $H[i] = j$). Diesen Swap können wir ausführen und aus dem Spiel nehmen.

Sollte es weitere Swaps geben, die ebenfalls dieses Element verarbeiten möchten, werden diese verzögert. In jedem Schritt wird also pro Abhängigkeitskette mindestens ein Swap verarbeitet. Die Haupterkenntnis der Autoren ist, dass Fisher-Yates einen Abhängigkeitswald impliziert, der mit hoher Wahrscheinlichkeit eine Tiefe von $\mathcal{O}(\log n)$ hat.

In der Praxis ist der Algorithmus leider nicht besonders effizient: Echte Maschinen sind in der Regel keine Priority-Write-CRCW – dies muss mittels atomaren Compare-and-Swap-Operationen implementiert werden. Diese Zugriffe sind überdies auch noch wahlfrei und verursachen für große Eingaben Cache-Misses. Letztlich ist auch der Speicherbedarf suboptimal:

1. Die Eingabe H benötigt mindestens $\Omega(n \log n)$ Bits.
2. Die Datenstruktur zum Finden der Kollisionen benötigt $\Omega(n \log P)$ Bits, wobei P die Anzahl an Prozessoren bezeichnet.

Der Algorithmus ist also nicht in-place und benötigt viele Prozessoren, um überhaupt einen Speed-up zu erzielen.

In einem weiteren Papier zeigten die Autoren, dass die Kollisionen mittels $o(n)$ Bits gefunden werden können. Unter der Annahme, dass H zur Eingabe gehört, handelt es sich also um einen In-Place-Algorithmus. Im Vergleich zu Fisher-Yates wird jedoch weiterhin mindestens linear viel Speicher benötigt – denn Fisher-Yates benötigt H nicht.

6.2 Paralleles Shuffling

Vorsicht: In diesem Abschnitt verwenden wir nullindizierte Sequenzen $[x_0, \ldots, x_{n-1}]$ und Prozessoren $\{p_0, \ldots, p_{P-1}\}$.

Ein einfacher und deshalb besonders praktischer paralleler Algorithmus basiert auf [19]. Der Aufsatz diskutiert u. a. einen Algorithmus zum Shufflen auf verteilten Computern – jeder Prozessor interagiert mit seiner Umwelt durch explizites Senden von Nachrichten. Der Algorithmus eignet sich mit geringen Modifikationen auch für parallele Maschinen mit gemeinsam genutztem Speicher und funktioniert auch im External-Memory-Modell.

Als Eingabe erhalten wir Elemente $X = [x_0, \ldots, x_{n-1}]$, die arbiträr auf P Prozessoren partitioniert sind, s. d. jeder Prozessor $\Theta(n/P)$ Elemente erhält. Diese sollen in eine uniform zufällige Permutation überführt werden. Die Ausgabereihenfolge ergibt sich

dadurch, dass wir die lokalen Sequenzen der Prozessoren in aufsteigender Reihenfolge konkatenieren – erst kommen alle Elemente, für die Prozessor 0 im Ergebnis zuständig ist, dann jene von Prozessor 1 usw. usf. Hierzu führt Prozessor ℓ das folgende Programm aus:

1. Sende jedes lokale Element x_i der Eingabe zu einem Prozessor, der unabhängig und uniform zufällig gewählt wurde.
2. Empfange alle Elemente $T = [t_0, \ldots, t_{k_\ell - 1}]$.
3. Permutiere T lokal z. B. mit Fisher-Yates.
4. Speichere T als ℓ-ten Teil der Ausgabe.

Die Korrektheit des Algorithmus folgt aus dem folgendem Theorem.

Theorem 6.2 (nach [19]). Jede mögliche Permutation wird mit Wahrscheinlichkeit $1/n!$ erzeugt. Der Algorithmus erzeugt also eine uniforme Permutation. ◀

Beweis wird nachgereicht.

Am längsten benötigt der Algorithmus für das zufällige Versenden der Elemente und das lokale Permutieren der empfangen Elemente. Die Laufzeit des ersten Schrittes ist linear in der größten Partition, die nach Voraussetzung auf $\Theta(n/P)$ beschränkt ist. Das lokale Permutieren benötigt dann Zeit $\max_\ell k_\ell$, d. h. linear in der Größe der größten empfangenen Partition.

