Algorithmen und Wahrscheinlichkeit

D-INFK, ETH Zürich

Angelika Steger
Emo Welzl

Stand: 11. März 2024

Inhaltsverzeichnis

Notationen

$\mathbb{N}$ die natürlichen Zahlen: $\mathbb{N} = \{1, 2, 3, \ldots\}$

$\mathbb{N}_0$ die natürlichen Zahlen und die Null: $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$

$[n]$ die natürlichen Zahlen von 1 bis n: $[n] = \{1, 2, \ldots, n\}$

$\mathbb{Z}$ die ganzen Zahlen: $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

$\mathbb{R}$ die reellen Zahlen

$\mathbb{R}^+$ die positiven reellen Zahlen

$\mathbb{R}_{\geq 0}$ die nicht-negativen reellen Zahlen

ln bezeichnet den natürlichen Logarithmus zur Basis $e = 2.71828\ldots$

log bezeichnet den Logarithmus zur Basis 2

$|A|$ bezeichnet die Kardinalität (Anzahl Elemente) einer Menge A

$A \times B$ das kartesische Produkt der Mengen A und B:
$A \times B = \{(a, b) \mid a \in A, b \in B\}$

2^A die Potenzmenge von A: $2^A = \{X \mid X \subseteq A\}$

$\binom{A}{k}$ die Menge der k-elementigen Teilmengen von A:
$\binom{A}{k} = \{X \mid X \subseteq A, |X| = k\}$

$G = (V, E)$ ein (ungerichteter) Graph mit Knotenmenge V und Kantenmenge $E \subseteq \binom{V}{2}$

$D = (V, A)$ ein gerichteter Graph mit Knotenmenge V und Kantenmenge $A \subseteq V \times V$

$\ldots \stackrel{!}{=} \min$ der Ausdruck auf der linken Seite soll minimiert werden

Kapitel 1

Graphentheorie

In der Vorlesung *Algorithmen und Datenstrukturen* des vergangenen Semesters haben wir bereits einige Graphalgorithmen kennen gelernt. In diesem Kapitel fassen diese nochmals zusammen und fügen einige weitere hinzu. Zunächst wiederholen wir die Sprache der Graphentheorie, die es uns oft erlaubt, komplizierte Zusammenhänge elegant auszudrücken.

1.1 Grundbegriffe & Notationen

Ein *Graph* G ist ein Tupel (V, E), wobei V eine endliche, nichtleere Menge von *Knoten* (engl. *vertices*) ist. Die Menge E ist eine Teilmenge der zweielementigen Teilmengen von V, also $E \subseteq \binom{V}{2} := \{\{x, y\} \mid x, y \in V, x \neq y\}$. Die Elemente der Menge E bezeichnet man als *Kanten* (engl. *edges*).

Einen Graphen kann man anschaulich sehr schön darstellen indem man jeden Knoten des Graphen durch einen Punkt (oder Kreis) repräsentiert und diese genau dann durch eine Linie verbindet, wenn im Graphen die entsprechenden Knoten durch eine Kante verbunden sind. Abbildung 1.1 illustriert dies an einigen Beispielen.

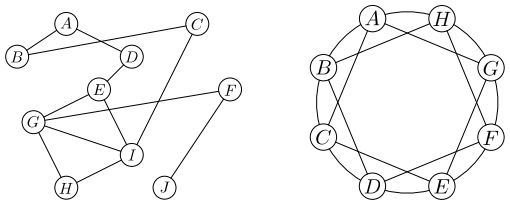


Abbildung 1.1: Zwei Beispiele für Graphen.

Wenn wir vor allem auf die Struktur des Graphen, aber nicht so sehr auf die Bezeichnung der Knoten eingehen wollen, so lassen wir zuweilen der Übersichtlichkeit halber die Bezeichnung der Knoten weg. Abbildung 1.2 zeigt einige Beispiele hierfür: ein *vollständiger Graph* (engl. *complete graph*) K_n besteht aus n Knoten, die alle paarweise miteinander verbunden sind. Ein *Kreis* (engl. *cycle*) C_n besteht aus n Knoten, die zyklisch miteinander verbunden sind. Ein *Pfad* (engl. *path*) P_n entsteht aus einem Kreis auf n Knoten, in dem wir eine beliebige Kante weglassen. Der d-*dimensionale Hyperwürfel* Q_d hat die Knotenmenge $\{0,1\}^d$, also die Menge aller Sequenzen von d Nullen und Einsen, wobei zwei Knoten $(a_1,\ldots,a_d)$ und $(b_1,\ldots,b_d)$ genau dann durch eine Kante verbunden werden, wenn es eine und nur eine Koordinate i gibt, für die $a_i \neq b_i$ gilt. Man überprüft leicht, dass Q_3 das „Skelett“ eines herkömmlichen (3-dimensionalen) Würfels ist.

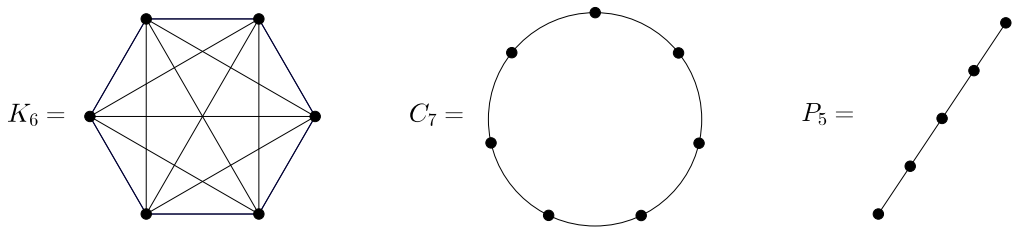


Abbildung 1.2: Der vollständige Graph K_6, der Kreis C_7 und der Pfad P_5.

Ein Graph heisst *bipartit* (engl. *bipartite*), wenn sich die Knotenmenge in zwei disjunkte Mengen A und B zerlegen lässt (wir verwenden die Notation $V = A \uplus B$ dafür), sodass Kanten von G nur zwischen A und B verlaufen. Jede Kante $e \in E$ muss also einen Endpunkt in A und einen Endpunkt in B haben. Zum Beispiel sind Hyperwürfel und Pfade bipartit, und Kreise sind genau dann bipartit, wenn sie gerade Länge haben.

Zuweilen betrachtet man auch eine etwas allgemeinere Form von Graphen und erlaubt sogenannte *Schlingen* (engl. *loops*), d.h. Kanten, die einen Knoten mit sich selbst verbinden. Ausserdem können Graphen mit *Mehrfachkanten* (engl. *multiple edges*) betrachtet werden, bei denen ein Knotenpaar durch mehr als eine Kante verbunden sein kann. Graphen, in denen auch Schleifen und Mehrfachkanten vorkommen können, nennt man auch *Multigraphen*. Wir werden solche in dieser Vorlesung jedoch, von wenigen Ausnahmen abgesehen, auf die wir dann besonders hinweisen, nicht

betrachten. Wenn wir daher von einem Graphen $G = (V, E)$ sprechen meinen wir immer einen Graphen ohne Schleifen und ohne Multikanten.

Für einen Knoten $v \in V$ eines Graphen $G = (V, E)$ definieren wir die *Nachbarschaft* (engl. *neighbourhood*) $N_G(v)$ eines Knotens $v \in V$ durch

$$N_G(v) := \{u \in V \mid \{v, u\} \in E\}.$$

Der *Grad* (engl. *degree*) von v bezeichnet die Größe der Nachbarschaft von v, also $\deg_G(v) := |N_G(v)|$. Die Schreibweisen „$N_G(v) := \ldots$“ und „$\deg_G(v) := \ldots$“ verdeutlichen, dass wir die Nachbarschaft oder den Grad des Knotens im Graphen G meinen. Wenn sich der Graph aus dem Zusammenhang eindeutig ergibt, lassen wir den Index meist weg und schreiben einfach $\deg(v)$ für den Grad von v bzw. $N(v)$ für die Nachbarschaft von v.

Ein Graph G heisst k-*regulär* (engl. k-*regular*), falls für alle Knoten $v \in V$ gilt, dass $\deg(v) = k$.

Beispiel 1.1. Der vollständige Graph K_n ist $(n-1)$-regulär, die Kreise C_n sind jeweils 2-regulär, und der Hyperwürfel Q_d ist d-regulär.

Zwei Knoten u und v heissen *adjazent* (engl. *adjacent*), wenn sie durch eine Kante verbunden sind. Die Knoten u und v nennt man dann auch die *Endknoten* der Kante $\{u, v\}$. Ein Knoten u und eine Kante e heissen *inzident* (engl. *incident*), wenn u einer der Endknoten von e ist.

Der folgende Satz stellt eine einfache, aber wichtige Beziehung zwischen den Knotengraden und der Gesamtanzahl der Kanten eines Graphen auf.

Satz 1.2. Für jeden Graphen $G = (V, E)$ gilt $\sum_{v \in V} \deg(v) = 2|E|$.

Beweis. Wir verwenden die Regel des doppelten Abzählens. Auf der linken Seite wird jede Kante $\{u, v\}$ genau zweimal gezählt: einmal, wenn $\deg(u)$ betrachtet wird und zum zweiten Mal, wenn $\deg(v)$ betrachtet wird. Auf der rechten Seite wird jede Kante ebenfalls genau zweimal gezählt. □

In k-regulären Graphen gilt $\deg(v) = k$ für alle Knoten v und daher $2|E| = |V|k$. Insbesondere kann es einen k-regulären Graphen auf n Knoten überhaupt nur dann geben, wenn das Produkt nk gerade ist. Allgemeiner lässt sich mit Hilfe des gerade bewiesenen Satzes leicht zeigen, dass in jedem beliebigen (auch nicht-regulären) Graphen die Anzahl der Knoten mit ungeradem Grad gerade sein muss.

Korollar 1.3. Für jeden Graphen $G = (V, E)$ gilt: Die Anzahl der Knoten mit ungeradem Grad ist gerade.

Beweis. Wir teilen die Knotenmenge V in zwei Teile, die Menge V_g der Knoten mit geradem Grad und die Menge V_u der Knoten mit ungeradem Grad. Die Summe von beliebig vielen geraden Zahlen ist immer gerade. Die Summe von k ungeraden Zahlen ist hingegen genau dann gerade, wenn k gerade ist. Also ist

$$\sum_{v \in V} \deg(v) = \sum_{v \in V_g} \deg(v) + \sum_{v \in V_u} \deg(v)$$

genau dann gerade, wenn $|V_u|$ dies ist. Nach Satz 1.2 muss obige Summe aber gerade sein, denn $2|E|$ ist ja immer gerade. Also ist $|V_u|$ gerade. □

In der Formel aus Satz 1.2 werden auf der linken Seite alle Knotengrade aufsummiert. Teilen wir daher beide Seiten durch die Anzahl $|V|$ aller Knoten, so erhalten wir links den durchschnittlichen Knotengrad und rechts $2|E|/|V|$. Da es bei der Durchschnittsbildung immer Werte gibt die höchstens so gross wie der Durchschnitt sind und solche die mindestens so gross sind, erhalten wir aus Satz 1.2 sofort auch die folgende Konsequenz.

Korollar 1.4. In jedem Graph $G = (V, E)$ ist der durchschnittliche Knotengrad gleich $2|E|/|V|$. Insbesondere gibt es daher Knoten $x, y \in V$ mit $\deg(x) \leq 2|E|/|V|$ und $\deg(y) \geq 2|E|/|V|$. □

Teilgraphen. Ein Graph $H = (V_H, E_H)$ heisst (schwacher) *Teilgraph* eines Graphen $G = (V_G, E_G)$, falls

$$V_H \subseteq V_G \quad \text{und} \quad E_H \subseteq E_G$$

gilt. Ist H ein Teilgraph von G, so schreiben wir auch $H \subseteq G$. Gilt sogar $E_H = E_G \cap \binom{V_H}{2}$, so nennen wir H einen *induzierten Teilgraphen* von G und schreiben $H = G[V_H]$. Abbildung 1.3 illustriert dies an einigen Beispielen. Einen Teilgraphen eines Graphen erhält man also, indem man aus dem ursprünglichen Graphen beliebig Kanten und/oder Knoten (und alle inzidenten Kanten) entfernt. Um einen induzierten Teilgraphen zu erhalten, muss

man etwas mehr aufpassen. Hier wird verlangt, dass je zwei Knoten, die im ursprünglichen Graphen verbunden waren und die beide im Teilgraph vorhanden sind, auch im induzierten Teilgraphen miteinander verbunden sind. Um einen induzierten Teilgraphen zu erhalten, darf man somit nur Knoten (und die mit ihnen verbundenen Kanten) aus dem ursprünglichen Graphen entfernen, man darf jedoch *nicht* eine Kante zwischen Knoten entfernen und beide Knoten im Graphen belassen.

Ist $G = (V, E)$ ein Graph und v ein Knoten von G, so bezeichnen wir mit $G - v := G[V \setminus \{v\}]$ den Teilgraphen, den wir erhalten indem wir v und alle zu v inzidenten Kanten löschen.

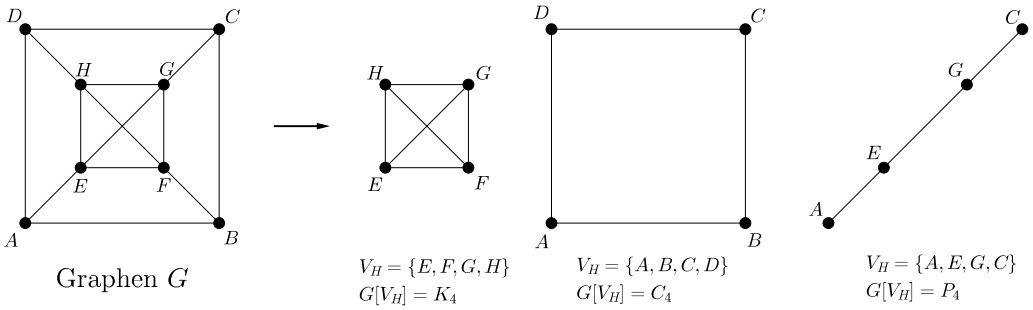


Abbildung 1.3: Der Graph G (links) und einige induzierte Teilgraphen des Graphen G (rechts).

1.1.1 Zusammenhang und Bäume

Eine Sequenz von Knoten $\langle v_1, \ldots, v_k \rangle$ heisst *Weg* (engl. *walk*), wenn für jedes $i \in \{1, \ldots, k-1\}$ eine Kante von v_i nach v_{i+1} existiert. Für eine Kante e und einen Weg $W = \langle v_1, \ldots, v_k \rangle$ schreiben wir $e \in W$ wenn es ein i gibt, so dass $e = \{v_i, v_{i+1}\}$ gilt. Die *Länge* eines Weges ist die Anzahl der Schritte, also in diesem Fall $k - 1$. Die Knoten v_1 und v_k nennt man auch den Start- bzw. Endknoten des Weges; einen Weg mit Startknoten s und Endknoten t nennen wir auch kurz einen s-t-Weg. Zuweilen, wenn es auf die Richtung nicht ankommt, bezeichnen wir auch beide Knoten als Endknoten des Weges. Ein *Pfad* von s nach t (oder auch kurz s-t-Pfad, engl. s-t-*path*) s-t-Weg, der keinen Knoten mehrfach benutzt. Ein Weg $\langle v_1, \ldots, v_k \rangle$ mit $v_1 = v_k$ heisst *Zyklus* (oder auch: geschlossener Weg, engl. *closed walk*). Ein Zyklus ist also ein Weg in dem Start- und Endknoten übereinstimmen.

Ein Zyklus $\langle v_1, \ldots, v_k \rangle$ heisst *Kreis* (engl. *cycle*), wenn er Länge mindestens 3 hat und die Knoten $v_1, \ldots, v_{k-1}$ alle paarweise verschieden sind.

Es zeigt sich leicht, dass zwei Knoten s und t genau dann durch einen Pfad verbunden sind, wenn sie durch einen Weg verbunden sind; in der Tat ist nämlich ein *kürzester* s-t-Weg (wenn denn einer existisiert) stets auch ein s-t-Pfad.

Wir nennen einen Graphen $G = (V, E)$ *zusammenhängend* (engl. *connected*) falls es für je zwei Knoten $s, t \in V$ einen s-t-Pfad in G gibt. Einen zusammenhängenden Teilgraphen $C \subseteq G$ der bezüglich dieser Eigenschaft maximal ist[1] nennt man eine *Zusammenhangskomponente*. Die Knotenmengen der verschiedenen Zusammenhangskomponenten von G sind genau die Äquivalenzklassen der Äquivalenzrelation

$$R := \{(s, t) \in V^2 \mid G \text{ enthält einen s-t-Pfad}\}.$$

von G; insbesondere bilden sie eine Partition der Knotenmenge V.

Ein Graph der keinen Kreis enthält, heisst *kreisfrei*. Ist ein Graph $G = (V, E)$ zusammenhängend und kreisfrei, so nennt man ihn *Baum* (engl. *tree*). Aber Achtung: diese Definition heisst nicht, dass er auch wie ein Baum aussieht. Zum Beispiel sind alle in Abbildung 1.4 dargestellten Graphen Bäume.

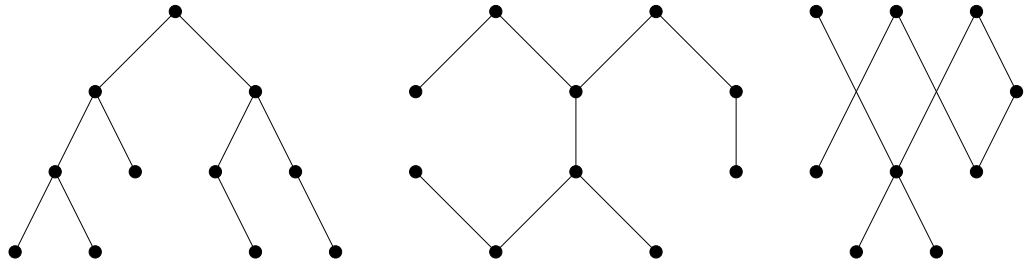


Abbildung 1.4: Einige Beispiele für Bäume.

Im vergangenen Semester haben Sie bereits *Suchbäume* als eine wichtige Datenstruktur für eine effiziente Organisation von Daten kennen gelernt. Suchbäume tragen das Wort 'Baum' im Namen und sie erfüllen beide Kriterien der obigen Definition: sie sind zusammenhängend und kreisfrei. Darüber hinaus haben sie aber noch eine weitere wichtige Eigenschaft: sie

[1] D.h. jeder Teilgraph $H \neq C$ mit $C \subseteq H \subseteq G$ ist nicht zusammenhängend.

besitzen eine *Wurzel* an der der Suchbaum quasi 'aufgehängt' ist. In der Sprache der Graphentheorie nennt man einen Suchbaum daher auch einen *gewurzelten Baum* (engl. *rooted tree*). In dieser Vorlesung folgen wir der in der Wissenschaft üblichen Sprechweise: in der Graphentheorie ist ein Baum *nicht* gewurzelt.

Für Bäume gilt die schöne und wichtige Eigenschaft, dass es für beliebige Knoten $x, y \in V$ genau einen x-y-Pfad gibt. Dass es mindestens einen solchen Pfad gibt folgt aus der Tatsache, dass ein Baum zusammenhängend ist, dass es nur höchstens einen solchen Pfad gibt folgt aus der Tatsache, dass ein Baum kreisfrei ist. Ist $T = (V, E)$ ein Baum und $v \in V$ ein Knoten mit Grad $\deg(v) = 1$, so heisst v ein *Blatt* (engl. *leaf*).

Lemma 1.5. Ist $T = (V, E)$ ein Baum mit $|V| \geq 2$ Knoten, so gilt:
a) T enthält mindestens zwei Blätter,
b) ist $v \in V$ ein Blatt, so ist der Graph $T - v$ ebenfalls ein Baum.

Beweis. Wir zeigen zunächst Eigenschaft a). Wähle eine beliebige Kante $e \in E$ und laufe von den beiden Knoten der Kante aus durch den Graphen „bis es nicht weiter geht". Da T ein Baum und daher kreisfrei ist, kann man beim Weiterlaufen nie auf einen Knoten stossen, der schon in dem Pfad enthalten ist (denn sonst hätten wir einen Kreis gefunden) und somit müssen beide Richtungen in je einem Blatt enden.

Für die Eigenschaft b) überlegen wir uns Folgendes: $T' := T - v$ ist sicherlich kreisfrei, da T kreisfrei war, und durch die Wegnahme von Knoten und Kanten keine Kreise entstehen können. Um einzusehen, dass T' auch zusammenhängend ist, betrachten wir zwei beliebige Knoten $x, y \in V \setminus \{v\}$ und zeigen, dass x und y in T' durch einen Pfad verbunden sind. Da T ein Baum ist, ist T sicherlich zusammenhängend. In T gibt es also einen Pfad P, der x und y verbindet. Da in einem Pfad alle inneren Knoten Grad zwei haben, kann v in diesem Pfad nicht enthalten sein. Der Pfad P ist somit auch im Graphen T' enthalten, was zu zeigen war. □

Es ist nicht schwer daraus auch noch einen weitere wichtige Eigenschaft eines Baumes herzuleiten: ein Baum mit Knotenmenge V enthält genau $|V| - 1$ Kanten. Der folgende Satz fasst die verschiedenen Eigenschaften von Bäumen zusammen.

Satz 1.6. Ist $G = (V, E)$ ein Graph auf $|V| \geq 1$ Knoten, so sind die folgenden Aussagen äquivalent:
(a) G ist ein Baum,
(b) G ist zusammenhängend und kreisfrei,
(c) G ist zusammenhängend und $|E| = |V| - 1$,
(d) G ist kreisfrei und $|E| = |V| - 1$,
(e) für alle $x, y \in V$ gilt: G enthält genau einen x-y-Pfad.

Beweis. Die folgenden Implikationen zeigen die gewünschte Äquivalenz.

(a) $\Leftrightarrow$ (b) Das gilt per Definition eines Baums.

(a) $\Rightarrow$ (c) Wir müssen zeigen, dass jeder Baum auf n Knoten genau $n-1$ Kanten besitzt. Dazu verwenden wir vollständige Induktion nach n. Für $n = 1$ ist die Aussage klar. Sei also $n \geq 2$. Wir nehmen an, dass (c) für alle Bäume mit $n-1$ Knoten gilt, und wollen (c) für alle Bäume mit n Knoten zeigen. Sei daher $T = (V, E)$ ein beliebiger Baum mit n Knoten. Nach Lemma 1.5 enthält T ein Blatt u, und der Graph $T - v = (V', E')$ ist ein Baum mit $|V'| = n-1$ Knoten. Da u ein Blatt war, besitzt T' ausserdem $|E'| = |E| - 1$ Kanten. Nun können wir auf T' die Induktionsvoraussetzung anwenden, und erhalten $|E| - 1 = |E'| \overset{\text{I.V.}}{=} |V'| - 1 = n - 2$. Daraus folgt $|E| = n - 1$, wie gewünscht.

(c) $\Rightarrow$ (d) Sei G ein zusammenhängender Graph mit n Knoten und $n-1$ Kanten. Zu zeigen ist, dass G kreisfrei ist. Angenommen, G enthielte einen Kreis $C = \langle u_1, u_2, \ldots, u_k, u_1 \rangle$. Betrachten wir den Graphen $G = (V, E \setminus \{u_1, u_k\})$, der aus G durch Löschen der Kante $\{u_1, u_k\}$ entsteht. Dieser ist immer noch zusammenhängend, denn für je zwei Knoten $v, w \in V$ gibt es (weil G zusammenhängend ist) einen Weg von v nach w in G, und falls dieser Weg über die Kante $\{u_1, u_k\}$ führt, kann man diese in G' durch den Weg $u_1, u_2, \ldots, u_k$ ersetzen. Wir können also bei Existenz eines Kreise eine Kante entfernen, sodass der Graph immer noch zusammenhängend bleibt. Dies wiederholen wir so lange, bis keine Kreise mehr übrig sind. Danach ist der entstehende Graph G' aber zusammenhängend und kreisfrei, also ein Baum. Wegen der schon bewiesenen Implikation (a) $\Rightarrow$ (c) hat G' genau $n-1$ Kanten. Dies steht im Widerspruch dazu, dass G auch $n-1$ Kanten

hatte, und wir mindestens eine Kante (im ersten Schritt) gelöscht haben. Also kann G keinen Kreis enthalten haben.

(d) ⇒ (a) Sei G ein kreisfreier Graph mit n Knoten und $n-1$ Kanten. Es ist zu zeigen, dass G zusammenhängend ist. Jede Zusammenhangskomponente von G ist zusammenhängend (nach Definition) und kreisfrei (da G kreisfrei ist), also ein Baum. Seien die Komponenten $C_1 = (V_1, E_1), \ldots, C_k = (V_k, E_k)$. Dann ist $V = \dot{\bigcup}_{i=1}^{k} V_i$ und $E = \dot{\bigcup}_{i=1}^{k} E_i$. Da die Komponenten Bäume sind, besagt die schon bewiesene Richtung (a) ⇒ (c) aber $|E_i| = |V_i| - 1$ für alle $i \in \{1, \ldots, k\}$. Daher gilt

$$|V| - 1 = |E| = \sum_{i=1}^{k} |E_i| = \sum_{i=1}^{k} (|V_i| - 1) = \left(\sum_{i=1}^{k} |V_i| \right) - k = |V| - k.$$

Vergleicht man die linke und die rechte Seite, so erhält man $k = 1$. Es gibt also genau eine Zusammenhangskomponente, und G ist zusammenhängend.

(a) ⇒ (e) Sei G ein Baum. Da G zusammenhängend ist, gibt es für jedes Paar von Knoten u, v *mindestens* einen Pfad von u nach v in G. Angenommen es gäbe für ein Paar u, v zwei verschiedene Pfade von u nach v, zum Beispiel $\langle u_1, u_2, \ldots, u_r \rangle$ und $\langle v_1, v_2, \ldots, v_s \rangle$, wobei $u_1 = v_1 = u$ und $u_r = v_s = v$, und wobei wir $s \leq r$ annehmen. Sei $i \in \{2, \ldots, s\}$ minimal mit $u_i \neq v_i$ (ein solches i existiert, weil die Pfade verschieden sind und im gleichen Knoten starten bzw. enden). Sei $j \in \{i+1, \ldots, r\}$ minimal mit $u_j \in \{v_{i+1}, \ldots, v_s\}$ und $k \in \{i+1, \ldots, r\}$ minimal mit $v_k = u_j$. Dann kann man sich leicht überlegen, dass $C := \langle u_{i-1}, u_i, \ldots, u_{j-1}, u_j, v_{k-1}, v_{k-2}, \ldots, v_{i+1}, v_i \rangle$ ein Kreis ist. Dies widerspricht der Kreisfreiheit von G. Daher gibt es *genau* einen u-v-Pfad für jedes Paar u, v.

(e) ⇒ (a) Da G für jedes Paar u, v einen Pfad von u nach v enthält, ist G zusammenhängend. Wir nehmen zum Widerspruch an, dass es einen Kreis $C = \langle u_1, \ldots, u_k, u_1 \rangle$ in G gäbe. Dann wären aber $P_1 := \langle u_1, u_2, \ldots, u_k \rangle$ und $P_2 := \langle u_1, u_k \rangle$ zwei verschiedene Pfade von u_1 nach u_k, denn C hat per Definition mindestens Länge 3. Dies ist ein Widerspruch zu der Eindeutigkeit des u_1-u_k-Pfades, also muss G auch kreisfrei gewesen sein. □

Bäume sind Graphen, die kreisfrei und zusammenhängend sind. Verzichtet man auf die Bedingung 'zusammenhängend' so nennt man die ent-

sprechenden Graphen Wälder. Ein *Wald* $W = (V, E)$ ist also ein Graph, der kreisfrei ist. Jede Zusammenhangskomponente eines Waldes ist dann natürlich ein Baum. Und die Anzahl der Zusammenhangskomponenten lässt sich unmittelbar aus der Kantenanzahl ablesen:

Lemma 1.7. Ein Wald $G = (V, E)$ enthält genau $|V| - |E|$ viele Zusammenhangskomponenten.

Beweis. Wie beweisen das Lemma durch Induktion über $|E|$. Wenn $E = \emptyset$ ist, dann ist jeder Knoten eine einzelne Zusammenhangskomponente und der Graph G besteht daher aus genau $|V|$ Komponenten. Die Aussage des Lemmas ist in diesem Fall also richtig. Nehmen wir für den Induktionsschritt daher an, dass die Aussage für einen Graphen $G' = (V, E')$ gilt und $e \notin E'$ eine Kante ist, so dass $G = (V, E' \cup \{e\})$ noch immer kreisfrei ist. Dann kann die Kante e nicht innerhalb einer Zusammenhangskomponente von G' verlaufen (denn dann würden wir einen Kreis erhalten), sie verbindet daher zwei verschiedene Komponenten – weshalb sich die Anzahl Zusammenhangskomponenten um eins reduziert. Die Behauptung des Lemmas gilt daher auch für den Graphen $G = (V, E' \cup \{e\})$, was zu zeigen war. □

1.1.2 Gerichtete Graphen

In Graphen sind Kanten durch zweielementige Mengen von Knoten gegeben. Insbesondere gilt daher (wie immer bei Mengen), dass $\{u, v\} = \{v, u\}$ gilt. In sogenannten *gerichteten Graphen* sind die Kanten zusätzlich noch orientiert, eine Kante wird also nicht mehr durch eine zweielementige Menge, sondern durch ein geordnetes Paar dargestellt. Dies führt zu folgender Definition: ein *gerichteter Graph* oder auch kurz *Digraph* (engl. *directed graph* oder auch kurz *digraph*) D ist ein Tupel (V, A), wobei V eine (endliche) Menge von Knoten ist und $A \subseteq V \times V$ eine Menge von gerichteten Kanten (engl. *arcs*).

Man beachte, dass der Ausdruck 'gerichteter Graph' grammatisch etwas irreführend ist, da er den Eindruck erweckt, ein gerichteter Graph sei eine spezielle Art von Graph. Das ist jedoch nicht so. Graphen und gerichtete Graphen sind verschiedene (wenn auch eng verwandte) Konzepte. Wenn wir besonders betonen wollen, dass wir das erste Konzept meinen,

dann sprechen wir auch von 'ungerichteten Graphen'. Allgemein ist aber mit 'Graph' immer 'ungerichteter Graph' gemeint. Man beachte auch den Unterschied in der Schreibweise von Kanten in ungerichteten und gerichteten Graphen. In einem ungerichteten Graphen ist eine Kante zwischen zwei Knoten u und v eine zweielementige Teilmenge von V und wir notieren sie daher in der üblichen Mengenschreibweise $\{u, v\}$. In gerichteten Graphen ist eine Kante hingegen ein Element des Kreuzproduktes $V \times V$. Entsprechend verwenden wir hier die Tupelschreibweise (u, v). Graphisch wird eine gerichtete Kante (u, v) als Pfeil von u nach v dargestellt.

Gemäss Definition sind in gerichteten Graphen formal auch Schleifen (x, x) zugelassen. Analog zu ungerichteten Graphen werden wir in diesem Skript jedoch davon ausgehen, dass ein gerichteter Graph weder Schleifen noch Mehrfachkanten enthält. Man beachte aber, dass es bei gerichteten Graphen dennoch zwischen zwei Knoten x und y mehr als eine Kante geben kann, nämlich (x, y) und (y, x). Diese gelten nicht als Mehrfachkanten, denn es sind ja verschieden orientierte Kanten.

Für Knoten v definieren wir den *Aus-Grad* durch

$$\deg^+(v) := |\{(x, y) \in A \mid x = v\}|$$

und den *In-Grad* durch

$$\deg^-(v) := |\{(x, y) \in A \mid y = v\}|.$$

Analog zu Satz 1.2 für ungerichtete Graphen zeigt man:

Satz 1.8. Für jeden gerichteten Graphen $D = (V, A)$ gilt:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |A| \ .$$

□

Die meisten Definitionen für Graphen lassen sich sinngemäss auch auf gerichtete Graphen übertragen. Wir werden im Folgenden nur einige davon näher betrachten.

Ein *gerichteter Weg* in einem gerichteten Graphen $D = (V, A)$ ist eine Folge $W = \langle v_1, \ldots, v_k \rangle$ von Knoten aus V, so dass $(v_i, v_{i+1}) \in A$ für alle $i = 1, \ldots, k-1$. Die *Länge* des gerichteten Weges entspricht wie im ungerichteten Fall der Anzahl der Schritte, ist also hier $k-1$. Die Knoten v_1 und v_k nennt man Start- bzw. Endknoten des gerichteten Weges. Ein

gerichteter Pfad ist ein gerichteter Weg, in dem alle Knoten paarweise verschieden sind. Ein *gerichteter Zyklus* ist ein gerichteter Weg in dem Start- und Endknoten identisch sind. Ein *gerichteter Kreis* ist ein gerichteter Weg $C = \langle v_1, \ldots, v_k, v_1 \rangle$ der Länge mindestens 2, in dem die Knoten $v_1, \ldots, v_k$ paarweise verschieden sind. Beachte, dass diese Definition eines Kreises von der Definition in ungerichteten Graphen etwas abweicht: ein ungerichteter Kreis muss Länge mindestens 3 haben. Insbesondere ist also in einem gerichteten Graphen $D = (V, A)$ die Folge $\langle v, u, v \rangle$ ein Kreis, wenn (u, v) und (v, u) in A sind; andererseits ist in einem ungerichteten Graphen der Zyklus $\langle v, u, v \rangle$ nie ein Kreis.

Azyklische Graphen; stark und schwach zusammenhängende Graphen. Kreisfreie ungerichtete Graphen sind als Bäume bzw. Wälder bekannt. Wir werden sie im Abschnitt 1.2 genauer studieren. Bei gerichteten Graphen müssen wir bei der Definition von Kreisfreiheit sehr präzise vorgehen, denn hier ist a priori überhaupt nicht klar, was kreisfrei bedeuten soll. Betrachten wir hierzu die beiden Digraphen in Abbildung 1.5: Der Linke ist offensichtlich kreisfrei. Wie sieht es aber mit dem Rechten aus? Ignorieren wir die Orientierung der Kanten, so enthält er Kreise. Andererseits enthält er aber keinen gerichteten Kreis. Solche Graphen haben einen eigenen Namen.

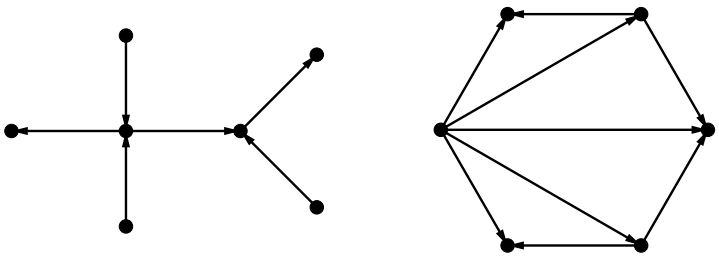


Abbildung 1.5: Zwei DAGs.

Ein gerichteter Graph $D = (V, A)$ heisst *azyklisch* (engl. *acyclic*), wenn er keinen gerichteten Kreis enthält. Azyklische gerichtete Graphen werden oft auch kurz als DAG bezeichnet, von engl. *Directed Acyclic Graph*.

DAGs (und nur diese) haben die schöne Eigenschaft, dass man ihre Knoten so nummerieren kann, dass alle Kanten vom kleineren zum grösseren Knoten zeigen. Man nennt so eine Nummerierung auch *topologische Sortierung*. Im vergangenen Semester haben Sie gesehen, wie man in DAGs

so eine Sortierung bestimmen kann: man gebe die kleinste Nummer einem Knoten mit Eingangsgrad Null und wiederhole dann diese Vorgehnsweise auf dem Teilgraphen ohne den kleinsten Knoten. Wenn man sich die Grade merkt (und geschickt updated) erhält man so einen Algorithmus dessen Laufzeit linear in der Anzahl Knoten und Kanten ist.

Satz 1.9. Für jeden DAG $D = (V, A)$ kann man in Zeit $O(|V| + |A|)$ eine topologische Sortierung berechnen. □

Jedem gerichteten Graphen kann man einen ungerichteten Graphen zuordnen, indem man die Orientierung der Kanten ignoriert (und ggf. entstehende Multikanten durch eine einzige Kante ersetzt). Diesen Graphen nennt man auch den *zugrunde liegenden* Graphen.

Bei der Übertragung einer Definition von Graphen auf Digraphen erhält man oft zwei verschiedene Definitionen, abhängig davon, ob man die Orientierung der Kanten berücksichtigt oder lediglich den zugrunde liegenden Graphen betrachtet.

Betrachten wir dies am Beispiel von u-v-Pfaden. Für zwei Knoten u und v kann man drei Fälle unterscheiden: a) es gibt einen gerichteten u-v-Pfad, b) es gibt keinen gerichteten u-v-Pfad, aber es gibt einen u-v-Pfad im zugrunde liegenden Graphen und c) es gibt überhaupt keinen u-v-Pfad.

Da der Zusammenhangsbegriff über die Existenz von u-v-Pfaden definiert ist, ergeben sich konsequenter Weise auch verschiedene Zusammenhangsbegriffe.

Ein gerichteter Graph $D = (V, A)$ heisst *stark zusammenhängend* (engl. *strongly connected*), wenn für jedes Paar von Knoten $u, v \in V$ ein gerichteter u-v-Pfad existiert. Ein gerichteter Graph $D = (V, A)$ heisst *(schwach) zusammenhängend* (engl. *weakly connected*), wenn der zugrunde liegende Graph zusammenhängend ist.

Beispiel 1.10. Die folgende Abbildung 1.6 zeigt links einen schwach zusammenhängenden Digraphen, der nicht stark zusammenhängend ist (was daran zu erkennen ist, dass es keine Kante gibt, die vom rechten Dreieck zum linken gerichtet ist) und rechts einen stark zusammenhängenden Digraphen.

Beispiel 1.11. Jeder DAG dessen zugrunde liegender Graph zusammenhängend ist, ist schwach zusammenhängend, aber nicht stark zusammenhängend. Letzteres sieht man ein, in dem man sich zunächst überlegt, dass die Definition des starken Zusammenhangs impliziert, dass der Graph gerichtete Kreise enthält. Betrachtet man beispielsweise eine beliebige Kante (x, y), so bildet diese zusammen mit einem gerichteten y-x-Pfad (der nach

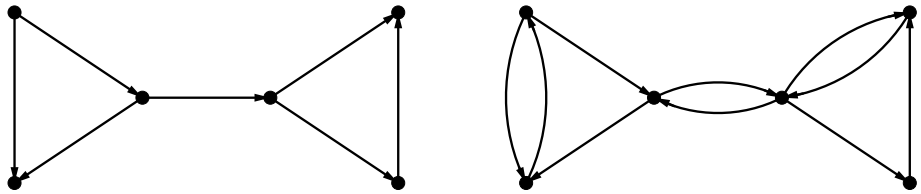


Abbildung 1.6: Ein schwach zusammenhängend Graph (links) und ein stark zusammenhängend Graph (rechts).

Definition des starken Zusammenhangs existieren muss) einen gerichteten Kreis. Gerichtete Kreise kann ein azyklischer Graph anderseits nach Definition nicht enthalten, d.h. ein DAG kann nicht stark zusammenhängend sein.

1.1.3 Datenstrukturen

Im vorigen Semester haben Sie bereits die beiden grundlegenden Möglichkeiten kennen gelernt, wie man einen Graphen (gerichtet oder ungerichtet) speichern kann: mit einer Adjazenzmatrix oder mit Adjazenzlisten. Beides ist besonders einfach, wenn die Knotenmenge $V = [n] = \{1, \ldots, n\}$ ist, wobei $n = |V|$. Die *Adjazenzmatrix* A_G ist dann eine $n \times n$ Matrix gegeben durch

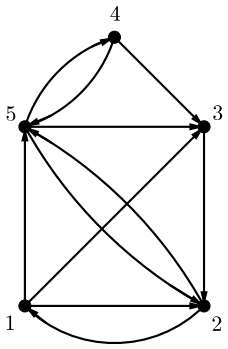
$$A_G = (a_{ij})_{i,j=1}^{n} \quad \text{mit} \quad a_{ij} := \begin{cases} 1 & \text{falls } \{i,j\} \in E, \\ 0 & \text{sonst.} \end{cases}$$

Für ungerichtete Graphen ist A_G eine symmetrische 0-1-Matrix mit Nullen auf der Hauptdiagonalen. Man überlegt sich leicht, dass umgekehrt auch jede solche Matrix einem Graphen entspricht.

Für gerichtete Graphen passt man obige Definition sinngemäss an und enthält dann ebenfalls eine 0-1-Matrix mit Nullen auf der Hauptdiagonalen, die aber nunmehr nicht symmetrisch sein muss. Wieder gilt auch: jede 0-1-Matrix mit Nullen auf der Hauptdiagonalen entspricht genau einem gerichteten Graphen.