Da die Prozessoren unabhängig uniform gewählt werden, handelt es sich also um ein klassisches Gedankenexperiment, bei dem n Bälle auf P Eimer verteilt werden sollen. Man kann daher mit einem Chernoff-Argument zeigen, dass für $\beta > 0$ gilt:

$$\mathbb{P}\left[\max_\ell k_i > \frac{n}{P} + \sqrt{2\frac{n}{p}(\beta \ln n + \ln P)} + \mathcal{O}(\ln n)\right] < n^{-\beta}. \tag{6.1}$$

Unter realistischen Annahmen ($n/P = \Omega(\ln n)$) folgt also eine Laufzeit von $\mathcal{O}(n/P)$ mit hoher Wahrscheinlichkeit.

Kapitel 7

Communities

Bereits sehr früh in der Vorlesung stellten wir fest, dass beobachtete Netze in der Regel dünn sind. Auf ein Netzwerk mit menschlichen Teilnehmern bezogen könnte man dies so interpretieren, dass nicht jede Person mit jeder anderen sprechen kann (würde man mit jedem Menschen weltweit genau eine Sekunde lang sprechen, wäre man damit mehr als 250 Jahre beschäftigt). Menschen neigen aber dazu, sich in Gruppen bzw. Gemeinschaften (engl. communities) zu organisieren; Beispiele sind die Familie, Freunde, Arbeitskollegen, Sportclubs, Lern- und Forschungsgruppen. Wir finden aber Communities auch in anderen Netzwerken (z. B. im WWW oder biochemischen Netzwerken).

Hierbei handelt es sich in der Regel um verhältnismäßig kleine Gruppen, die jedoch eng interagieren. Wir sehen aber bereits an den Beispielen, dass ein Mensch zu mehreren Gruppen gehören kann. Man spricht dann von überlappenden Communities. Überlappende Communities können auch hierarchisch angelegt sein: Eine Firma mag in verschiedene Standorte mit je verschiedenen Sparten mit je verschiedenen Teams organisiert sein. Der Einfachheit halber konzentrieren wir uns aber auf den nichtüberlappenden Fall, in dem jeder Teilnehmer einer Gruppe zugeordnet wird.

Leider gibt es selbst für diesen einfachen Fall keine scharfe und allgemeingültige Definition, was eine Community darstellt. Um sie dennoch mit Werkzeugen der Network Science greifen zu können, treffen wir zwei Annahmen (nach [3]):

H1) *Der betrachtete Graph enthält die Struktur des beobachteten Netzwerks.* Auch wenn dieser Punkt offensichtlich erscheint, müssen wir immer berücksichtigen, dass der betrachtete Graph eines beobachteten Netzwerks eine verlustbehaftete Modellierung darstellt. So ist es nicht unwahrscheinlich, dass wir in einem Freundschafts- oder Kommunikationsnetz Freundeskreise identifizieren können. Hingegen werden wir in einem Stammbaum im Allgemeinen keine Arbeitsgruppen finden.

H2) Eingangs argumentierten wir, dass eine Community einen starken internen Zusammenhalt hat. In Verbindung mit der vorherigen Annahme ergibt sich dann: *Eine Community hat – im Vergleich zum Gesamtgraphen – einen relativ hohen internen Grad.*

Beginnen wir mit einem Extrem von Annahme H2: Eine Clique hat immer den

maximalen internen Grad, daher liegt es nahe, jede maximale (d. h. nicht erweiterbare) Clique als eine Community aufzufassen. Im Kleinen funktioniert das gut: In sozialen Netzen etwa gibt es viele Dreiecke. Wenn Person A zwei Freunde B und C hat, dann ist es nicht unwahrscheinlich, dass auch B und C in Kontakt stehen. Das entstehende Dreieck ist eine 3-Clique.

Größere Gruppen müssen aber im Allgemeinen nicht vollständig verbunden sein; man stelle sich etwa eine Schulklasse im Kommunikationsnetz einer Schule vor. Nicht jeder Schüler wird ständig mit jedem anderem sprechen. Im Schnitt sollten aber – schon aufgrund der räumlichen Nähe über längere Zeiträume – innerhalb der Klasse mehr Verbindungen vorhanden sein als zu anderen Klassen oder gar Stufen.