Beispiel 1.12. Die folgende Abbildung 1.7 zeigt einen gerichteten Graphen G und die zugehörige Adjazenzmatrix A_G.

Im vorigen Semester haben Sie bereits Beispiele dafür gesehen, dass es, je nach Aufgabe, sinnvoll sein kann die eine oder die andere Datenstruktur zu verwenden. Wollen wir zum Beispiel mit Breiten- oder Tiefensuche testen, ob ein Graph zusammenhängend ist, so bieten sich Adjazenzlisten an.

$G =$ 

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Abbildung 1.7: Der Graph G und die zugehörige Adjazenzmatrix A_G.

Denn dann ist die Laufzeit $O(|V| + |E|)$, während die Verwendung eine Adjazenzmatrix immer zu einer Laufzeit von mindestens $O(|V|^2)$ führt. Wenn daher $|E| \ll |V|^2$ gilt (was immer gegeben ist, wenn der Durchschnittsgrad $o(|V|)$ ist), ist eine Breiten- oder Tiefensuche also schneller (unter Umständen sogar viel schneller) wenn mann Adjazenzlisten statt einer Adjazenzmatrix verwendet.

Die Verwendung von Adjazenzmatrizen erlaub andererseits die Verwendung von Hilfsmitteln aus der linearen Algebra. So haben Sie haben zum Beispiel auch bereits gesehen, dass man mit Hilfe von Matrizenmultiplikationen die Anzahl Wege der Länge k zwischen allen Knotenpaaren $i, j \in V$ bestimmen kann:

Satz 1.13. Sei $G = (V, E)$ bzw. $D = (V, A)$ ein ungerichteter bzw. gerichteter Graph mit Knotenmenge $V = [n]$ und M die zugehörige Adjazenzmatrix. Dann entspricht der Eintrag m_{ij}^k der i-ten Zeile und j-ten Spalte der Matrix M^k der Anzahl gerichteter Wege der Länge (genau) k von i nach j.

Auch zu anderen Parametern einer Matrix, wie zum Beispiel dem Rang oder den Eigenwerten, kann man Eigenschaften des zugehörigen Graphen in Verbindung bringen. Dieses überlassen wir aber weiterführenden Vorlesungen.

1.2 Bäume

Im vorigen Abschnitt haben wir Bäume als eine wichtige graphentheoretische Struktur kennen gelernt: ein Graph ist ein Baum, wenn er kreisfrei und zusammenhängend ist. Ebenfalls kennen gelernt haben wir den Begriff eines Teilgraphen: ein Graph $H = (V_H, E_H)$ ist (schwacher) Teilgraph eines Graphen $G = (V, E)$, wenn $V_H \subseteq V$ gilt und zusätzlich auch $E_H \subseteq E$. In diesem Sinne könnten wir also beispielsweise nach einem *grössten* Baum fragen der in G enthalten ist, wobei wir „Grösse“ interpretieren als „möglichst viele Kanten“.

Betrachten wir als Beispiel den vollständigen Graphen auf der Knotenmenge $V = [n]$. Wir erinnern uns: vollständig heisst, dass alle möglichen Kanten auch tatsächlich vorhanden sind. Für $n \geq 3$ enthält der vollständige Graph aber (je nach Grösse von n sogar sehr viele) Kreise. Ein Baum ist aber kreisfrei. Eine Möglichkeit einen Baum zu erhalten besteht daher darin, einfach so lange Kanten aus Kreisen zu entfernen, bis wir einen Baum erhalten. Dieses Verfahren funktioniert in der Tat, und zwar nicht nur für den vollständigen Graphen, sondern für jeden zusammenhängenden Graphen.

Satz 1.14. Ist $G = (V, E)$ ein zusammenhängender Graph, so findet der folgende Algorithmus einen Baum $T = (V, E_T)$ mit $E_T \subseteq E$:

$G' \leftarrow G$
while (G' enthält einen Kreis)
    wähle einen beliebigen Kreis C in G' und entferne eine beliebige Kante des Kreises C aus der Kantenmenge von G'
return G'

Um die Korrektheit dieses Satzes zu beweisen, überlegen wir uns zunächst eine wichtige Eigenschaften eines zusammenhängenden Graphen.

Lemma 1.15. Ist $G = (V, E)$ ein zusammenhängender Graph, C ein Kreis in G und e eine Kante in C, so ist der Graph $G_e = (V, E \setminus \{e\})$ (sprich: der Graph den wir aus G erhalten in dem wir die Kante e löschen) ebenfalls zusammenhängend.

Beweis. Zu zeigen ist, dass es für je zwei Knoten x, y in V mindestens

einen Weg von x nach y in G_e gibt. Da G zusammenhängend ist, gibt es aber in G auf jeden Fall einen x-y-Pfad, nennen wir ihn P. Falls die Kante $e = \{u, v\}$ nicht zu P gehört, so ist P auch ein Pfad in G_e; in diesem Fall sind wir fertig. Im anderen Fall enthält P die Kante e. Insbesondere gibt es in G_e dann einen Pfad $\langle x, a_1, \ldots, a_k, u\rangle$ von x zu einem der Endpunkte der Kante e, sagen wir zu u. Ebenso gibt enthält G_e einen Pfad $\langle y, b_1, \ldots, b_\ell, v\rangle$ von y zu dem anderen Endpunkt von e, hier also v. Da jedoch (nach Annahme) die Kante e auf einem Kreis C liegt, enthält G_e ausserdem noch einen u-v-Pfad $\langle u, c_1, \ldots, c_m, v\rangle$. Insgesamt enthält G_e also den x-y-Weg $\langle x, a_1, \ldots, a_k, u, c_1, \ldots, c_m, v, b_\ell, \ldots, b_1, y\rangle$, was zu zeigen war. □

Mit diesem Lemma ist der Beweis von Satz 1.14 nunmehr ein Kinderspiel.

Beweis von Satz 1.14. Da nach Annahme G zusammenhängend ist, ist der Graph G' nach der Initialisierung (als Graph G) sicherlich auch zusammenhängend. Lemma 1.15 garantiert uns weiterhin, dass wir den Zusammenhang von G' während der while-Schleife nicht zerstören. Nach Beendigung der while-Schleife ist der Graph G' also noch immer zusammenhängend, er ist aber auch kreisfrei (weil die while-Schliefe ja terminiert hat), also ist G' kreisfrei und zusammenhängend, sprich: ein Baum. □

Aus Satz 1.14 folgt: jeder zusammenhängende Graph enthält einen Baum auf der gleichen Knotenmenge. Solch einem Baum nennt man auch *Spannbaum* (engl. *spanning tree*). Auch wenn wir jetzt wissen, dass jeder zusammenhängende Graph (mindestens) einen Spannbaum enthält, so bleibt die Frage: wie finden wir einen „schönsten“ Spannbaum? – Bevor wir diese Frage beantworten können müssen wir uns zunächst darauf verständigen, was „schön“ überhaupt meint. Dies ist a priori nicht so klar. Wir könnten zum Beispiel darauf bestehen, dass alle Grade so klein wie möglich sind. Also beispielsweise einen Baum suchen in dem alle Grade kleiner gleich 2 sind, also (Übung!) einen Baum suchen, der de facto ein Pfad ist. Wie wir in Abschnitt 1.5.2 sehen werden ist dies kein ganz einfaches Problem. Zunächst wollen wir daher keine Restriktion über die Grade in dem Baum erheben sondern betrachten statt dessen eine gewichtete Variante des Spannbaumproblems.

MINIMALER SPANNBAUM

GEGEBEN: ein zusammenhängender Graph $G = (V, E)$ und eine Kostenfunktion $c\colon E \to \mathbb{R}$.
GESUCHT: ein Baum $T = (V, E_T)$ mit $E_T \subseteq E$ und $\sum_{e \in E_T} c(e) \stackrel{!}{=} \min$.

Einen solchen Spannbaum T, der die Summe $\sum_{e \in E_T} c(e)$ minimiert, nennt man auch einen *minimalen Spannbaum* (engl. *minimum spanning tree* oder *MST*). Bevor wir uns konkrete Algorithmen für dieses Problem ansehen, versuchen wir zunächst etwas Gefühl für dieses Problem zu entwickeln. Ein *Schnitt* in einem Graphen ist gegeben durch eine Menge $S \subseteq V$ mit $\emptyset \neq S$ und $S \neq V$ und besteht aus allen Kanten mit einem Endknoten in S und dem anderen in $V \setminus S$. Mit $E(S, V \setminus S)$ bezeichnen wir die Menge dieser Kanten (siehe Abbildung 1.8). Da jeder Pfad von einem Knoten in S zu einem Knoten in $V \setminus S$ mindestens eine Kante aus $E(S, V \setminus S)$ enthält, wissen wir insbesondere dass auch jeder (minimale) Spannbaum mindestens eine Kante aus $E(S, V \setminus S)$ enthalten muss. Im Idealfall ist das diejenige mit dem kleinsten Gewicht unter allen Kanten in $E(S, V \setminus S)$. Betrachten wir statt einem Schnitt einen Kreis C, so wissen wir umgekehrt, dass jeder (minimale) Spannbaum mindestens eine Kante von C *nicht* enthält. Im Idealfall ist das diejenige mit dem grösstem Gewicht unter allen Kanten in C. Diese Beobachtungen machen wir jetzt präzise.

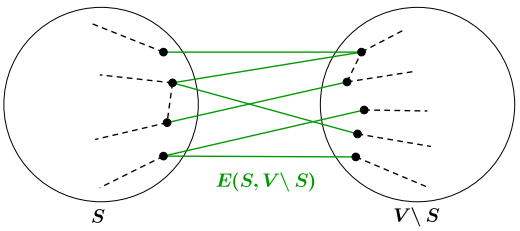


Abbildung 1.8: Ein Schnitt $E(S, V \setminus S)$ in dem Graphen $G = (V, E)$.

Lemma 1.16. Sei $G = (V, E)$ ein zusammenhängender Graph und sei $S \subseteq V$ mit $\emptyset \neq S$ und $S \neq V$. Für jede Kante $\hat{e}$ in $E(S, V \setminus S)$ mit $c(\hat{e}) = \min\{c(e) \mid e \in E(S, V \setminus S)\}$ gilt dann:
a) es gibt einen minimalen Spannbaum T^* in G, der $\hat{e}$ enthält.
b) gilt $c(\hat{e}) < c(e)$ für alle Kanten $e \in E(S, V \setminus S)$, $e \neq \hat{e}$, so ist $\hat{e}$ in *allen* minimalen Spannbäume in G enthalten.

Beweis. Da G nach Annahme zusammenhängend ist, enthält G sicherlich

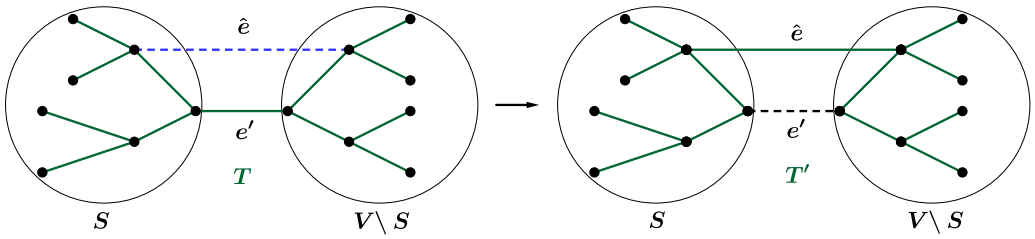


Abbildung 1.9: Illustration zum Beweis von Lemma 1.16.

einen Spannbaum (nach Satz 1.14). Es gibt daher mindestens einen minimalen Spannbaum (vielleicht auch mehrere). Sei T ein solcher. Ist $\hat{e}$ in T enthalten, so haben wir a) bereits gezeigt. Nehmen wir daher an, dass $\hat{e}$ nicht in T enthalten ist. Nach Annahme gehört einer der Endknoten von $\hat{e}$ zur Menge S, der andere nicht: sei daher $\hat{e} = \{x, y\}$ mit $x \in S$ und $y \notin S$. Da T ein Spannbaum ist, gibt es in T einen Pfad von x nach y. Wegen $x \in S$ und $y \notin S$ muss dieser daher eine Kante $e' = \{x', y'\}$ enthalten mit $x' \in S$ und $y' \notin S$ und somit $e' \in E(S, V \setminus S)$. Nach Wahl von e gilt daher $c(\hat{e}) \leq c(e')$. Der Baum $T' := T + \hat{e} - e'$ ist daher ebenfalls ein minimaler Spannbaum, der $\hat{e}$ im Gegensatz zu T enthält, womit a) bewiesen ist (Abbildung 1.9). Die Aussage b) folgt jetzt ganz analog: gäbe es einen minimalen Spannbaum T, der $\hat{e}$ nicht enthält, so könnten wir wie zuvor einen Spannbaum T' konstruieren, wobei aber jetzt, wegen der Annahme in b), gelten würde: $c(T') < c(T)$, was nicht sein kann, da T bereits ein minimaler Spannbaum war. Dieser Widerspruch zeigt, dass b) gilt. □

Lemma 1.17. Sei $G = (V, E)$ ein zusammenhängender Graph und sei C ein Kreis in G. Für jede Kante $\hat{e} \in C$ mit $c(\hat{e}) = \max\{c(e) \mid e \in C\}$ gilt dann:

a) es gibt einen minimalen Spannbaum T^* in G, der $\hat{e}$ nicht enthält.

b) gilt $c(\hat{e}) > c(e)$ für alle Kanten $e \in C$, $e \neq \hat{e}$, so gilt $\hat{e} \notin T$ für *alle* minimalen Spannbäume in G.

Beweis. Wie zuvor betrachten wir einen beliebigen minimalen Spannbaum T in G. Sei $\hat{e} = \{x, y\}$ und $C = \langle x, a_1, \ldots, a_k, y, x\rangle$. Ist $\hat{e}$ nicht in T enthalten, so haben wir a) bereits gezeigt. Nehmen wir daher an, dass $\hat{e}$ in T enthalten ist. Wir entfernen $\hat{e}$ aus T, sei also $T' := T - \hat{e}$. Im Folgenden bezeichnen wir mit S die Menge der Knoten, die in T' durch einen Pfad mit x verbunden

sind, und mit S' die Menge der Knoten, die in T' durch einen Pfad mit y verbunden sind. Nach Lemma 1.7 besteht T' aus genau zwei Komponenten, weshalb $S' = V \setminus S$ gilt.

Wir betrachten nun den x-y-Pfad $C' := \langle x, a_1, \ldots, a_k, y \rangle$, der aus C entsteht, indem wir die Kante $\hat{e}$ weglassen. Da $x \in S$ und $y \in V \setminus S$ ist, enthält C' eine Kante $e' = \{x', y'\}$ mit $x' \in S$ und $y' \in V \setminus S$. Wir sehen nun, dass $T'+e'$ ein Baum sein muss (siehe Abbildung 1.10): erstens ist $T'+e'$ zusammenhängend, denn die Kante e' verbindet ja die Zusammenhangskomponenten S und $V \setminus S$ von T', und zweitens ist $T'+e'$ kreisfrei, denn sonst hätte schon T' eine Kante enthalten, die zwischen S und $V \setminus S$ verläuft. Nach Wahl von $\hat{e}$ gilt $c(\hat{e}) \geq c(e')$ und damit $c(T' + e') = c(T) - c(\hat{e}) + c(e') \leq c(T)$. Da T nach Annahme ein minimaler Spannbaum war, ist daher auch $T' + e'$ ein minimaler Spannbaum, und zwar einer der $\hat{e}$ nicht enthält. Womit wir a) gezeigt haben. Die Aussage b) folgt jetzt wieder ganz analog: gäbe es einen minimalen Spannbaum T, der $\hat{e}$ enthält, könnten wir wie zuvor einen Spannbaum $T' + e'$ konstruieren, wobei aber jetzt, wegen der Annahme in b), gelten würde: $c(T' + e') < c(T)$, was nicht sein kann, da T bereits ein minimaler Spannbaum war. Dieser Widerspruch zeigt, dass b) gilt. □

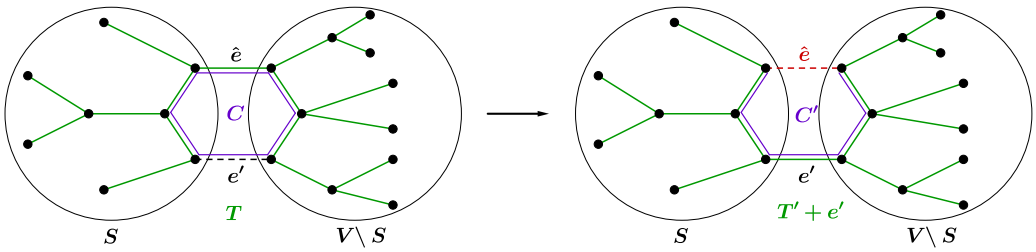


Abbildung 1.10: Illustration zum Beweis von Lemma 1.17.

Aus diesen beiden Lemmas leiten wir jetzt einen Algorithmus ab. Ausgehend von einem Graphen $G = (V, E)$ mit Kostenfunktion $c\colon E \to \mathbb{R}$ und ungefärbten Kanten, färben wir die Kanten iterativ (so lange es geht) in dem wir eine (beliebige) der beiden folgenden Regeln anwenden:

Blaue Regel

Gegeben: eine Menge $S \subseteq V$ mit $\emptyset \neq S$ und $S \neq V$, so dass keine Kante in $E(S, V \setminus S)$ blau gefärbt ist.

Färbe: eine ungefärbte Kante $\hat{e} \in E(S, V \setminus S)$ mit $c(\hat{e}) = \min\{c(e) \mid e \in E(S, V \setminus S)\}$ *blau*.

ROTE REGEL
Gegeben: ein Kreis C in G, so dass keine Kante in C rot gefärbt ist.
Färbe: eine ungefärbte Kante $\hat{e} \in C$ mit $c(\hat{e}) = \max\{c(e) \mid e \in C\}$ *rot*.

Satz 1.18. Ist $G = (V, E)$ ein zusammenhängender Graph mit Kostenfunktion $c\colon E \to \mathbb{R}$, so gilt für jede Ausführungsreihenfolge der blauen und roten Regel: lässt sich keine der beiden Regeln mehr anwenden, so sind alle Kanten gefärbt und die blauen Kanten bilden einen minimalen Spannbaum.

Beweis. Wir zeigen zunächst, dass die folgenden beiden Invarianten gelten:

SCHNITT: für jede Menge $S \subseteq V$ mit $\emptyset \neq S$ und $S \neq V$ gibt es eine Kante $\hat{e} \in E(S, V \setminus S)$ mit $c(\hat{e}) = \min\{c(e) \mid e \in E(S, V \setminus S)\}$, die ungefärbt oder blau gefärbt ist.

KREIS: für jeden Kreis C in G gibt es eine Kante $\hat{e} \in C$ mit $c(\hat{e}) = \max\{c(e) \mid e \in C\}$, die ungefärbt oder rot gefärbt ist.

Offenbar gelten beide Bedingungen zu Beginn des Algorithmus, da dann alle Kanten noch ungefärbt sind. Wir zeigen daher, dass die Anwendung der roten oder blauen Regel nicht dazu führen kann, dass eine der Invarianten nicht mehr gilt. Dazu überlegen wir uns zunächst, dass eine Anwendung der blauen Regel die Invariante SCHNITT nicht zerstören kann (da wir hier ja eine Kante blau färben) und umgekehrt eine Anwendung der roten Regel die Invariante KREIS nicht zerstören kann. Weiter überlegen wir uns, dass ein Kreis C und ein Schnitt $E(S, V \setminus S)$ sich immer in einer *geraden* Anzahl Kanten überschneiden, denn der Kreis muss ja genauso oft von S nach $V \setminus S$ gehen wie von $V \setminus S$ nach S. Wenn wir daher die blaue Regel anwenden, so gilt für jeden Kreis C: entweder ist keine Kante von C durch die blaue Regel tangiert oder es muss es noch eine weitere Kante in C geben, die (nach Definition der blauen Regel) nicht blau gefärbt ist und deren Gewicht mindestens so gross ist wie das Gewicht derjenigen Kante, die blau gefärbt wird. Eine Anwendung der blauen Regel kann daher die Invariante KREIS nicht zerstören. Wenn wir andererseits die rote Regel anwenden, so gilt für jeden Schnitt $E(S, V \setminus S)$: entweder ist keine Kante des Schnittes $E(S, V \setminus S)$ durch die rote Regel tangiert oder es muss es noch eine weitere Kante in $E(S, V \setminus S)$ geben, die (nach Definition der roten Regel) nicht rot gefärbt ist und deren Gewicht höchstens so gross ist wie das Gewicht derjenigen

Kante, die rot gefärbt wird. Eine Anwendung der roten Regel kann daher die Invariante SCHNITT nicht zerstören.

Somit folgt: die Invarianten SCHNITT und KREIS gelten insbesondere auch am Ende des Algorithmus. Wir betrachten jetzt den Teilgraphen $H = (V, E_{blau})$, wobei mit E_{blau} die Menge der blauen Kanten gemeint ist. Wäre dieser nicht zusammenhängend, so gäbe es eine Menge S so dass alle Kanten aus $E(S, V \setminus S)$ entweder rot oder ungefärbt sind. Wegen der Invariante SCHNITT gibt es dann eine ungefärbte Kante $\hat{e} \in E(S, V \setminus S)$ mit $c(\hat{e}) = \min\{c(e) \mid e \in E(S, V \setminus S)\}$, und daher könnten wir die blaue Regel auf die Menge S anwenden. Also wissen wir, dass der Teilgraph aus den blauen Kanten zusammenhängend ist. Ebenso kann H keinen Kreis enthalten, denn die Invariante KREIS impliziert insbesondere, dass jeder Kreis eine ungefärbte oder rote Kante enthalten muss, also nicht ganz blau sein kann. Der blaue Teilgraph H ist also ein Spannbaum. Wegen der Invariante KREIS wissen wir ausserdem, dass es keine ungefärbte Kante $\{x, y\}$ geben kann, denn auf den Kreis, der aus $\{x, y\}$ und dem x-y-Pfad in H besteht, könnten wir die rote Regel anwenden. Daraus folgt, dass H der einzige Spannbaum ist, der keine rote Kante enthält. Wegen Lemma 1.17 wissen wir zusätzlich: es gibt einen minimalen Spannbaum, der keine der rot gefärbten Kanten enthält (denn statt die Kante rot zu färben, könnten wir sie alternativ auch löschen). Der blaue Teilgraph ist daher nicht nur ein Spannbaum, sondern sogar ein *minimaler* Spannbaum, was zu zeigen war. □

In den nächsten beiden Abschnitten werden wir Satz 1.18 anwenden, um zwei klassische Algorithmen für das Finden minimaler Spannbäume herzuleiten.

1.2.1 Algorithmus von Prim

Als erstes betrachten wir einen Algorithmus, der de facto ohne die rote Regel auskommt. Genauer gesagt wird der Algorithmus zunächst $n-1$ mal die blaue Regel anwenden. Und da aus Satz 1.18 insbesondere folgt, dass danach nur noch die rote Regel anwendbar ist, können wir uns diese Schritte dann auch sparen. In der Literatur ist dieser Algorithmus als Algorithmus von Prim bekannt, benannt nach ROBERT PRIM, einem amerikanischen Wissenschaftler, der einen entsprechenden Algorithmus 1957 publizierte,

offenbar in Unkenntnis eines ähnlichen Verfahrens, das VOJTĚCH JARNÍK bereits 1930 auf tschechisch veröffentlichte. Wir gehen folgendermassen vor:

wähle einen beliebigen (Start-)Knoten $v \in V$
$S := \{v\}$
while $S \neq V$ **do**
  wähle eine (beliebige) Kante $\hat{e} \in E(S, V \setminus S)$ mit
    $c(\hat{e}) = \min\{c(e) \mid e \in E(S, V \setminus S)\}$
  färbe $\hat{e}$ blau
  sei $\hat{e} = \{\hat{v}, \hat{x}\}$ mit $\hat{v} \in S$ und $\hat{x} \notin S$: füge $\hat{x}$ zu S hinzu

Offenbar wächst die Menge S in jedem Durchlauf der while-Schleife um genau einen Knoten, und daher wird die Schleife genau $n - 1$ mal ausgeführt. Wir überlegen uns zunächst, dass dieser Algorithmus wirklich einen minimalen Spannbaum bestimmt. Hierfür zeigen wir: liegen zu Beginn einer Iteration der while-Schleife alle blau gefärbten Kanten in dem durch die Menge S induzierten Subgraphen, so gilt dies auch am Ende. In der Tat: aus der Konstruktion des Algorithmus folgt unmittelbar, dass alle Endpunkte der neu blau gefärbten Kante $\hat{e}$ am Ende der while-Schleife ebenfalls zu S gehören (und alle andere tun das nach Annahme sowieso). Verwenden wir nun noch, dass zu Beginn des Algorithmus keine Kante blau gefärbt ist und daher die Menge $S = \{v\}$ alle (nicht vorhandenen) blauen Kanten enthält, so können wir somit folgern, dass jede der $n-1$ Iterationen der while-Schleife einer korrekten Anwendung der blauen Regel entspricht. Daher bilden die blauen Kanten am Ende einen minimalen Spannbaum.

Nachdem wir die Korrektheit des Algorithmus eingesehen haben, überlegen wir uns nun noch, wie man den Algorithmus effizient implementieren kann. Eine naive Implementierung würde einfach in jeder der $n-1$ Iterationen der while-Schleife die komplette Kantenliste durchlaufen und für jede Kante (in konstanter Zeit) entscheiden, ob sie zu dem aktuellen Schnitt $E(S, V \setminus S)$ gehört und unter allen solchen eine minimale Kante auswählen. Damit erhalten wir einen Algorithmus mit Laufzeit $O(|V| \cdot |E|)$.

Um dies zu verbessern, überlegen wir uns zunächst, dass sich die Menge S in jedem Durchlauf der while-Schleife nur um genau einen Knoten ändert. Um diese Knoten effizient zu finden, definieren wir zwei Parameter:

$$\begin{aligned} \forall x \notin S: \qquad \sigma[x] &:= \min\{c(\hat{v}, x) \mid \hat{v} \in N(x) \cap S\}, \\ \operatorname{pred}[x] &:= \text{ein } \hat{v} \in S \text{ so dass } c(\hat{v}, x) = \sigma[x]. \end{aligned}$$

Anschaulich bedeutet dies: für jeden Knoten $x \in V \setminus S$ bezeichnet $\sigma[x]$ das Gewicht einer leichtesten Kante von x in die Menge S (falls es keine solche Kante gibt, das Minimum also über eine leere Menge geht, so ist $\sigma[x]$ definitionsgemäss gleich $+\infty$). Beachte: $\sigma[x]$ gibt nur das *Gewicht* einer leichtesten Kanten an. Um eine solche Kante auch wirklich zu finden (um sie in den minimalen Spannbaum einfügen zu können), müssen wir uns auch noch einen Knoten aus S merken, der zusammen mit x solch eine billigste Kante bildet. Dies ist die Aufgabe des Parameters $\text{pred}[x]$.

Jetzt müssen wir uns noch überlegen, wie wir diese Parameter effizient berechnen. Die Initialisierung ist einfach: für alle $x \in N(v)$ setzen wir $\sigma[x] = c(v, x)$ und $\text{pred}[x] = v$, für alle übrigen Knoten $x \in V \setminus S$ setzen wir $\sigma[x] = \infty$ und $\text{pred}[x] = \text{undef}$ (für: undefined), um anzudeuten, dass es von diesem x keine Kante in die Menge S gibt. Auch der Update der Werte innerhalb der while-Schleife lässt sich einfach realisieren. Da in der while-Schleife jeweils nur der Knoten $\hat{x}$ zur Menge S hinzugefügt wird, müssen wir nur die Nachbarn von $\hat{x}$ in die Menge $V \setminus S$ betrachten, denn nur für diese kann $\sigma[x]$ sich ändern. Für jeden solchen Knoten $x \in N(\hat{x}) \cap (V \setminus S)$ vergleichen wir die Kosten $c(\hat{x}, x)$ der Kante von $\hat{x}$ zu x mit dem Wert $\sigma[x]$, dem Gewicht einer leichtesten Kante von x in die Menge S. Falls $c(\hat{x}, x) < \sigma[x]$, so setzen wir $\sigma[x] = c(\hat{x}, x)$ und $\text{pred}[x] = \hat{x}$.

Was uns jetzt noch fehlt, ist ein Verfahren, das es uns erlaubt innerhalb der while-Schleife jeweils den „besten“ Knoten $\hat{x}$ effizient zu bestimmen. Hierfür bietet sich die Datenstruktur der sogenannten *PriorityQueues* an. Diese unterstützen drei Arten von Operationen: INSERT, EXTRACTMIN, DECREASEKEY, die jeweils genau das tun was ihr Name besagt: ein neues Element mit einem dazugehörenden Schlüssel einfügen, dasjenige Element mit dem kleinsten Schlüssel auslesen (und aus der Datenstruktur entfernen), und den Schlüssel eines Elementes verringern.

Nach all diesen Vorüberlegungen erhalten wir jetzt folgende Implementierung des Algorithmus von Prim.

ALGORITHMUS VON PRIM (G, c)

1: wähle einen beliebigen (Start-)Knoten $v \in V$
2: $S \leftarrow \{v\}$
3: **for all** $x \in V \setminus S$ **do**
4: **if** $x \in N(v)$ **then**
5: $\sigma[x] \leftarrow c(v, x)$ und $\text{pred}[x] \leftarrow v$
6: **else**
7: $\sigma[x] \leftarrow \infty$ und $\text{pred}[x] \leftarrow \text{undef}$
8: $T \leftarrow \emptyset$ ▷ Initialisiere Spannbaum
9: $Q \leftarrow \emptyset$ ▷ Initialisiere PriorityQueue
10: **for all** $x \in V \setminus S$ **do**
11: INSERT$(Q, x, \sigma[x])$ ▷ füge x mit dem Schlüssel $\sigma[x]$ ein
12: **while** $S \neq V$ **do**
13: $\hat{x} \leftarrow$ EXTRACTMIN(Q)
14: $S \leftarrow S \cup \{\hat{x}\}$
15: $T \leftarrow T + \{\hat{x}, \text{pred}[\hat{x}]\}$
16: **for all** $x \in N(\hat{x}) \cap (V \setminus S)$ **do**
17: **if** $c(\hat{x}, x) < \sigma[x]$ **then**
18: $\sigma[x] \leftarrow c(\hat{x}, x)$ und $\text{pred}[x] \leftarrow \hat{x}$
19: DECREASEKEY$(Q, x, \sigma[x])$
20: **return** T ▷ T is ein minimaler Spannbaum

Die Laufzeit des Algorithmus von Prim lässt sich jetzt einfach abschätzen. Die Initialisierung wird dominiert durch die $n-1$ INSERT-Aufrufe. Innerhalb der while-Schleife (die $n-1$ mal durchlaufen wird, einmal für jede Kante des Spannbaums) wird in jeder Iteration ein Mal die Operation EXTRACTMIN aufgerufen. Wenn der Graph mit Adjazenzlisten abgespeichert ist, so benötigt die forall-Schleife für jeden Knoten $\hat{x}$ Lautzeit proportional zu $\deg(\hat{x}) \cdot \langle$Laufzeit von DECREASEKEY$\rangle$. Wegen $\sum_{\hat{x} \in V} \deg(\hat{x}) = 2|E|$ (vgl. Lemma 1.2) erhalten wir folgende Schranke für Laufzeit des Algorithmus von Prim:

$$O(|V| \cdot \langle \text{Laufzeit von INSERT} \rangle + |V| \cdot \langle \text{Laufzeit von EXTRACTMIN} \rangle + |E| \cdot \langle \text{Laufzeit von DECREASEKEY} \rangle).$$

Die konkrete Laufzeit des Algorithmus von Prim hängt davon ab, wie wir die PriorityQueue implementieren. Wir könnten zum Beispiel ein Array

verwenden. Dann benötigt jedes INSERT und jedes DECREASEKEY nur $O(1)$ Zeit, ein EXTRACTMIN jedoch $O(|V|)$. Als Laufzeit für den Algorithmus von Prim erhalten wir damit $O(|V| + |V|^2 + |E|) = O(|V|^2)$. Alternativ können wir die aus dem ersten Teil der Vorlesung bekannten Min-Heaps verwenden. Für diese benötigen alle drei Operationen nur Laufzeit $O(\log |V|)$. Damit erhalten wir:

Satz 1.19. Der Algorithmus von Prim berechnet für zusammenhängende Graphen $G = (V, E)$ mit Gewichtsfunktion $c\colon E \to \mathbb{R}$, die mit Adjazenzlisten gespeichert sind, in Zeit $O(\min\{|V|^2, |E| \cdot \log |V|\})$ einen minimalen Spannbaum. □

Die beste bekannte Laufzeit erhält man, wenn man statt den Min-Heaps sogenannte Fibonacci-Heaps verwendet. Damit reduziert sich die asymptotische Laufzeit des Algorithmus von Prim auf $O(|V| \cdot \log |V| + |E|)$. Allerdings sind hierbei die in der O-Notation verborgenen Konstanten relativ gross, so dass für praktische Zwecke die Verwendung eines Min-Heaps oft effizienter ist.

1.2.2 Algorithmus von Kruskal

Der amerikanischer Mathematiker JOSEPH KRUSKAL hat 1956 folgenden Algorithmus für das Minimale Spannbaum Problem vorgestellt:

sortiere die Kantenmenge E so dass gilt:
$E = \{e_1, \ldots, e_m\}$ und $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$
$S := \{v\}$
for $i = 1$ **to** m **do**
  wenn die Endpunkte von e_i durch einen blauen Pfad verbunden sind:
    färbe e_i rot
  ansonsten:
    färbe e_i blau

Wir überlegen uns zunächst, dass der Algorithmus die rote bzw. blaue Regel korrekt anwendet. In der Tat: wenn die Endpunkte Kante der Kante e_i durch einen blauen Pfad verbunden sind, so können wir wegen der aufsteigenden Sortierung der Kanten die rote Regel für den zugehörigen Kreis anwenden. Ansonsten befinden sich die Endpunkte der Kante e_i in

verschiedenen Zusammenhangskomponenten S und S' des durch die blau gefärbten Kanten gegebenen Teilgraphen. Offenbar gibt es dann auch keine roten Kanten mit einem Endpunkt in S und einem in $V \setminus S$, denn wir färben eine Kante ja nur rot, wenn zwischen ihren Endpunkten ein blauer Pfad läuft. Wegen der aufsteigenden Sortierung der Kanten können wir die blaue Regel zum Beispiel für Menge S anwenden.

Satz 1.18 garantiert uns also, dass der Algorithmus von Kruskal einen minimalen Spannbaum findet. Wie aber können wir ihn effizient implementieren? Eine naive Implementierung würde in jeder Iteration der for-Schleife eine Breiten- oder Tiefensuche für den Teilgraphen mit den blau gefärbten Kanten ausführen, um zu entscheiden ob die Endpunkte der gegebenen Kante durch einen blauen Pfad verbunden sind. Dies führt zu einer Laufzeit von $O(|E| \cdot |V|)$ (da der blaue Teilgraph höchstens $|V| - 1$ Kanten enthält).

Eine effizientere Implementierung erhalten wir wiederum mit der Verwendung einer geeigneten Datenstruktur. In diesem Fall ist dies die sogenannten *Union-Find-Struktur*. Dies ist eine Datenstruktur für folgendes Problem. Gegeben sei eine endliche Menge X, partitioniert in (disjunkte) Mengen X_i, wobei jede Menge X_i durch ein in ihr enthaltenes Element repräsentiert wird. Die Datenstruktur unterstützt nun die folgenden drei Operationen:

- INSERT(X, x): füge das Element x zur Menge X hinzu; dieses bildet eine neue (einelementige) Menge X_i,
- FIND(X, x): gib den Repräsentanten derjenigen Menge X_i aus, die x enthält,
- UNION(X, x, y): vereinige die beiden Mengen X_i und X_j, die x bzw. y enthalten.

Die grundlegende Idee für die Realisierung dieser Datenstruktur ist sehr einfach. Jede Menge wird durch einen gewurzelten und zur Wurzel hin gerichteten Baum (einen sogenannten *intree*) dargestellt. Speichern wir dann an der Wurzel des Baumes den Repräsentanten der zugehörigen Menge X_i, so kann man beispielsweise ein FIND dadurch realisieren, dass man den Baum entlang der zur Wurzel hin gerichteten Kanten durchläuft und den dort gespeicherten Repräsentanten ausgibt.

Die einzelnen Versionen dieser Datenstruktur unterscheiden sich darin, wie man die Bäume im Detail aufbaut. Hier wollen wir eine ganz einfache

Version betrachten, in der alle Bäume Tiefe 1 haben. Ein FIND lässt sich daher immer in $O(1)$ Zeit ausführen. Zeitaufwendiger ist jedoch die UNION Operation, denn hier müssen wir ja alle Knoten eines der beiden Bäume unter die Wurzel des anderen Baumes hängen. Um den Aufwand hierfür wenigstens etwas unter Kontrolle zu halten, vereinbaren wir noch Folgendes: an der Wurzel jedes Baumes speichern wir die Anzahl der Knoten ab, die in dem Baum enthalten sind. Dann können wir also immer die Knoten des kleineren Bäume unter die Wurzel des grösseren Baumes hängen. Auf den ersten Blick hilft dies nicht viel: für zwei Bäume mit je $|X|/2$ Knoten benötigt ein einziger Aufruf der UNION Operation trotzdem Laufzeit $\Theta(|X|)$. Eine *amortisierte Analyse* erlaubt uns jedoch folgende Beobachtung: jeder Knoten aus X kann während einer beliebigen Folge von UNION-Aufrufen höchstens $\log |X|$ oft umgehängt werden. Dies sieht man wie folgt ein: nach dem Einfügen in die Datenstruktur (mit INSERT) bildet der Knoten seine eigene Menge, also einen Baum mit genau einem Knoten. Jedes Mal wenn der Knoten bei einem Aufruf von UNION an eine neue Wurzel angehängt, so befindet er sich danach in einem Baum mit mindestens doppelt so vielen Knoten wie vorher (da wir ja die Knoten des kleineren Baumes umhängen). Nach k-maligem Umhängen befindet sich der Knoten also in einem Baum mit mindestens 2^k vielen Knoten. Da aber kein Baum mehr als $|X|$ viele Knoten enthalten kann, können wir den Knoten höchstens $\log |X|$ oft umhängen. Damit erhalten wir:

Satz 1.20. Eine Union-Find-Struktur für eine Menge X kann so implementiert werden, dass jedes INSERT und jedes FIND nur Zeit $O(1)$ benötigt und eine beliebige Folge von UNION Operationen eine Gesamt-Laufzeit von $O(|X| \cdot \log |X|)$ hat. □

Für den Algorithmus von Kruskal können wir diese Datenstruktur verwenden, in dem wir als Menge X die Knotenmenge V unseres Graphen wählen und als Mengen X_i die Knotenmengen der Zusammenhangskomponenten des blauen Teilgraphen. Damit ergibt sich dann folgende Implementierung:

ALGORITHMUS VON KRUSKAL (G, c)

1: sortiere die Kantenmenge E so dass gilt:
2: $\quad E = \{e_1, \ldots, e_m\}$ und $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$
3: $T \leftarrow \emptyset$ ▷ Initialisiere Spannbaum
4: $X \leftarrow \emptyset$ ▷ Initialisiere Union-Find-Struktur
5: **for all** $v \in V$ **do**
6: $\quad$ INSERT(X, v)
7: **for** $i = 1$ **to** m **do**
8: $\quad$ seien x und y die Endpunkte von e_i, also $e_i = \{x, y\}$
9: $\quad$ **if** FIND$(X, x) \neq$ FIND(X, y) **then**
10: $\quad\quad T \leftarrow T + e_i$
11: $\quad\quad$ UNION(X, x, y)
12: **return** T ▷ minimaler Spannbaum

Die Analyse der Laufzeit ist jetzt sehr einfach: die Sortierung der Kantenmenge (z.B. mit Quicksort) erfordert $O(|E| \log |E|) = O(|E| \log |V|)$ Operationen, da $|E| \leq |V|^2$ ist. Die for-schleife wird $|E|$ mal durchlaufen, die beiden FIND-Aufrufe benötigen hierbei jeweils nur $O(1)$ Zeit; die Gesamtlaufzeit für alle UNION-Aufrufe ist nach Satz 1.20 beschränkt durch $O(|V| \cdot \log |V|)$. Somit erhalten wir:

Satz 1.21. Der Algorithmus von Kruskal berechnet für zusammenhängende Graphen $G = (V, E)$ mit Gewichtsfunktion $c\colon E \to \mathbb{R}$, die mit Adjazenzlisten gespeichert sind, in Zeit $O(|E| \cdot \log |V|)$ einen minimalen Spannbaum. □

1.2.3 Exkurs: Shannon's Switching Game

Zum Abschluss dieses Abschnittes über Spannbäume betrachten wir noch ein Spiel bei dem wir einige der Begriffe und Argumentationsweisen dieses Abschnittes nochmals vertiefen können. Zwei Spieler, nennen wir sie BLAU und ROT färben die Kanten eines zu Beginn ungefärbten Graphen $G = (V, E)$ abwechselnd mit ihrer Farbe: BLAU färbt eine Kante blau, dann ROT eine Kante rot, dann wieder BLAU eine Kante blau, usw. Wobei beide Spieler immer nur ungefärbte Kanten färben dürfen. Gewonnen hat wer

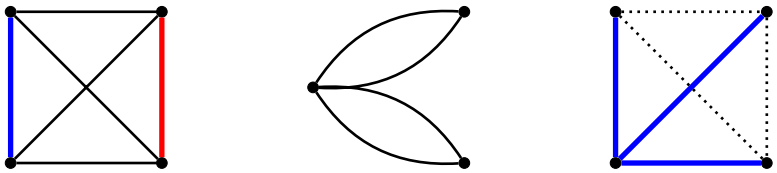
zuerst einen Spannbaum in seiner Farbe hat. Leider ist dieses Spiel für den Spieler ROT ziemlich unattraktiv: denn ROT kann nicht gewinnen. Dies sieht man wie folgt ein. Angenommen es gäbe eine Strategie mit der ROT gewinnen könnte. Strategie heisst hier: für jede Spielsituation (Teil der Kanten blau bzw. rot gefärbt) eine Wahl für die nächste rot zu färbende Kante. Wenn es so eine Strategie für ROT gäbe ... dann könnte auch BLAU danach spielen (mit den die Farben blau und rot vertauscht). Und da BLAU beginnt, hätte dann BLAU zuerst einen Spannbaum. Dieses Argument nennt man *Strategie-Klau* (engl. *strategy stealing*). Es zeigt, dass es bei einem solchen, wie man sagt symmetrischen, Spiel nur zwei mögliche Ergebnisse geben kann: der erste Spieler gewinnt oder das Spiel endet unentschieden.

Um das Spiel für beide Spieler attraktiv zu halten, betrachten man daher häufig statt einer symmetrischen Variante eine unsymmetrische. Die beiden Spieler nennt man dabei hierbei oft auch MAKER und BREAKER. Wir werden gleich sehen wieso. In unserem Fall hat MAKER die Aufgabe einen Spannbaum zu erzeugen. MAKER gewinnt, wenn ihm das gelingt. Gelingt ihm das nicht, so hat BREAKER gewonnen. Die Regeln sind dabei analog zu oben: MAKER färbt die Kanten in seiner Farbe, BREAKER darf jeweils eine ungefärbte Kante löschen. Wir nehmen an, dass BREAKER den ersten Zug hat. Für welche Graphen kann MAKER gewinnen? – Der Satz von Shannon beantwortet diese Frage:

Satz 1.22. Für einen $G = (V, E)$ gibt es genau dann eine Gewinnstrategie für MAKER, wenn G zwei kantendisjunkten Spannbäume enthält.

Beweis. Wir zeigen zunächst, dass es eine Gewinnstrategie für MAKER gibt, wenn G zwei kantendisjunkte Spannbäume enthält. Hierfür benötigen wir das Konzept der *Kantenkontraktion*, das für Multigraphen definiert ist. Bei der *Kontraktion* einer Kante $e = \{u, v\}$ verschmelzen wir die beiden Knoten u und v zu einem neuen Knoten $x_{u,v}$, der nun zu allen Kanten inzident ist, zu denen u oder v inzident war, wobei wir alle Kanten zwischen u und v löschen.

Um die Strategie für MAKER zu erläutern betrachten wir zunächst ein Beispiel. Nehmen wir an, dass G der vollständige Graph auf vier Knoten ist, also $G = K_4$. BREAKER beginnt und entfernt eine beliebige Kante. Als nächstes wählt MAKER die Kante zwischen den anderen beiden Knoten. Die folgende Abbildung illustriert diese Vorgehensweise.



In der linken Abbildung markiert die rote Kante die von BREAKER gewählte Kante und die blaue Kante die von MAKER gewählte Kante. In der nächsten Abbildung haben wir die Kante von BREAKER gelöscht und die Kante von MAKER kontrahiert. Jetzt klar wie MAKER spielen muss: sobald BREAKER eine der parallelen Kanten nimmt, nimmt MAKER die andere. Die rechte Abbildung zeigt einen der möglichen Spannbäume für MAKER (welcher genau sich ergibt, hängt davon ab, welche Kanten BREAKER entfernt).

Mit dieser Vorüberlegung sind wir jetzt bereit, die Strategie für MAKER zu formulieren. Genauer zeigen wir durch Induktion über die Anzahl Knoten: ist $G = (V, E)$ ein *Multi*graph mit zwei kantendisjunkten Spannbäumen T und T', so gibt es eine Gewinnstrategie für MAKER. Für $|V| = 2$ ist dies offensichtlich richtig. Denn die Annahme, dass es zwei *kantendisjunkte* Spannbäume gibt, impliziert, dass es zwischen den beiden Knoten aus V mindestens *zwei* Kanten geben muss. Von diesen kann BREAKER in seinem ersten Zug nur eine entfernen und MAKER kann daher ebenfalls eine solche Kante wählen, die dann bereits einen Spannbaum bildet.

Betrachten wir nun den Induktionsschritt. Sei e die Kante, die BREAKER in seinem ersten Zug wählt. Ohne Einschränkung sei diese in T. (Falls e weder in T noch in T' ist, so wählen wir als e eine beliebige Kante aus T). Durch Entfernen von e zerfällt T in zwei Komponenten. Da T' ebenfalls ein Spannbaum ist, enthält T' mindestens eine Kante, die zwischen diesen beiden Komponenten verläuft. Solch eine Kante wählt MAKER. Nennen wir sie e'. Wenn wir nun e' kontrahieren, so sind $T - e$ und $T' - e'$ kantendisjunkte Spannbäume im kontrahierten Graphen. Und da dieser Graph einen Knoten weniger enthält als G folgt aus der Induktionsannahme, dass es eine Gewinnstrategie für MAKER im kontrahierten Graphen gibt. Das war es, was wir zeigen wollten!

Sei nun andererseits G ein Graph, der keine zwei kantendisjunkte Spannbäume enthält. Wenn G gar keinen Spannbaum enthält, so muss sich BREAKER nicht sonderlich anstrengen um zu gewinnen. Nehmen wir daher an, dass G mindestens einen Spannbaum enthält. BREAKER wählt sich einen solchen Spannbaum, wir nennen ihn T, beliebig. Nach Annahme gibt es kei-

nen zu T kantendisjunkten zweiten Spannbaum. BREAKER wählt sich daher nun einen zu T kantendisjunkten Wald F mit möglichst vielen Kanten. Falls F aus weniger als $|V|-2$ Kanten besteht, so zeigt sich BREAKER grosszügig und fügt noch in G noch einige zusätzliche Kanten hinzu, bis es einen zu T kantendisjunkten Wald mit $|V|-2$ Kanten gibt. Man beachte: wenn der Graph mehr Kanten enthält, so wird es für MAKER leichter zu gewinnen. Wir werden aber zeigen, dass MAKER trotz dieser zusätzlichen Kanten noch immer nicht gewinnen kann. Hier ist die Strategie für BREAKER. Da F ein Wald mit $|V|-2$ Kanten ist, besteht F aus zwei Zusammenhangskomponenten. Da T ein Spannbaum ist, gibt es daher mindestens eine Kante in T, die diese beiden Komponenten verbindet. Solch ein Kante e wählt BREAKER für seinen ersten Zug. Danach kontrahiert er (in Gedanken) diese Kante. Der neu entstandene Graph G' enthält jetzt zwei kantendisjunkte Spannbäume: F und $T-e$. Im Graph G' hat jetzt MAKER den ersten Zug. BREAKER kann daher jetzt so tun als wäre er MAKER – und die Strategie aus dem ersten Teil des Beweises anwenden – um in dem Graphen G' einen Spannbaum zu erhalten. Zusammen mit der Kante e ist dies ein Spannbaum in G. Und da es, nach Annahme, keinen dazu kantendisjunkten zweiten Spannbaum in G gibt, können MAKERs Kanten nicht ebenfalls einen Spannbaum enthalten. Also hat BREAKER gewonnen! □

1.3 Pfade

Im ersten Teil der Vorlesung wurden verschiedene Algorithmen vorgestellt, um kürzeste Pfade zwischen zwei gegebenen Knoten (oder auch allen Knotenpaaren) zu finden.

KÜRZESTE PFADE
GEGEBEN: ein zusammenhängender Graph $G = (V, E)$, zwei Knoten $s, t \in V$ und eine Kostenfunktion $c\colon E \to \mathbb{R}$.
GESUCHT: ein s-t-Pfad P in G mit $\sum_{e\in P} c(e) \stackrel{!}{=} \min$.

In diesem Skript führen wir fürs erste nur die Namen der dort vorgestellten Algorithmen an:

- Algorithmus von Dijkstra
- Algorithmus von Floyd-Warshall

- Algorithmus von Bellman-Ford
- Algorithmus von Johnson

1.4 Zusammenhang

Wir erinnern uns: ein Graph ist genau dann zusammenhängend, wenn es zwischen je zwei Knoten des Graphen einen Pfad gibt. Dieser Begriff des Zusammenhangs sagt aber nichts über den Grad oder die Intensität des Zusammenhangs aus. Darauf wollen wir in diesem Abschnitt eingehen.

Definition 1.23. Ein Graph $G = (V, E)$ heisst k-*zusammenhängend*, falls $|V| \geq k + 1$ und für alle Teilmengen $X \subseteq V$ mit $|X| < k$ gilt: Der Graph $G[V \setminus X]$ ist zusammenhängend.

Eine Teilmenge $X \subseteq V$ der Knoten, für die $G[V \setminus X]$ nicht zusammenhängend ist, nennt man auch *(Knoten-)Separator*. Genauer: Sind $u, v \in V$ und ist $X \subseteq V \setminus \{u, v\}$ eine Knotenmenge, für die u und v in verschiedenen Zusammenhangskomponenten von $G[V \setminus X]$ liegen, so nennt man X einen u-v-Separator. Ein Graph $G = (V, E)$ ist also genau dann k-zusammenhängend, wenn jeder Separator mindestens Grösse k hat (die einzige Ausnahme ist der vollständige Graph mit k Knoten, der per Definition $k-1$ zusammenhängend ist). Je grösser daher k, desto „zusammenhängender“ ist der Graph. Insbesondere ist ein k zusammenhängender Graph auch $k - 1$ zusammenhängend (usw.). Statt die Entfernung von Knoten zu erlauben, könnten wir auch nur Kanten entfernen. Dann erhält man den Begriff des Kanten-Zusammenhangs:

Definition 1.24. Ein Graph $G = (V, E)$ heisst k-*kanten-zusammenhängend*, falls für alle Teilmengen $X \subseteq E$ mit $|X| < k$ gilt: Der Graph $(V, E \setminus X)$ ist zusammenhängend.

Auch hier gilt also: Ein Graph $G = (V, E)$ ist genau dann k-kanten-zusammenhängend, wenn man mindestens k Kanten (oder mehr) entfernen muss, um den Zusammenhang des Graphen zu zerstören. Analog zu Knotenseparatoren nennt man eine solche Kantenmenge $X \subseteq V$ einen *Kanten-*

separator (bzw. einen u-v-Kantenseparator, wenn $u, v \in V$ nach Entfernen von X in verschiedenen Zusammenhangskomponenten liegen.)

Eine andere Möglichkeit einen starken Zusammenhang zu fordern wäre, für alle Knotenpaare $u, v \in V$, $u \neq v$, statt nur einen Pfad (wie in der Definition des Zusammenhangs) die Existenz von k Pfaden zu fordern, die (ausser u und v) paarweise keine Knoten bzw. Kanten gemeinsam haben. Man spricht hier auch von k intern-knoten-disjunkten bzw. kantendisjunkten Pfaden. Der Satz von KARL MENGER (1902-1985) zeigt, dass beide Begriffe in der Tat äquivalent sind. Wir werden später sehen, dass der Satz von Menger nur ein Spezialfall eines viel allgemeineren Satzes ist. Deshalb verschieben wir den Beweis auf später (Abschnitt 3.1.2).

Satz 1.25 (Menger). Sei $G = (V, E)$ ein Graph und $u, v \in V, u \neq v$. Dann gilt:

a) Jeder u-v-Knotenseparator hat Grösse mindestens $k \iff$ Es gibt mindestens k intern-knotendisjunkte u-v-Pfade.

b) Jeder u-v-Kantenseparator hat Grösse mindestens $k \iff$ Es gibt mindestens k kantendisjunkte u-v-Pfade.

1.4.1 Artikulationsknoten

Ist ein Graph zusammenhängend, aber nicht 2-zusammenhängend, so gibt es mindestens einen Knoten v mit der Eigenschaft, dass $G[V \setminus \{v\}]$ nicht zusammenhängend ist. Knoten mit dieser Eigenschaft nennt man *Artikulationsknoten*. In diesem Abschnitt werden wir zeigen, wie man die Tiefensuche so modifizieren kann, dass sie *en passant* auch alle Artikulationsknoten bestimmt. Hier zur Erinnerung die rekursive Variante der Tiefensuche, wie sie im letzten Semester eingeführt wurde.

DFS-VISIT(G, v)

```
1: Markiere v als besucht
2: for all {v, w} ∈ E do
3:     if w ist noch nicht besucht then
4:         DFS-VISIT(G, w)
```

Ein Aufruf von DFS-VISIT(G, v) besucht alle Knoten in der Zusammen-

hangskomponente, die v enthält. Wenn der Graph zusammenhängend ist, was wir in diesem Abschnitt annehmen wollen, so besucht ein Aufruf von DFS-VISIT daher alle Knoten. Als erstes modifizieren wir den Algorithmus so, dass er jedem Knoten statt einem Label „besucht“ eine Nummer zuordnet, und zwar in der Reihenfolge in der die Knoten besucht werden. D.h. der Startknoten s erhält die Nummer 1, der erste Nachbar von s, der besucht wird, die Nummer 2, usw.[2] Diese Nummern speichern wir in einem Array `dfs[ ]` ab. Dieses Array initialisieren wir in einem Rahmenprogramm:

DFS(G, s)

1: $\forall v \in V:$ `dfs`$[v] \leftarrow 0$
2: `num` $\leftarrow 0$
3: $T \leftarrow \emptyset$
4: DFS-VISIT(G, s)

Insbesondere bedeutet daher `dfs`$[w] = 0$, dass der Knoten w noch nicht besucht wurde. In der Variable `num` speichern wir zu jedem Zeitpunkt die Anzahl der bereits besuchten Knoten und in der Menge die gefundenen Kanten des Tiefensuchbaumes. Die Prozedur DFS-VISIT verändern wir hierfür wie folgt:

DFS-VISIT(G, v)

1: `num` $\leftarrow$ `num` $+ 1$
2: `dfs`$[v] \leftarrow$ `num`
3: **for all** $\{v, w\} \in E$ **do**
4: **if** `dfs`$[w] = 0$ **then**
5: $T \leftarrow T + \{v, w\}$
6: DFS-VISIT(G, w)

Nach diesen Vorarbeiten kommen wir auf unser Artikulationsknotenproblem zurück. Einen Artikulationsknoten kann es nur dann geben, wenn der Graph zusammenhängend ist. Dies können wir in der Routine DFS(G, s) leicht testen: nach Rückkehr von DFS-VISIT(G, s) muss `dfs`$[v] > 0$ für alle

[2]Diese DFS-Nummern entsprechen fast genau den Pre-Nummern aus dem letzten Semester. Allerdings waren diese mit den Post-Nummern verwoben, wodurch die Pre-Nummern eine Teilmenge von $\{1, \ldots, 2|V|\}$ bilden. Die DFS-Nummern bilden dagegen die Menge $\{1, \ldots, |V|\}$. Die daraus resultierende *Sortierung* der Knoten ist jedoch dieselbe.

Knoten $v \in V$ gelten. Wir nehmen im Folgenden an, dass dies gilt und stellen uns die Frage, wie wir einem Knoten $v \in V$ ansehen können, ob er ein Artikulationsknoten ist.

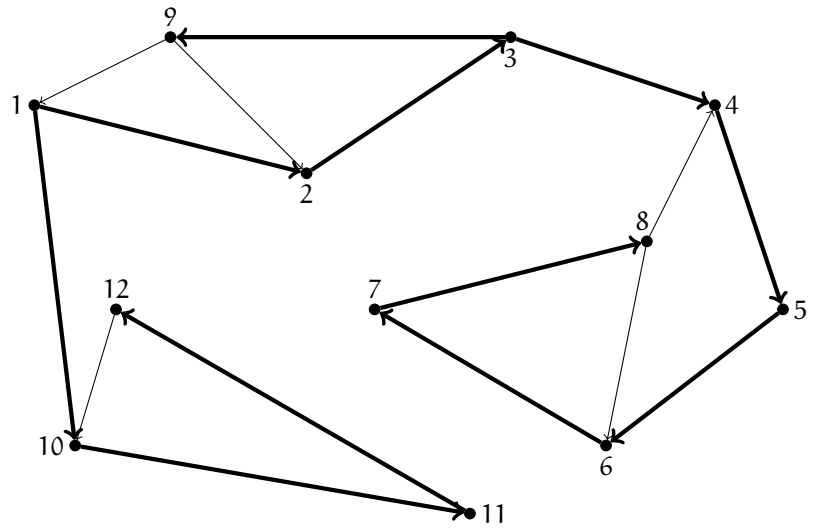
Für den Startknoten s ist dies einfach: er ist genau dann ein Artikulationsknoten, wenn s in dem Tiefensuchbaum T Grad mindestens zwei hat. (Den einfachen Beweis hierfür überlassen wir dem Leser.) Für einen Knoten $v \neq s$ ist dies etwas komplizierter. Um etwas Intuition zu gewinnen richten wir die Kanten des Graphen wie folgt: alle Kanten des Tiefensuchbaum T richten wir von dem Knoten mit der kleineren `dfs`-Nummer zu dem Knoten mit der grösseren `dfs`-Nummer; wir nennen diese Kanten auch *Baumkanten*. Alle übrigen Kanten richten wir umgekehrt: vom Knoten mit der grösseren `dfs`-Nummer zu dem Knoten mit der kleineren `dfs`-Nummer; entsprechend nennen wir diese Kanten *Restkanten*. Im letzten Semester wurden die Restkanten noch weiter in forward-/back-/cross-Kanten unterteilt. Diese Unterteilung ist für uns aber nicht relevant. Ausserdem wurde DFS auch für gerichtete Graphen diskutiert, in welchen die Orientierung der Rest-Kanten komplizierter ist. Man kann sich aber leicht überlegen, dass sich für ungerichtete Graphen die Orientierung der Kanten aus dem Vorsemester zu der hier beschriebenen Orientierung vereinfacht.

Mit Hilfe dieser gerichteten Kanten können wir jedem Knoten neben seiner `dfs`-Nummer eine weitere Nummer zuordnen. Dazu definieren wir für alle Knoten $v \in V$:

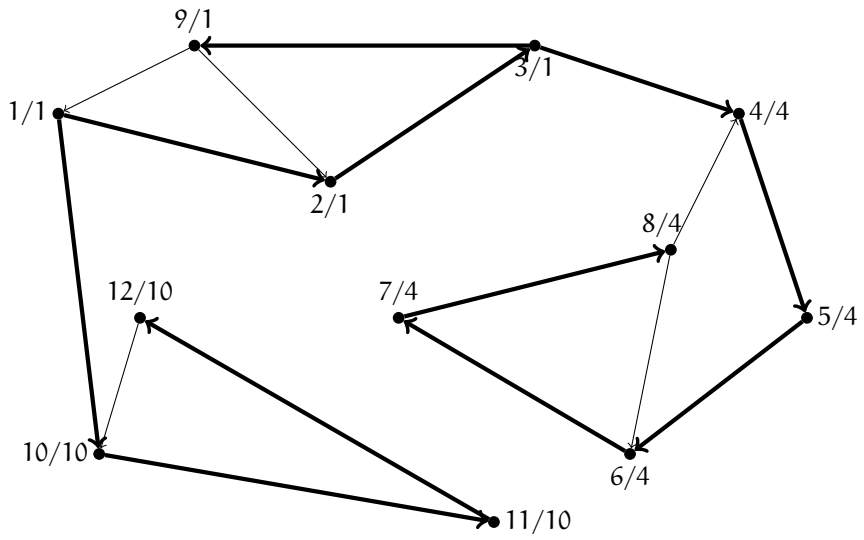
$\texttt{low}[v] :=$ kleinste `dfs`-Nummer, die man von v aus durch einen Pfad aus (beliebig vielen) Baumkanten und maximal einer Restkante erreichen kann.

Beachte, dass aus der Definition folgt: für alle Knoten $v \in V$: $\texttt{low}[v] \leq \texttt{dfs}[v]$, denn diesen Wert erhält man, wenn man den leeren Pfad betrachtet. Abgesehen vom leeren Pfad kann man sich auf Pfade beschränken, bei denen es genau eine Restkante gibt und diese Restkante den Abschluss des Pfades bildet. Denn einen Pfad, der mit einer Baumkante endet, kann man immer durch Weglassen dieser letzten Baumkante verbessern, da eine Baumkante ja die `dfs`-Nummer vergrössert.

Beispiel 1.26. Um diese Definition zu veranschaulichen, betrachten wir ein Beispiel. Die folgende Abbildung zeigt einen Graphen und ein mögliches Ergebnis einer Tiefensuche, die wir in dem Knoten ganz links starten. Die Zahl an jedem Knoten entspricht dem `dfs`-Wert des Knotens, die fetten Kanten sind die Kanten des gefundenen Spannbaums T:



Die `low`-Werte lassen sich jetzt daraus berechnen. In der nachfolgenden Abbildung sind sie jeweils an zweiter Stelle eingetragen:



Die Artikulationsknoten des Graphen sind die Knoten mit den `dfs`-Nummern $1, 3, 4$ und 10. Für den Startknoten 1 kennen wir schon den Grund: er hat in dem Baum T Grad 2. Für die Knoten $3, 4$ und 10 (und nur für diese!) gilt: im Baum T gibt es einen Nachbarknoten, dessen `low`-Wert mindestens so gross ist wie der `dfs`-Wert des (Artikulations-)Knotens. Für den Knoten 3 ist dies der Knoten 4, für den Knoten 4 der Knoten 5 und für den Knoten 10 der Knoten 11.

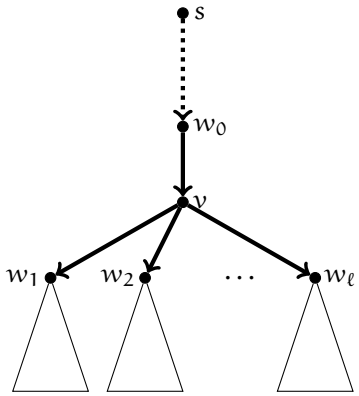
Betrachten wir einen Knoten $v \neq s$. Wenn v ein Artikulationsknoten ist, so besteht $G[V \setminus \{v\}]$ aus mindestens zwei Zusammenhangskomponenten, eine – wir nennen sie Z_1 – die den Startknoten s enthält und mindestens

einer weiteren Z_2. Da jeder Pfad von s zu einem Knoten in Z_2 den Knoten v enthalten muss, wissen wir dass gelten muss:

$$1 = \texttt{dfs}[s] < \texttt{dfs}[v] < \texttt{dfs}[w] \qquad \text{für alle } w \in Z_2.$$

Und da $G[Z_2]$ eine Zusammenhangskomponente in $G[V \setminus \{v\}]$ ist, kann es keine Kanten von einem Knoten w aus Z_2 zu einem Knoten in $V \setminus (\{v\} \cup Z_2)$ geben. Der `low`-Wert eines jeden Knotens in Z_2 ist daher mindestens $\texttt{dfs}[v]$.

Nehmen wir nun an, dass der Knoten $v \neq s$ kein Artikulationsknoten ist. Seien $w_0, w_1, \ldots, w_s$ die Nachbarknoten von v in T. Genau einer dieser Knoten hat einen `dfs`-Wert kleiner als derjenige von v. Ohne Einschränkung sei dies w_0. Der Baum T sieht also wie folgt aus:



Nach Konstruktion des DFS-Algorithmus kann es keine Kante zwischen irgend zwei der an den Knoten $w_1, \ldots, w_\ell$ verankerten Teilbäume geben. Da andererseits nach Annahme v kein Artikulationsknoten ist, muss es aus jedem dieser Bäume eine Restkante zu einem Knoten mit kleinerer `dfs`-Nummer als v geben, denn sonst wäre dieser Baum in $G[V \setminus \{v\}]$ nicht in derselben Komponente wie w_0. Insgesamt haben wir daher gezeigt:

v ist Artikulationsknoten

$\iff$ $v = s$ und s hat in T Grad mindestens zwei, oder
$v \neq s$ und es gibt $w \in V$ mit $\{v, w\} \in E(T)$ und $\texttt{low}[w] \geq \texttt{dfs}[v]$.

Die `low`-Werte kann man effizient berechnen, in dem man diese jeweils mit $\texttt{dfs}[v]$ initialisiert, für jeden Nachfolgerknoten gegebenenfalls anpasst und den kleinsten gefundenen Wert retourniert. Damit ergibt sich dann folgender Algorithmus:

DFS-VISIT(G, v)

```
1: num ← num + 1
2: dfs[v] ← num
3: low[v] ← dfs[v]
4: isArtVert[v] ← FALSE
5: for all {v, w} ∈ E do
6:     if dfs[w] = 0 then
7:         T ← T + {v, w}
8:         val ← DFS-VISIT(G, w)
9:         if val ≥ dfs[v] then
10:            isArtVert[v] ← TRUE
11:        low[v] ← min{low[v], val}
12:    else dfs[w] ≠ 0 and {v, w} ∉ T
13:        low[v] ← min{low[v], dfs[w]}
14: return low[v]
```

Wir müssen dabei beachten, dass die Berechnung von `isArtVert`$[v]$ durch DFS-VISIT(G, v) für den Startknoten $v = s$ falsch sein könnte. Wir müssen die Rahmenprozedur deshalb noch durch eine Abfrage ergänzen, um diesen Sonderfall zu korrigieren:

DFS(G, s)

```
1: ∀v ∈ V :   dfs[v] ← 0
2: num ← 0
3: T ← ∅
4: DFS-VISIT(G, s)
5: if s hat in T Grad mindestens zwei then
6:     isArtVert[s] ← TRUE
7: else
8:     isArtVert[s] ← FALSE
```

Damit erhalten wir folgenden Satz:

Satz 1.27. Für zusammenhängende Graphen $G = (V, E)$, die mit Adjazenzlisten gespeichert sind, kann man in Zeit $O(|E|)$ alle Artikula-

tionsknoten berechnen.

1.4.2 Brücken

Artikulationsknoten sind Zertifikate dafür, dass ein zusammenhängender Graph nicht 2-zusammenhängend ist. Analog hierfür sind Brücken Zertifikate, dass ein zusammenhängender Graph nicht 2-kanten-zusammenhängend ist. Formal definieren wir hierfür: eine Kante $e \in E$ in einem zusammenhängenden Graphen $G = (V, E)$ heisst *Brücke*, wenn der Graph $(V, E \setminus \{e\})$ nicht zusammenhängend ist.

Beispiel 1.26 *(Fortsetzung)* Der Graph aus Beispiel 1.26 enthält zwei Brücken: die Kanten zwischen den Knoten mit `dfs`-Nummern 3 und 4 bzw. 1 und 10.

Aus der Definition einer Brücke folgt sofort, dass ein Spannbaum immer alle Brücken des Graphen enthalten muss. Insbesondere kommen daher als Brücken nur die Kanten eines Tiefensuchbaumes in Frage. Ebenfalls sieht man sofort, dass jeder in einer Brücke enthaltene Knoten entweder ein Artikulationsknoten des Graphen ist oder in G Grad Eins hat. Verwenden wir jetzt wieder die im vorigen Abschnitt eingeführten Nummerierungen `dfs` und `low`, so sieht man leicht ein: eine (gerichtete) Kante (v, w) des Tiefensuchbaumes T ist genau dann eine Brücke, wenn $\texttt{low}[w] > \texttt{dfs}[v]$. Damit lässt sich also der Tiefensuchalgorithmus aus dem vorigen Abschnitt so anpassen, dass er nicht nur alle Artikulationsknoten sondern zudem auch alle Brücken berechnet.

Satz 1.28. Für zusammenhängende Graphen $G = (V, E)$, die mit Adjazenzlisten gespeichert sind, kann man in Zeit $O(|E|)$ alle Artikulationsknoten und alle Brücken berechnen.

1.4.3 Block-Zerlegung

Wie können wir die Berechnung von Artikulationsknoten und Brücken nun algorithmisch nutzen? Hierzu zunächst eine Proposition/Definition des Konzepts *Block*.[3]

[3]Manchmal nennt man Blöcke auch 2-Zusammenhangskompenten. Das ist aber nicht ganz konsequent, denn ein Block kann auch aus einer einzigen Kante bestehen. Diese ist dann automatisch eine Brücke.

Definition/Proposition 1.29. Sei $G = (V, E)$ ein zusammenhängender Graph. Für $e, f \in E$ definieren wir eine Relation durch

$e \sim f :\Longleftrightarrow e = f$ oder es gibt einen gemeinsamen Kreis durch e und f.

Dann ist diese Relation eine Äquivalenzrelation. Wir nennen die Äquivalenzklassen *Blöcke*.

Beweis. Wir müssen uns überlegen, dass die Relation eine Äquivalenzrelation ist. Sie ist offenbar reflexiv und symmetrisch. Die Transitivität ist etwas schwieriger direkt zu zeigen ist. Hier hilft aber der Satz von Menger weiter. Nehmen wir an, dass für $e = \{u_e, v_e\}, f = \{u_f, v_f\}, g = \{u_g, v_g\} \in E$ gilt, dass $e \sim f$ und $f \sim g$. Dann müssen wir zeigen, dass $e \sim g$ gilt. Wir können annehmen, dass e, f, g paarweise verschieden sind, da die Aussage sonst trivial wird. Wir fügen nun einen "Mittelknoten" w_e in die Kante e ein, d.h. w_e ist zu u_e und v_e adjazent, und diese beiden Verbindungen ersetzen die Kante e. Ebenso fügen wir Mittelknoten w_f bzw. w_g in f bzw. g ein. Dann impliziert $e \sim f$, dass es im neuen Graphen zwei intern knotendisjunkte Wege von w_e nach w_f gibt. (Die beiden Abschnitte des Kreises durch e und f.) Ebenso gibt es wegen $f \sim g$ zwei intern knotendisjunkte Wege von w_f nach w_g. Nach dem Satz von Menger gibt es also keinen w_e-w_f-Separator der Grösse 1, und auch keinen w_f-w_g-Separator der Grösse 1. Da man weder w_e von w_f noch w_f von w_g trennen kann, gibt es auch keinen w_e-w_g-Separator der Grösse 1. Wiederum nach dem Satz von Menger heisst das, dass es zwei intern knotendisjunkte Pfade von w_e nach w_g gibt. Damit liegen e und g in G auf einem Kreis. □

Man überlegt sich nun leicht, dass sich zwei Blöcke – wenn überhaupt – nur in einem Artikulationsknoten schneiden können. Wir können für einen Graphen G daher einen Graphen T, die sogenannte *Block-Zerlegung* von G, wie folgt definieren, siehe auch Abbildung 1.11. T ist ein bipartiter Graph mit Knotenmenge $V = A \uplus B$, wobei A die Menge der Artikulationsknoten von G und B die Menge der Blöcke von G ist. (Jeder Block entspricht also einem Knoten in T). Wir verbinden einen Artikulationsknoten $a \in A$ mit einem Block $b \in B$ genau dann, wenn a inzident zu einer Kante in b ist. Dann ist T immer noch zusammenhängend, wenn es T war. Ausserdem ist T kreisfrei, denn jeder Kreis in T liesse sich in einen Kreis in G übersetzen,

der Kanten aus mindestens zwei verschiedenen Blöcken enthält – ein Widerspruch, denn Kanten auf einem Kreis liegen ja per Definition immer im selben Block. Der Graph T ist also ein Baum.

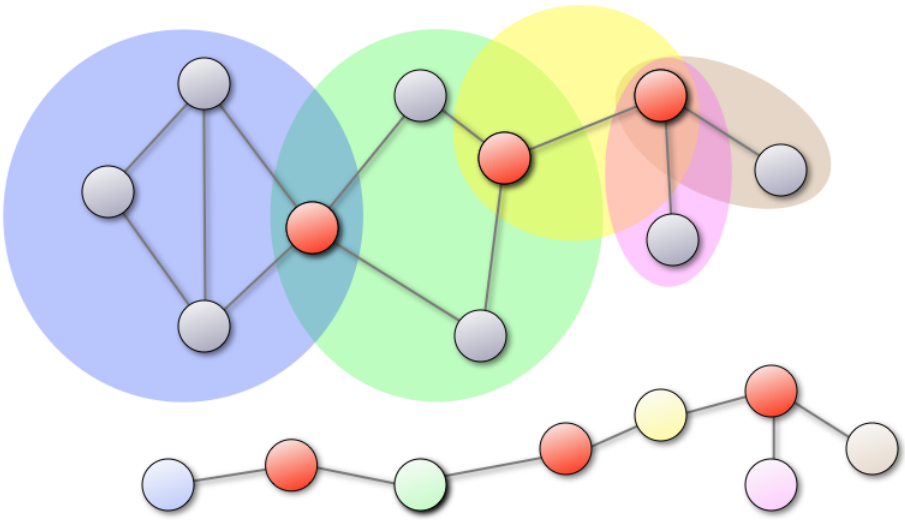


Abbildung 1.11: Ein Graph und seine Block-Zerlegung.

Mit den Algorithmen aus den vorigen Abschnitten lässt sich die Block-Zerlegung eines Graphen in linearer Zeit bestimmen. Eine solche Block-Zerlegung kann trivial sein und nur aus einem einzigen Block bestehen. Wenn sie aber aus mehreren Blöcken besteht, dann hat man algorithmisch viel gewonnen. Die meisten Probleme lassen sich auf natürlich Art und Weise an Artikulationsknoten in Teilprobleme zerlegen, sodass man einen Ansatzpunkt für Divide-and-Conquer oder für Dynamische Programmierung hat. Dynamische Programmierung funktioniert übrigens oft immer noch, wenn man etwas grössere Separatoren hat. Auf manchen Graphklassen hat man immer vergleichsweise kleine Separatoren. So hat zum Beispiel jeder planare Graph einen Separator der Grösse $O(\sqrt{|V|})$, der sich auch effizient bestimmen lässt. Wir verweisen auf weiterführende Vorlesungen zur Graphentheorie für Details. Es ist aber eine gute Übung, sich zu überlegen, wie man klassische Probleme (all-pairs shortest paths, Matchings, Färbbarkeit, ...) in Teilprobleme zerlegen kann, wenn man einen Separator der Grösse $s = 2$ in einem Graphen hat.

1.5 Kreise

In diesem Abschnitt wollen wir zwei Arten von besonderen "Kreisen" in Graphen betrachten: Kreise die alle Kanten bzw. alle Knoten des Graphen genau einmal enthalten.

1.5.1 Eulertouren

Die erste Art eines solchen "Kreises" (wie wir gleich sehen werden ist dieser Kreis genau genommen ein Zyklus, daher die Anführungsstriche) geht auf Euler zurück. LEONHARD EULER (1707–1783) war ein Schweizer Mathematiker. Er gilt als einer der bedeutendsten und produktivsten Mathematiker, die je gelebt haben. Er hat zu zahlreichen Gebieten der Mathematik und Physik bedeutende Beiträge geliefert. Im Jahre 1736 veröffentlichte er eine Arbeit, die sich mit dem Problem beschäftigte, ob es möglich sei, einen Rundgang durch Königsberg zu machen, bei dem man alle Brücken über die Pregel genau einmal überquert. Diese Arbeit gilt als Ursprung der Graphentheorie.

Definition 1.30. Eine *Eulertour* in einem Graphen $G = (V, E)$ ist ein geschlossener Weg (Zyklus), der jede Kante des Graphen genau einmal enthält. Enthält ein Graph eine Eulertour, so nennt man ihn *eulersch*.

Enthält G eine Eulertour, so ist der Grad $\deg(v)$ aller Knoten $v \in V$ gerade, denn aus jedem Knoten geht man genauso oft „hinein" wie „heraus". Für zusammenhängende Graphen gilt sogar die Umkehrung. Hiervon überzeugen wir uns, indem wir einen Algorithmus angeben, der in einem zusammenhängenden Graphen $G = (V, E)$, in dem alle Knotengrade gerade sind, eine Eulertour findet. Hierfür gehen wir wie folgt vor.

Zunächst wählen wir einen beliebigen Knoten $v \in V$ und stellen uns einen Läufer vor, der in v startet. Dieser läuft einen beliebigen Weg W ab und löscht jede benutzte Kante aus dem Graphen. Insbesondere benutzt er also jede Kante höchstens einmal. Der Läufer bleibt erst stehen, wenn er einen isolierten Knoten erreicht, also einen Knoten, von dem keine Kante mehr ausgeht. Wir werden nun zeigen, dass der Weg W ein Zyklus ist, also wieder in v endet. Dazu nehmen wir zwecks Widerspruch an, dass der Weg

in einem Knoten $u \neq v$ enden würde. Nehmen wir weiter an, dass W den Knoten u zuvor schon k-mal besucht hatte, für ein $k \in \mathbb{N}_0$. Dann wurden bei jedem der k Besuche zwei zu u inzidente Kanten gelöscht (eine beim Hinein- und eine beim Hinauslaufen). Ausserdem wurde beim finalen Besuch eine inzidente Kante gelöscht (beim Hereinlaufen). Insgesamt wurden also $2k+1$ zu u inzidente Kanten gelöscht, was eine ungerade Anzahl ist. Da der Grad von u in G, aber gerade war, ist er nach Löschen der Kanten in W noch ungerade, also insbesondere nicht 0. Dies ist aber ein Widerspruch, weil der Läufer ja nur in einem isolierten Knoten enden darf. Also haben wir gezeigt, dass W ein Zyklus ist.

Falls W alle Kanten des Graphen enthält, so ist W die gesuchte Eulertour. Sonst beobachten wir dass im verbleibenden Graphen nach wie vor alle Grade gerade sind, weil es zu jedem Knoten in W eine gerade Anzahl von inzidenten Kanten gibt (möglicherweise 0). Nun suchen wir auf W einen Knoten v', von dem noch mindestens eine Kante ausgeht. Es muss einen solchen Knoten geben, weil sonst die von W besuchten Knoten nicht in derselben Zusammenhangskomponente wie die restlichen Kanten wären, was im Widerspruch zum Zusammenhang von G steht. Wie zuvor lassen wir den Läufer einen Zyklus W' suchen, diesmal aber von v' aus startend. Nach wie vor löscht der Läufer jede besuchte Kante aus dem Graphen. Dann vereinen wir die beiden Zyklen W und W' zu einem neuen Zyklus W_2, indem wir zunächst W bis zum Knoten v', dann den Zyklus W' und dann den Rest des Zyklus W ablaufen. Wiederum gilt: falls W_2 alle Kanten des Graphen enthält, so ist W_2 die gesuchte Eulertour. Falls nicht, finden wir einen Knoten v_2' auf W_2, von dem noch mindestens eine Kante ausgehen, und und verfahren wiederum wie oben.

Zur Realisierung des Algorithmus müssen wir $|E|$-mal von einem gegebenen Knoten eine inzidente Kante finden, und diese Kante anschliessend aus dem Graphen löschen. Mit geeigneten Datenstrukturen sind bei Operationen in Zeit $O(1)$ möglich. Ebenso können wir zwei Zyklen in Zeit $O(1)$ zu einem Zyklus kombinieren, wenn wir sie als verlinkte Listen speichern. Die einzige Schwierigkeit ist es, jeweils einen Knoten v' auf W zu finden, von dem noch Kanten ausgehen. Dazu benutzen wir folgenden Trick. Neben dem bisherigen Läufer, den wir als *schnellen Läufer* S bezeichnen, benutzen wir auch einen *langsamen Läufer* L. Der langsame Läufer startet im selben Knoten v wie der schnelle Läufer. Er bewegt sich jedoch grundsätzlich nur, wenn der schnelle Läufer in einem isolierten Knoten feststeckt. In

Abbildung 1.12: Der erste (schwarz) und zweite (rot) Weg des schnellen Läufers S. Nach dem ersten Weg bleibt der schnelle Läufer stecken. Danach bewegt sich der langsame Läufer entlang der ersten Kante nach v und entdeckt, dass es von dort noch unbenutzte Kanten gibt. Dann läuft der schnelle Läufer von v aus die rote Tour ab und vereinigt die beiden. Anschliessend läuft der langsame Läufer die restliche Tour ab (erst den roten Teil, dann den restlichen schwarzen), entdeckt aber keine weitere ausgehenden Kanten.