Wir fassen daher den Begriff weiter und unterscheiden zwischen starken und schwachen Communities (nach [3]). Sei $C \subseteq V$ eine Community, die wir als Knotenteilmenge modellieren. Dann sei $\deg_C^{\text{int}}(v)$ die Anzahl der Nachbarn von v in C und $\deg_C^{\text{ext}}(v)$ die Anzahl der Nachbarn außerhalb von C. Es gilt also

$$\deg_C^{\text{int}}(v) + \deg_C^{\text{ext}}(v) = \deg(v) \qquad \forall v \in V \tag{7.1}$$

- Wir nennen C eine *starke Community*, falls jeder Knoten in C strikt mehr interne als externe Nachbarn hat, d. h.

$$\deg_C^{\text{int}}(v) > \deg_C^{\text{ext}}(v) \qquad \forall v \in C. \tag{7.2}$$

- Wir nennen C eine *schwache Community*, falls der gesamte interne Grad die Anzahl der externen Nachbarn übersteigt, d. h.

$$\sum_{v \in C} \deg_C^{\text{int}}(v) > \sum_{v \in C} \deg_C^{\text{ext}}(v). \tag{7.3}$$

7.1 Erkennen von Communities

Das Erkennen von Communities ist eine wichtige Aufgabe in Network Science, die zwei verwandte Algorithmenklassen umfasst: Community-Detection-Algorithmen und Graph-Partitioning-Algorithmen. Beide Typen haben gemein, dass sie einen Graphen in möglichst zusammengehörige Untergruppen teilen sollen.

Der Unterschied ist in der Anwendung und Eingabe. Ein Partitionierungsalgorithmus wird z. B. verwendet, wenn ein großer Graph auf mehrere Prozessoren verteilt werden soll. Wir wissen, dass wir P Teile brauchen, und die Aufgabe ist es z. B., die Anzahl der Kanten zwischen den Partitionsklassen zu minieren; ggf. werden weitere Einschränkungen gesetzt, z. B. dass die Teile ungefähr gleich groß sind (z. B. minimal balanced edge cuts). Oft handelt es sich also um Unterroutinen, um eine Berechnung zu „vereinfachen“, und ist damit besonders im Algorithmen-Design zu finden.

Community-Detection-Algorithmen hingegen müssen selbst „herausfinden“, wie viele Communities vorhanden sind – die resultierende Zerlegung ist also vor allem durch die Eingabe bestimmt. Im Kontext von Network Science sind also Community-Detection-Algorithmen interessant.

Leider sollte es wenig überraschen, dass die meisten Probleme in Partitionierung und Community-Detection $\mathcal{NP}$-schwer sind. Konsequenterweise steigt auch die Anzahl der möglichen Partition schon bei Bipartitionen in $\Omega(2^{|V|})$.

Wir müssen daher in der Regel Approximationsalgorithmen nutzen, um Communities zu finden. Um einen iterativen Optimierungsalgorithmus zu konstruieren, benötigen wir ein Qualitätsmaß für eine Graphpartitionierung. Hierbei ist die sog. Modularity eine gängige Wahl.

7.2 Modularity

Sei $A = (a_{ij})_{ij}$ die Adjazenzmatrix eines ungerichteten Graphen $G = (V, E)$. Sei $C = (c_1, \ldots, c_n) \in [n_c]^n$ eine Auflistung der Communities $1, \ldots, n_c$, zu denen die Knoten jeweils gehören. Dann ist eine übliche Definition der Modularity Q des Graphen

$$Q = \frac{1}{2m} \sum_{i,j \in V} \left(a_{ij} - \frac{\deg(i)\deg(j)}{2m} \right) \delta(c_i, c_j), \tag{7.4}$$

mit Kronecker-Delta $\delta(x, y) = 1$, falls $x = y$, und $\delta(x, y) = 0$, falls $x \neq y$. Die Modularity kann einen Wertebereich von $[-1/2, 1]$ einnehmen. Wir nehmen an, dass eine gute Partitionierung des Graphen einen hohen Modularity-Wert annimmt.

Aufgabe 7.1. Drücke Q unter Verwendung von $\deg_C^{\text{int}}(v)$ und $\deg_C^{\text{ext}}(v)$ aus. ◀

Versuchen wir zunächst, die Aussage von Q intuitiv zu verstehen. Der erste Term, d. h. $\sum_{i,j} a_{ij}\delta(c_i, c_j)$, zählt die Kanten innerhalb und kann nur über das Kronecker-Delta beeinflusst werden. Er wird maximal, falls alle Knoten in einer Community sind – das ist nicht sonderlich nützlich.

Der Beitrag des zweiten Terms, d. h. $-\sum_{i,j} \deg(i)\deg(j)\delta(c_i, c_j)/2m$, ist nichtpositiv. Wir maximieren ihn, indem wir ihn möglichst nahe an 0 bringen. Das geschieht, indem wir viele Communities erzeugen, die alle einen möglichst geringen internen Gesamtgrad haben. Das Optimieren der Modularity ist also immer ein Kompromiss dieser beiden Größen.

7.2.1 Ein Greedy-Algorithmus

Das Finden von Community-Strukturen mit optimaler Modularity ist – natürlich – $\mathcal{NP}$-schwer (und trivialerweise in $\mathcal{NP}$). [?] Daher müssen wir für praktische Netzwerke eine Approximation nutzen, z. B. folgenden Greedy-Algorithmus:

1. Weise jedem Knoten seine eigene Community zu.
2. Suche unter allen Paaren von mit mindestens einer Kante verbundenen Communities ein Paar, dessen Verschmelzung Q am meisten erhöht.
3. Verschmilz die beiden Communities und wiederhole die Schritte 2 und 3, bis nur noch eine Community verbleibt.

4. Gebe unter allen erzeugten Zwischenergebnisse das Clustering mit maximaler Modularity aus.

Diese Art der Modularity-Maximierung leidet unter dem sog. *resolution limit* (siehe [3] für Details). Stark vereinfacht besagt es, dass für zwei verbundene Knoten i und j mit $\deg(i)\deg(j) < 2m$ die erwartete Kantenanzahl zwischen ihnen kleiner als die tatsächliche ist. Die Modularity steigt daher zwangsweise durch deren Vernetzung. Eine Konsequenz hieraus ist, dass der Greedy-Algorithmus Communities mit einem internen Grad von weniger als $\sqrt{2m}$ verschmilzt, sobald sie mit einer Kante verbunden sind.

In vielen beobachteten Netzwerken gibt es jedoch deutlich kleinere Gruppen, die nicht auf diese Weise gefunden werden können. Das Problem kann z. B. mit dem Louvain-Algorithmus relativiert werden.

Literaturverzeichnis

[1] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences*, 97(21):11149–11152, 2000. URL: https://www.pnas.org/doi/abs/10.1073/pnas.200327197, arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.200327197, doi:10.1073/pnas.200327197.

[2] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[3] Albert-László Barabási. Network science book. *Network Science*, 625, 2014.

[4] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[5] Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. In *ESA*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[6] Corrie Jacobien Carstens. Topology of complex networks: Models and analysis. *Bulletin of the Australian Mathematical Society*, 95(2):347–349, 2017. doi:10.1017/S0004972716001003.

[7] Paul Erdős, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[8] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[9] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/o-efficient generation of massive graphs following the *LFR* benchmark. *ACM J. Exp. Algorithmics*, 23, 2018.

[10] Lorenz Hübschle-Schneider and Peter Sanders. Parallel weighted random sampling. *ACM Trans. Math. Softw.*, 48(3):29:1–29:40, 2022.

[11] Michael Krivelevich and Benny Sudakov. The phase transition in random graphs: A simple proof. *Random Struct. Algorithms*, 43(2):131–138, 2013.

[12] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110, 2008.

[13] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019.

[14] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. Dynamic generation of discrete random variates. *Theory Comput. Syst.*, 36(4):329–358, 2003.

[15] Manuel Penschuck. *Scalable generation of random graphs.* Frankfurt [Hessen], 2020.

[16] Derek J. De Solla Price. Networks of scientific papers. *Science*, 149(3683):510–515, 1965. URL: http://www.jstor.org/stable/1716232.

[17] A. Ramachandra Rao, Rabindranath Jana, and Suraj Bandyopadhyay. A markov chain monte carlo method for generating random (0, 1)-matrices with given marginals. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 58(2):225–242, 1996. URL: http://www.jstor.org/stable/25051102.

[18] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: https://networkrepository.com.

[19] Peter Sanders and Christian Schulz. Scalable generation of scale-free graphs. *Information Processing Letters*, 116(7):489–491, 2016. doi:https://doi.org/10.1016/j.ipl.2016.02.004.

[20] Wolfgang Eugen Schlauch, Emoke-Ágnes Horvát, and Katharina Anna Zweig. Different flavors of randomness: comparing random graph models with fixed degree sequences. *Soc. Netw. Anal. Min.*, 5(1):36:1–36:14, 2015.

[21] Isabelle Stanton and Ali Pinar. Sampling graphs with a prescribed joint degree distribution using markov chains. In *ALENEX*, pages 151–163. SIAM, 2011.

[22] Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. *J. Complex Networks*, 4(1):15–37, 2016.