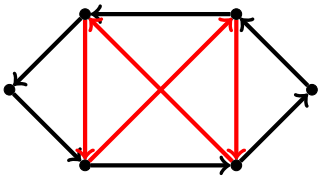


Abbildung 1.13: Die Wege von S und L vereint.

dem Fall läuft der langsame Läufer den Weg W entlang. An jedem Knoten u prüft er, ob noch Kanten von u ausgehen. Falls ja, startet S einen weiteren Zyklus von u ausgehend und aktualisiert W. Dann setzt L seine Suche mit dem aktualisierten Weg W fort. Wir beachten, dass sich der Teil von W, den L schon zurückgelegt hat, durch die Operationen von S nicht ändert, da von diesem Teilweg ja keine Kanten mehr ausgehen. Dies setzen wir fort, bis L den Weg W einmal komplett abgelaufen hat. Danach wissen wir, dass von den Knoten auf W keine Kanten mehr ausgehen, dass W also die finale Eulertour ist. Die zusätzliche Zeit, die wir für L aufwenden müssen, ist $O(|E|)$, da er ja jede Kante im Graphen genau einmal abläuft. Insgesamt läuft der Algorithmus damit in Zeit $O(|E|)$. In diesem Beispielcode sieht man in $\textsc{RandomTour}_G$, wie aus dem Graphen die Kanten nach und nach entfernt werden:

EULERTOUR(G, v_{start})

1: // *Schneller Läufer*
2: $W \leftarrow$RANDOMTOUR$_G(v_{start})$
3: // *Langsamer Läufer*
4: $v^{langsam} \leftarrow$ Startknoten von W.
5: **while** $v^{langsam}$ ist nicht letzter Knoten in W **do**
6: $v \leftarrow$ Nachfolger von $v^{langsam}$ in W
7: **if** $N_G(v) \neq \emptyset$ **then**
8: $W' \leftarrow$ RANDOMTOUR$_G(v)$
9: // *Ergänze* $W = W_1 + \langle v \rangle + W_2$ *um die*
10: // *Tour* $W' = \langle v'_1 = v, v'_2, \ldots, v'_{\ell-1}, v'_\ell = v \rangle$
11: $W \leftarrow W_1 + W' + W_2$
12: $v^{langsam} \leftarrow$ Nachfolger von $v^{langsam}$ in W
13: **return** W

RANDOMTOUR$_G(v_{start})$

1: $v \leftarrow v_{start}$
2: $W \leftarrow \langle v \rangle$
3: **while** $N_G(v) \neq \emptyset$ **do**
4: Wähle v_{next} aus $N_G(v)$ beliebig.
5: Hänge v_{next} an die Tour W an.
6: $e \leftarrow \{v, v_{next}\}$
7: Lösche e aus G.
8: $v \leftarrow v_{next}$
9: **return** W

Wir haben damit den folgenden Satz gezeigt.

Satz 1.31. a) Ein zusammenhängender Graph $G = (V, E)$ ist genau dann eulersch, wenn der Grad aller Knoten gerade ist.
b) In einem zusammenhängenden, eulerschen Graphen kann man eine Eulertour in Zeit $O(|E|)$ finden.

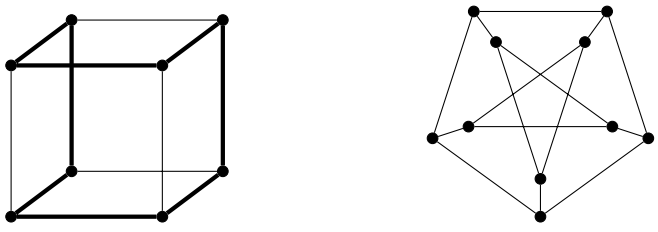
Dieser Satz gilt generell auch für Multigraphen, da der obige Algorithmus analog auch auf Multigraphen funktioniert.

Damit ist über Eulertouren eigentlich alles gesagt: Wir wissen, wie wir einem Graphen ansehen können, ob er eine solche Tour enthält. Und wie wir die Tour, wenn sie denn existiert, effizient finden können. Damit können wir uns dem zweiten Problem zuwenden: der Suche nach einem Kreis, der jeden Knoten des Kreises genau einmal enthält. Wie wir gleich sehen werden, ist dieses Problem ungleich schwieriger.

1.5.2 Hamiltonkreise

In diesem Abschnitt interessieren wir uns für Kreise (diesmal wirklich für Kreise!), die jeden Knoten des Graphen genau einmal enthalten. Als Anwendungsbeispiel kann man sich hier ein Rechnernetz vorstellen, in dem man im Hintergrund einen Prozess laufen lassen möchte, der zyklisch alle Knoten durchläuft und so für einen Informationsaustausch oder eine Synchronisation der Rechner sorgt. Gesucht ist hierfür ein Kreis, der alle Knoten des Graphen enthält. Erstmals wurde dieses Problem von WILLIAM HAMILTON (1805–1865) betrachtet. Hamilton war ein irisches Universalgenie: In seiner Jugend entwickelte er den Ehrgeiz, genauso viele Sprachen zu sprechen wie er Jahre alt war (dies hat er angeblich bis zu seinem siebzehnten Lebensjahr durchgehalten), schon vor Beendigung seines Studiums wurde er zum Königlichen Irischen Astronomen ernannt. Die nach ihm benannten Kreise untersuchte er im Zusammenhang mit einem von ihm entwickelten Geschicklichkeitsspiel.

Definition 1.32. Ein *Hamiltonkreis* in einem Graphen $G = (V, E)$ ist ein Kreis, der alle Knoten von V genau einmal durchläuft. Enthält ein Graph einen Hamiltonkreis, so nennt man ihn *hamiltonsch*.



Beispiel 1.33. Nicht jeder Graph enthält einen Hamiltonkreis. Für den Würfel H_3 ist es nicht schwer einen Hamiltonkreis zu finden (durch fette Kanten symbolisiert). Der rechts dargestellte, sogenannte *Petersen-Graph*, benannt nach dem dänischen Mathematiker JULIUS PETERSEN (1839–1910), enthält andererseits keinen Hamiltonkreis.

Ein klassisches Anwendungsbeispiel ist das Problem des Handlungsreisenden (engl. Traveling Salesman Problem). Ein Vertreter möchte zum Besuch seiner Kunden seine Rundreise möglichst effizient organisieren. Stellt man sich hier die zu besuchenden Städte als Knoten eines Graphen vor und verbindet zwei Städte, wenn es eine direkte Flugverbindung zwischen diesen beiden Städten gibt, so gibt es genau dann eine Rundreise durch alle Städte, bei der keine Stadt mehrfach besucht wird, wenn der Graph einen Hamiltonkreis enthält. (Für realistischere Varianten dieses Problems versieht man die Kanten des Graphen beispielsweise noch mit Kosten und wird dann nach einem Hamilitonkreis suchen, der die Gesamtkosten minimiert.)

Ein exponentieller Algorithmus

Wie sieht es mit Algorithmen für das Finden eines Hamiltonkreises aus? Dies stellt sich als schwieriges Problem heraus. RICHARD KARP hat 1972 bewiesen, dass das zugehörige Entscheidungsproblem „*Gegeben ein Graph* $G = (V, E)$*, enthält* G *einen Hamiltonkreis?*“ $\mathcal{NP}$-vollständig ist. Auf die Definition und Bedeutung des Begriffs $\mathcal{NP}$-Vollständigkeit kann hier nicht weiter eingegangen werden (das ist Thema einer Einführungsvorlesung zur Theoretischen Informatik). Gesagt sei hier nur, dass dies impliziert, dass man davon ausgeht[4], dass es keinen Algorithmus gibt, der entscheidet, ob ein Graph hamiltonsch ist und dessen Laufzeit polynomiell in der Grösse des Graphen (also in der Anzahl Knoten und Kanten) ist.

Ob ein Graph einen Hamiltonkreis enthält, lässt sich natürlich dadurch testen, dass man für jeden möglichen Hamiltonkreis (davon gibt es $\frac{1}{2}(n-1)!$ viele – Übung!) nachsieht, ob alle für ihn notwendigen Kanten in dem Graphen vorhanden sind. Statt die Details hierfür auszuführen, geben wir im Folgenden einen Algorithmus an, dessen Komplexität “nur” exponentiell in der Anzahl Knoten des Graphen ist[5]. Hierfür kommt einmal mehr das

[4]In der Vorlesung *Theoretische Informatik* werden Sie die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ kennen lernen. $\mathcal{P}$ steht für Probleme, die man in polynomieller Laufzeit (in der Grösse der Eingabe) lösen bzw. entscheiden kann. $\mathcal{NP}$-vollständige Probleme sind die “schwierigsten” Probleme in der Klasse $\mathcal{NP}$. Man geht allgemein davon aus, dass $\mathcal{P} \neq \mathcal{NP}$, aber dies wurde noch nicht bewiesen.

[5]Zur Einordnung: Es gilt $k! \geq (k/2)^{k/2}$ für alle $k \in \mathbb{N}$. Im Binärsystem ist die Stellenzahl von $k!$ daher $\log_2(k!) \geq k/2 \log(k/2) = \Theta(k \log k)$. Dagegen hat 2^k nur k Stellen im Binärsystem. Die Stellenzahl von $k!$ ist also asymptotisch grösser als die von 2^k. Das bedeutet, dass $\frac{1}{2}(n-1)!$ asymptotisch viel(!) grösser ist als 2^n. Zum Vergleich: n^2 hat 'nur'

Prinzip der dynamischen Programmierung zum Einsatz.

Sei $G = (V, E)$ ein Graph, wobei wir annehmen wollen, das $V = [n]$ gilt. Zunächst überlegen wir uns, dass es genügt zu testen, ob es ein $x \in N(1)$ gibt, für das es einen 1-x-Pfad gibt, der alle Knoten aus $V = [n]$ enthält. In der Tat: Wenn es einen Hamiltonkreis in G gibt, so enthält dieser Kreis den Knoten 1 und eine Kante von 1 zu einem Nachbar $x \in N(1)$. Der Hamitonkreis ohne diese Kante ist dann der gesuchte 1-x-Pfad.

Um zu testen, ob es solch ein $x \in N(1)$ gibt, definieren wir für alle Teilmengen $S \subseteq [n]$ mit $1 \in S$ und für alle $x \in S$ mit $x \neq 1$:

$$P_{S,x} := \begin{cases} 1, & \text{es gibt in G einen 1-}x\text{-Pfad, der genau die Knoten aus S enthält} \\ 0, & \text{sonst.} \end{cases}$$

Dann gilt:

$$\text{G enthält einen Hamiltonkreis} \quad \Longleftrightarrow \quad \exists\, x \in N(1) \text{ mit } P_{[n],x} = 1$$

und wir können die Werte $P_{S,x}$ mit Hilfe der dynamischen Programmierung berechnen. Dazu überlegen wir uns zunächst, dass für die Mengen $S = \{1, x\}$ offenbar gilt: $P_{S,x} = 1$ genau dann, wenn $\{1, x\} \in E$. Damit haben wir die Initialisierung geschafft. Für die Rekursion von Mengen der Grösse s auf Mengen der Grösse $s + 1$, für $s \geq 2$, überlegen wir uns zunächst folgendes: $P_{S,x}$ ist genau dann gleich Eins, wenn es ein $x' \in N(x)$ gibt mit $x' \neq 1$ und $P_{S\setminus\{x\},x'} = 1$, denn ein 1-$x$-Pfad mit Knoten aus S muss ja einen Nachbarn x' von x als vorletzten Knoten enthalten. Und der Pfad von 1 bis zu diesem Knoten x' ist dann ein 1-x'-Pfad, der genau die Knoten aus $S \setminus \{x\}$ enthält. Es gilt also:

$$P_{S,x} = \max\{P_{S\setminus\{x\},x'} \mid x' \in S \cap N(x), x' \neq 1\}.$$

Damit erhalten wir folgenden Algorithmus:

---

doppelt so viele Stellen wie n, der Unterschied zwischen n^2 und n ist also viel geringer.

HAMILTONKREIS ($G = ([n], E)$)

1: // *Initialisierung*
2: **for all** $x \in [n]$, $x \neq 1$ **do**
3: $\quad P_{\{1,x\},x} := \begin{cases} 1, & \text{falls } \{1,x\} \in E \\ 0, & \text{sonst} \end{cases}$
4: // *Rekursion*
5: **for all** $s = 3$ **to** n **do**
6: $\quad$ **for all** $S \subseteq [n]$ mit $1 \in S$ und $|S| = s$ **do**
7: $\quad\quad$ **for all** $x \in S$, $x \neq 1$ **do**
8: $\quad\quad\quad P_{S,x} = \max\{P_{S\setminus\{x\},x'} \mid x' \in S \cap N(x), x' \neq 1\}$.
9: // *Ausgabe*
10: **if** $\exists\, x \in N(1)$ mit $P_{[n],x} = 1$ **then**
11: $\quad$ **return** G enthält Hamiltonkreis
12: **else**
13: $\quad$ **return** G enthält keinen Hamiltonkreis

Satz 1.34. Algorithmus HAMILTONKREIS ist korrekt und benötigt Speicher $O(n \cdot 2^n)$ und Laufzeit $O(n^2 \cdot 2^n)$, wobei $n = |V|$.

Beweis. Die Korrektheit des Algorithmus haben wir bereits gezeigt. Der benötigte Speicherplatz wird dominiert durch die Anzahl Werte $P_{S,x}$ die wir berechnen müssen. Dies sind 2^{n-1} viele Mengen S und jeweils höchstens $n-1$ viele verschiedene Werte für x. (Genau genommen müssen wir uns jeweils nur die Werte für alle Mengen der Grösse s und diejenigen der Grösse $s+1$ merken. Wenn man dies durchrechnet, führt dies zu $O(\sqrt{n} \cdot 2^n)$, aber darauf wollen wir hier nicht eingehen.) Die Laufzeit wird dominiert durch die dreifache for-Schleife. Diese lässt sich wie folgt abschätzen:

$$\sum_{s=3}^{n} \sum_{S \subseteq [n], 1 \in S, |S|=s} \sum_{x \in S, x \neq 1} O(n) = \sum_{s=3}^{n} \binom{n-1}{s-1} \cdot (s-1) \cdot O(n) = O(n^2 2^n),$$

wie behauptet. Hier haben wir verwendet, dass $\sum_{s=0}^{n-1} \binom{n-1}{s} = 2^{n-1}$ ist. □

Verbesserung: exponentielle Laufzeit, aber polynomieller Speicher[6]

Wir haben also einen Algorithmus gesehen, der exponentielle Laufzeit und Speicher benötigt. Da das Problem $\mathcal{NP}$-vollständig ist, wäre es auch höchst überraschend, wenn wir einen Algorithmus mit polynomieller Laufzeit finden könnten. Aber ist der hohe Speicherbedarf ebenfalls notwendig? Tatsächlich ist ein zu hoher Speicherbedarf in der Regel noch verheerender als eine hohe Laufzeit. Für Hamiltonkreise stellt sich heraus, dass man den Speicherbedarf elegant und drastisch verringern kann, indem man *zählt*, wie viele Hamiltonkreise es gibt. Dazu brauchen wir ein zentrales Prinzip aus der Kombinatorik, das *Prinzip der Inklusion und Exklusion*, auch *Siebformel* genannt.

Satz 1.35. *(Siebformel, Prinzip der Inklusion/Exklusion)* Für endliche Mengen $A_1, \ldots, A_n$ $(n \geq 2)$ gilt:

$$\begin{aligned}\left|\bigcup_{i=1}^{n} A_i\right| &= \sum_{l=1}^{n} (-1)^{l+1} \sum_{1 \leq i_1 < \cdots < i_l \leq n} |A_{i_1} \cap \cdots \cap A_{i_l}| \\ &= \sum_{i=1}^{n} |A_i| - \sum_{1 \leq i_1 < i_2 \leq n} |A_{i_1} \cap A_{i_2}| + \sum_{1 \leq i_1 < i_2 < i_3 \leq n} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| \\ &\quad - \cdots + (-1)^{n+1} \cdot |A_1 \cap \cdots \cap A_n|.\end{aligned}$$

Es wäre nicht schwer, wenn auch etwas technisch, Satz 1.35 direkt zu beweisen. Wir werden jedoch später sehen, dass die Siebformel in der obigen Form nur ein Spezialfall von einer allgemeinen Aussage aus der Wahrscheinlichkeitstheorie ist (Satz 2.5). Diese Verallgemeinerung hat einen eleganteren und natürlichen Beweis (Beispiel 2.36), deshalb verzichten wir an dieser Stelle auf den Beweis von Satz 1.35.

Wir illustrieren die grundlegenden Ideen hinter Satz 2.5 jedoch, indem wir die Spezialfälle $n = 2$ und $n = 3$ explizit betrachten. Für $n = 2$ lautet die Formel

$$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|.$$

Diese ist sehr leicht zu begründen: Wenn wir die Elemente aus $A_1 \cup A_2$ zählen wollen, dann können wir erst die Elemente aus A_1 zählen, und dann

[6] Dieser Abschnitt wird in der Vorlesung nicht behandelt und ist daher auch nicht prüfungsrelevant.

die Elemente aus A_2. Wenn wir das tun, haben wir alle Elemente aus $A_1 \cap A_2$ jedoch doppelt gezählt, und müssen diese daher wieder abziehen.

Für $n = 3$ lautet die Formel

$$|A_1 \cup A_2 \cup A_3| = |A_1| + |A_2| + |A_3| - |A_1 \cap A_2| - |A_1 \cap A_3| - |A_2 \cap A_3| + |A_1 \cap A_2 \cap A_3|.$$

Abbildung 1.14 veranschaulicht den Fall $n = 3$. Man überzeuge sich, dass durch die im Satz angegebene Summe die Elemente in jeder der sieben farblich markierten Teilmengen $A_1 \setminus (A_2 \cup A_3), \ldots, A_1 \cap A_2 \cap A_3$ jeweils genau einmal gezählt werden.

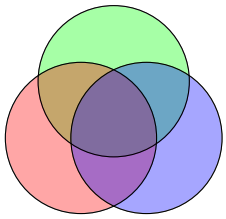


Abbildung 1.14: Illustration zur Inklusion-Exklusion-Formel für $n = 3$.

Wie können wir das Prinzip der Inklusion und Exklusion nun ausnutzen, um Hamiltonkreise zu zählen? Sei dazu wieder $G = (V, E)$ ein Graph mit Knotenmenge $V = [n]$. Erinnern wir uns, dass das Zählen von *Wegen* deutlich einfacher als das Zählen von *Pfaden* ist. Daher wählen wir zunächst einen beliebigen Startknoten $s \in V$ und definieren für alle Teilmengen $S \subseteq [n]$ mit $v \notin S$ die folgende Grösse:

$$W_S := \{\textit{Wege} \text{ der Länge } n \text{ in } G \text{ mit Start- und Endknoten } s, \text{ der keine Knoten in } S \text{ besucht}\}.$$

Zunächst halten wir fest, dass wir die Grösse der Menge W_S effizient bestimmen können: Ist A_S die Adjazenzmatrix des induzierten Teilgraphen $G[V \setminus S]$, dann steht $|W_S|$ im Eintrag (s, s) der Matrix $(A_S)^n$, siehe Satz 1.13. Zur Berechnung von $(A_S)^n$ brauchen wir $n - 1$ Matrixmultiplikationen (oder $O(\log n)$ Matrixmultiplikation durch iteriertes Quadrieren), und jede Matrixmultiplikation können wir mit dem Algorithmus von Strassen in Zeit $O(n^{2.81})$ ausführen. Insgesamt brauchen wir also Zeit $O(n^{3.81})$ (oder $O(n^{2.81} \log n)$) und Speicher $O(n^2)$.

Um die Verbindung zu Hamiltonkreisen herzustellen, beobachten wir für $S = \emptyset$, dass die Menge $W_\emptyset$ jeden Weg der Länge n mit Start- und

Endknoten s enthält. Insbesondere sind also alle Hamiltonkreise darin enthalten – sogar je zweimal, denn man kann jeden Hamiltonkreis in beiden Richtungen ablaufen. Die Menge $W_\emptyset$ enthält aber noch viele ungewünschte Wege, nämlich Wege der Länge n, die nicht alle Knoten besuchen. Diese ungewünschten Wege können wir beschreiben als $\bigcup_{i=1}^{n} W_{\{i\}}$, denn die Menge $W_{\{i\}}$ enthält ja gerade die Wege, die den Knoten i nicht enthalten. Wir halten also fest:

$$\begin{aligned}\text{Zahl der Hamiltonkreise} &= \frac{1}{2}\left(|W_\emptyset| - \left|\bigcup_{i=1}^{n} W_{\{i\}}\right|\right)\\ &= \frac{1}{2}\left(|W_\emptyset| + \sum_{l=1}^{n}(-1)^l \sum_{1\le i_1<\cdots<i_l\le n} |W_{\{i_1\}} \cap \cdots \cap W_{\{i_l\}}|\right)\\ &= \frac{1}{2}\left(|W_\emptyset| + \sum_{l=1}^{n}(-1)^l \sum_{1\le i_1<\cdots<i_l\le n} |W_{\{i_1,\ldots,i_l\}}|\right),\end{aligned}$$

wobei wir im zweiten Schritt die Siebformel eingesetzt haben. Im dritten Schritt haben wir $W_{\{i_1\}} \cap \cdots \cap W_{\{i_l\}} = W_{\{i_1,\ldots,i_l\}}$ benutzt, was eine reine Umschreibung ist – in beiden Fällen geht es um die Menge der Wege, die keinen der Knoten $i_1, \ldots, i_l$ enthalten.

Dank dieser Formel können wir leicht einen speichereffizienten Algorithmus zum Zählen von Hamiltonkreisen angeben.

ZÄHLE_HAMILTONKREISE $(G = ([n], E))$

1: $s := 1$. // *willkürlich gewählt*
2: $Z := |W_\emptyset|$.
3: **for all** $S \subseteq [n]$ mit $s \notin S$ und $S \neq \emptyset$ **do**
4: Berechne $|W_S|$. // *mit der Adjazenzmatrix von* $G[V \setminus S]$.
5: $Z := Z + (-1)^{|S|}|W_S|$.
6: $Z := Z/2$.
7: **return** Z // Zahl der Hamiltonkreise in G

Da die for-Schleife über $O(2^n)$ Teilmengen läuft, erhalten wir folgenden Satz.[7]

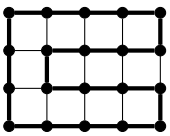
[7] In unserem Modell, dass jede ganze Zahl mit Platz $O(1)$ gespeichert werden kann. Die hier auftretenden Zahlen sind sehr gross. Zum Beispiel ist das Endergebnis für den vollständigen Graphen ja $\frac{1}{2}(n-1)!$, was eine Zahl mit $\Theta(n \log n)$ vielen Stellen ist. Aber auch unter Berücksichtigung dieser grossen Zahlen bleibt der Speicherbedarf polynomiell.

Satz 1.36. Der Algorithmus ZÄHLE_HAMILTONKREISE berechnet die Zahl der Hamiltonkreise in $G = (V, E)$ und benötigt Speicher $O(n^2)$ und Laufzeit $O(n^{2.81} \log n \cdot 2^n)$, wobei $n = |V|$.

1.5.3 Spezialfälle

Erinnern wir uns: Ob ein Graph eine Eulertour enthält kann man einfach dadurch entscheiden, dass man sich die Grade aller Knoten ansieht. Für das Problem des Hamiltonkreises ist nichts ähnliches bekannt und (da das Problem $\mathcal{NP}$-vollständig ist) geht man auch davon aus, dass es keine ähnliche schöne und elegante Charakterisierung gibt. Für einige Spezialfälle gibt es eine Charakterisierung jedoch sehr wohl. In diesem Abschnitt wollen wir einige solche Ergebnisse vorstellen. Als erstes betrachten wir Gittergraphen $M_{m,n}$.

Beispiel 1.37. Wann ist ein Gittergraph $M_{m,n}$ hamiltonsch? – Betrachten wir zunächst ein Beispiel. Für das 4×5 Gitter ist ein Hamiltonkreis schnell gefunden:

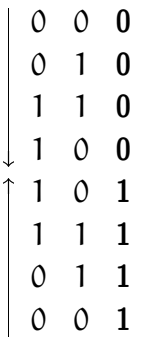


Schnell sieht man auch ein, dass man auf ähnliche Weise immer dann einen Hamiltonkreis in $M_{m,n}$ finden kann, falls m oder n gerade ist. Falls hingegen sowohl m als auch n ungerade, so gibt es keinen Hamiltonkreis. Dies können wir wie folgt einsehen. Sei $[m] \times [n]$ die Knotenmenge des $m \times n$ Gitters. Dann sind zwei Knoten (i, j) und (k, ℓ) genau dann benachbart, falls $|i-k|+|j-\ell| = 1$. Bezeichnen wir $i+j \bmod 2$ die *Parität* des Knoten (i, j), folgt sofort, dass benachbarte Knoten unterschiedliche Parität haben. Damit ergibt sich: In einem Weg gerader Länge haben Anfangs- und Endpunkt gleiche Parität, in einem Weg ungerader Länge ist die Parität unterschiedlich. Betrachte nun einen Hamiltonkreis, falls denn so einer existiert. Er hat die Länge mn. Ein solcher Kreis ist ein Weg der einerseits im gleichen Knoten endet wie er beginnt, andererseits, ist mn ungerade, ist die Parität von Anfangs- und Endknoten unterschiedlich – offensichtlich ein Widerspruch.

In obigem Beispiel haben wir ein sogenanntes *Paritätsargument* verwendet. Dieses können wir noch verallgemeinern. Den (einfachen) Beweis hierfür überlassen wir dem Leser.

Lemma 1.38. Ist $G = (A \uplus B, E)$ ein bipartiter Graph mit $|A| \neq |B|$, so kann G keinen Hamiltonkreis enthalten. □

Als nächstes betrachten wir einen d-dimensionalen Hyperwürfel H_d. Wir erinnern uns: Die Knotenmenge von H_d ist $\{0,1\}^d$, also die Menge aller 0-1-Folgen der Länge d. Und zwei Knoten sind genau dann durch eine Kante verbunden, wenn ihre beiden 0-1-Folgen sich an genau einer Stelle unterscheiden. Für $d = 3$ haben wir bereits auf Seite 47 gesehen, dass H_3 einen Hamiltonkreis enthält. Für $d = 2$ ist das ebenfalls leicht einzusehen, da ein H_2 nichts anderes ist als ein Kreis der Länge vier. Es gilt aber ganz allgemein: Der d-dimensionale Hyperwürfel enthält für jedes $d \geq 2$ einen Hamiltonkreis. Wir zeigen dies durch Induktion über d. Für $d = 2, 3$ haben wir dies bereits gezeigt. Für den Induktionsschritt $d \Rightarrow d+1$ gehen wir wie folgt vor. Zunächst betrachten wir alle Knoten im $(d+1)$-dimensionalen Hyperwürfel, für welche die letzte Koordinate 0 ist. Diese Punkte induzieren (indem man die letzte Koordinate "ignoriert") einen d-dimensionalen Hyperwürfel und wir wissen daher nach der Induktionsannahme, dass es einen Kreis durch alle diese Punkte gibt. Statt diesem Kreis betrachten wir jetzt nur einen Pfad durch alle diese Knoten, und zwar jenen, der im Knoten $00\ldots0$ des H_d beginnt. Schreiben wir diesen Pfad jetzt zweimal hin, und zwar einmal *vorwärts* und einmal *rückwärts*, so können wir die erste Kopie hinten um eine Null ergänzen und die zweite Kopie durch eine Eins, und erhalten so einen Pfad von $0\ldots00$ nach $0\ldots01$ durch alle Knoten des H_{d+1}. Und da die beiden Knoten $0\ldots00$ und $0\ldots01$ benachbart sind, ist dies dann ein Hamiltonkreis für H_{d+1}. Das folgende Schaubild illustriert diese Konstruktion für $d = 2$:



Einen Hamiltonkreis im d-dimensionalen Hyperwürfel kann man auch benutzen, um eine Codierung der ersten 2^d natürlichen Zahlen zu erhalten, die eine oftmals sehr nützliche zusätzliche Eigenschaft haben.

Beispiel 1.39. Üblicherweise codiert man natürliche Zahlen durch den sogenannten Binärcode, dem die Darstellung einer Zahl als Summe von Zweierpotenzen zugrunde liegt: 45 codiert man wegen $45 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$ beispielsweise als 101101. Für manche Anwendungen hat der Binärcode jedoch einige gravierende Nachteile. Überträgt man beispielsweise die einzelnen Ziffern der Zahl parallel, so können leichte Zeitverschiebungen bei der Übertragung zu seltsamen Effekten führen. Ändert sich beispielsweise der Wert 3 (Binärcode 011) auf den Wert 4 (Binärcode 100), so kann eine zu schnelle Übertragung der führenden Ziffer dazu führend, dass wir kurzzeitig den Wert 7 (Binärcode 111) empfangen. Solche Artefakte verhindert der sogenannte Gray-Code, benannte nach dem amerikanischen Physiker FRANK GRAY (1887-1969), welcher 1953 ein Patent auf dieses Verfahren erhielt. Der Gray-Code ist ein stetiger Code, bei dem sich benachbarte Codewörter nur in einer einzigen binären Ziffer unterscheiden. Dies erreichen wir, indem wir die Zahl i als den i-ten Knoten eines Hamiltonkreises im H_d codieren.

Mit den Gittergraphen und dem d-dimensionalen Hyperwürfel haben wir zwei spezielle Graphklassen kennengelernt, für die wir sehr leicht entscheiden können, ob es einen Hamiltonkreis gibt und, falls ja, einen solchen auch sofort angeben können. Ähnliches gilt für Graphen, die sehr viele Kanten haben.

Satz 1.40 (Dirac). Wenn $G = (V, E)$ ein Graph mit $|V| \geq 3$ Knoten ist, in dem jeder Knoten mindestens $|V|/2$ Nachbarn hat, dann ist G hamiltonsch.

Beweis. Wir beweisen den Satz durch Widerspruch. Nehmen wir also an, es gibt einen Graphen $G = (V, E)$ mit $|V| \geq 3$, in dem jeder Knoten mindestens $|V|/2$ Nachbarn hat, der aber nicht hamiltonsch ist. Zunächst überlegen wir uns, dass G zusammenhängend ist. Dazu betrachten wir zwei beliebige Knoten $x, y \in V$, $x \neq y$, und zeigen, dass es in G einen x-y-Pfad gibt. Wenn $\{x, y\} \in E$, so ist das klar. Ansonsten folgt aus $deg(x), deg(y) \geq |V|/2$, dass $N(x) \cap N(y)$ nicht leer sein kann; es gibt dann also einen x-y-Pfad der Länge zwei.

Nun betrachten wir einen (beliebigen) längsten Pfad P in G. Mit anderen Worten: Wir nehmen an, dass $P = \langle v_1, \ldots, v_k \rangle$ ein Pfad ist und es keinen Pfad in G gibt, der mehr Kanten enthält als der Pfad P. Schnell überlegt man sich, dass alle Nachbarn von v_1 und v_k Knoten des Pfades P sein müssen, denn sonst könnten wir den Pfad P ja verlängern. Als nächstes überlegen wir uns, dass es ein $2 \leq i \leq k$ geben muss mit $v_i \in N(v_1)$ und $v_{i-1} \in N(v_k)$. Dies gilt, da nach Annahme v_1 zu mindestens $|V|/2$ vielen Knoten v_i mit $2 \leq i \leq k$ benachbart ist. Wäre v_k zu keinem der entsprechen-

den Knoten v_{i-1} benachbart, könnte v_k nur zu höchstens $k-1-|V|/2 < |V/2|$ Knoten benachbart sein, im Widerspruch zu unserer Annahme. Es gibt also solch einen Knoten v_i. Und $\langle v_1, v_i, v_{i+1}, \ldots, v_k, v_{i-1}, v_{i-2}, \ldots, v_2\rangle$ ist dann ein Kreis der Länge k. Für $k = n$ wäre dies ein Hamiltonkreis, den es nach Annahme nicht gibt. Also ist $k < n$ und somit existieren Knoten in V, welche nicht auf diesem Kreis liegen. Da wir aber bereits wissen, dass G zusammenhängend ist, wissen wir auch, dass mindestens einer der Knoten dieses Kreises mit einem solchen Knoten $\hat{v}$ ausserhalb des Kreises benachbart sein muss – und wir erhalten daher einen Pfad der Länge $k+1$, indem wir von $\hat{v}$ zum Kreis laufen und von dort dann einmal den Kreis entlang. Da es aber, nach Wahl von P, in G keine Pfade der Länge $k+1$ geben kann, haben wir unseren gewünschten Widerspruch. □

Unser Widerspruchsbeweis zeigt: Jeder Graph $G = (V, E)$ mit minimalem Grad $|V|/2$ ist hamiltonsch. Wie aber findet man einen Hamiltonkreis effizient? – Dazu überlegen wir uns zunächst, dass unser Beweis ja eigentlich recht konstruktiv war. Denn wir haben ein Verfahren angegeben, mit dem wir in Zeit $O(|V|)$ aus einem Pfad der Länge $k < n$ einen Pfad der Länge $k+1$ machen können bzw. aus einem Pfad der Länge n einen Hamiltonkreis. Beginnen wir daher mit einem beliebigen Pfad der Länge 1 (einer Kante), so erhalten wir auf diese Weise nach $O(|V|^2)$ Schritten einen Hamiltonkreis.

Beispiel 1.41. Die Schranke $|V|/2$ aus Satz 1.40 ist bestmöglich, wie das folgende Beispiel zeigt. Nehmen wir an, dass $|V|$ ungerade ist. Wir partitionieren V in V_1 und V_2 mit $|V_1| = (|V|-1)/2$ und $|V_2| = (|V|+1)/2$ und betrachten den vollständig bipartiten Graphen $G = (V_1 \uplus V_2, V_1 \times V_2)$. Dann hat G minimalen Grad $(|V|-1)/2$, kann aber keinen Hamiltonkreis enthalten, da jeder Kreis in G abwechselnd Knoten aus V_1 und V_2 enthalten muss, V_1 aber nur $(|V|-1)/2$ viele Knoten enthält. (Die Konstruktion für $|V|$ gerade ist ähnlich und sei dem Leser überlassen.)

1.5.4 Das Travelling Salesman Problem

Eine Verallgemeinerung der Frage, ob ein Graph hamiltonsch ist, ist das sogenannte Travelling Salesman Problem (oder deutsch: das Problem des Handlungsreisenden).

TRAVELLING SALESMAN PROBLEM

GEGEBEN: ein vollständiger Graph K_n und eine Funktion $\ell : \binom{[n]}{2} \to \mathbb{N}_0$, die jeder Kante des Graphen eine Länge zuordnet

GESUCHT: ein Hamiltonkreis C in K_n mit

$$\sum_{e \in C} \ell(e) = \min\{\sum_{e \in C'} \ell(e) \mid C' \text{ ist ein Hamiltonkreis in } K_n\}.$$

Leicht sieht man ein, dass das Travelling Salesman Problem allgemeiner ist als die Frage ob ein Graph hamiltonsch ist. Dazu definieren wir für einen Graphen $G = (V, E)$ mit $V = [n]$ Knoten eine Gewichtsfunktion ℓ durch

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

Dann gilt offenbar: die Länge eines minimalen Hamiltonkreises in K_n bzgl. der Funktion ℓ ist genau dann Null, wenn G einen Hamiltonkreis enthält. Das Travelling Salesman Problem ist daher sicherlich nicht einfacher zu lösen als die Frage, ob ein Graph hamiltonsch ist. Allerdings erlaubt uns die Formulierung als Optimierungsproblem eine differenziertere Antwort: statt nur JA oder NEIN können wir jetzt auch die Güte einer nicht optimalen Lösung bewerten. Sei dazu wie oben K_n ein vollständiger Graph mit Gewichtsfunktion $\ell : \binom{[n]}{2} \to \mathbb{N}_0$. Bezeichnen wir mit $\mathrm{opt}(K_n, \ell)$ die Länge einer optimale Lösung, also

$$\mathrm{opt}(K_n, \ell) := \min\{\sum_{e \in C} \ell(e) \mid C \text{ ist ein Hamiltonkreis in } K_n\}$$

so können wir nunmehr jeden Hamiltonkreis in K_n mit der optimalen Lösung vergleichen. Entsprechend spricht man von einem α-Approximationsalgorithmus, wenn der Algorithmus immer einen Hamiltonkreis C findet mit

$$\sum_{e \in C} \ell(e) \leq \alpha \cdot \mathrm{opt}(K_n, \ell).$$

Leider gilt jedoch:

Satz 1.42. Gibt es für ein $\alpha > 1$ einen α-Approximationsalgorithmus für das TRAVELLING SALESMAN PROBLEM mit Laufzeit $O(f(n))$, so gibt es auch einen Algorithmus, der für alle Graphen auf n Knoten in Laufzeit $O(f(n))$ entscheidet, ob sie hamiltonsch sind.

Beweis. Um diesen Satz einzusehen, müssen wir uns nur daran erinnern, dass für die Gewichtsfunktion ℓ, die wir eben eingeführt haben, gilt: $\mathrm{opt}(K_n, \ell) = 0$ genau dann wenn G hamiltonsch ist. Wegen $\alpha \cdot 0 = 0$, für alle $\alpha > 1$, muss ein α-Approximationsalgorithmus daher, wenn G hamiltonsch ist, immer einen Hamiltonkreis in G finden. □

Fügen wir andererseits eine eigentlich sehr natürliche Annahme an die Gewichtsfunktion ℓ hinzu, so ändert sich die Situation grundlegend. Hierfür definieren wir das, wie man sagt, *metrische* Problem des Handlungsreisenden wie folgt:

METRISCHES TRAVELLING SALESMAN PROBLEM

GEGEBEN: ein vollständiger Graph K_n und eine Funktion $\ell : \binom{[n]}{2} \to \mathbb{N}_0$ mit $\ell(\{x,z\}) \leq \ell(\{x,y\}) + \ell(\{y,z\})$ für alle $x, y, z \in [n]$

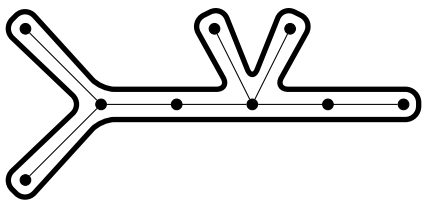
GESUCHT: ein Hamiltonkreis C in K_n mit

$$\sum_{e \in C} \ell(e) = \min\{\sum_{e \in C'} \ell(e) \mid C' \text{ ist ein Hamiltonkreis in } K_n\}.$$

Die Bedingung $\ell(\{x,z\}) \leq \ell(\{x,y\}) + \ell(\{y,z\})$ nennt man auch *Dreiecksungleichung*. Sie besagt anschaulich: die direkte Verbindung zwischen zwei Knoten x und z darf nicht länger sein als der "Umweg" über einen Knoten y.

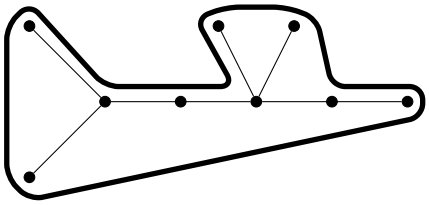
Satz 1.43. Für das METRISCHE TRAVELLING SALESMAN PROBLEM gibt es einen 2-Approximationsalgorithmus mit Laufzeit $O(n^2)$.

Beweis. Wir berechnen zunächst einen minimalen Spannbaum; dies geht in Zeit $O(n^2)$, vgl. Satz 1.19. Anschliessend verwenden wir diesen Spannbaum um daraus einen Hamiltonkreis zu berechnen. Die folgende Zeichnung illustriert dies:



Wir laufen also den Baum 'aussen rum' ab. Dabei wird jede Kante zwei Mal durchlaufen. Die Gesamtlänge des entsprechenden (geschlossenen) Weges

ist daher $2\text{mst}(K_n, \ell)$, wobei $\text{mst}(K_n, \ell)$ die Länge eines minimalen Spannbaum für den K_n mit Gewichtsfunktion ℓ sei. Als nächstes laufen wir diesen Weg nochmals ab, lassen dabei aber alle Knoten aus, die wir bereits besucht haben. Laufen wir zum Beispiel bei dem Knoten links oben los, so erhalten wir dadurch den folgenden Hamiltonkreis:



Nach Annahme erfüllt ℓ die Dreiecksungleichung. Daher wird die Länge des Weges durch das Auslassen von Knoten sicher nicht länger (vielleicht sogar kürzer, aber das interessiert uns hier nicht). Wir erhalten somit auf diese Weise einen Hamiltonkreis mit Länge *höchstens* $2\text{mst}(K_n, \ell)$.

Aus jedem Hamiltonkreis wird durch Weglassen einer beliebigen Kante ein Spannbaum. Daher gilt für die Länge $\text{opt}(K_n, \ell)$ eines minimalen Hamiltonkreises:

$$\text{mst}(K_n, \ell) \leq \text{opt}(K_n, \ell).$$

Insbesondere gilt daher für die Länge $\ell(C) = \sum_{e \in C} \ell(e)$ des von uns gefunden Hamiltonkreises C, dass

$$\ell(C) \leq 2\text{opt}(K_n, \ell).$$

Wir müssen uns nun noch überlegen, wie wir unser obiges anschauliche Argument algorithmisch realisieren können. Dies ist verblüffend einfach: wir verdoppeln einfach jede Kante des minimalen Spannbaums. Dadurch erhält jeder Knoten geraden Grad und wir können daher in Zeit $O(n)$ eine Eulertour finden (vgl. Satz 1.31); diese entspricht dem Weg aus unserer ersten Zeichnung. Ausgehend von einem beliebigen Knoten durchlaufen wir anschliessend die Eulertour (die nach Konstruktion aus genau $2(n-1)$ Kanten besteht); dabei merken wir uns für jeden Knoten, ob wir ihn schon durchlaufen haben. Immer wenn wir einen Knoten treffen, in dem wir schon einmal waren, lassen wir diesen Knoten aus (formal: wir ersetzen die beiden Kanten vor und nach diesem Knoten durch die entsprechende direkte Verbindung). Auf diese Weise erhalten wir dann, wiederum in Zeit $O(n)$, einen Hamiltonkreis mit Länge $\leq 2\text{opt}(K_n, \ell)$,. □

Im nächsten Abschnitt werden wir sehen, dass wir – mit einem zusätzlichen Trick – den Faktor 2 aus Satz 1.43 sogar durch $3/2$ ersetzen können.

1.6 Matchings

Betrachten wir das folgende Zuordnungsproblem. Gegeben ist eine Menge von Rechnern mit verschiedenen Leistungsmerkmalen (Speicher, Geschwindigkeit, Plattenplatz, etc.) und eine Menge von Jobs mit unterschiedlichen Leistungsanforderungen an die Rechner. Gibt es eine Möglichkeit, die Jobs so auf die Rechner zu verteilen, dass alle Jobs gleichzeitig bearbeitet werden können? Graphentheoretisch können wir das Problem wie folgt formulieren: Wir symbolisieren jeden Job und jeden Rechner durch einen Knoten und verbinden einen Job mit einem Rechner genau dann, wenn der Rechner die Leistungsanforderungen des Jobs erfüllt. Gesucht ist dann eine Auswahl der Kanten, die jedem Job genau einen Rechner zuordnet und umgekehrt jedem Rechner höchstens einen Job. Eine solche Teilmenge der Kanten nennt man ein Matching des Graphen.

Definition 1.44. Eine Kantenmenge $M \subseteq E$ heisst *Matching* in einem Graphen $G = (V, E)$, falls kein Knoten des Graphen zu mehr als einer Kante aus M inzident ist, oder formal ausgedrückt, wenn

$$e \cap f = \emptyset \quad \text{für alle } e,f \in M \text{ mit } e \neq f.$$

Man sagt ein Knoten v wird von M *überdeckt*, falls es eine Kante $e \in M$ gibt, die v enthält. Ein Matching M heisst *perfektes Matching*, wenn jeder Knoten durch genau eine Kante aus M überdeckt wird, oder, anders ausgedrückt, wenn $|M| = |V|/2$.

Beispiel 1.45. Ein Graph enthält im Allgemeinen sehr viele Matchings. Beispielsweise ist $M = \{e\}$ für jede Kante $e \in E$ ein Matching. Abbildung 1.15. zeigt ein Matching (links) und ein perfektes Matching (Mitte). Nicht jeder Graph enthält jedoch ein perfektes Matching. Für Graphen mit einer ungeraden Anzahl an Knoten ist dies klar. Es gibt aber sogar Graphen mit beliebig vielen Knoten, deren grösstes Matching aus einer einzigen Kante besteht. Als Beispiel, die sogenannten *Sterngraphen* (im Bild rechts), deren Kantenmenge genau aus den zu einem Knoten inzidenten Kanten besteht.

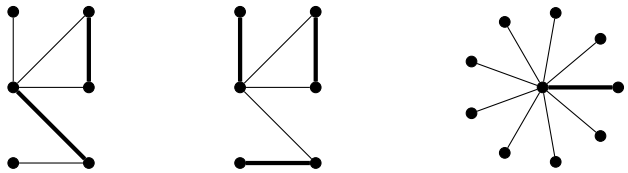


Abbildung 1.15: Matchings

Meist interessiert man sich dafür in einem Graphen ein möglichst grosses Matching zu finden. Hier ist es zunächst wichtig, dass man sich klar macht, dass es zwei verschiedene Arten gibt, wie man "möglichst gross" interpretieren kann.

Definition 1.46. Sei $G = (V, E)$ ein Graph und M ein Matching in G.

- M heisst *inklusionsmaximal*, falls gilt $M \cup \{e\}$ ist kein Matching für alle Kanten $e \in E \setminus M$.
- M heisst *kardinalitätsmaximal*, falls gilt $|M| \geq |M'|$ für alle Matchings M' in G.

(In der englischsprachigen Literatur hat sich die folgende abkürzende Schreibweise eingebürgert: Ein *maximal matching* bezeichnet ein inklusionsmaximales Matching, während ein *maximum matching* ein kardinalitätsmaximales Matching ist.)

Ein inklusionsmaximales Matching hat also die Eigenschaft, dass man keine Kante mehr hinzufügen kann, ohne die Matching-Eigenschaft zu zerstören. Ein solches Matching muss aber nicht unbedingt kardinalitätsmaximal sein. Dies sieht man sehr schön an einem Pfad mit drei Kanten: Das Matching, das nur aus der mittleren Kante besteht ist inklusionsmaximal, aber nicht kardinalitätsmaximal. Ein kardinalitätsmaximales Matching erhält man, wenn man die beiden äusseren Kanten wählt. Ein kardinalitätsmaximales Matching ist immer auch inklusionsmaximal.

1.6.1 Algorithmen

Ein inklusionsmaximales Matching kann man sehr einfach mit einem Greedy-Algorithmus bestimmen:

GREEDY-MATCHING (G)

1: $M \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: wähle eine beliebige Kante $e \in E$
4: $M \leftarrow M \cup \{e\}$
5: lösche e und alle inzidenten Kanten in G

Satz 1.47. Der Algorithmus GREEDY-MATCHING bestimmt in Zeit $O(|E|)$ ein inklusionsmaximales Matching M_{Greedy} für das gilt:

$$|M_{Greedy}| \geq \frac{1}{2}|M_{max}|,$$

wobei M_{max} ein kardinalitätsmaximales Matching sei.

Beweis. Dass M_{Greedy} ein inklusionsmaximales Matching ist, folgt unmittelbar aus der Konstruktion des Matchings: Wenn immer wir eine Kante zum Matching hinzufügen, löschen wir genau alle Kanten aus dem Graphen, die zukünftig nicht mehr zum Matching hinzugefügt werden können. Und wir wiederholen dies, bis keine Kante mehr vorhanden ist.

Um die Ungleichung zu zeigen, betrachten wir die *exklusive Disjunktion* (auch *exklusives Oder* oder *Exor* genannt) $M_{Greedy} \oplus M_{max} := (M_{Greedy} \cup M_{max}) \setminus (M_{Greedy} \cap M_{max})$ der beiden Matchings. Da M_{Greedy} inklusionsmaximal ist, muss jede Kante aus M_{max} mindestens eine Kante aus M_{Greedy} berühren. Da M_{max} ein Matching ist, kann andererseits jeder Knoten aus M_{Greedy} (von denen es $2|M_{Greedy}|$ viele gibt), nur eine Kante von M_{Max} berühren. Aus beiden Fakten zusammen folgt daher $|M_{max}| \leq 2|M_{Greedy}|$, woraus sich die behauptete Ungleichung durch eine elementare Umformung ergibt. □

Im Beweis von Satz 1.47 ist uns erstmals ein Konzept begegnet, das sich bei der Behandlung von Matchings als immens nützlich erwiesen hat: die Betrachtung des Exor von zwei Matchings. Wir wollen dies hier noch etwas genauer ausführen.

Seien M_1 und M_2 zwei beliebige Matchings in einem Graphen G. Was können wir über $M_1 \oplus M_2$ sagen? – Zum einen wissen wir, dass jeder Knoten in dem Graphen $G_M = (V, M_1 \oplus M_2)$ Grad höchstens zwei hat. Die Zusam-

menhangskomponenten des Graphen G_M sind daher alles Pfade und/oder Kreise, wobei alle Kreise gerade Länge haben müssen. (Warum?!).

Nehmen wir nun zusätzlich an, dass $|M_1| < |M_2|$ gilt. Was können wir nun über die Zusammenhangskomponenten in der Vereinigung $M_1 \cup M_2$ sagen? – Wir wissen bereits: Die Zusammenhangskomponenten sind Kreise und/oder Pfade. Jede Komponente, die ein Kreis oder ein Pfad mit gerader Länge ist, enthält jeweils gleich viele Kanten aus M_1 und M_2. Wegen $|M_1| < |M_2|$ muss es daher in $M_1 \cup M_2$ eine Zusammenhangskomponente geben, die ein Pfad P ist, der *mehr* Kanten aus M_2 als aus M_1 enthält. Da sich Kanten aus M_1 und M_2 abwechseln, kann ein solcher Pfad höchstens eine Kante mehr aus M_2 enthalten, nämlich wenn die beiden äusseren Kanten von P (d.h., die erste und die letzte Kante von P) zu M_2 gehören. Ist $|M_2| = |M_1| + k$, muss es sogar mindestens k solche Pfade geben. Wir können einen solchen Pfad P verwenden, um aus dem Matching M_1 ein neues Matching M_1' zu erhalten, das eine Kante mehr enthält. Dazu tauschen wir die Kanten in P: Wir setzen $M_1' := M_1 \oplus P$, wobei wir P hier als Menge von Kanten behandeln. Man sagt daher auch: P ist ein M_1-augmentierender Pfad. Formal ist ein M-augmentierender Pfad (für ein beliebiges Matching M) definiert durch: Die beiden Endknoten von P werden durch M nicht überdeckt (haben also in M Grad Null) und P besteht abwechselnd aus Kanten, die nicht zu M gehören, und Kanten aus M. Insbesondere haben wir damit den folgenden Satz gezeigt.

Satz 1.48 (Satz von Berge). Ist M ein Matching in einem Graphen $G = (V, E)$, das nicht kardinalitätsmaximal ist, so existiert ein augmentierender Pfad zu M.

Beweis. Ist M nicht kardinalitätsmaximal, so gibt es ein grösseres Matching M'. Wie eben schon gezeigt, gibt es in $M \oplus M'$ eine Zusammenhangskomponente, die mehr Kanten aus M' als aus M enthält. Dieser ist ein augmentierender Pfad für M. □

Mit diesen Ideen erhält man wie folgt einen Algorithmus zur Bestimmung eines kardinalitätsmaximalen Matchings: Wir starten mit einem Matching, das aus einer einzigen (beliebigen) Kante besteht. Solange das Matching noch nicht kardinalitätsmaximal ist, gibt es einen augmentierenden Pfad, der es uns erlaubt, das Matching zu vergrössern. Spätestens nach

$|V|/2 - 1$ vielen solcher Schritte ist das Matching dann maximal, denn ein Matching kann ja nicht mehr als $|V|/2$ viele Kanten enthalten. Bleibt die Frage, wie man augmentierende Pfade effizient bestimmen kann. Zumindest für bipartite Graphen geht das recht einfach mit einer modifizierten Breitensuche. Damit erhält man dann einen Algorithmus mit Laufzeit $O((|V| + |E|) \cdot |E|)$. Schauen wir uns dazu die Subroutine genauer an, die zu einem gegebenen Matching in einem bipartiten Graphen einen augmentierenden Pfad findet.

AUGMENTING_PATH $(G = (A \uplus B, E), M)$

```
1:  L_0 := {unüberdeckte Knoten in A}
2:  Markiere alle Knoten aus L_0 als besucht.
3:  if L_0 = ∅ then
4:      return M ist maximal
5:  for all i = 1 to n do
6:      if i ungerade then
7:          L_i := {unbesuchte Nachbarn von L_{i-1} via Kanten in E \ M}
8:      else
9:          L_i := {unbesuchte Nachbarn von L_{i-1} via Kanten in M}
10:     Markiere alle Knoten aus L_i als besucht.
11:     if L_i enthält unüberdeckten Knoten v then
12:         Finde Pfad P von L_0 nach v durch backtracking
13:         return P // terminiert Algorithmus
14: return M ist schon maximal
```

Schauen wir uns die Layers L_i noch etwas genauer an, die der Algorithmus erzeugt, siehe Abbildung 1.16. Zunächst halten wir fest, dass ein augmentierender Pfad immer ungerade Länge hat, denn er hat eine Kante aus $E \setminus M$ mehr als Kanten in M. Deshalb muss solch ein Pfad in bipartiten Graphen immer einen Endpunkt in A und einen Endpunkt in B haben. Deshalb genügt es, mit unüberdeckten Knoten aus A zu starten. Danach wechseln die Layer zwischen den partiten Mengen A und B ab. Knoten aus B erreichen wir daher grundsätzlich über Kanten aus $E \setminus M$, und Knoten aus A über die (eindeutig bestimmte) inzidente Kante aus M. Insbesondere liegen alle unüberdeckten Knoten, die der Algorithmus besucht, im ersten und letzten Layer.

Mit dieser modifizierten Breitensuche kann man daher einen augmentie-

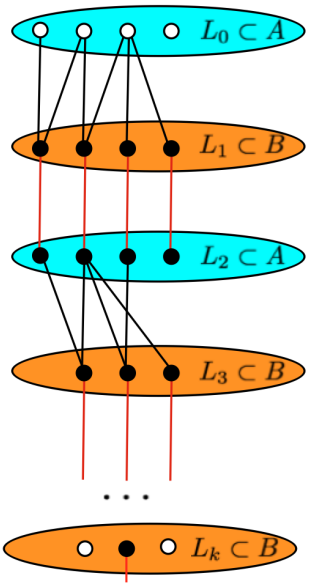


Abbildung 1.16: Der Layer-Graph, der von der Breitensuche zum Auffinden eines augmentierenden Pfades erzeugt wird.

renden Pfad, wenn es denn einen gibt (wenn es keinen gibt, ist das aktuelle Matching kardinalitätsmaximal), in Zeit $O(|E|)$ finden. Da wir höchstens $|V|/2$ oft augmentieren müssen, erhalten wir damit insgesamt einen Algorithmus mit Laufzeit $O(|V||E|)$. Mit einer kleinen Modifikation (und einer genaueren Analyse) kann man die Laufzeit sogar auf $O(\sqrt{|V|}|E|)$ reduzieren. Dies ist der sogenannte Algorithmus von Hopcroft und Karp.

Der Algorithmus von Hopcroft und Karp

Die oben geschilderte Breitensuche hat einen schönen Nebeneffekt. Sie findet nicht nur irgendeinen augmentierenden Pfad, sondern wir können sie auch benutzen um eine (inklusions-) maximale Menge *kürzester* augmentierenden Pfad(e) zu finden. Hierfür stellen wir fest, dass wir die Layer-Struktur mit etwas Glück nicht nur einen augmentierenden Pfad liefert, sondern möglicherweise mehrere, falls im letzten Layer L_k mehrere unüberdeckte Knoten liegen. Dazu wählen wir einen solchen Knoten v_1 aus L_k aus, finden einen augmentierenden Pfad P_1 von L_0 zu v_1, und löschen P_1 aus der Layerstruktur. Danach wählen wir einen weiteren unüberdeckten Knoten v_2 aus L_k (falls vorhanden) und schauen, ob es zu diesem noch einen aug-

mentierenden Pfad gibt. Dies können wir tun, indem wir alle Kanten im Layergraph nach oben richten, und in diesem gerichteten Graphen dann eine Tiefensuche von v_2 aus starten, bis wir auf L_0 stossen oder die Tiefensuche erfolglos abbricht. Falls wir L_0 erreichen, haben wir einen weiteren augmentierenden Pfad P_2 gefunden, der disjunkt zu P_1 ist. Dann löschen wir alle besuchten Knoten aus der Layerstruktur und wiederholen das Ganze, bis wir alle unüberdeckten Knoten aus L_k abgearbeitet haben. Der Gesamtaufwand bleibt bei $O(|V| + |E|)$, weil wir ja jede Kante und jeden Knoten nach Besuch löschen.

Auf diese Weise finden wir nicht nur einen einzelnen Pfad, sondern eine Menge S aus augmentierenden Pfaden mit:

- Alle Pfade in S haben dieselbe minimale Länge k (d.h. k ist die Länge eines kürzesten augmentierenden Pfades).
- Alle Pfade in S sind paarweise disjunkt.
- S ist inklusionsmaximal mit dieser Eigenschaft, d.h. es lässt sich kein weiterer augmentierender Pfad der Länge k zu S hinzufügen, ohne die zweite Bedingung zu verletzen.

Da die Pfade disjunkt sind, können wir M entlang aller dieser Pfade parallel augmentieren, denn ein augmentierender Pfad P ändert den Status überdeckt/unüberdeckt ja nur für die Knoten auf P. Fügen wir diesen kleinen Trick zu unserem Algorithmus hinzu, so erhalten wir den folgenden Algorithmus von Hopcroft und Karp zur Berechnung eines maximalen Matchings in bipartiten Graphen. Wir formulieren den Pseudocode so, dass er gut lesbar ist, auch wenn eine Implementierung mehrere Schritte miteinander verzahnen würde.

MAXIMAL_MATCHING $(G = (A \oplus B, E))$ (Hopcroft und Karp)

```
1: M := {e} für irgendeine Kante e ∈ E.
2: while es gibt noch augmentierende Pfade do
3:     k := Länge eines kürzesten augmentierenden Pfades
4:     Finde eine inklusionsmaximale Menge S von paarweise disjunkten
   augmentierenden Pfaden der Länge k.
5:     for all P aus S do
6:         M := M ⊕ P. // augmentiere entlang der Pfade aus S
7: return M
```

Erstaunlicherweise macht dieser harmlos scheinende Trick den Algorithmus wesentlich schneller.

Satz 1.49. Der Algorithmus von Hopcroft und Karp durchläuft die while-Schleife nur $O(\sqrt{|V|})$ Mal. Er berechnet daher ein maximales Matching in einem bipartiten Graphen in Zeit $O(\sqrt{|V|} \cdot (|V| + |E|))$.

Beweis. Wir haben uns bereits von der Korrektheit überzeugt und skizziert, warum jeder Durchlauf der while-Schleife in Zeit $O(|V|+|E|)$ möglich ist. Die spannende Frage bleibt daher, warum der Algorithmus die while-Schleife nur $O(\sqrt{|V|})$ Mal durchläuft. Dazu sammeln wir einige weitere Erkenntnisse zu augmentierenden Pfaden. Im Folgenden behandeln wir Pfade weiter als Kollektion von Kanten, d.h. mit $|P|$ bezeichnen wir die Zahl der Kanten im Pfad P.

1) Ist M ein Matching, P ein kürzester augmentierender Pfad von M, und P' ein augmentierender Pfad für $M \oplus P$, so gilt

$$|P'| \geq |P| + 2|P \cap P'|.$$

Insbesondere: Augmentieren wir M sukzessive mit *kürzesten* augmentierenden Pfaden, so können die Längen dieser Pfade nicht abnehmen.

Um 1) zu zeigen, schauen wir uns das Matching $\tilde{M} := M \oplus P \oplus P'$ an, das wir nach Augmentieren mit P und P' erhalten. Wir wissen, dass $|\tilde{M}| = |M| + 2$. Deshalb muss $|M \oplus \tilde{M}|$ mindestens zwei disjunkte Pfade enthalten, die augmentierende Pfade für M sind. Aber weil P ein kürzester augmentierender Pfad für M ist, haben diese Pfade je mindestens Länge $|P|$. Andererseits erfüllt die Verknüpfung $\oplus$ die normalen Kommutativitäts- und Assoziativgesetze, daher ist $M \oplus \tilde{M} = M \oplus M \oplus P \oplus P' = P \oplus P'$. Insgesamt ist also $|P \oplus P'| = |M \oplus \tilde{M}| \geq 2|P|$. Schliesslich setzen wir noch ein, dass $|P \oplus P'| = |P \cup P'| - |P \cap P'| = |P| + |P'| - 2|P \cap P'|$ nach Definition von $\oplus$ ist, und erhalten $|P| + |P'| - 2|P \cap P'| = |P \oplus P'| \geq 2|P|$, was nach Umformen 1) ergibt.

2) Mit jedem Durchlaufen der while-Schleife erhöht sich k um mindestens 2.

Dies ist eine Konsequenz von 1), wie wir nun sehen werden. Zunächst betrachten wir, was passiert, während wir M mit den Pfaden der Länge k aus S augmentieren. Nachdem wir mit dem ersten Pfad $P \in S$ augmentiert haben, erhalten wir ein Matching $M' = M \oplus P$. Laut 1) hat dieses keinen augmentierenden Pfad, der echt kürzer als k wäre. Damit sind die weiteren Pfade in S also nicht nur kürzeste augmentierende Pfade für M, sondern auch für M' und induktiv auch für alle weiteren Matchings, die wir in diesem Schritt erhalten. Der Algorithmus von Hopcroft und Karp augmentiert also immer entlang kürzester Pfade, auch in allen Zwischenschritten.

Sei nun $\tilde{M}$ das Matching, das wir erhalten, nachdem wir mit allen Pfaden aus S augmentiert haben. 2) behauptet, dass der kürzeste augmentierende Pfad für $\tilde{M}$ mindestens Länge $k+2$ hat. Sei $\tilde{P}$ ein solcher Pfad. Wir haben gerade schon gesehen, dass $|\tilde{P}| \geq k$ ist. Wir unterscheiden zwei Fälle. Der erste Fall ist, dass $\tilde{P}$ disjunkt ist zu allen Pfaden in S. Da S inklusionsmaximal ist, muss dann $|\tilde{P}| \geq k+1$ sein, und sogar $|\tilde{P}| \geq k+2$, weil augmentierende Pfade ungerade Länge haben. Der andere Fall ist, dass $\tilde{P}$ einen Pfad P aus S in mindestens einem Knoten v schneidet. $\tilde{M}$ überdeckt alle Knoten aus P, da es ja durch Augmentieren mit P entstanden ist. Also ist auch v durch eine Kante $e \in \tilde{M}$ überdeckt, die durch P hinzugefügt wurde. Die Kante e ist also einerseits in P. Andererseits ist auch $e \in \tilde{P}$, weil augmentierende Pfade wie $\tilde{P}$ die Matching-Kanten von allen überdeckten Knoten auf dem Pfad enthalten. Also ist $e \in P \cap \tilde{P}$, und damit $P \cap \tilde{P} \neq \emptyset$. Schliesslich können wir ohne Einschränkung annehmen, dass der Pfad P von allen Pfaden aus S als letztes verarbeitet wurde, weil die Reihenfolge der Pfade aus S ja beliebig ist. Damit sind wir in der Situation von 1), und können folgern, dass $|\tilde{P}| \geq |P| + 2|P \cap \tilde{P}| \geq k+2$ ist. Wir haben in beiden Fällen also $|\tilde{P}| \geq k+2$ gesehen, und damit ist 2) bewiesen.

3) Sei M ein Matching, für das der kürzeste augmentierende Pfad Länge k hat, und M' ein beliebiges anderes Matching. Dann gilt

$$|M'| \leq |M| + \frac{|V|}{k+1}.$$

Um dies zu sehen, nehmen wir an, dass $|M'| > |M|$ ist, da die Aussage sonst trivial ist. Dann wissen wir schon, dass $M \oplus M'$ mindestens $|M'| - |M|$ disjunkte Pfade besitzt, die augmentierende Pfade für M sind. Auf jedem solchen Pfad liegen mindestens $k+1$ Knoten (da er Länge mindestens k

hat). Zusammen liegen auf diesen Pfaden also mindestens $(|M'|-|M|)\cdot(k+1)$ viele Knoten. Andererseits gibt es im Graphen nur $|V|$ viele Knoten. Also ist $(|M'|-|M|)\cdot(k+1) \leq |V|$, woraus 3) durch Umformung folgt.

Aus 1), 2) und 3) können wir nun das Theorem beweisen. Zunächst besagt 2), dass nach den ersten $\lceil\sqrt{|V|}/2\rceil$ Durchläufen der while-Schleife $k \geq \sqrt{|V|}$ gilt. An diesem Punkt besagt 3) aber, dass ein maximale Matching M_{max} erfüllt: $|M_{max}| \leq |M|+|V|/(k+1) \leq |M|+\sqrt{|V|}$, bzw. $|M| \geq |M_{max}|-\sqrt{|V|}$. Mit jedem weiteren Durchlauf der while-Schleife erhöht sich aber die Grösse von M um mindestens 1. Nach spätestens $\sqrt{|V|}$ weiteren Durchläufen erreicht $|M|$ also die Grösse $|M_{max}|$, und der Algorithmus terminiert. □

Weitere Matching-Algorithmen

In Kapitel 3 der Vorlesung werden wir noch weitere, alternative Algorithmen für das Finden von Matchings in bipartiten Graphen kennenlernen. Mit (deutlich) mehr Aufwand kann man auch zeigen, dass man sogar in beliebigen Graphen ein (kardinalitäts-)maximales Matching in Zeit $O(\sqrt{|V|}\cdot(|V|+|E|))$ finden kann. Das überlassen wir aber Spezialvorlesungen. Stattdessen betrachten wir hier noch kurz eine gewichtete Variante des Problems:

Satz 1.50. Ist n gerade und $\ell : \binom{[n]}{2} \to \mathbb{N}_0$ ein Gewichtsfunktion des vollständigen Graphen K_n, so kann man in Zeit $O(n^3)$ ein minimales perfektes Matching finden, also ein perfektes Matching M mit

$$\sum_{e\in M} \ell(e) = \min\{\sum_{e\in M'} \ell(e) \mid M' \text{ perfektes Matching in } K_n\}.$$

Der Beweis dieses Satz sprengt ebenfalls den Rahmen dieser Einführungsvorlesung. Wir wollen hier jedoch noch eine Konsequenz dieses Satzes herleiten.

Satz 1.51. Für das Metrische Travelling Salesman Problem gibt es einen 3/2-Approximationsalgorithmus mit Laufzeit $O(n^3)$.

Beweis. Die Grundidee dieses Algorithmus ist ähnlich zu der Idee des 2-Approximationsalgorithmus aus Satz 1.43: Wir berechnen zunächst einen

minimalen Spannbaum T, dann modifizieren wir diesen, sodass wir in dem neuen Graphen eine Eulertour finden können. In einem letzten Schritt laufen wir dann diese Eulertour ab und verkürzen sie hierbei schrittweise (indem wir Knoten, in denen wir schon waren, auslassen) so lange, bis wir einen Hamiltonkreis haben. Der Unterschied der beiden Algorithmen besteht darin, wie wir die Modifikation des Spannbaums vornehmen, damit der Graph eulersch wird. In Satz 1.43 haben wir einfach jede Kante von T verdoppelt. Nun gehen wir wie folgt vor. Wir bezeichnen mit S die Menge derjenigen Knoten, die in T einen ungeraden Grad haben. Da $|S|$ gerade ist (vgl. Korollar 1.3), gibt es in $K_n[S]$ ein perfektes Matching. Unter allen solchen Matchings wählen wir eines mit minimalem Gewicht. Aus Satz 1.50 wissen wir, dass wir ein solches minimales perfektes Matching, nennen wir es M, in Zeit $O(n^3)$ finden können. Im (Multi-) Graph $T \cup M$ hat dann jeder Knoten geraden Grad und wir können daher, jetzt wieder ganz analog zum Beweis von Satz 1.43, einen Hamiltonkreis C finden mit

$$\ell(C) \leq \ell(M) + \ell(T).$$

Wir wissen bereits, dass $\ell(T) = \mathrm{mst}(K_n, \ell) \leq \mathrm{opt}(K_n, \ell)$. Wir zeigen nun noch, dass $\ell(M) \leq \frac{1}{2}\mathrm{opt}(K_n, \ell)$, woraus dann folgt, dass $\ell(C) \leq \frac{3}{2}\mathrm{opt}(K_n, \ell)$, und der Algorithmus also ein 3/2-Approximationsalgorithmus ist.

Um einzusehen, dass $\ell(M) \leq \frac{1}{2}\mathrm{opt}(K_n, \ell)$ gilt, betrachten wir einen Hamiltonkreis C_{opt} der Länge $\ell(C_{\mathrm{opt}}) = \mathrm{opt}(K_n, \ell)$. Die Knoten aus S zerlegen C_{opt} in $|S|$ viele Pfade. Da ℓ die Dreiecksungleichung erfüllt, können wir jeden dieser Pfade zu einer Kante reduzieren, ohne die Länge des Kreises zu erhöhen. D.h., wir haben jetzt einen Kreis C_s, dessen Knoten genau die Knoten aus S sind und für den gilt: $\ell(C_s) \leq \ell(C_{\mathrm{opt}}) = \mathrm{opt}(K_n, \ell)$. Der Kreis C_s lässt sich als Vereinigung von zwei Matchings (mit jeweils $|S|/2$ Kanten) schreiben: $C_s = M_1 \cup M_2$, wobei mindestens eines dieser beiden Matchings Länge $\leq \frac{1}{2}\ell(C_s)$ haben muss (denn sonst wäre die Summe ja grösser als $\ell(C_s)$). Da wir M als minimales perfektes Matching in $K_n[S]$ gewählt haben, gilt daher $\ell(M) \leq \frac{1}{2}\ell(C_s) \leq \frac{1}{2}\mathrm{opt}(K_n, \ell)$, wie behauptet. □

1.6.2 Der Satz von Hall

Wir erinnern uns: Ein Graph $G = (V, E)$ heisst bipartit, wenn man die Knotenmenge V so in zwei Mengen A und B partitionieren kann, dass alle Kanten in E je einen Knoten aus A und einen Knoten aus B enthalten.

Bipartite Graphen schreiben wir dann entsprechend als $G = (A \uplus B, E)$. Der folgende Satz von PHILIP HALL (1904–1982) gibt eine notwendige und hinreichende Bedingung an, unter der ein Matching in einem bipartiten Graphen existiert, welches alle Knoten einer Partition überdeckt. Zur Formulierung des Satzes führen wir noch eine abkürzende Schreibweise für die *Nachbarschaft einer Knotenmenge* $X \subseteq V$ ein:

$$N(X) := \bigcup_{v \in X} N(v).$$

Satz 1.52 (Satz von Hall, Heiratssatz). Für einen bipartiten Graphen $G = (A \uplus B, E)$ gibt es genau dann ein Matching M der Kardinalität $|M| = |A|$, wenn gilt

$$|N(X)| \geq |X| \quad \text{für alle } X \subseteq A. \tag{1.1}$$

Beweis. Wir beweisen zuerst die notwendige Bedingung (also die „⇒"-Richtung des Satzes). Sei M ein Matching der Kardinalität $|M| = |A|$. In dem durch M gegebenen Teilgraphen $H = (A \uplus B, M)$ hat jede Teilmenge $X \subseteq A$ nach Definition eines Matchings genau $|X|$ Nachbarn. Wegen $M \subseteq E$ gilt daher auch $|N(X)| \geq |X|$ für alle $X \subseteq A$.

Die hinreichende Bedingung (die „⇐"-Richtung) beweisen wir durch Induktion über die Kardinalität $a = |A|$ der Menge A. Für $a = 1$ impliziert die Bedingung (1.1), angewandt auf die Menge $X = A$, dass der (einzige) Knoten in A zu mindestens einer Kante inzident ist. Jede solche Kante ist dann ein Matching, das die Bedingung des Satzes erfüllt. Nehmen wir daher an, dass $a > 1$ und dass die Behauptung für alle bipartiten Graphen mit $|A| < a$ gilt. Wir unterscheiden zwei Fälle. Gilt die Bedingung (1.1) für alle Mengen $\emptyset \neq X \subsetneq A$ sogar mit $>$ statt nur mit $\geq$, so wählen wir eine beliebige Kante $e = \{x, y\}$ des Graphen, fügen e zum Matching hinzu und betrachten den Graphen G', den wir erhalten, wenn wir die Knoten x und y (und alle inzidenten Kanten) löschen. Da die Bedingung (1.1) für alle Mengen $\emptyset \neq X \subsetneq A$ mit $>$ erfüllt war (und wir nur einen Knoten aus der Menge B gelöscht haben), ist die Bedingung (1.1) für den Graphen G' noch immer erfüllt. Aus der Induktionsannahme folgt daher, dass wir die Kante e durch ein Matching M' so ergänzen können, dass dann alle Knoten in A überdeckt werden.

Falls die Bedingung (1.1) nicht für alle Mengen $\emptyset \neq X \subsetneq A$ mit $>$ erfüllt ist, so gibt es mindestens eine Menge $\emptyset \neq X_0 \subsetneq A$, für die $|N(X_0)| = |X_0|$ gilt. Unser Plan ist jetzt, die Induktionsannahme auf die beiden induzierten, knotendisjunkten Graphen $G' = G[X_0 \uplus N(X_0)]$ und $G'' = G[A \setminus X_0 \uplus B \setminus N(X_0)]$ anzuwenden. Die Bedingung (1.1) ist offensichtlich für den Graphen G' erfüllt. Sie gilt aber auch für den Graphen G'', wie wir uns jetzt noch überlegen. Betrachte eine beliebige Menge $X \subseteq A \setminus X_0$. Da (1.1) für den Graphen G gilt, wissen wir

$$|X| + |X_0| = |X \cup X_0| \overset{(1.1)}{\leq} |N(X \cup X_0)| = |N(X_0)| + |N(X) \setminus N(X_0)|.$$

Wegen $|N(X_0)| = |X_0|$ folgt daraus $|X| \leq |N(X) \setminus N(X_0)| = |N(X) \cap (B \setminus N(X_0))|$. D.h., die Nachbarschaft von X in dem Graphen G'' besteht aus mindestens $|X|$ Knoten. Und da dies für alle Mengen $X \subseteq A \setminus X_0$ gilt, ist die Bedingung (1.1) daher für den Graphen G'' erfüllt. Aus der Induktionsannahme folgt daher: Es gibt ein Matching M' in G', das alle Knoten in X_0 überdeckt, und ein Matching M'' in G'', das alle Knoten in $A \setminus X_0$ überdeckt. $M = M' \cup M''$ ist dann ein Matching in G mit $|M| = |A|$. □

Aus dem Satz von Hall folgt unmittelbar, dass k-reguläre bipartite Graphen immer ein perfektes Matching enthalten. Es gilt sogar noch mehr: Wir können die Kantenmenge als Vereinigung von perfekten Matchings schreiben!

Satz 1.53. Sei $G = (A \uplus B, E)$ ein k-regulärer bipartiter Graph. Dann gibt es $M_1, \ldots, M_k$ so dass $E = M_1 \uplus \ldots \uplus M_k$ und alle M_i, $1 \leq i \leq k$, perfekte Matchings in G sind.

Beweis. Wir überlegen uns zunächst, dass jeder k-reguläre bipartite Graph in der Tat ein perfektes Matching enthält. Dazu betrachten wir eine beliebige Menge $X \subseteq A$ und den induzierten Graphen $G' = G[X \cup N(X)]$. Offenbar ist G' auch bipartit; die entsprechende Partition ist gegeben durch $X \uplus N(X)$. Da G k-regulär ist, hat in G' jeder Knoten in X ebenfalls Grad k und jeder Knoten in $N(X)$ Grad höchstens k. Da in bipartiten Graphen die Summe der Grade der Knoten "links" immer gleich der Summe der Grade "rechts" ist, folgt daraus: $|N(X)| \geq |X|$ und die Bedingung (1.1) des Satzes von Hall ist damit erfüllt. Daraus folgt also, dass G ein perfektes Matching M_1 enthält. Entfernen wir dieses, so erhalten wir einen $(k-1)$-regulären Graphen,

der somit ebenfalls ein perfektes Matching M_2 enthält. Fahren wir entsprechend fort, bis der Graph 1-regulär ist (und daher ein perfektes Matching ist), so erhalten wir dadurch die im Satz behauptete Partitionierung in perfekte Matchings. □

Satz 1.53 garantiert, dass ein k-regulärer bipartiter Graph ein perfektes Matching enthält. Aber kann man ein solches effizient finden? Ja, man kann: Es gibt einen Algorithmus, der in Zeit $O(|E|)$ ein solches Matching findet. Wir zeigen dies hier nur für Werte für k, die eine Zweierpotenz sind. Hierfür ist der Algorithmus sogar sehr elegant. Der allgemeine Fall ist deutlich schwieriger.

Satz 1.54. Ist $G = (V, E)$ ein 2^k-regulärer bipartiter Graph, so kann man in Zeit $O(|E|)$ ein perfektes Matching bestimmen.

Beweis. Da G 2^k-regulär und bipartit ist, erfüllt jede Zusammenhangskomponente von G die Euler-Bedingung. Wir können daher in Zeit $O(|E|)$ für jede Komponente eine Eulertour bestimmen (vgl. Satz 1.31). Wir laufen diese Touren jetzt einmal ab, wobei wir jede zweite Kante in der Tour aus dem Graphen entfernen. Der übrig gebliebene Graph ist jetzt 2^{k-1}-regulär - und wir können daher das Verfahren wiederholen. Nach k Iterationen ist der Graph 2^0-regulär – und also ein perfektes Matching.

Um die Laufzeit abzuschätzen gehen wir wie folgt vor. Wir wissen aus Satz 1.31, dass man eine Eulertour in Zeit $O(|E|)$ bestimmen kann. Wollen wir in jeder Zusammenhangskomponente eine Eulertour finden, so sieht man leicht ein, dass das ebenfalls in Gesamtzeit $O(|E|)$ möglich ist. Statt der $O()$-Notation verwenden wir jetzt die folgende Schreibweise: Es gibt ein $C > 0$, sodass der Algorithmus aus Satz 1.31 für jeden eulerschen Graphen $G = (V, E)$ in höchstens $C|E|$ Schritten eine Eulertour pro Komponente bestimmt. Für unseren Ausgangsgraphen benötigt der Algorithmus daher höchstens $C|E|$ Schritte. Dann laufen wir die Eulertouren einmal ab und löschen jede zweite Kante. Dies geht sicherlich in $C'|E|$ Schritten. Nun haben wir einen Graphen mit $|E|/2$ Kanten. In der nächsten Iteration benötigen wir daher nur noch höchstens $C|E|/2$ Schritte, um die Eulertouren zu bestimmen und höchstens $C'|E|/2$ Schritte für das Löschen von jeder zweiten Kante. Allgemein gilt daher: In der i-ten Iteration kann man in $(C + C')|E|/2^{i-1}$ Schritten die Anzahl Kanten um die Hälfte reduzieren.

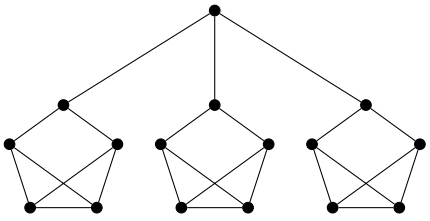
Insgesamt benötigen wir daher höchstens

$$\sum_{i=1}^{k}(C + C')|E|/2^{i-1} \leq \sum_{i\geq 0}(C + C')|E|/2^{i} = 2(C + C')|E|$$

Schritte, um ein perfektes Matching zu bestimmen. □

Es ist verlockend anzunehmen, dass der Algorithmus auch für nicht-bipartite Graphen so funktioniert. Dies ist aber nicht so: Nach Wegnahme jeder zweiten Kante der Eulertour kann es passieren, dass der übrig gebliebene Graph nicht mehr zusammenhängend ist. In bipartiten Graphen ist das nicht weiter schlimm: Wir können den Algorithmus einfach auf jede Zusammenhangskomponente anwenden. Bei nicht-bipartiten Graphen kann es aber für $k = 1$ passieren, dass eine solche Komponente eine ungerade Anzahl an Knoten, und damit auch an Kanten enthält. In dem Fall hat auch die Eulertour eine ungerade Länge, wir können also nicht jede zweite Kante löschen. Da die Komponente eine ungerade Anzahl an Knoten hat, kann auch überhaupt kein perfektes Matching mehr existieren.

Beispiel 1.55. Aus Satz 1.53 wissen wir, dass jeder k-reguläre bipartite Graph ein perfektes Matching enthält. Für nicht-bipartite Graphen gilt dies jedoch nicht immer: Ungerade Kreise oder vollständige Graphen auf ungerade vielen Knoten sind hier einfache Gegenbeispiele. Dass diese Graphen kein perfektes Matching enthalten, sieht man sofort daran, dass sie ungerade viele Knoten enthalten. Für Gegenbeispiele auf gerade vielen Knoten muss man sich etwas mehr Mühe geben. Die folgende Abbildung zeigt ein Beispiel für $k = 3$:



Die hier verwendete Idee lässt sich auch leicht verallgemeinern. Wir erläutern sie noch für $k = 4$ und überlassen alle weiteren Fälle dem Leser. Man wähle sich einen beliebigen Graphen auf sieben Knoten, in dem sechs der Knoten Grad vier haben und einer Grad zwei. Dann nimmt man acht Kopien hiervon und vier weitere Knoten. Die vier neuen Knoten und die acht Knoten vom Grad zwei verbinde man durch einen bipartiten Graphen, in dem die neuen Knoten Grad vier haben, die übrigen acht Knoten jeweils Grad zwei. Und schon haben wir einen 4-regulären Graphen, der kein perfektes Matching haben kann.

1.7 Färbungen

Viele Probleme kann man darauf zurückführen, dass man in einem entsprechend definierten Graphen eine Partition der Knotenmenge findet, sodass Kanten nur zwischen Knoten in verschiedenen Klassen der Partition verlaufen. Im Mobilfunk erhält man so beispielsweise eine Zuordnung von Frequenzen zu Sendern, bei der benachbarte Sender verschiedene Frequenzen benutzen. Im Compilerbau verwendet man diesen Ansatz, um eine Zuordnung von Variablen auf die Register des Prozessors zu finden, sodass gleichzeitig verwendete Variablen in verschiedenen Registern gespeichert werden, und in der Stunden- oder Prüfungsplanung entspricht eine Partition einer Menge von Kursen oder Prüfungen, die gleichzeitig stattfinden können.

Statt von einer Partition der Knotenmenge spricht man meist von einer *Knotenfärbung* bzw. nur kurz *Färbung* des Graphen. In diesem Abschnitt führen wir diese formal ein und zeigen einige grundlegende Eigenschaften.

Definition 1.56. Eine *(Knoten-)Färbung* (engl. *(vertex) colouring*) eines Graphen $G = (V, E)$ mit k Farben ist eine Abbildung $c\colon V \to [k]$, sodass gilt

$$c(u) \neq c(v) \quad \text{für alle Kanten } \{u, v\} \in E.$$

Die *chromatische Zahl* (engl. *chromatic number*) $\chi(G)$ ist die minimale Anzahl Farben, die für eine Knotenfärbung von G benötigt wird.

Beispiel 1.57. Ein vollständiger Graph auf n Knoten hat chromatische Zahl n. Kreise gerader Länge haben chromatische Zahl 2, Kreise ungerader Länge haben chromatische Zahl 3. Bäume auf mindestens zwei Knoten haben immer chromatische Zahl 2. (Warum?!)

Graphen mit chromatischer Zahl k nennt man auch k-*partit* (engl. k-*partite*). Die Motivation für diese Namensgebung sollte klar sein: Ein Graph $G = (V, E)$ ist genau dann k-partit, wenn man seine Knotenmenge V so in k Mengen $V_1, \ldots, V_k$ partitionieren kann, dass alle Kanten Knoten aus verschiedenen Mengen verbinden. Besonders wichtig ist der Fall $k = 2$. In diesem Fall nennt man den Graphen *bipartit*. Der folgende Satz stellt eine einfache, aber wichtige Charakterisierung bipartiter Graphen dar.

Satz 1.58. Ein Graph $G = (V, E)$ ist genau dann bipartit, wenn er keinen Kreis ungerader Länge als Teilgraphen enthält.

Beweis. Die eine Richtung folgt sofort aus Beispiel 1.57: ungerade Kreise haben chromatische Zahl 3, sie können daher in einem bipartiten Graphen nicht enthalten sein. Die andere Richtung sieht man ebenfalls schnell ein. Man startet einfach in einem beliebigen Knoten s eine Breitensuche (ohne Einschränkung sei G zusammenhängend) und färbt einen Knoten genau dann mit Farbe 1 (bzw. 2), wenn sein Abstand $d[v]$ zu s gerade (ungerade) ist. Da es keinen Kreis ungerader Länge gibt, kann es keine Kante geben, deren Endknoten dadurch die gleiche Farbe erhalten. □

Ein klassisches Graphfärbungsproblem ist das Färben von politischen Landkarten, bei dem benachbarte Länder unterschiedliche Farben bekommen sollen. Historisch geht dieses Problem bis ins 19. Jahrhundert zurück. Lange wurde vermutet, dass für das Färben von Landkarten vier Farben immer ausreichen, aber erst 1977 wurde dieses sogenannte Vierfarbenproblem von APPEL und HAKEN gelöst. Hierbei nimmt man an, dass das Gebiet eines jeden Landes zusammenhängend ist und dass Länder, die nur in einem einzigen Punkt aneinanderstossen, gleich gefärbt werden dürfen.

Satz 1.59 (Vierfarbensatz). Jede Landkarte lässt sich mit vier Farben färben.

Der Beweis dieses Satzes ist sehr aufwendig. Er besteht aus einem theoretischen Teil, in dem das allgemeine Problem auf endlich viele Probleme reduziert wird, und einem Computerprogramm, das alle endlichen Fälle überprüft.

Wie kann man eine Färbung eines Graphen mit möglichst wenigen Farben bestimmen? Für die Entscheidung ob ein Graph bipartit ist, genügt eine einfache Breitensuche. Im Allgemeinen ist das Färben von Graphen jedoch ein schwieriges Problem. Schon die scheinbar einfache Frage *„Gegeben ein Graph $G = (V, E)$, gilt $\chi(G) \leq 3$?“* ist $\mathcal{NP}$-vollständig. Das heisst aber, dass es (unter der Annahme $\mathcal{P} \neq \mathcal{NP}$) keinen Algorithmus gibt, der die chromatische Zahl in polynomieller Laufzeit berechnet. In der Praxis wird man sich daher mit Annäherungen an die optimale Lösung zufrieden geben müssen.

Der folgende Algorithmus berechnet eine Färbung, indem er die Knoten des Graphen in einer beliebigen Reihenfolge $v_1, v_2, \ldots, v_n$ besucht und dem aktuellen Knoten jeweils die kleinste Farbe zuordnet, die noch nicht für einen benachbarten Knoten verwendet wird.

GREEDY-FÄRBUNG (G)

1: wähle eine beliebige Reihenfolge der Knoten: $V = \{v_1, \ldots, v_n\}$
2: $c[v_1] \leftarrow 1$
3: **for** $i = 2$ **to** n **do**
4: $\quad c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c(u) \text{ für alle } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

Es ist klar, dass der Algorithmus eine zulässige Färbung berechnet, denn die Farbe eines Knotens unterscheidet sich nach Konstruktion immer von den Farben seiner Nachbarn. Wie viele Farben verwendet der Algorithmus im schlimmsten Fall? Da jeweils die kleinste Farbe gewählt wird, die nicht schon für einen Nachbarknoten verwendet wird, tritt der schlimmste Fall ein, wenn die Nachbarknoten von v_i in den Farben $1, \ldots, \deg(v_i)$ gefärbt sind. In diesem Fall bekommt der neue Knoten die Farbe $\deg(v_i)+1$. Insgesamt werden vom Algorithmus also höchstens $\Delta(G)+1$ Farben verwendet. Dabei bezeichnet $\Delta(G) := \max_{v \in V} \deg(v)$ den *maximalen Grad* eines Knotens in G.

Satz 1.60. Sei G ein zusammenhängender Graph. Für die Anzahl Farben $C(G)$, die der Algorithmus GREEDY-FÄRBUNG benötigt, um die Knoten des Graphen G zu färben, gilt

$$\chi(G) \leq C(G) \leq \Delta(G) + 1.$$

Ist der Graph als Adjazenzliste gespeichert, findet der Algorithmus die Färbung in Zeit $O(|E|)$.

Beweis. Dass der Algorithmus mit $\Delta(G)+1$ Farben auskommt, haben wir bereits eingesehen. Nach Definition der chromatischen Zahl können auch nicht weniger als $\chi(G)$ verwendet worden sein, weshalb die behauptete Ungleichung folgt. Wie sieht es nun mit der Implementierung des Algorithmus aus? Hierzu überlegen wir uns, dass ein Knoten mit Grad d höchstens d bereits gefärbte Nachbarn haben kann und daher insbesondere eine der ersten $d+1$ Farben unter seinen Nachbarn nicht vorkommt. Um den Knoten

v_i zu färben, können wir daher wie folgt vorgehen: Wir initialisieren ein Array der Länge $\deg(v_i)+1$ mit $false$, laufen dann über alle Nachbarn von v und setzen für jeden mit einer Farbe $\leq \deg(v_i)+1$ gefärbten Nachbarn den entsprechenden Eintrag im Array auf $true$. Anschliessend durchlaufen wir das Array bis wir einen Eintrag $false$ finden (den es geben muss) und verwenden diese Farbe für den Knoten v_i. Wir können also v_i in Zeit $O(\deg(v_i))$ färben und somit alle Knoten in Zeit $O(\sum_{v\in V}\deg(v)) = O(|E|)$, wie behauptet. □

Die Anzahl Farben, die der Algorithmus tatsächlich verwendet, hängt im Allgemeinen stark von der Reihenfolge ab, in der die Knoten betrachtet werden. Es gibt beispielsweise immer eine Reihenfolge, bei der man mit $\chi(G)$ Farben auskommt (Übungsaufgabe!), aber da wir diese Reihenfolge nicht kennen, kann der Algorithmus auch deutlich mehr Farben verwenden.

Beispiel 1.61. Betrachten wir den Graphen B_n mit $2n$ Knoten, der aus dem vollständigen bipartiten Graphen $K_{n,n}$ entsteht, indem man die Kanten zwischen gegenüberliegenden Knoten entfernt. Da der Graph B_n bipartit ist, könnte er eigentlich mit zwei Farben gefärbt werden; es ist aber nicht schwer einzusehen, dass es auch eine Reihenfolge der Knoten gibt, für die der Greedy-Algorithmus n Farben benötigt (Übung!).

Im Allgemeinen ist es sehr schwer, eine gute Reihenfolge für den Greedy-Algorithmus zu bestimmen. Es gibt jedoch zwei Situationen, in denen man in Laufzeit $O(|E|)$ eine Reihenfolge der Knoten bestimmen kann, bei der zumindest eine Farbe eingespart wird.

Beispiel 1.62. Sei $G = (V, E)$ ein zusammenhängender Graph mit Maximalgrad $\Delta(G)$. Weiter nehmen wir an, dass es einen Knoten $v \in V$ gibt mit $\deg(v) < \Delta(G)$. Wenn wir jetzt eine Breiten- oder Tiefensuche in v starten und die Knoten in *umgekehrter* Reihenfolge nummerieren, wie sie vom Algorithmus durchlaufen werden (der Knoten v ist also der Knoten v_n), so hat jeder Knoten v_i mit $i < n$ mindestens einen Nachbarn v_j mit $j > i$ und daher höchstens $\Delta(G)-1$ gefärbte Nachbarn. Der Knoten v_n hat nach Wahl ebenfalls nur $\Delta(G)-1$ gefärbte Nachbarn. Der Greedy-Algorithmus benötigt daher für diese Reihenfolge der Knoten höchstens $\Delta(G)$ Farben.

Beispiel 1.63. Sei $G = (V, E)$ ein zusammenhängender k-regulärer Graph, in dem es mindestens einen Artikulationsknoten gibt. Wir wissen bereits aus Abschnitt 1.4.1, dass wir mit einer modifizierten Tiefensuche in Zeit $O(|E|)$ einen solchen Artikulationsknoten v bestimmen können. Seien $V_1, \ldots, V_s$ die Knotenmengen der Zusammenhangskomponenten von $G - v$, wobei $s \geq 2$. Dann erfüllen alle Graphen $G_i := G[V_i \cup \{v\}]$, $1 \leq i \leq s$, die Annahme von Beispiel 1.62 – und können daher jeweils mit k Färben gefärbt werden. Durch eventuelles Vertauschen von Farben können wir zudem sicherstellen, dass in allen Graphen $G_1, \ldots, G_s$ der Knoten v die gleiche Farbe bekommen hat. Die Färbungen der Graphen G_i ergeben daher zusammen eine k-Färbung des Graphen G.

Die beiden obigen Beispiele zeigen Situationen, in denen der Greedy-Algorithmus für Graphen mit Maximalgrad k mit k Farben auskommt. Es gibt jedoch auch k-reguläre Graphen, für die wir $k+1$ Farben benötigen. Vollständige Graphen und ungerade Kreise sind solche Beispiele. Für beide gilt: $\chi(G) = \Delta(G) + 1$. Brooks hat 1941 bewiesen, dass dies die beiden einzigen Graphtypen sind, für die $\chi(G) = \Delta(G) + 1$ gilt:

Satz 1.64 (Satz von Brooks). Ist $G = (V, E)$ ein zusammenhängender Graph, der weder vollständig ist noch ein ungerader Kreis ist, also $G \neq K_n$ und $G \neq C_{2n+1}$, so gilt

$$\chi(G) \leq \Delta(G)$$

und es gibt einen Algorithmus, der die Knoten des Graphen in Zeit $O(|E|)$ mit $\Delta(G)$ Farben färbt.

Beweis. Sei G ein von K_n und C_{2n+1} verschiedener zusammenhängender Graph. Wegen Beispiel 1.62 und 1.63 wissen wir, dass wir annehmen dürfen, dass alle Knoten Grad $\Delta(G)$ haben und es keinen Artikulationsknoten gibt. Da G kein vollständiger Graph ist, aber dennoch zusammenhängend ist, muss es einen Knoten v geben, der zwei Nachbarn $v_1, v_2 \in N(v)$ mit $v_1 \neq v_2$ und $\{v_1, v_2\} \notin E$ besitzt. Nun unterscheiden wir zwei Fälle. Wenn $G \setminus \{v_1, v_2\}$ zusammenhängend ist, so können wir wie in Beispiel 1.62 den Graphen $G[V \setminus \{v_1, v_2\}]$ ausgehend von v mit einer Breiten- oder Tiefensuche durchlaufen. Nummerieren wir die Knoten dann wie folgt: Zuerst v_1 und v_2, dann alle übrigen Knoten, wieder in umgekehrter Reihenfolge wie sie von der Breiten- oder Tiefensuche gefunden wurden (sodass also der Knoten v wiederum der letzte Knoten v_n ist), so folgt leicht, dass der Greedy-Algorithmus für diese Reihenfolge der Knoten mit $\Delta(G)$ Farben auskommt.

Was ist nun, wenn $G \setminus \{v_1, v_2\}$ nicht zusammenhängend ist? – Dann gehen wir ähnlich vor wie in Beispiel 1.63. Seien $V_1, \ldots, V_s$ die Knotenmengen der Zusammenhangskomponenten von $G[V \setminus \{v_1, v_2\}]$, wobei $s \geq 2$. Betrachte die Graphen $G_i := G[V_i \cup \{v_1, v_2\}]$, $1 \leq i \leq s$. Wenn in all diesen Graphen einer der beiden Knoten v_1 und v_2 Grad höchstens $\Delta(G) - 2$ hat, dann können wir zu all diesen Graphen jeweils die Kante $\{v_1, v_2\}$ hinzufügen und die Graphen dann wie in Beispiel 1.62 mit $\Delta(G)$ Farben färben, wobei v_1 und

v_2 wegen der Kante $\{v_1, v_2\}$ jeweils verschiedene Farben bekommen müssen. Durch eventuelles Vertauschen von Farben können wir daher sicherstellen, dass in allen Graphen $G_1, \ldots, G_s$ die Knoten v_1 und v_2 jeweils die gleichen, voneinander verschiedenen, Farben haben. Die Färbungen der Graphen G_i ergeben daher zusammen eine $\Delta(G)$-Färbung des Graphen G. Was nun, wenn in einem Graphen, sagen wir in G_1, beide Knoten v_1 und v_2 Grad grösser als $\Delta(G) - 2$ haben? Die Grade von v_1 in den verschiedenen G_i können sich zu höchstens $\Delta(G)$ aufsummieren, daher kann der Grad von v_1 und v_2 in den anderen G_i jeweils höchstens eins sein. Wenn der Grad von v_1 in einem der G_i Null wäre, so wäre v_2 ein Artikulationsknoten, und umgekehrt. Da wir dies ausgeschlossen haben, muss also $s = 2$ gelten, und v_1 und v_2 haben beide jeweils exakt einen Nachbarn u_1 bzw. u_2 in G_2. Dann färben wir G_1 und G_2 wie in Beispiel 1.62 jeweils mit $\Delta(G)$ Farben. Falls $\Delta(G) > 2$, so können wir nach einem eventuellen Farbtausch in G_2 die Knoten u_1 und u_2 so färben, dass sie nicht dieselbe Farbe wie v_1 bzw. v_2 in G_1 haben. Wieder bilden daher die Färbungen der Graphen G_i zusammen eine $\Delta(G)$-Färbung des Graphen G. Falls hingegen $\Delta(G) = 2$ ist, so handelt es sich bei G um einen Kreis, da G zusammenhängend ist. Da wir ungerade Kreise ausgeschlossen haben, muss G ein Kreis mit einer geraden Knotenzahl sein, und damit ist $\chi(G) = 2 = \Delta(G)$. □

Es ist nicht besonders schwer, aber eine gute Übung, sich zu überlegen, dass es beliebig grosse Graphen mit Maximalgrad k gibt, für die die chromatische Zahl gleich k ist. Satz 1.64 ist daher bestmöglich. Andererseits ist es nicht immer so, dass der maximale Grad die chromatische Zahl bestimmt. Ein Stern ist hierfür ein schönes Beispiel: den Grad des zentralen Knotens können wir beliebig gross machen, der Stern ist dennoch immer mit zwei Farben färbbar.

Unser nächster Satz gibt ein Kriterium für Situationen an, in denen wir die chromatische Zahl unabhängig vom maximalen Grad beschränken können.

Satz 1.65. Ist $G = (V, E)$ ein Graph und $k \in \mathbb{N}$ eine natürliche Zahl mit der Eigenschaft, dass jeder induzierte Subgraph von G einen Knoten mit Grad höchstens k enthält, so gilt $\chi(G) \leq k + 1$ und eine $(k+1)$-Färbung lässt sich in Zeit $O(|E|)$ finden.

Beweis. Nach Annahme gibt es einen Knoten mit Grad höchstens k. Diesen nennen wir v_n (falls es mehrere solche Knoten gibt, wählen wir einen beliebig) und entfernen v_n aus dem Graphen und adaptieren die Grade der übrigen Knoten entsprechend. Wieder gilt: Es gibt einen Knoten mit Grad höchstens k, wir wählen einen solchen, nennen ihn v_{n-1}, entfernen ihn aus dem Graphen und adaptieren die Grade der übrigen Knoten entsprechend. Fahren wir analog fort, so erhalten wir eine Reihenfolge $v_1, \ldots, v_n$ der Knoten, sodass für alle $2 \leq i \leq n$ gilt: v_i hat in dem durch die Knotenmenge $\{v_1, \ldots, v_i\}$ induzierten Subgraphen Grad höchstens k. Färben wir daher die Knoten $v_1, \ldots, v_n$ in dieser Reihenfolge, so wissen wir dass für alle $2 \leq i \leq n$ gilt: Wenn wir v_i färben wollen, so hat v_i höchstens k bereits gefärbte Nachbarn. Wenn uns daher $k+1$ Farben zur Verfügung stehen, kommt mindestens eine dieser Farben nicht bei den Nachbarn von v_i vor und kann daher für v_i verwendet werden. Mit anderen Worten: $k+1$ Farben genügen, um mit diesem Verfahren für jeden Knoten eine zulässige Farbe zu finden.

Wir überlegen uns nun noch, dass wir dieses Verfahren tatsächlich mit Laufzeit $O(|E|)$ implementieren können. Dazu benötigen wir zwei Ideen. Zum einen führen wir ein Array $d[]$ ein, das für jeden Knoten und Zeitpunkt $1 \leq i \leq n$ den Grad des Knoten v in dem durch die noch nicht entfernten Knoten induzierten Graphen $G[V \setminus \{v_{i+1}, \ldots, v_n\}]$ enthält. Dies ist einfach: zu Beginn des Algorithmus entspricht $d[v]$ genau dem Grad $\deg(v)$ des Knoten v im Graphen G. Für jeden neu gefundenen Knoten $v_n, v_{n-1}, \ldots$ benötigen wir Zeit $O(\deg(v_i))$, um das Array anzupassen (wir müssen ja genau für jeden Nachbarn von v_i den Wert um eins reduzieren). Insgesamt benötigen wir hierfür also Zeit $O(|E|)$. Nun müssen wir uns noch überlegen, wie wir jeweils einen Knoten vom Grad höchstens k in dem Subgraphen $G[V \setminus \{v_{i+1}, \ldots, v_n\}]$ effizient finden können. Dazu müssen wir zum einen wissen, welche Knoten wir schon entfernt haben. Dies lässt sich einfach durch ein Boolesches Array removed[] realisieren, das wir für jeden Knoten $v \in V$ mit $false$ initialisieren. Damit könnten wir die Knoten v_i jeweils finden, in dem wir jeweils das Array $d[]$ nach einem Knoten v durchsuchen mit $d[v] \leq k$ und $\text{removed}[v] = false$. Dies würde aber jeweils Zeit $O(|V|)$ benötigen und, für alle Knoten zusammen, zu einer Laufzeit von $O(|V|^2)$ führen. Effizienter geht es, wenn wir zusätzlich noch eine Datenstruktur Q verwenden, die zu jedem Zeitpunkt alle noch nicht entfernten Knoten mit Grad höchstens k enthält. Genauer gehen wir wie folgt vor: Nach der In-

itialisierung von d[] und immer, wenn wir die Werte von d[] updaten, fügen wir alle Knoten mit $d[v] \leq k$ und $\text{removed}[v] = \mathit{false}$ in Q ein und setzen für diese Knoten $\text{removed}[v]$ auf true, damit sie später nicht nochmals in Q eingefügt werden. Als neuen Knoten v_i können wir dann in jeder Runde einen beliebigen Knoten aus Q wählen, d.h. wir können Q zum Beispiel als Stapel oder auch als Schlange implementieren. Durch die Verwendung von Q reduziert sich die Wahl von v_i von $O(|V|)$ auf $O(1)$ und die Gesamtlaufzeit für das Bestimmen der Reihenfolge der Knoten $v_1, \ldots, v_n$ ist daher $O(|E|)$. Wie bereits im Beweis von Satz 1.60 können wir das eigentliche Färben des Graphen ebenfalls in Zeit $O(|E|)$ implementieren. □

Satz 1.66 (Mycielski-Konstruktion). Für alle $k \geq 2$ gibt es einen dreiecksfreien Graphen G_k mit $\chi(G_k) \geq k$.

Beweis. Wir beweisen den Satz durch Induktion über k. Für $k = 2$ ist nichts zu zeigen: Der Graph, der aus einer einzigen Kante besteht, erfüllt bereits alle Bedingungen des Satzes. Sei also $k \geq 2$ und G_k ein dreiecksfreier Graph mit $\chi(G_k) \geq k$. Wir konstruieren daraus einen neuen Graphen G_{k+1}, indem wir zu G_k einige zusätzliche Knoten und Kanten hinzufügen. Seien $v_1 \ldots, v_n$ die Knoten des Graphen G_k. Für jeden Knoten v_i fügen wir einen neuen Knoten w_i hinzu und verbinden ihn mit allen Nachbarn von v_i in G_k. Zusätzlich fügen wir noch einen Knoten z hinzu und verbinden ihn mit allen Knoten $w_1, \ldots, w_n$.

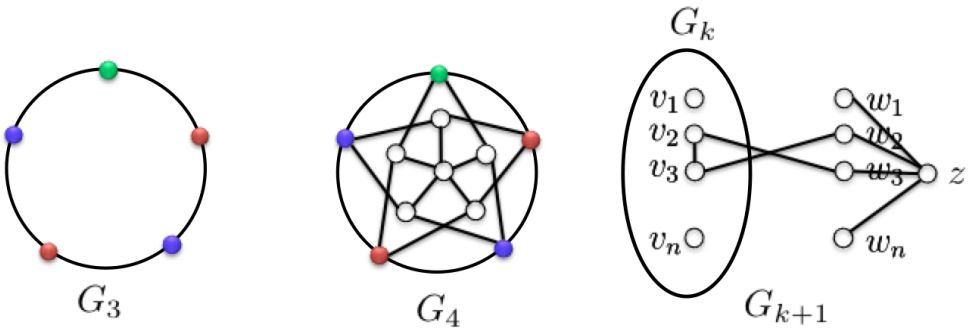


Abbildung 1.17: Die Mycielski-Konstruktion für $k = 3$ und $k = 4$, und die allgemeine Konstruktionsvorschrift von k auf $k + 1$.

Wir überlegen uns zunächst, dass der neue Graph kein Dreieck enthält. Da die Knoten w_i nicht untereinander verbunden sind, ist der Knoten z sicherlich in keinem Dreieck enthalten. Analog kann es kein Dreieck geben, das zwei Knoten w_i und w_j enthält. Da aber w_i in G_k genau zu den Nachbarn von v_i verbunden ist, kann w_i auch in keinem Dreieck mit zwei Knoten aus G_k liegen (denn G_k ist ja nach Annahme dreiecksfrei). Also ist der neue Graph G_{k+1} in der Tat dreiecksfrei. Nehmen wir nun an, er würde sich mit k Farben färben lassen. Dann kommt wegen dem Knoten z mindestens eine der k Farben nicht unter den Nachbarn von z vor. Ohne Einschränkung sei dies die Farbe k. Da alle Knotenpaare v_i und w_i dieselben Nachbarn in G_k haben, können wir die mit k gefärbten Knoten in G_k sicherlich in die Farbe des 'Partnerknotens' w_i umfärben, woraus folgt, dass sich G_k mit $k-1$ Farben färben lässt, im Widerspruch zur Annahme. Also gibt es keine Färbung von G_{k+1} mit k Farben und die chromatische Zahl von G_{k+1} ist daher mindestens $k+1$, was zu zeigen war. □

Man kann sogar zeigen: Für alle $k, \ell \geq 2$ gibt es Graphen $G_{k,\ell}$, sodass $G_{k,\ell}$ keinen Kreis der Länge höchstens ℓ enthält, aber dennoch gilt: $\chi(G_{k,\ell}) \geq k$. Den Beweis dieses Satzes überlassen wir aber Spezialvorlesungen.

Den Abschnitt schliessen wir mit einem überraschend schwierigen algorithmischen Problem. Nehmen wir an, jemand verspricht uns, dass der zu färbende Graph chromatische Zahl drei hat. Dann wissen wir: Es gibt eine Reihenfolge der Knoten für die der Greedy-Algorithmus nur drei Farben benötigt. Diese kennen wir aber nicht. Dennoch wollen wir mit möglichst wenigen Farben auskommen. Der folgende Satz zeigt uns, dass uns zumindest $O(\sqrt{|V|})$ genügen. In Anbetracht der Tatsache, dass der Graph ja in Wirklichkeit 3-färbbar ist, ist dies nicht besonders eindrucksvoll. Es ist aber fast das beste was bekannt ist: Es ist bekannt, dass das Färben mit vier Farben NP-schwer ist und dass andererseits $O(|V|^{0.211})$ (statt der von uns benötigten $O(|V|^{0.5})$) Farben ausreichen.

Satz 1.67. Jeden 3-färbbaren Graphen $G = (V, E)$ kann man in Zeit $O(|E|)$ mit $O(\sqrt{|V|})$ Farben färben.

Beweis. Wir überlegen uns zunächst: Ist $G = (V, E)$ ein 3-färbbarer Graph und $v \in V$ ein beliebiger Knoten in G, so ist der durch die Nachbarschaft von v induzierte Subgraph $G[N(v)]$ bipartit. In der Tat: In einer 3-Färbung

von G müssen alle Nachbarn von v eine andere Farbe bekommen als der Knoten v; für die Knoten in $N(v)$ stehen daher nur zwei Farben zur Verfügung – und der Graph $G[N(v)]$ ist daher bipartit und wir können die Nachbarschaft daher mit einer Breitensuche in linearer Zeit (genauer: linear in der Anzahl Kanten in der Nachbarschaft von v) mit zwei Farben färben. Damit ergibt sich folgende Idee für einen Algorithmus: Solange es einen Knoten mit "grossem" Grad gibt, wählen wir einen solchen und färben ihn und seine Nachbarn mit drei (neuen) Farben. Sobald es keinen Knoten mit grossem Grad mehr gibt, wenden wir auf den restlichen Graphen den Satz von Brooks an. Eine gute Wahl für "grossen" Grad ist der Wert $\sqrt{n}$. Denn dann färben wir höchstens $n/\sqrt{n} = \sqrt{n}$ Knoten nach der ersten Regel (und benötigen dafür maximal $3\sqrt{n}$ Farben) und für die Anwendung des Satzes von Brooks benötigen wir ebenfalls nur $\sqrt{n}$ Farben. Insgesamt kommen wir also sicher mit $4\sqrt{n}$ Farben aus. (Die Konstante 4 lässt sich durch eine bessere Wahl des Grenzwertes noch leicht reduzieren, an der Grössenordnung $\sqrt{n}$ ändert sich dadurch jedoch nichts.) □

Kapitel 2

Wahrscheinlichkeitstheorie und randomisierte Algorithmen

Über die Jahre haben stochastische Konzepte in der Informatik eine wachsende Bedeutung gewonnen. Einige (algorithmische) Beispiele wurden im ersten Teil der Vorlesung bereits aufgegriffen. Die Grundlagen des Hashings beruhen beispielsweise auf Aussagen über die Verteilung bestimmter Ereignisse. Auch dass der Sortieralgorithmus QuickSort zu Recht das Wort „schnell“ in seinem Namen trägt (und nicht etwa „SlowSort“ genannt wird, was in Anbetracht seiner WorstCase Laufzeit von $\Omega(n^2)$ auf den ersten Blick durchaus angebracht scheint), verdankt er seiner sehr effizienten Performance bei einer zufälligen Wahl der Pivotelemente.

Der Einfluss der Stochastik auf die Informatik geht aber weit über die Algorithmik hinaus. Jegliche Art von Kryptographie, wie wir sie heutzutage in vielen Bereichen des täglichen Lebens verwenden, wäre ohne Stochastik so nicht möglich. Aber beim sogenannten verteilten Rechnen oder auch bei der Entwicklung von Verfahren für Roboter, die sich eigenständig koordinieren sollen, spielt der Zufall eine grosse Rolle.

In diesem Kapitel werden wir die Grundlagen der Stochastik entwickeln und an Beispielen illustrieren.

2.1 Grundbegriffe und Notationen

Einem stochastischen Experiment liegt immer ein Wahrscheinlichkeitsraum (eine Menge Ω) zugrunde, zusammen mit Wahrscheinlichkeiten für die Elemente dieser Menge. Die folgende Definition formalisiert dies.

Definition 2.1. Ein *diskreter Wahrscheinlichkeitsraum* ist bestimmt durch eine *Ergebnismenge* $\Omega = \{\omega_1, \omega_2, \ldots\}$ von *Elementarereignissen*. Jedem Elementarereignis ω_i ist eine *(Elementar-)Wahrscheinlichkeit* $\Pr[\omega_i]$ zugeordnet, wobei wir fordern, dass $0 \leq \Pr[\omega_i] \leq 1$ und

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

Eine Menge $E \subseteq \Omega$ heisst *Ereignis*. Die Wahrscheinlichkeit $\Pr[E]$ eines Ereignisses ist definiert durch

$$\Pr[E] := \sum_{\omega \in E} \Pr[\omega].$$

Ist E ein Ereignis, so bezeichnen wir mit $\bar{E} := \Omega \setminus E$ das *Komplementärereignis* zu E.

Ein Wahrscheinlichkeitsraum mit $\Omega = \{\omega_1, \ldots, \omega_n\}$ heisst *endlicher Wahrscheinlichkeitsraum*. Wir werden uns in diesem Kapitel oft auf endliche Wahrscheinlichkeitsräume beschränken. Bei unendlichen Wahrscheinlichkeitsräumen werden wir gewöhnlich nur den Fall $\Omega = \mathbb{N}_0$ betrachten.

Man kann den Begriff des Wahrscheinlichkeitsraumes auch auf überabzählbare Mengen wie $\Omega = \mathbb{R}$ erweitern. Hierbei treten jedoch einige zusätzliche Schwierigkeiten auf und wir werden die Behandlung dieses Themas daher auf weiterführende Vorlesungen verschieben. Für die Wahrscheinlichkeitstheorie für diskrete (endliche oder abzählbar unendliche) Mengen verwendet man oft auch den Begriff (elementare) Stochastik. Auf diese werden wir uns in dieser Vorlesung beschränken.

Aus der Definition 2.1 folgen sofort einige elementare, aber sehr nützliche Konsequenzen.

Lemma 2.2. Für Ereignisse A, B gilt:

1. $\Pr[\emptyset] = 0$, $\Pr[\Omega] = 1$.
2. $0 \leq \Pr[A] \leq 1$.
3. $\Pr[\bar{A}] = 1 - \Pr[A]$.

4. Wenn $A \subseteq B$, so folgt $\Pr[A] \leq \Pr[B]$.

Ebenso elementar ist der folgende Satz, der nichtsdestrotrotz einen hochtrabenden Namen trägt.

Satz 2.3 (Additionssatz). Wenn die Ereignisse $A_1, \ldots, A_n$ paarweise disjunkt sind (also wenn für alle Paare $i \neq j$ gilt, dass $A_i \cap A_j = \emptyset$), so gilt

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i].$$

Für eine unendliche Menge von disjunkten Ereignissen $A_1, A_2, \ldots$ gilt analog

$$\Pr\left[\bigcup_{i=1}^{\infty} A_i\right] = \sum_{i=1}^{\infty} \Pr[A_i].$$

Die Annahme aus Satz 2.3, dass die Ereignisse paarweise disjunkt sind, ist essentiell. Ohne diese ist die Aussage im Allgemeinen nicht wahr.

Beispiel 2.4. Wir werfen einen normalen sechsseitigen Würfel. Hier ist $\Omega = \{1,2,3,4,5,6\}$ und jedes der sechs Elementarereignisse hat die Wahrscheinlichkeit $1/6$. Betrachten wir jetzt die Ereignisse $A = \{1,3,5\}$ (Augenzahl ist ungerade) und $B = \{5,6\}$ (Augenzahl ist mindestens fünf), so sind diese Ereignisse nicht disjunkt. Tatsächlich gilt $\Pr[A \cup B] = \Pr[\{1,3,5,6\}] = \frac{4}{6} \neq \frac{3}{6} + \frac{2}{6} = \Pr[A] + \Pr[B]$.

Für den allgemeinen Fall gilt jedoch der folgende Satz.

Satz 2.5. *(Siebformel, Prinzip der Inklusion/Exklusion)*
Für Ereignisse $A_1, \ldots, A_n$ $(n \geq 2)$ gilt:

$$\begin{aligned}
\Pr\left[\bigcup_{i=1}^{n} A_i\right] &= \sum_{l=1}^{n} (-1)^{l+1} \sum_{1 \leq i_1 < \cdots < i_l \leq n} \Pr[A_{i_1} \cap \cdots \cap A_{i_l}] \\
&= \sum_{i=1}^{n} \Pr[A_i] - \sum_{1 \leq i_1 < i_2 \leq n} \Pr[A_{i_1} \cap A_{i_2}] \\
&\quad + \sum_{1 \leq i_1 < i_2 < i_3 \leq n} \Pr[A_{i_1} \cap A_{i_2} \cap A_{i_3}] - \cdots \\
&\quad + (-1)^{n+1} \cdot \Pr[A_1 \cap \cdots \cap A_n].
\end{aligned}$$

Ein besonderer Speziallfall tritt auf, wenn wir Satz 2.5 auf den Wahrscheinlichkeitsraum $\Omega = A_1 \cup \cdots \cup A_n$ mit $\Pr[\omega] = 1/|\Omega|$ anwenden, wo-

bei $A_1, \ldots, A_n$ beliebige endliche Mengen sind. Dann erhalten wir nämlich (nach ausmultiplizieren mit $|\Omega|$) die nützliche Formel

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{l=1}^{n} (-1)^{l+1} \sum_{1 \le i_1 < \cdots < i_k \le n} |A_{i_1} \cap \cdots \cap A_{i_l}|,$$

die ebenfalls oft als *Siebformel* bezeichnet wird.

Es ist auch nicht schwer, wenn auch etwas technisch, Satz 2.5 direkt zu beweisen. Statt dies jedoch hier zu tun, verschieben wir den Beweis auf ein späteres Kapitel (Beispiel 2.36), in dem uns einige dann zur Verfügung stehende zusätzliche Techniken ermöglichen etwas Rechnung einzusparen. Wir illustrieren die grundlegenden Ideen hinter Satz 2.5 jedoch, indem wir noch die Spezialfälle $n = 2$ und $n = 3$ explizit betrachten. Für $n = 2$ setzen wir $X := A_1 \setminus A_2 = A_1 \setminus (A_1 \cap A_2)$. X ist so gewählt, dass X und $A_1 \cap A_2$ sowie X und A_2 disjunkt sind. Deshalb können wir den Additionssatz anwenden:

$$\Pr[A_1] = \Pr[X \cup (A_1 \cap A_2)] = \Pr[X] + \Pr[A_1 \cap A_2].$$

Wegen $A_1 \cup A_2 = X \cup A_2$ folgt daraus

$$\Pr[A_1 \cup A_2] = \Pr[X \cup A_2] = \Pr[X] + \Pr[A_2] = \Pr[A_1] - \Pr[A_1 \cap A_2] + \Pr[A_2]$$

und wir haben die Behauptung für $n = 2$ gezeigt.

Abbildung 2.1 veranschaulicht den Fall $n = 3$. Man überzeuge sich, dass durch die im Satz angegebene Summe die Elementarereignisse in jeder der sieben Teilmengen $A \setminus (B \cup C), \ldots, A \cap B \cap C$ jeweils genau einmal gezählt werden.

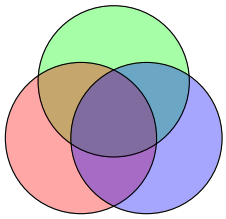


Abbildung 2.1: Illustration zur Inklusion-Exklusion-Formel für $n = 3$.

Für $n \ge 4$ werden die Formeln aus Satz 2.5 recht lang und umständlich. In diesem Fall gibt man sich deshalb oft mit der folgenden einfachen Abschätzung zufrieden, die in der Literatur nach George Boole (1815–1864) benannt ist. In der Informatikliteratur wird hierfür oft auch der Begriff

„Union Bound“ verwendet, der sehr schön beschreibt, was die Ungleichung besagt: Wir beschränken die Wahrscheinlichkeit der Vereinigung durch die Summe der Einzelwahrscheinlichkeiten.

Korollar 2.6. *(Boolesche Ungleichung, Union Bound)* Für Ereignisse $A_1, \ldots, A_n$ gilt

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr[A_i].$$

Analog gilt für eine unendliche Folge von Ereignissen $A_1, A_2, \ldots$, dass $\Pr\left[\bigcup_{i=1}^{\infty} A_i\right] \leq \sum_{i=1}^{\infty} \Pr[A_i]$.

Beweis. Wir betrachten zunächst den endlichen Fall. Für jedes $i \geq 1$ setzen wir $B_i := A_i \setminus (A_1 \cup \cdots \cup A_{i-1})$; dann gilt offenbar $\Pr[B_i] \leq \Pr[A_i]$. Ausserdem sind je zwei Mengen B_i und B_j mit $i \neq j$ disjunkt und es gilt $\bigcup_{i=1}^{n} A_i = \bigcup_{i=1}^{n} B_i$. Nach dem Additionssatz ist dann

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \Pr\left[\bigcup_{i=1}^{n} B_i\right] = \sum_{i=1}^{n} \Pr[B_i] \leq \sum_{i=1}^{n} \Pr[A_i].$$

Die entsprechende Aussage für eine unendliche Folge von Ereignissen beweist man analog. □

Wahl der Wahrscheinlichkeiten

Wenn wir Wahrscheinlichkeitsräume und die damit verbundene Theorie einsetzen wollen, müssen wir zunächst die Frage beantworten, wie bei einer konkreten Anwendung die Wahrscheinlichkeiten der Elementarereignisse sinnvoll festgelegt werden können. Einen ersten Anhaltspunkt liefert ein Prinzip, das nach Pierre-Simon Laplace (1749–1827) benannt ist. Laplace leistete bedeutende Beiträge zu zahlreichen Gebieten. Insbesondere beschäftigte er sich neben der Mathematik mit Astronomie, Physik und Chemie. Unter Napoleon war er auch kurz als Minister des Inneren tätig, wurde allerdings bereits nach sechs Wochen wieder abgelöst, da er sich auch der unbedeutendsten Probleme selbst annahm.

Prinzip von Laplace: *Wenn nichts dagegen spricht, gehen wir davon aus, dass alle Elementarereignisse gleich wahrscheinlich sind.*

Bei der Anwendung des Prinzips von Laplace gilt für alle Elementarereignisse $\Pr[\omega] = 1/|\Omega|$. Daraus erhalten wir für ein beliebiges Ereignis E die Formel

$$\Pr[E] = \frac{|E|}{|\Omega|}.$$

Wir sagen dann auch, dass das Ergebnis des modellierten Zufallsexperiments auf Ω *uniform verteilt* oder *gleichverteilt* ist.

Im informationstheoretischen Sinn besitzt der Wahrscheinlichkeitsraum mit $\Pr[\omega] = 1/|\Omega|$ für alle $\omega \in \Omega$ die grösstmögliche Entropie („Unordnung"). Jede Abweichung von der Gleichwahrscheinlichkeit bedeutet, dass wir in das Modell zusätzliche Information einfliessen lassen (und dadurch die Entropie verringern). Das Prinzip von Laplace besagt nun, dass es nicht sinnvoll ist, bei der Modellierung eines Systems Wissen „vorzugaukeln", wenn man nicht über entsprechende Anhaltspunkte verfügt.

Wenn zusätzliches Wissen über das zu modellierende Experiment vorhanden ist und die Bedingungen für die einzelnen Elementarereignisse somit nicht mehr als symmetrisch angesehen werden können, so müssen die Elementarwahrscheinlichkeiten diesen Umständen angepasst werden. Wenn beispielsweise auf einem Würfel die Seite mit der „Sechs" durch eine weitere „Eins" ersetzt wird, so ist anschaulich klar, dass die Eins nun eine doppelt so grosse Wahrscheinlichkeit erhalten sollte wie alle anderen Elementarereignisse.

Beschreibung von Wahrscheinlichkeitsräumen und Ereignissen

Die vollständige und mathematisch exakte Darstellung eines Wahrscheinlichkeitsraumes, sowie entsprechender Ereignisse, ist bei vielen Anwendungen recht kompliziert. Aus diesem Grund haben sich dafür einige Konventionen eingebürgert, auf die wir im Folgenden kurz eingehen werden.

Als Beispiel für einen etwas komplizierteren Wahrscheinlichkeitsraum betrachten wir ein Kartenspiel mit zwei Spielern, die wir A und B nennen. Jeder Spieler erhält fünf Karten. Nach dem Prinzip von Laplace gehen wir davon aus, dass jede Auswahl der zweimal fünf Karten aus den 52 Karten im gesamten Kartenspiel (französisches Blatt mit den Farben Kreuz, Pik, Herz, Karo und den Werten 2, 3, ..., 9, 10, Bube, Dame, König, Ass, also $4 \cdot 13 = 52$ Karten) gleich wahrscheinlich ist.

Als Ergebnismenge könnten wir beispielsweise definieren

$$\begin{aligned}\Omega \;:=\; & \{(X,Y) \mid X,Y \subseteq C,\ X \cap Y = \emptyset,\ |X| = |Y| = 5,\\ & \text{wobei } C = \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\} \times \{2,3,\ldots,9,10,B,D,K,A\}\}.\end{aligned}$$

Die Komponenten X und Y eines Elementarereignisses $(X,Y) \in \Omega$ entsprechen hierbei den Karten, die A und B erhalten.

An diesem Beispiel sieht man, dass es oft recht mühsam ist, eine Kodierung für die Ergebnismenge aufzuschreiben. In der Praxis verzichtet man deshalb häufig darauf und auch wir werden in Zukunft nicht immer eine explizite Darstellung von Ω angeben. Allerdings sollte man sich stets klarmachen, wie eine solche Darstellung im Prinzip auszusehen hätte. Wem bei einem Beispiel nicht klar ist, auf welche Weise Ω kodiert werden könnte, sollte sich auf jeden Fall über eine exakte Darstellung Gedanken machen und gegebenenfalls versuchen, diese aufzuschreiben.

Wenn man keine formale Darstellung von Ω angibt, muss man auch die Ereignisse informell angeben. Wenn wir beispielsweise die Wahrscheinlichkeit untersuchen wollen, dass Spieler A vier Asse erhält, so „definieren“ wir das entsprechende Ereignis E durch

$$E := \text{„Spieler } A \text{ hat vier Asse“},$$

anstatt zu schreiben

$$\begin{aligned}E := \{(X,Y) \in \Omega \;\mid\; & X = \{(f_1,w_1),\ldots,(f_5,w_5)\}\\ & \text{und } w_1 = \ldots = w_4 = A\}.\end{aligned}$$

Wenn wir uns die Mühe sparen wollen, einem Ereignis einen Namen zu geben, schreiben wir oft auch nur

$$\Pr[\text{„Spieler } A \text{ hat vier Asse“}]$$

für die Wahrscheinlichkeit, dass Spieler A vier Asse hat.

2.2 Bedingte Wahrscheinlichkeiten

Durch das Bekanntwerden zusätzlicher Information verändern sich Wahrscheinlichkeiten. Nehmen wir zum Beispiel an, dass wir bei einem Würfel die geraden Zahlen rot markieren, verdeckt würfeln und dann den Würfel

aus der Ferne betrachten. In diesem Fall können wir zwar die gewürfelte Augenzahl nicht ablesen, aber wir können bereits an der Farbe erkennen, ob eine gerade oder eine ungerade Augenzahl gefallen ist. Dadurch werden manche Elementarereignisse wahrscheinlicher, während andere unwahrscheinlicher bzw. unmöglich werden.

Beispiel 2.7. Zwei Spieler, wir nennen sie wieder A und B, spielen eine Runde Poker. Die beiden verwenden dazu das auf Seite 91 vorgestellte Experiment (52 Karten, 5 Karten pro Spieler, keine getauschten Karten).

A ist sehr zufrieden mit seinen Karten, denn er hält vier Asse und eine Herz Zwei in der Hand. B kann dieses Blatt nur überbieten, wenn sie einen Straight Flush (fünf Karten *einer* Farbe in aufsteigender Reihenfolge, zum Beispiel Kreuz 9, 10, Bube, Dame, König) hat. Die Wahrscheinlichkeit für das Ereignis $F :=$ „B hat einen Straight Flush“ beträgt

$$\Pr[F] = \frac{|F|}{|\Omega|} = \frac{3 \cdot 8 + 7}{\binom{52-5}{5}} = \frac{31}{1533939} = 2{,}02\ldots \cdot 10^{-5}.$$

Diese Rechnungen bedürfen noch einiger Erläuterung: Das Blatt von B wird zufällig aus den $52 - 5$ Karten gewählt, die A nicht besitzt. Bei allen Farben ausser Herz kann der Straight Flush bei den acht Karten 2, 3, 4, 5, 6, 7, 8 oder 9 beginnen. Bei Herz fällt die Zwei weg, da diese Karte ja im Besitz von A ist.

Die äusserst geringe Wahrscheinlichkeit von F würde A sehr beruhigen, wenn er nicht die Karten gezinkt hätte und deshalb erkennen könnte, dass B nur Kreuz in der Hand hält. A beginnt also noch einmal zu rechnen: Wir setzen nun $|\Omega'| = \binom{12}{5}$, da das Blatt von B aus den zwölf Kreuzkarten gewählt wird, die nicht in der Hand von A sind. Ferner bezeichne F' das Ereignis, dass B einen Straight Flush der Farbe Kreuz hat. Wir erhalten mit derselben Argumentation wie oben $|F'| = 8$. In unserem neuen Wahrscheinlichkeitsraum gilt

$$\Pr[F'] = \frac{|F'|}{|\Omega'|} = \frac{8}{\binom{12}{5}} = \frac{8}{792} \approx 0{,}01\,.$$

Die Wahrscheinlichkeit für einen Sieg von B ist also drastisch gestiegen, wenn sie auch absolut gesehen noch immer nicht besonders gross ist.

Beispiel 2.7 zeigt, wie zusätzliche Information den Wahrscheinlichkeitsraum und damit die Wahrscheinlichkeit eines Ereignisses beeinflussen kann. Mit $A|B$ (sprich: „A bedingt auf B“ oder „A gegeben B“) bezeichnen wir das Ereignis, dass A eintritt, wenn wir bereits wissen, dass das Ereignis B auf jeden Fall eintritt.

Beispiel 2.7 *(Fortsetzung)* Sei K das Ereignis, dass B nur Kreuzkarten in der Hand hat. In unserem Beispiel entspricht F' im neuen Wahrscheinlichkeitsraum Ω' somit dem Ereignis $F|K$ im Wahrscheinlichkeitsraum Ω.

Welche Eigenschaften sollte eine sinnvolle Definition von $\Pr[A|B]$ erfüllen? Die folgenden Punkte macht man sich zu dieser Frage recht schnell klar:

1. $\Pr[B|B] = 1$, denn wenn wir wissen, dass B sicher eintritt, dann sollte die Wahrscheinlichkeit für das Eintreten von B gleich Eins sein. Ebenso fordern wir $\Pr[B|\bar{B}] = 0$.

2. Die Bedingung auf Ω sollte keine Auswirkungen auf die Wahrscheinlichkeit eines beliebigen Ereignisses A haben, da die Aussage „Ω ist eingetreten“ keine zusätzliche Information liefert. Wir fordern also $\Pr[A|\Omega] = \Pr[A]$.

3. Wenn wir wissen, dass B eingetreten ist, dann kann ein Ereignis A nur dann eintreten, wenn zugleich auch $A \cap B$ eintritt. Die Wahrscheinlichkeiten $\Pr[A|B]$ sollten daher für ein festes B proportional zu $\Pr[A \cap B]$ sein.

Diese Überlegungen führen zu folgender Definition.

Definition 2.8. A und B seien Ereignisse mit $\Pr[B] > 0$. Die *bedingte Wahrscheinlichkeit* $\Pr[A|B]$ von A gegeben B ist definiert durch

$$\Pr[A|B] := \frac{\Pr[A \cap B]}{\Pr[B]}.$$

Der Leser überzeuge sich, dass Definition 2.8 die zuvor aufgelisteten Eigenschaften erfüllt.

Beispiel 2.7 *(Fortsetzung)* Erinnern wir uns: Da A die Karten gezinkt hat, kann er erkennen, dass B nur Kreuzkarten in der Hand hält, das Ereignis K also eingetreten ist. Gesucht ist die Wahrscheinlichkeit des Ereignisses F, dass B einen Straight Flush in der Hand hält, unter dieser Bedingung. Gemäss Definition 2.8 berechnen wir dazu zunächst

$$\Pr[F \cap K] = \frac{8}{|\Omega|} \quad \text{und} \quad \Pr[K] = \frac{\binom{12}{5}}{|\Omega|} = \frac{|\Omega'|}{|\Omega|}.$$

Daraus folgt

$$\Pr[F|K] = \frac{\Pr[F \cap K]}{\Pr[K]} = \frac{\frac{8}{|\Omega|}}{\frac{|\Omega'|}{|\Omega|}} = \frac{8}{|\Omega'|},$$

und wir erhalten also dasselbe Ergebnis wie bei unseren vorigen, auf direkten Überlegungen basierenden Rechnungen.

Die bedingten Wahrscheinlichkeiten der Form $\Pr[\,\cdot\,|B]$ bilden für ein beliebiges Ereignis $B \subseteq \Omega$ mit $\Pr[B] > 0$ einen neuen Wahrscheinlichkeitsraum über Ω. Die Wahrscheinlichkeiten der Elementarereignisse ω_i berech-

nen sich durch $\Pr[\omega_i|B]$. Man überprüft leicht, dass dadurch Definition 2.1 erfüllt ist:

$$\sum_{\omega\in\Omega}\Pr[\omega|B]=\sum_{\omega\in\Omega}\frac{\Pr[\omega\cap B]}{\Pr[B]}=\sum_{\omega\in B}\frac{\Pr[\omega]}{\Pr[B]}=\frac{\Pr[B]}{\Pr[B]}=1.$$

Damit gelten alle Rechenregeln für Wahrscheinlichkeiten auch für bedingte Wahrscheinlichkeiten. Beispielsweise erhalten wir die Regeln $\Pr[\emptyset|B]=0$ oder $\Pr[\bar{A}|B]=1-\Pr[A|B]$.

Den bedingten Wahrscheinlichkeitsraum kann man sich so vorstellen, dass die Wahrscheinlichkeiten für Elementarereignisse ausserhalb von B auf Null gesetzt werden. Die Wahrscheinlichkeiten für Elementarereignisse in B werden dann so skaliert, dass die Summe aller Wahrscheinlichkeiten wieder Eins ergibt. Zur Skalierung ist der Faktor $1/\Pr[B]$ nötig, da wir die Wahrscheinlichkeit für alle Elementarereignisse $\omega\in\bar{B}$ auf Null setzen und die Summe der verbleibenden Elementarwahrscheinlichkeiten somit gleich $\Pr[B]$ ist.

Beim Umgang mit Wahrscheinlichkeiten im Allgemeinen und mit bedingten Wahrscheinlichkeiten im Besonderen ist es erforderlich, sehr sorgfältig vorzugehen und nie die formalen Definitionen aus den Augen zu verlieren, da man sonst leicht zu voreiligen Schlüssen verleitet wird. Das folgende Problem stellt ein berühmtes Beispiel hierfür dar.

Beispiel 2.9. *(Zweikinderproblem)* Wir sind zu Gast bei einer Familie mit zwei Kindern. Wir nehmen an, dass bei der Geburt eines Kindes beide Geschlechter gleich wahrscheinlich sind. Wie gross ist die Wahrscheinlichkeit, dass beide Kinder der Familie Mädchen sind, wenn wir wissen dass sie mindestens ein Mädchen haben?

Die Frage verführt zur spontanen Antwort $\frac{1}{2}$, da für das Geschlecht des unbekannten Kindes immer noch zwei Möglichkeiten bestehen. Von diesen ist scheinbar keine bevorzugt, da das Geschlecht des Geschwisterkindes keine Auswirkung auf das Geschlecht des bislang unbekannten Kindes hat.

Bei genauerem Hinsehen stellt man aber fest, dass die Ergebnismenge so zu definieren ist: $\Omega:=\{mm,mj,jm,jj\}$. Hierbei ist das Geschlecht (j für „Junge“ und m für „Mädchen“) der Kinder in der Reihenfolge ihrer Geburt angetragen. Wir bedingen auf das Ereignis $M:=\{mm,mj,jm\}$ und interessieren uns für $A:=\{mm\}$ und die Wahrscheinlichkeit $\Pr[A|M]$. Aus der Definition der bedingten Wahrscheinlichkeit folgt

$$\Pr[A|M]=\frac{\Pr[A\cap M]}{\Pr[M]}=\frac{1/4}{3/4}=\frac{1}{3}.$$

Die Wahrscheinlichkeit $1/3$ folgt daraus, dass wir wussten, dass eines der beiden Kinder ein Mädchen ist (aber nicht, welches). Wissen wir, dass das ältere Kind ein Mädchen ist,

so folgt

$$\Pr[A|\text{„älteres Kind ist ein Mädchen“}] = \frac{\Pr[\{mm\}]}{\Pr[\{mj, mm\}]} = \frac{1/4}{2/4} = \frac{1}{2}.$$

Kennen wir andererseits noch gar kein Kind, so erhalten wir

$$\Pr[A] = \Pr[\{mm\} = \frac{1}{4}.$$

Wir sehen: die Wahrscheinlichkeit eines Ereignisses hängt sehr davon ab, welche Informationen uns bereits bekannt sind. Es ist daher wichtig, sich immer sehr genau zu überlegen, ob bzw. auf welches Ereignis wir bedingen müssen.

Häufig verwendet man Definition 2.8 in der Form

$$\Pr[A \cap B] = \Pr[B|A] \cdot \Pr[A] = \Pr[A|B] \cdot \Pr[B]. \tag{2.1}$$

Anschaulich bedeutet dies: Um auszurechnen, mit welcher Wahrscheinlichkeit A und B zugleich eintreten, genügt es, die Wahrscheinlichkeiten zu multiplizieren, dass zunächst A eintritt und dann noch B unter der Bedingung, dass A schon eingetreten ist. Bei mehr als zwei Ereignissen führt dies zu folgender Rechenregel.

Satz 2.10. *(Multiplikationssatz)* Seien die Ereignisse $A_1, \ldots, A_n$ gegeben. Falls $\Pr[A_1 \cap \cdots \cap A_n] > 0$ ist, gilt

$$\Pr[A_1 \cap \cdots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \cdots \Pr[A_n|A_1 \cap \cdots \cap A_{n-1}].$$

Beweis. Zunächst halten wir fest, dass alle bedingten Wahrscheinlichkeiten wohldefiniert sind, da $\Pr[A_1] \geq \Pr[A_1 \cap A_2] \geq \ldots \geq \Pr[A_1 \cap \cdots \cap A_n] > 0$. Die rechte Seite der Aussage im Satz können wir gemäss der Definition der bedingten Wahrscheinlichkeit umschreiben zu

$$\frac{\Pr[A_1]}{1} \cdot \frac{\Pr[A_1 \cap A_2]}{\Pr[A_1]} \cdot \frac{\Pr[A_1 \cap A_2 \cap A_3]}{\Pr[A_1 \cap A_2]} \cdots \frac{\Pr[A_1 \cap \ldots \cap A_n]}{\Pr[A_1 \cap \ldots \cap A_{n-1}]}.$$

Offensichtlich kürzen sich alle Terme bis auf $\Pr[A_1 \cap \ldots \cap A_n]$. □

Mit Hilfe von Satz 2.10 können wir ein klassisches Problem der Wahrscheinlichkeitsrechnung lösen:

Beispiel 2.11. *(Geburtstagsproblem)* Wir möchten folgende Frage beantworten: Wie gross ist die Wahrscheinlichkeit, dass in einer m-köpfigen Gruppe zwei Personen am selben Tag Geburtstag haben? Dieses Problem formulieren wir folgendermassen um: Man werfe m Bälle zufällig und gleich wahrscheinlich in n Körbe. Wie gross ist die Wahrscheinlichkeit, dass nach dem Experiment jeder Ball allein in seinem Korb liegt? Im Fall des Geburtstagsproblems gilt (wenn man von Schaltjahren absieht und Gleichwahrscheinlichkeit der Geburtstage annimmt) $n = 365$.

Für $m > n$ folgt aus dem Schubfachprinzip, dass es immer einen Korb mit mehr als einem Ball gibt. Wir fordern deshalb $0 < m \leq n$. Zur Berechnung der Lösung stellen wir uns vor, dass die Bälle nacheinander geworfen werden. A_i bezeichne das Ereignis „Ball i landet in einem noch leeren Korb". Das gesuchte Ereignis „Alle Bälle liegen allein in einem Korb" bezeichnen wir mit A. Nach Satz 2.10 können wir $\Pr[A]$ berechnen durch

$$\begin{aligned}\Pr[A] &= \Pr\left[\cap_{i=1}^{m} A_i\right] \\ &= \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_2 \cap A_1] \cdots \Pr[A_m| \cap_{i=1}^{m-1} A_i].\end{aligned}$$

$\Pr[A_j| \cap_{i=1}^{j-1} A_i]$ bezeichnet die Wahrscheinlichkeit, dass der j-te Ball in einer leeren Urne landet, wenn bereits die vorherigen $j-1$ Bälle jeweils allein in einer Urne gelandet sind. Wenn unter dieser Bedingung A_j eintritt, so muss der j-te Ball in eine der $n-(j-1)$ leeren Urnen fallen, die aus Symmetriegründen jeweils mit derselben Wahrscheinlichkeit gewählt werden. Daraus folgt

$$\Pr[A_j| \cap_{i=1}^{j-1} A_i] = \frac{n-(j-1)}{n} = 1 - \frac{j-1}{n}.$$

Mit der Abschätzung $1 - x \leq e^{-x}$ und wegen $\Pr[A_1] = 1$ erhalten wir

$$\Pr[A] = \prod_{j=2}^{m} \left(1 - \frac{j-1}{n}\right) \leq \prod_{j=2}^{m} e^{-(j-1)/n} = e^{-(1/n)\cdot\sum_{j=1}^{m-1} j} = e^{-m(m-1)/(2n)}.$$

Abbildung 2.2 zeigt den Verlauf der Funktion

$$f(m) := e^{-m(m-1)/(2\cdot 365)}.$$

Bei 50 Personen ist die Wahrscheinlichkeit, dass mindestens zwei Personen am selben Tag Geburtstag haben, bereits grösser als 95%.

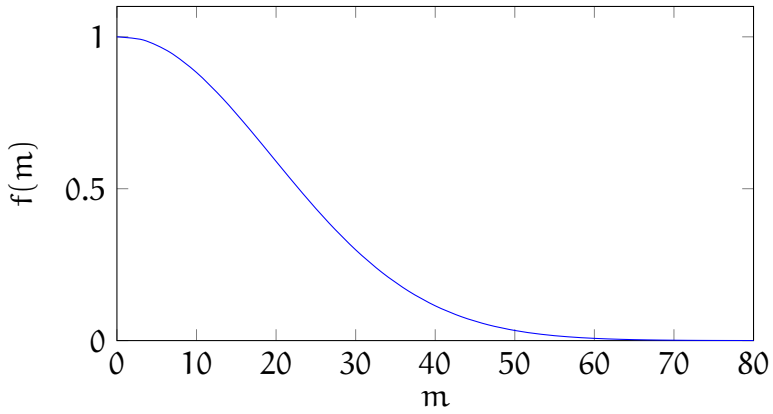


Abbildung 2.2: Die Funktion $f(m) = e^{-m(m-1)/(2 \cdot 365)}$.

Analysen ähnlich der in Beispiel 2.11 werden insbesondere für die Untersuchung von Hash-Verfahren benötigt. Aufgabe von Hash-Verfahren ist es, m Datensätze möglichst gut (und effizient) in n Speicherplätze einzuordnen. Wenn zwei Datensätze demselben Speicherplatz zugeordnet werden, so spricht man von einer Kollision. Da Kollisionen unerwünschte Ereignisse darstellen, möchte man das Verfahren so auslegen, dass Kollisionen nur selten auftreten. Das Geburtstagsproblem beantwortet die Frage, mit welcher Wahrscheinlichkeit keine einzige Kollision auftritt, wenn man die Datensätze *zufällig* verteilen würde. Die in der Praxis verwendeten Verfahren (die so genannten Hash-Funktionen) garantieren zwar keine „völlige“ Gleichverteilung, aber die Abweichungen sind im Allgemeinen so gering, dass die obigen Abschätzungen zumindest näherungsweise zutreffen.

Beispiel 2.12. Wir nehmen an, dass die Datensätze aus einem Universum $\mathcal{K}$ stammen. Da Datensätze auf einem Rechner durch eine Folge von Bits codiert werden, nehmen wir im Folgenden an, dass $\mathcal{K} \subseteq \mathbb{N}_0$ gilt. Nehmen wir weiter an, dass $\mathcal{K} = [p]$, wobei p eine Primzahl ist, so können wir für jedes Paar $a, b \in \mathcal{K}$ mit $a \neq 0$ eine Hash-Funktion wie folgt definieren:

$$\begin{aligned} h_{ab} \quad &: \quad \mathcal{K} \to [n] \\ &\qquad k \mapsto ((ak + b) \bmod p) \bmod n. \end{aligned}$$

Die Annahme, dass p eine Primzahl ist, impliziert, dass es für jedes $k' \in \mathcal{K}$ genau ein $k \in \mathcal{K}$ gibt mit $k' = (ak + b) \bmod p$ (für Nicht-Primzahlen p gilt dies nicht). Die Funktion h_{ab} hat daher für alle Paare $a, b \in \mathcal{K}$ mit $a \neq 0$ die Eigenschaft, dass $|h_{ab}^{-1}(i)| \leq \lceil p/n \rceil$. Das heisst, die Funktion h_{ab} verteilt die Elemente aus $\mathcal{K}$ gleichmässig auf den Speicher $[n]$.

Aus der Vorlesung *Algorithmen und Datenstrukturen* wissen Sie schon, dass die Funktionen h_{ab} sogar eine universelle Familie von Hashfunktionen bildet. Dies bedeutet, dass für alle $k_1, k_2 \in \mathcal{K}$ mit $k_1 \neq k_2$ und für eine uniform zufällig gezogene Hashfunktion gilt: $\Pr[h_{ab}(k_1) = h_{ab}(k_2)] \leq 1/n$.

Ausgehend von der multiplikativen Darstellung der bedingten Wahrscheinlichkeit in (2.1) zeigen wir den folgenden Satz.

Satz 2.13. *(Satz von der totalen Wahrscheinlichkeit)* Die Ereignisse $A_1, \ldots, A_n$ seien paarweise disjunkt und es gelte $B \subseteq A_1 \cup \ldots \cup A_n$. Dann folgt

$$\Pr[B] = \sum_{i=1}^{n} \Pr[B|A_i] \cdot \Pr[A_i].$$

Analog gilt für paarweise disjunkte Ereignisse $A_1, A_2, \ldots$ mit $B \subseteq \bigcup_{i=1}^{\infty} A_i$, dass

$$\Pr[B] = \sum_{i=1}^{\infty} \Pr[B|A_i] \cdot \Pr[A_i].$$

Beweis. Wir zeigen zunächst den endlichen Fall. Wir halten fest, dass

$$B = (B \cap A_1) \cup \cdots \cup (B \cap A_n).$$

Da für beliebige i, j mit $i \neq j$ gilt, dass $A_i \cap A_j = \emptyset$ ist, sind auch die Ereignisse $B \cap A_i$ und $B \cap A_j$ disjunkt. Wegen (2.1) gilt $\Pr[B \cap A_i] = \Pr[B|A_i] \cdot \Pr[A_i]$. Wir wenden nun den Additionssatz an:

$$\begin{aligned} \Pr[B] &= \Pr[B \cap A_1] + \cdots + \Pr[B \cap A_n] \\ &= \Pr[B|A_1] \cdot \Pr[A_1] + \cdots + \Pr[B|A_n] \cdot \Pr[A_n] \end{aligned}$$

und haben damit die Behauptung gezeigt.

Da der Additionssatz auch für unendlich viele Ereignisse $A_1, A_2, \ldots$ gilt, kann dieser Beweis direkt auf den unendlichen Fall übertragen werden. □

Der Satz von der totalen Wahrscheinlichkeit ermöglicht häufig eine einfachere Berechnung komplexer Wahrscheinlichkeiten, indem man die Ergebnismenge Ω geschickt in mehrere Fälle zerlegt und diese getrennt betrachtet. Das folgende bekannte Problem lässt sich mit dieser Technik recht einfach lösen.

Beispiel 2.14. *(Ziegenproblem)* Die Kandidatin einer Fernsehshow darf zwischen drei Türen wählen, um ihren Gewinn zu ermitteln. Hinter einer davon befindet sich ein teures Auto, während hinter den beiden anderen als Trostpreis jeweils eine Ziege wartet. Um die Spannung zu steigern, öffnet der Showmaster, nachdem die Kandidatin gewählt hat, eine der beiden übrigen Türen, hinter der sich (wie er weiss) eine Ziege befindet, und bietet

der Kandidatin an, die Tür noch einmal zu wechseln. Würden Sie an ihrer Stelle dieses Angebot annehmen?

Wir betrachten die Ereignisse $A :=$ „Kandidatin hat bei der ersten Wahl das Auto gewählt“ und $G :=$ „Kandidatin gewinnt nach Wechseln der Tür“. Zu berechnen ist $\Pr[G]$. Offensichtlich gilt $\Pr[G|A] = 0$, da die Kandidatin nach dem Wechseln die „richtige“ Tür verlässt. Ferner erhalten wir $\Pr[G|\bar{A}] = 1$, da die Kandidatin nach der ersten Wahl vor einer Ziege stand und die zweite Ziege vom Showmaster aufgedeckt wurde. Folglich muss sich hinter der verbleibenden Tür das Auto befinden. Mit dem Satz von der totalen Wahrscheinlichkeit schliessen wir, dass

$$\Pr[G] = \Pr[G|A] \cdot \Pr[A] + \Pr[G|\bar{A}] \cdot \Pr[\bar{A}] = 0 \cdot \frac{1}{3} + 1 \cdot \frac{2}{3} = \frac{2}{3}.$$

Es zahlt sich also aus, die Tür zu wechseln.

Mit Hilfe von Satz 2.13 erhalten wir leicht einen weiteren nützlichen Satz.

Satz 2.15. *(Satz von Bayes)* Die Ereignisse $A_1, \ldots, A_n$ seien paarweise disjunkt. Ferner sei $B \subseteq A_1 \cup \cdots \cup A_n$ ein Ereignis mit $\Pr[B] > 0$. Dann gilt für ein beliebiges $i = 1, \ldots, n$

$$\Pr[A_i|B] = \frac{\Pr[A_i \cap B]}{\Pr[B]} = \frac{\Pr[B|A_i] \cdot \Pr[A_i]}{\sum_{j=1}^{n} \Pr[B|A_j] \cdot \Pr[A_j]}.$$

Analog gilt für paarweise disjunkte Ereignisse $A_1, A_2, \ldots$ mit $B \subseteq \bigcup_{i=1}^{\infty} A_i$, dass

$$\Pr[A_i|B] = \frac{\Pr[A_i \cap B]}{\Pr[B]} = \frac{\Pr[B|A_i] \cdot \Pr[A_i]}{\sum_{j=1}^{\infty} \Pr[B|A_j] \cdot \Pr[A_j]}.$$

□

Mit dem Satz von Bayes kann man gewissermassen die Reihenfolge der Bedingung umdrehen. Dieses Verfahren kommt insbesondere bei der Entwicklung von medizinischen Tests sehr oft zur Anwednung.

Beispiel 2.16. Wir betrachten einen medizinischen Test zur Früherkennung einer bestimmten Krebsart. Der Test ist binär, das heisst, er kann entweder positiv oder negativ ausfallen. Wir definieren uns einen Wahrscheinlichkeitsraum dadurch, dass wir den Test auf einen

zufälligen Patienten anwenden. Dann können folgende Ereignisse eintreten:

$$\begin{aligned} P &= \text{„der Test ist positiv“}, \\ N &= \text{„der Test ist negativ“}, \\ K &= \text{„der Patient hat Krebs“}, \\ G &= \text{„der Patient hat nicht Krebs“}. \end{aligned}$$

Es gibt nun zwei Arten von Fehlern, die bei dem Test auftreten können: Entweder der Test fällt bei einem gesunden Patienten positiv aus (Fehler 1. Art, *false positive*) oder der Test fällt bei einem kranken Patienten negativ aus (Fehler 2. Art, *false negative*). Bezeichnen wir mit F_1 und F_2 die Ereignisse, dass ein Fehler 1. bzw. 2. Art eintritt, so gilt also

$$\Pr[F_1] = \Pr[P|G] \quad \text{und} \quad \Pr[F_2] = \Pr[N|K].$$

Bei einem guten Test sind offenbar beide Wahrscheinlichkeiten möglichst klein.

Die Fehlerwahrscheinlichkeiten lassen sich empirisch ermitteln. Für dieses Beispiel nehmen wir an, dass ein Fehler 1. Art mit Wahrscheinlichkeit $\Pr[F_1] = 0.02$ und ein Fehler 2. Art mit Wahrscheinlichkeit $\Pr[F_2] = 0.01$ eintritt, also beide Arten von Fehler relativ selten sind. Was bedeutet es nun für einen Patienten, wenn sein Test positiv ausfällt? Hierfür müssen wir $\Pr[K|P]$ betrachten, also die Wahrscheinlichkeit, dass der Patient tatsächlich krank ist, gegeben, dass sein Test positiv war. Dabei ist noch die Angabe wichtig, wie häufig die Krankheit insgesamt ist. Hier nehmen wir an, dass von dieser speziellen Krebsart etwa 0.5% der Bevölkerung betroffen ist, das heisst also $\Pr[K] = 0.005$.

Nach dem Satz von Bayes (mit $B = P$, $A_1 = K$ und $A_2 = G$), gilt nun

$$\Pr[K|P] = \frac{\Pr[P|K] \cdot \Pr[K]}{\Pr[P|K] \cdot \Pr[K] + \Pr[P|G] \cdot \Pr[G]}$$

Nach den Angaben haben wir $\Pr[P|G] = \Pr[F_1] = 0.02$, $\Pr[K] = 0.005$ und $\Pr[G] = 1 - \Pr[K] = 0.995$. Nach Definition der bedingten Wahrscheinlichkeit gilt ausserdem $\Pr[N|K] + \Pr[P|K] = (\Pr[N \cap K] + \Pr[P \cap K])/\Pr[K]$ und, da P und K komplementäre Ereignisse sind, somit $\Pr[N \cap K] + \Pr[P \cap K] = \Pr[K]$ und also $\Pr[N|K] + \Pr[P|K] = 1$. Damit folgt $\Pr[P|K] = 1 - \Pr[F_2] = 0.99$. Daher ist

$$\Pr[K|P] = \frac{0.99 \cdot 0.005}{0.99 \cdot 0.005 + 0.02 \cdot 0.995} = 0.1991...,$$

es ist also immer noch eher unwahrscheinlich, dass der Patient wirklich krank ist.

2.3 Unabhängigkeit

Bei einer bedingten Wahrscheinlichkeit $\Pr[A|B]$ kann der Fall auftreten, dass die Bedingung auf B, also das Vorwissen, dass B eintritt, keinen Einfluss auf die Wahrscheinlichkeit hat, mit der wir das Eintreten von A erwarten. In diesem Fall gilt also $\Pr[A|B] = \Pr[A]$ und wir nennen die Ereignisse A und B *unabhängig*. Bevor wir diesen Begriff formalisieren, betrachten wir zunächst ein einführendes Beispiel.

Beispiel 2.17. Wir untersuchen das Zufallsexperiment „Zweimaliges Würfeln mit einem sechsseitigen Würfel“. Als Ergebnismenge verwenden wir

$$\Omega := \{(i,j) \mid 1 \leq i,j \leq 6\}.$$

Dabei bezeichnet das Elementarereignis (i,j) den Fall, dass im ersten Wurf die Zahl i und im zweiten Wurf die Zahl j gewürfelt wurde. Alle Elementarereignisse erhalten nach dem Prinzip von Laplace die Wahrscheinlichkeit $\frac{1}{36}$. Ferner definieren wir die Ereignisse

$$A := \text{Augenzahl im ersten Wurf ist gerade,}$$
$$B := \text{Augenzahl im zweiten Wurf ist gerade.}$$

Es gilt $\Pr[A] = \Pr[B] = \frac{1}{2}$. Wie gross ist $\Pr[B|A]$? Nach unserer Intuition sollte gelten $\Pr[B|A] = \Pr[B] = \frac{1}{2}$, da der Ausgang des ersten Wurfs den zweiten Wurf nicht beeinflusst (der Würfel ist „gedächtnislos“). Folglich gewinnen wir durch die Bedingung, dass A eingetreten ist, keine wesentliche Information in Bezug auf das Ereignis B hinzu. Wir rechnen dies nun auch noch formal nach. Es gilt

$$B \cap A = \{(2,2),(2,4),(2,6),(4,2),(4,4),(4,6),(6,2),(6,4),(6,6)\}$$

und somit

$$\Pr[B|A] = \frac{\Pr[B \cap A]}{\Pr[A]} = \frac{\frac{9}{36}}{\frac{1}{2}} = \frac{1}{2} = \Pr[A].$$

Damit haben wir nachgewiesen, dass das Eintreffen des Ereignisses A keine Auswirkungen hat auf die Wahrscheinlichkeit, mit der das Ereignis B eintrifft.

Die folgende Definition formalisiert den Begriff der Unabhängigkeit.

Definition 2.18. Die Ereignisse A und B heissen *unabhängig*, wenn gilt

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B].$$

Wenn $\Pr[B] \neq 0$ ist, so können wir Definition 2.18 umformen zu

$$\Pr[A] = \frac{\Pr[A \cap B]}{\Pr[B]} = \Pr[A|B].$$

$\Pr[A|B]$ zeigt also für unabhängige Ereignisse A und B das Verhalten, das wir erwartet haben.

Beispiel 2.17 *(Fortsetzung)* Bei den Ereignissen A und B ist die Unabhängigkeit klar, da offensichtlich kein kausaler Zusammenhang zwischen den Ereignissen besteht. Wir betrachten nun noch ein weiteres Ereignis:

$$C := \text{Summe der Augenzahlen beider Würfe beträgt 7.}$$

Da das Ereignis C beide Würfe berücksichtigt, könnte es durchaus sein, dass C von A beeinflusst wird. Wir rechnen jedoch nach, dass

$$A \cap C = \{(2,5),(4,3),(6,1)\}$$

und damit

$$\Pr[A \cap C] = \frac{3}{36} = \frac{1}{12} = \frac{1}{2} \cdot \frac{1}{6} = \Pr[A] \cdot \Pr[C] \quad \text{bzw.} \quad \Pr[C|A] = \Pr[C].$$

Damit haben wir nachgewiesen, dass A und C unabhängig sind. Analog zeigt man die Unabhängigkeit von B und C.

Die Unabhängigkeit von A und C bedeutet intuitiv: Wenn wir wissen, dass A eingetreten ist, so ändert sich dadurch nichts an der Wahrscheinlichkeit, mit der wir das Ereignis C erwarten. Bei der Untersuchung von Ereignis C ist es uns deshalb egal, ob A eintritt oder nicht. In Beispiel 2.17 haben wir gesehen, dass für die zwei Ereignisse A und C die Bedingung der Unabhängigkeit erfüllt ist, obwohl sie nicht wie die Ereignisse A und B in Beispiel 2.17 „physikalisch" getrennt sind.

Unabhängige Ereignisse haben viele „schöne" Eigenschaften, von denen wir noch einige kennen lernen werden. Die Analyse eines Zufallsexperiments wird deshalb stark vereinfacht, wenn man zeigen kann, dass die dabei betrachteten Ereignisse unabhängig sind.

Auch für mehr als zwei Ereignisse kann man Unabhängigkeit definieren. Besonders einfach ist dies wieder, wenn die Ereignisse „physikalisch" getrennt sind, da man für jedes eine neuen Münzwurf durchführt. Das folgende Beispiel zeigt eine interessante Anwendung hierfür.

Beispiel 2.19 (Visuelle Kryptographie). Sagen wir, wir haben ein Bild, das nur aus schwarzen und weissen Pixeln besteht. Mathematisch stellen wir uns das Bild als eine $n \times m$-Matrix von Nullen und Einsen vor, wobei eine 0 ein schwarzes Pixel darstellt. Zum Beispiel betrachten wir folgendes Bild des Buchstaben *W*:

Wir wollen dieses Bild nun verschlüsseln. Hierfür folgen wir einer Idee, die 1994 von MONI NAOR und ADI SHAMIR entwickelt wurde. Wir zerlegen unser Bild B in zwei Bilder B_1 und B_2, die jeweils doppelt so gross sind, also Dimension $2n \times 2m$ haben; hierfür ersetzen wir jedes Pixel von B durch 2×2-Blöcke in B_1 und B_2, wie folgt:

■ $\mapsto$ (▞, ▞) oder (▚, ▚)
□ $\mapsto$ (▞, ▞) oder (▚, ▚),

wobei die Auswahl für jedes Pixel durch einen fairen und unabhängigen Münzwurf getroffen wird. Für das obige Bild eines W erhalten wir zum Beispiel die Bilder:


Diese zwei Bilder sind, wie man leicht sieht, nicht voneinander unabhängig. Jedoch sind innerhalb von B_1 alle 2×2-Blöcke voneinander unabhängig, denn jeder Block ist ja ◪ oder ◩ mit Wahrscheinlichkeit je $1/2$, und zwar unabhängig davon, ob das ursprüngliche Pixel in B schwarz oder weiss ist. Das Gleiche gilt für B_2. Das heisst, auf sich allein gestellt enthalten weder B_1 noch B_2 irgend eine Information über das ursprüngliche Bild. Was passiert aber, wenn wir beide Bilder übereinanderlegen, wobei wir uns die weissen Stellen als Transparent vorstellen? Dann erhalten wir das Bild:

Wir haben also das ursprüngliche Bild wiedergefunden, wenn die Rekonstruktion auch nicht ganz perfekt ist, da die ursprünglich weissen Pixel jetzt 'grau' sind. Auf diese Art kann mit Transparenten ein schwarz-weisses Bild effizient und sicher verschlüsseln.

In Beispiel 2.17 haben wir gesehen, dass zwei Ereignisse unabhängig sein können, obwohl dies aus der Definition der Ereignisse nicht sofort ersichtlich ist. Wenn wir mehr als zwei Ereignisse betrachten, so wird die Situation noch komplizierter. Es kann zum Beispiel sein, dass von drei Ereignissen jeweils zwei unabhängig sind, dass aber die Bedingung auf zwei Ereignisse Einfluss auf die Wahrscheinlichkeit eines dritten Ereignisses hat. Das folgende Beispiel veranschaulicht dies.

Beispiel 2.20. Wir werfen zwei ideale Münzen M_1 und M_2. (Das Wort *ideal* bringt hier zum Ausdruck, dass wir annehmen wollen, dass bei jedem Münzwurf die beiden Ergebnisse „Kopf" K und „Zahl" Z mit gleicher Wahrscheinlichkeit fallen.) Unser Wahrscheinlichkeitsraum besteht also aus den vier Elementarereignissen (K,K), (K,Z), (Z,K), (Z,Z), wobei jedes mit Wahrscheinlichkeit $1/4$ eintritt. Wir betrachten die Ereignisse

$$A := \text{„}M_1 \text{ zeigt Kopf"} = \{(K,K),(K,Z)\}$$
$$B := \text{„}M_2 \text{ zeigt Kopf"} = \{(K,K),(Z,K)\}$$
$$C := \text{„die Resultate sind verschieden"} = \{(K,Z),(Z,K)\}.$$

Nun sind A und B voneinander unabhängig, denn

$$\Pr[A \cap B] = 1/4 = \Pr[A]\Pr[B].$$

Ebenso überprüft man leicht, dass B und C voneinander unabhängig sind; und genauso A und C. Allerdings sind A, B, und C zusammen nicht voneinander unabhängig, denn falls je zwei Ereignisse eintreten, so tritt auf keinen Fall das Dritte ein, also insbesondere

$$\Pr[A \cap B \cap C] = 0 \neq 1/8 = \Pr[A]\Pr[B]\Pr[C].$$

Beispiel 2.20 zeigt, dass es für die Definition der Unabhängigkeit von n Ereignissen $A_1, \ldots, A_n$ nicht genügt, die paarweise Unabhängigkeit der Ereignisse zu verlangen. Wir müssen zudem fordern, dass $\Pr[A_1 \cap \cdots \cap A_n] = \Pr[A_1] \cdots \Pr[A_n]$ gilt. Es ist allerdings auch nicht ausreichend nur diese Bedingung zu fordern, wir unser nächstes Beispiel zeigt.

Beispiel 2.21. Wir wählen eine zufällige Zahl zwischen 1 und 8 und betrachten die Ereignisse

$$A := \text{„die Zahl ist in } \{1,2,3,4\}\text{“} \quad \text{und} \quad B := \text{„die Zahl ist in } \{1,5,6,7\}\text{“}.$$

Ausserdem sei $C = B$. Dann gilt

$$\Pr[A \cap B \cap C] = \Pr[A \cap B] = 1/8 = \Pr[A]\Pr[B]\Pr[C],$$

aber $\Pr[A \cap B] = 1/8 \neq \Pr[A]\Pr[B]$, das heisst, A und B sind nicht unabhängig.

Die vorstehenden Beispiele motivieren, dass wir für die Definition der Unabhängikeit von mehreren Ereignissen fordern müssen, dass *alle Teilmengen* die Unabhängigkeitsbedingung ebenfalls erfüllen. Dies führt zu folgender Definition.

Definition 2.22. Die Ereignisse $A_1, \ldots, A_n$ heissen *unabhängig*, wenn für alle Teilmengen $I \subseteq \{1, \ldots, n\}$ mit $I = \{i_1, \ldots, i_k\}$ gilt, dass

$$\Pr[A_{i_1} \cap \cdots \cap A_{i_k}] = \Pr[A_{i_1}] \cdots \Pr[A_{i_k}]. \tag{2.2}$$

Eine unendliche Familie von Ereignissen A_i mit $i \in \mathbb{N}$ heisst unabhängig, wenn (2.2) für jede endliche Teilmenge $I \subseteq \mathbb{N}$ erfüllt ist.

Das folgende Lemma zeigt eine Bedingung, die äquivalent ist zu Definition 2.22, aber manchmal mit geringerem Aufwand überprüft werden kann.

Lemma 2.23. Die Ereignisse $A_1, \ldots, A_n$ sind genau dann unabhängig, wenn für alle $(s_1, \ldots, s_n) \in \{0,1\}^n$ gilt, dass

$$\Pr[A_1^{s_1} \cap \cdots \cap A_n^{s_n}] = \Pr[A_1^{s_1}] \cdots \Pr[A_n^{s_n}], \tag{2.3}$$

wobei $A_i^0 = \bar{A}_i$ und $A_i^1 = A_i$.

Beweis. Zunächst zeigen wir, dass aus (2.2) die Bedingung (2.3) folgt. Wir beweisen dies durch Induktion über die Anzahl der Nullen in $s_1, \ldots, s_n$. Wenn $s_1 = \cdots = s_n = 1$ gilt, so ist nichts zu zeigen. Andernfalls gelte ohne Einschränkung $s_1 = 0$. Aus dem Additionssatz folgt dann

$$\begin{aligned} \Pr[\bar{A}_1 \cap A_2^{s_2} \cap \cdots \cap A_n^{s_n}] &= \Pr[A_2^{s_2} \cap \cdots \cap A_n^{s_n}] \\ &\quad -\Pr[A_1 \cap A_2^{s_2} \cap \cdots \cap A_n^{s_n}]. \end{aligned}$$

Darauf können wir die Induktionsannahme anwenden und erhalten

$$\begin{aligned} &\Pr[\bar{A}_1 \cap A_2^{s_2} \cap \cdots \cap A_n^{s_n}] \\ &= \Pr[A_2^{s_2}] \cdots \Pr[A_n^{s_n}] - \Pr[A_1] \cdot \Pr[A_2^{s_2}] \cdots \Pr[A_n^{s_n}] \\ &= (1 - \Pr[A_1]) \cdot \Pr[A_2^{s_2}] \cdots \Pr[A_n^{s_n}], \end{aligned}$$

woraus die Behauptung wegen $1 - \Pr[A_1] = \Pr[\bar{A}_1]$ folgt.

Die Gegenrichtung zeigen wir nicht in voller Allgemeinheit, sondern rechnen nur nach, dass die Aussage $\Pr[A_1 \cap A_2] = \Pr[A_1] \cdot \Pr[A_2]$ aus (2.3) folgt. Der allgemeine Beweis verwendet genau denselben Ansatz, ist aber etwas umständlich aufzuschreiben. Es gilt wegen des Satzes von der totalen Wahrscheinlichkeit, dass

$$\begin{aligned} \Pr[A_1 \cap A_2] &= \sum_{s_3,\ldots,s_n \in \{0,1\}} \Pr[A_1 \cap A_2 \cap A_3^{s_3} \cap \cdots \cap A_n^{s_n}] \\ &= \sum_{s_3,\ldots,s_n \in \{0,1\}} \Pr[A_1] \cdot \Pr[A_2] \cdot \Pr[A_3^{s_3}] \cdots \Pr[A_n^{s_n}] \\ &= \Pr[A_1] \cdot \Pr[A_2] \cdot \sum_{s_3 \in \{0,1\}} \Pr[A_3^{s_3}] \cdot \ldots \cdot \sum_{s_n \in \{0,1\}} \Pr[A_n^{s_n}] \\ &= \Pr[A_1] \cdot \Pr[A_2], \end{aligned}$$

und es folgt die Behauptung. □

Definition 2.22 und Lemma 2.23 besagen anschaulich: Um zu überprüfen, ob n Ereignisse unabhängig sind, muss man entweder alle Teilmengen untersuchen oder den Schnitt aller Ereignisse betrachten, wobei an beliebiger Stelle Ereignisse komplementiert werden können. In beiden Fällen bleibt die Eigenschaft erhalten, dass die Wahrscheinlichkeit des Schnitts der Ereignisse dem Produkt der einzelnen Wahrscheinlichkeiten entspricht. Aus der Darstellung in Lemma 2.23 folgt die wichtige Beobachtung, dass für zwei unabhängige Ereignisse A und B auch die Ereignisse $\bar{A}$ und B (und analog auch A und $\bar{B}$ bzw. $\bar{A}$ und $\bar{B}$) unabhängig sind. Auch bei Ereignissen, die durch Vereinigung unabhängiger Ereignisse entstehen, können solche Aussagen getroffen werden, wie das folgende Lemma zeigt.

Lemma 2.24. Seien A, B und C unabhängige Ereignisse. Dann sind auch $A \cap B$ und C bzw. $A \cup B$ und C unabhängig.

Beweis. Die Unabhängigkeit von $A \cap B$ und C folgt aus

$$\Pr[(A \cap B) \cap C] = \Pr[A]\Pr[B]\Pr[C] = \Pr[A \cap B]\Pr[C].$$

Mit der Inklusion-Exklusion-Formel gilt

$$\begin{aligned}\Pr[(A \cup B) \cap C] &= \Pr[(A \cap C) \cup (B \cap C)] \\ &= \Pr[A \cap C] + \Pr[B \cap C] - \Pr[A \cap B \cap C] \\ &= \Pr[C] \cdot (\Pr[A] + \Pr[B] - \Pr[A \cap B]) = \Pr[A \cup B] \cdot \Pr[C],\end{aligned}$$

und daraus folgt die Unabhängigkeit von $A \cup B$ und C. □

Wie wir später noch genauer sehen werden, sind unabhängige Zufallszahlen für randomisierte Algorithmen von grosser Bedeutung. In der Theorie ist es kein Problem, die Existenz solcher Zufallszahlen anzunehmen, aber in der Praxis stellt sich die Frage, wie ein deterministischer Computer überhaupt zufällige Zahlen erzeugen kann. Die Antwort ist offenbar: überhaupt nicht, wenn wir von spezialisierter Hardware absehen, die etwa besondere physikalische Effekte ausnutzt. Dennoch hat jede Programmiersprache mindestens eine Funktion, um „Zufallszahlen“ zu generieren. Diese Zahlen sind aber nicht zufällig, sondern folgen einem deterministischen Gesetz; die dadurch erzeugte Folge sieht bloss zufällig aus. Etwas genauer ist ein *Pseudozufallszahlengenerator* (engl. *pseudorandom number generator*

oder kurz PRNG) für Zahlen mit n Bits ein Algorithmus, der eine Funktion $f\colon \{0,1\}^m \to \{0,1\}^m \to \{0,1\}^n$ implementiert, wobei m einen natürliche Zahl ist (die Zustandslänge). Ausgehend von einem Startwert $s_0 \in \{0,1\}^m$ (dem sogenannten *seed*) erzeugt der Algorithmus eine Folge von Tupeln $(s_1, a_1), (s_2, a_2), (s_3, a_3), \ldots$, wobei immer $(s_i, a_i) = f(s_{i-1})$ ist. Hierbei ist s_i der 'interne Zustand' des PRNG, den man als Anwender nicht zu Gesicht bekommt, und $a_1, a_2, a_3, \ldots$ ist die eigentliche Folge von generierten Pseudozufallszahlen. Für jeden Startwert s_0 kriegt man also eine eigene Folge. Hierbei ist es natürlich essentiell, dass die Folge $a_1, a_2, a_3, \ldots$ annähernd so aussieht, wie eine Folge von echten unabhängigen Zufallszahlen in $\{0,1\}^n$. Üblicherweise meint man damit, dass die Folge verschiedene statistische Tests erfüllt, die von unabhängigen Zufallszahlen ebenfalls erfüllt werden. So soll zum Beispiel jede fixe Zahl in einer genügend langen Teilfolge $a_1, a_2, a_3, \ldots, a_k$ etwa $k/|\{0,1\}|^n$ mal vorkommen. In weiterführenden Vorlesungen wird darauf genauer eingegangen.

In dieser Vorlesung werden wir zur Einfachheit immer annehmen, dass wir in einem Algorithmus echte Zufallszahlen benutzen können. Diese Annahme ist nicht ganz ungerechtfertigt, denn wenn der Algorithmus wesentlich schlechter mit Pseudozufallszahlen als mit echten Zufallszahlen funktionieren würde, so hätten wir einen effizienten Test gefunden, der die Pseudozufallszahlen von echten Zufallszahlen unterscheiden kann.

2.4 Zufallsvariablen

Bislang haben wir uns bei der Untersuchung eines Zufallsexperiments darauf beschränkt, die Wahrscheinlichkeiten bestimmter Ereignisse zu berechnen. Oftmals sind wir aber gar nicht an den Wahrscheinlichkeiten der einzelnen Ereignisse interessiert, sondern wir beobachten eine „Auswirkung" oder ein „Merkmal" des Experiments. Wir werden im Folgenden nur numerische Merkmale betrachten, das heisst, wir ordnen jedem Ausgang des Experiments eine bestimmte Zahl zu. Dabei kann es sich zum Beispiel um das bei einem Glückspiel gewonnene bzw. verlorene Geld handeln oder um die Anzahl korrekt übertragener Bytes auf einem unsicheren Kanal. So eine Zuordnung stellt mathematisch gesehen nichts anderes dar als eine Abbildung. Damit erhalten wir die folgende Definition.

Definition 2.25. Eine *Zufallsvariable* ist ein Abbildung $X\colon \Omega \to \mathbb{R}$, wobei Ω die Ergebnismenge eines Wahrscheinlichkeitsraumes ist.

Bei diskreten Wahrscheinlichkeitsräumen ist der *Wertebereich* einer Zufallsvariablen

$$W_X := X(\Omega) = \{x \in \mathbb{R} \mid \exists\ \omega \in \Omega \text{ mit } X(\omega) = x\}$$

immer endlich oder abzählbar unendlich, je nach dem, ob Ω endlich oder abzählbar unendlich ist. Die Bezeichnung *abzählbar* steht hierbei für die Tatsache, dass wir sämtliche Elemente des Wertebereichs in der Form $W_X = \{x_1, x_2, \ldots\}$ auflisten können. Formal definiert man: Eine Menge S heisst abzählbar unendlich genau dann, wenn es eine bijektive Abbildung von $\mathbb{N}$ nach S gibt. Die einzigen Mengen, die unendlich, aber nicht abzählbar unendlich sind, und mit denen wir es im Rahmen dieser Vorlesung zu tun haben werden, sind Teilmengen von $\mathbb{R}$.

Beispiel 2.26. Wir werfen eine ideale Münze dreimal. Als Ergebnismenge erhalten wir $\Omega := \{K, Z\}^3$. Die Zufallsvariable Y bezeichne die Gesamtanzahl der Würfe mit Ergebnis „Kopf“. Beispielsweise gilt also $Y(KZK) = 2$ und $Y(KKK) = 3$. Y hat den Wertebereich $W_Y = \{0, 1, 2, 3\}$.

Wenn man eine Zufallsvariable X untersucht, so interessiert man sich für die Wahrscheinlichkeiten, mit denen X bestimmte Werte annimmt. Für $W_X = \{x_1, \ldots, x_n\}$ bzw. $W_X = \{x_1, x_2, \ldots\}$ betrachten wir (für ein beliebiges $1 \leq i \leq n$ bzw. $x_i \in \mathbb{N}$) das Ereignis $\{\omega \in \Omega \mid X(\omega) = x_i\}$, das wir auch kurz durch $X^{-1}(x_i)$ schreiben können. Anstelle von $X^{-1}(x_i)$ verwendet man häufig auch die (intuitivere) Schreibweise "$X = x_i$". Analog setzt man

$$\Pr[\text{“}X \leq x_i\text{”}] = \sum_{x \in W_X\colon x \leq x_i} \Pr[\text{“}X = x\text{”}] = \Pr[\{\omega \in \Omega \mid X(\omega) \leq x_i\}].$$

In Zukunft werden wir zur weiteren Vereinfachung zusätzlich auf die Anführungszeichen verzichten und schreiben somit statt $\Pr[\text{“}X \leq x_i\text{”}]$ einfach $\Pr[X \leq x_i]$. Analog definiert man

$$\Pr[X \geq x_i], \quad \Pr[2 < X_i \leq 7], \quad \Pr[X^2 \geq 2]\ .$$

Mit Hilfe dieser Notation können wir jeder Zufallsvariablen auf natürliche Weise zwei reelle Funktionen zuordnen. Die Funktion

$$f_X : \mathbb{R} \to [0, 1], \quad x \mapsto \Pr[X = x] \tag{2.4}$$

nennt man *Dichte(funktion)* von X. Die Funktion

$$F_X : \mathbb{R} \to [0,1], \quad x \mapsto \Pr[X \leq x] = \sum_{x' \in W_X : x' \leq x} \Pr[X = x'] \tag{2.5}$$

heisst *Verteilung(sfunktion)* von X. Dichte bzw. Verteilungsfunktion beschreiben eine Zufallsvariable eindeutig. Oft genügt es daher, lediglich die Dichte einer Zufallsvariablen anzugeben. Wir werden dies in Abschnitt 2.5 genauer ausführen.

Beispiel 2.26 *(Fortsetzung)* Für die Zufallsvariable Y erhalten wir

$$\begin{aligned}
\Pr[Y = 0] &= \Pr[ZZZ] = \frac{1}{8}, \\
\Pr[Y = 1] &= \Pr[KZZ] + \Pr[ZKZ] + \Pr[ZZK] = \frac{3}{8}, \\
\Pr[Y = 2] &= \Pr[KKZ] + \Pr[KZK] + \Pr[ZKK] = \frac{3}{8}, \\
\Pr[Y = 3] &= \Pr[KKK] = \frac{1}{8}.
\end{aligned}$$

Abbildung 2.3 zeigt die Dichte und die Verteilung von Y. Bei der Darstellung der Dichte haben wir die Stellen, an denen f_Y von Null verschiedene Werte annimmt, durch breite Balken hervorgehoben. Formal gesehen befindet sich an diesen Stellen jedoch jeweils nur ein einzelner Punkt.

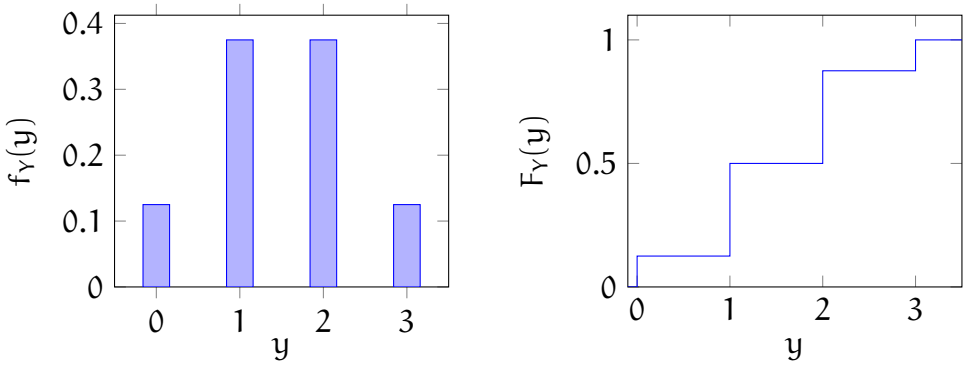


Abbildung 2.3: Dichte- und Verteilungsfunktion von Y.

2.4.1 Erwartungswert

Bei der Untersuchung einer Zufallsvariablen ist es interessant zu wissen, welches Ergebnis man „im Mittel“ erwarten kann. Wenn wir beispielsweise beim Würfeln für jede Sechs einen Euro gewinnen, so erwarten wir

bei n Würfen einen Gewinn von etwa $n/6$ Euro, da voraussichtlich bei ungefähr einem Sechstel der Würfe eine Sechs fallen wird. Man kann also sagen, dass wir bei jedem Spiel im Mittel $1/6$ Euro gewinnen.

Nun modifizieren wir das Spiel und gehen davon aus, dass wir für jede gerade Augenzahl zehn Euro erhalten. Mit derselben Argumentation wie zuvor folgt, dass unser Gewinn pro Spiel im Mittel $0{,}5 \cdot 10 = 5$ Euro beträgt. Diese Überlegungen lassen es sinnvoll erscheinen, den Wert der Zufallsvariable mit der Wahrscheinlichkeit für das Auftreten dieses Wertes zu gewichten.

Definition 2.27. Zu einer Zufallsvariablen X definieren wir den *Erwartungswert* $\mathbb{E}[X]$ durch

$$\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x],$$

sofern die Summe absolut konvergiert. Ansonsten sagen wir, dass der Erwartungswert undefiniert ist.

Beispiel 2.4 *(Fortsetzung)* Der Erwartungswert $\mathbb{E}[Y]$ für die Anzahl „Kopf“ bei dreimaligen Werfen einer idealen Münze ist

$$\begin{aligned}\mathbb{E}[Y] &= \sum_{i=0}^{3} i \cdot \Pr[Y = i] = 1 \cdot \Pr[Y = 1] + 2 \cdot \Pr[Y = 2] + 3 \cdot \Pr[Y = 3] \\ &= 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} = \frac{3}{2},\end{aligned}$$

wie wir auch intuitiv vermutet hätten.

Der Zusatz in Definition 2.27 über die Konvergenz der Summe mag auf den ersten Blick ein wenig merkwürdig erscheinen. In der Tat ist diese Bedingung bei endlichen Wahrscheinlichkeitsräumen trivialerweise erfüllt. Bei unendlichen Wahrscheinlichkeitsräumen ist jedoch Vorsicht geboten. In diesem Fall besitzen die Summen über $x \in W_X$ unendlich viele Glieder und ihr Wert kann deshalb unter Umständen nicht definiert sein. Wenn der Erwartungswert einer Zufallsvariablen definiert ist, so sagen wir auch, dass der Erwartungswert „existiert“. Unser nächstes Beispiel zeigt, welche Probleme auftreten können, wenn der Erwartungswert einer Zufallsvariablen nicht definiert ist.

Beispiel 2.28. In einem Casino wird folgendes Spiel angeboten: Eine Münze wird so lange

geworfen, bis sie zum ersten Mal „Kopf“ zeigt. Sei k die Anzahl durchgeführter Würfe. Wenn k ungerade ist, zahlt der Spieler an die Bank 2^k Euro. Andernfalls (k gerade) muss die Bank 2^k Euro an den Spieler auszahlen. Wir definieren die Zufallsvariable G für den Gewinn der Bank deshalb wie folgt:

$$G := \begin{cases} 2^k & \text{falls k ungerade,} \\ -2^k & \text{falls k gerade.} \end{cases}$$

Offenbar tritt das Ereignis „Anzahl Würfe = k“ genau dann ein, wenn die ersten k Münzwürfe zu der Ergebnisfolge

$$\underbrace{Z \ldots Z}_{k-1 \text{ Mal}} K$$

führen. Da das Ergebnis eines einzelnen Münzwurfes mit Wahrscheinlichkeit 1/2 jeweils Kopf oder Zahl ist und die einzelnen Münzwürfe unabhängig sind, erhalten wir:

$$\Pr[\text{„Anzahl Würfe} = k\text{“}] = (1/2)^k.$$

Damit erhalten wir für die Summe in Definition 2.27 den folgenden Ausdruck:

$$\sum_{k=1}^{\infty} (-1)^{k-1} \cdot 2^k \cdot \left(\frac{1}{2}\right)^k = +1 - 1 + 1 - 1 + \cdots .$$

Da diese Summe offensichtlich nicht konvergiert, ist der Erwartungswert $\mathbb{E}[G]$ in diesem Fall nicht definiert. Man überlege sich, dass ähnliches gilt, wenn uns die Bank anbietet, in jedem Fall 2^k Euro auszuzahlen, unabhängig von der Parität von k: dann ist jeder Summand gleich Eins und die Summe konvergiert daher ebenfalls nicht. Auch in diesem Fall ist der Erwartungswert daher nicht definiert.

In dieser Vorlesung werden wir nur Zufallsvariablen betrachten, für die der Erwartungswert existiert. Um die Formulierungen der Beispiele und Sätze im Folgenden nicht unnötig unübersichtlich zu gestalten, werden wir darauf nicht immer gesondert hinweisen, sondern dies ab jetzt immer stillschweigend annehmen.

In obiger Definition haben wir den Erwartungswert mit einer Summe über die Elemente im Wertebereich der Zufallsvariablen definiert. Alternativ können wir aber auch eine Summe über die Elemente des Wahrscheinlichkeitsraums verwenden.

Lemma 2.29. Ist X eine Zufallsvariable, so gilt:

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega].$$

Sehr oft wird der Wertebereich unserer Zufallsvariablen aus nichtnegativen ganzen Zahlen bestehen. Für solche Zufallsvariablen kann man den Erwartungswert sehr elegant mit folgender Formel ausrechnen.

Satz 2.30. Sei X eine Zufallsvariable mit $W_X \subseteq \mathbb{N}_0$. Dann gilt

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i].$$

Beweis. Nach Definition gilt

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=0}^{\infty} i \cdot \Pr[X = i] = \sum_{i=0}^{\infty} \sum_{j=1}^{i} \Pr[X = i] \\ &= \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} \Pr[X = i] = \sum_{j=1}^{\infty} \Pr[X \geq j].\end{aligned}$$

□

Beispiel 2.31. Wir werfen eine Münze, von der wir wissen, dass die Wahrscheinlichkeit von „Kopf“ gleich p ist, wobei $0 < p < 1$, so lange, bis wir das erste Mal Kopf sehen. X sei die Zufallsvariable, die zählt, wie oft wir die Münze werfen mussten. Wir verwenden Satz 2.30 um den Erwartungswert von X auszurechnen. Offenbar gilt $X \geq j$ dann und nur dann, wenn die ersten $j - 1$ Versuche jeweils „Zahl“ geliefert haben. Dies geschieht mit Wahrscheinlichkeit $(1-p)^{j-1}$ Somit gilt:

$$\mathbb{E}[X] = \sum_{j \geq 1} (1-p)^{j-1} = \sum_{j \geq 0} (1-p)^j = 1/p,$$

wobei wir verwendet haben, dass für alle $0 < x < 1$ gilt $\sum_{n \geq 0} x^n = 1/(1-x)$.

Aus Satz 2.30 ergibt sich sehr einfach (Übungsaufgabe!), dass für alle Zufallsvariablen X mit Wertebereich $W_X \subseteq \mathbb{N}_0$ und jedes $t > 0$ gilt:

$$\Pr[X \geq t] \leq \mathbb{E}[X]/t.$$

Diese Ungleichung nennt man *Markov-Ungleichung*, benannt nach dem russischen Mathematiker Andrei Markov (1856–1922). Sie ist von zentraler Bedeutung für viele Anwendungen in der Informatik. Wir werden sie in etwas allgemeinerer Form in Abschnitt 2.7.1 beweisen.

Bedingte Zufallsvariablen

In Beispiel 2.14 haben wir gesehen, dass der Satz von der totalen Wahrscheinlichkeit (Satz 2.13) es ermöglicht, die Wahrscheinlichkeit eines Ereignisses auszurechnen, indem wir auf verschiedene Spezialfälle bedingen. Für den Erwartungswert gilt dieses Prinzip analog. Um es zu formulieren, führen wir zunächst noch die Schreibweise einer bedingten Zufallsvariable

ein. Sei X eine Zufallsvariable und A ein Ereignis mit $\Pr[A] > 0$. Mit der Schreibweise $X|A$ deuten wir an, dass wir die Wahrscheinlichkeiten, mit denen die Zufallsvariable X bestimmte Werte annimmt bezüglich der auf A bedingten Wahrscheinlichkeiten berechnen. Es gilt also:

$$\Pr[(X|A) \leq x] = \Pr[X \leq x \mid A] = \frac{\Pr[\{\omega \in A : X(\omega) \leq x\}]}{\Pr[A]}$$

Hiermit ergibt sich dann folgendes Analogon zum Satz von der totalen Wahrscheinlichkeit für die Berechnung von Erwartungswerten.

Satz 2.32. Sei X eine Zufallsvariable. Für paarweise disjunkte Ereignisse $A_1, \ldots, A_n$ mit $A_1 \cup \cdots \cup A_n = \Omega$ und $\Pr[A_1], \ldots, \Pr[A_n] > 0$ gilt

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X|A_i] \cdot \Pr[A_i].$$

Für paarweise disjunkte Ereignisse $A_1, A_2, \ldots$ mit $\bigcup_{i=1}^{\infty} A_k = \Omega$ und $\Pr[A_1]$, $\Pr[A_2], \ldots > 0$ gilt analog

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} \mathbb{E}[X|A_i] \cdot \Pr[A_i].$$

Beweis. Mit Hilfe des Satzes von der totalen Wahrscheinlichkeit rechnen wir nach, dass

$$\begin{aligned} \mathbb{E}[X] &= \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{x \in W_X} x \cdot \sum_{i=1}^{n} \Pr[X = x|A_i] \cdot \Pr[A_i] \\ &= \sum_{i=1}^{n} \Pr[A_i] \cdot \sum_{x \in W_X} x \cdot \Pr[X = x|A_i] = \sum_{i=1}^{n} \Pr[A_i] \cdot \mathbb{E}[X|A_i]. \end{aligned}$$

Der Beweis für den unendlichen Fall verläuft analog. □

Beispiel 2.31 *(Fortsetzung)* Mit Hilfe von Satz 2.30 und der geometrischen Reihe haben wir in Beispiel 2.31 den Erwartungswert für die Anzahl Würfe bis zum ersten Mal „Kopf“ ausgerechnet (für eine Münze mit Wahrscheinlichkeit p für „Kopf“). Mit Hilfe von Satz 2.32 kann man dieses Ergebnis auf sehr elegante Weise auch ohne grosse „Rechnerei“ erhalten. Dazu definieren wir das Ereignis K_1 := „im ersten Wurf fällt Kopf“. Offensichtlich gilt $\mathbb{E}[X|K_1] = 1$. Nehmen wir also an, dass im ersten Wurf *nicht* „Kopf“ gefallen ist. Dann wird das Experiment de facto neu gestartet, da der erste Wurf keine Auswirkungen auf

die folgenden Würfe hat. Bezeichnen wir daher mit X' die Anzahl der Würfe bis zum ersten Auftreten von „Kopf“ im neu gestarteten Experiment, so gilt wegen der Gleichheit der Experimente $\mathbb{E}[X'] = \mathbb{E}[X]$. Damit schliessen wir $\mathbb{E}[X|\bar{K}_1] = 1 + \mathbb{E}[X'] = 1 + \mathbb{E}[X]$. Nun können wir Satz 2.13 anwenden und erhalten

$$\mathbb{E}[X] = \mathbb{E}[X|K_1] \cdot \Pr[K_1] + \mathbb{E}[X|\bar{K}_1] \cdot \Pr[\bar{K}_1] = 1 \cdot p + (1 + \mathbb{E}[X]) \cdot (1 - p).$$

Diese Gleichung können wir nach $\mathbb{E}[X]$ auflösen und erhalten das bereits bekannte Ergebnis $\mathbb{E}[X] = 1/p$.

Linearität des Erwartungswertes

Eine Zufallsvariable ist eine Funktion, die die Elementarereignisse eines Wahrscheinlichkeitsraumes Ω in die reellen Zahlen abbildet. Natürlich gibt es zu einem Wahrscheinlichkeitsraum Ω nicht nur eine Zufallsvariable, sondern sehr viele. Nehmen wir einmal an, wir hätten n Zufallsvariablen definiert:

$$X_1, \ldots, X_n \; : \; \Omega \to \mathbb{R}.$$

Für ein $\omega \in \Omega$ erhalten wir daher n reelle Zahlen $X_1(\omega), \ldots, X_n(\omega)$. Wenn wir uns jetzt noch eine Funktion $f\colon \mathbb{R}^n \to \mathbb{R}$ vorgeben, die aus n reellen Zahlen wieder eine einzige reelle Zahl macht, so sehen wir, dass die Konkatenation $f(X_1, \ldots, X_n)$ wiederum eine Zufallsvariable ist, denn es gilt:

$$f(X_1, \ldots, X_n)\colon \Omega \to \mathbb{R}.$$

Dies gilt für beliebige Funktionen $f\colon \mathbb{R}^n \to \mathbb{R}$. Insbesondere auch für affin lineare Funktionen:

$$\begin{aligned} f\colon \mathbb{R}^n &\to \mathbb{R} \\ (x_1, \ldots, x_n) &\mapsto a_1 x_1 + \cdots + a_n x_n + b, \end{aligned}$$

wobei $a_1, \ldots, a_n, b \in \mathbb{R}$ beliebige reelle Zahlen sind. In diesem Fall schreiben wir die Zufallsvariable $f(X_1, \ldots, X_n)$ üblicherweise explizit als

$$X := a_1 X_1 + \ldots + a_n X_n + b.$$

Der folgende Satz zeigt, dass wir den Erwartungswert einer Summe von Zufallsvariablen sehr einfach berechnen können:

Der Erwartungswert einer Summe von Zufallsvariablen ist die Summe der Erwartungswerte der Zufallsvariablen.

Satz 2.33. *(Linearität des Erwartungswerts)* Für Zufallsvariablen $X_1, \ldots, X_n$ und $X := a_1X_1 + \ldots + a_nX_n + b$ mit $a_1, \ldots, a_n, b \in \mathbb{R}$ gilt

$$\mathbb{E}[X] = a_1\mathbb{E}[X_1] + \ldots + a_n\mathbb{E}[X_n] + b.$$

Beweis. Aus dem Satz der totalen Wahrscheinlichkeit folgt

$$\mathbb{E}[X] = \sum_{i \in W_X} i \cdot \Pr[X = i] = \sum_{\omega \in \Omega} \sum_{i \in W_X} i \cdot \Pr[X = i \cap \omega] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega].$$

Die Behauptung folgt mit Hilfe einiger elementarer Umformungen leicht aus dieser Darstellung des Erwartungswerts:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{\omega \in \Omega} (a_1 \cdot X_1(\omega) + \ldots + a_n \cdot X_n(\omega) + b) \cdot \Pr[\omega] \\
&= a_1 \cdot \Big(\sum_{\omega \in \Omega} X_1(\omega) \cdot \Pr[\omega] \Big) + \ldots + a_n \cdot \Big(\sum_{\omega \in \Omega} X_n(\omega) \cdot \Pr[\omega] \Big) + b \\
&= a_1 \cdot \mathbb{E}[X_1] + \ldots + a_n \cdot \mathbb{E}[X_n] + b.
\end{aligned}$$

Hier haben wir ausserdem benutzt, dass $\sum_{\omega \in \Omega} \Pr[\omega] = 1$. □

Die Linearität des Erwartungswertes ermöglicht es oft, den Erwartungswert einer Zufallsvariablen sehr einfach zu berechnen, obwohl dies auf den ersten Blick nicht möglich scheint. Betrachten wir ein einfaches Beispiel:

Beispiel 2.34. Wir werfen eine ideale Münze m-mal. Mit X bezeichnen wir die Anzahl Teilfolgen, die aus dreimal Kopf bestehen. Beispiel: für $m = 6$ könnte die Ergebnisfolge KKKKZK lauten. In diesem Fall hätte X den Wert 2, denn eine Folge KKK startet sowohl an Position 1, wie auch an Position 2. Da die Vorkommen von KKK überlappen können, ist es auf den ersten Blick nicht klar, wie man $\mathbb{E}[X]$ berechnet. Der Trick ist, X als Summe von vielen Zufallsvariablen zu schreiben, deren Erwartungswerte wir leicht berechnen können. Hier bietet sich das Folgende an. Zunächst überlegen wir uns, dass eine Teilfolge KKK an jeder der Positionen $1, \ldots, m-2$ starten kann. (Die Positionen $m-1$ und m kommen nicht in Frage, da hier die verbleibende Teilfolge ja nur noch Länge 2 bzw. 1 hat.) Für jede dieser Positionen definieren wir eine Variable, die angibt, ob an dieser Stelle eine Teilfolge KKK startet: Für jedes $i = 1, \ldots, m-2$ sei

$$X_i := \begin{cases} 1, & \text{falls an Stelle } i \text{ eine Teilfolge KKK beginnt} \\ 0, & \text{sonst.} \end{cases}$$

Für die gesuchte Zufallsvariable X gilt dann: $X = X_1 + \ldots + X_{m-2}$. Der Erwartungswert von X_i lässt sich einfach berechnen, da X_i ja nur zwei verschiedene Werte annehmen kann:

$$\mathbb{E}[X_i] = 0 \cdot \Pr[X_i = 0] + 1 \cdot \Pr[X_i = 1] = \Pr[X_i = 1].$$

Nach Definition ist $X_i = 1$ genau dann, wenn an der Stelle i eine Teilfolge KKK beginnt, wenn also der ite und der $(i+1)$te und der $(i+2)$te Münzwurf jeweils Kopf ergeben haben. Die Wahrscheinlichkeit hierfür ist $(1/2)^3$. Somit gilt $\mathbb{E}[X_i] = 1/8$ für alle $i = 1, \ldots, m-2$ und daher $\mathbb{E}[X] = (m-2)/8$.

In Beispiel 2.34 haben wir die gewünschte Zufallsvariable als Summe von Zufallsvariablen geschrieben, die jeweils nur die Werte 0 oder 1 annehmen können. Solche Variablen nennt man auch *Indikatorvariablen*. Wie wir in Beispiel 2.34 formal nachgerechnet haben, ist der Erwartungswert einer Indikatorvariablen gleich der Wahrscheinlichkeit, dass das entsprechende Ereignis eintritt:

Beobachtung 2.35. Für ein Ereignis $A \subseteq \Omega$ ist die zugehörige *Indikatorvariable* X_A definiert durch:

$$X_A(\omega) := \begin{cases} 1, & \text{falls } \omega \in A \\ 0, & \text{sonst.} \end{cases}$$

Für den Erwartungswert von X_A gilt: $\mathbb{E}[X_A] = \Pr[A]$.

Mit Hilfe von Indikatorvariablen können wir nun einen einfachen Beweis des Prinzips der Inklusion und Exklusion angeben, der keine Eigenschaften der binomischen Summe erfordert, sondern nur mit der Linearität des Erwartungswertes auskommt.

Beispiel 2.36 (Beweis von Satz 2.5). Zur Erinnerung: Zu Ereignissen $A_1, \ldots, A_n$ wollen wir die Wahrscheinlichkeit $\Pr[B]$ des Ereignisses $B := A_1 \cup \ldots \cup A_n$ ermitteln. Wir betrachten die Indikatorvariablen $I_i := I_{A_i}$ der Ereignisse $A_1, \ldots, A_n$ und die Indikatorvariable $I_{\bar{B}}$ des Ereignisses $\bar{B} := \Omega \setminus B$. Wir wissen bereits (siehe Beobachtung 2.35), dass der Erwartungswert einer Indikatorvariablen eines Ereignisses gleich der Wahrscheinlichkeit ist, dass das dieses Ereignis eintrifft. Insbesondere gilt daher: $\mathbb{E}[I_{\bar{B}}] = \Pr[\bar{B}] = 1 - \Pr[B]$. Das Produkt $\prod_{i=1}^{n}(1 - I_{A_i})$ ist genau dann gleich Eins, wenn alle Variablen I_{A_i} gleich Null sind, das heisst wenn B nicht eintritt. Somit gilt $I_{\bar{B}} = \prod_{i=1}^{n}(1 - I_i)$ und wir erhalten durch Ausmultiplizieren

$$I_{\bar{B}} \;=\; 1 - \sum_{1 \le i \le n} I_i + \sum_{1 \le i_1 < i_2 \le n} I_{i_1} I_{i_2} - + \ldots + (-1)^n I_1 \cdot \ldots \cdot I_n.$$

Mit Hilfe der Linearität des Erwartungswertes (Satz 2.33) folgt daraus

$$\mathbb{E}[I_{\bar{B}}] = 1 - \sum_{1 \le i \le n} \mathbb{E}[I_{A_i}] + \sum_{1 \le i_1 < i_2 \le n} \mathbb{E}[I_{A_{i_1}} I_{A_{i_2}}] - + \ldots + (-1)^n \mathbb{E}[I_{A_1} \cdot \ldots \cdot I_{A_n}].$$

Wenn wir nun verwenden, dass das Produkt von Indikatorvariablen ebenfalls eine Indikatorvariable ist, nämlich der Indikator für den Schnitt der beteiligten Ereignisse, so erhalten

wir $\mathbb{E}[I_{i_1} \cdot \ldots \cdot I_{i_k}] = \Pr[A_{i_1} \cap \ldots \cap A_{i_k}]$, für alle Tupel $1 \leq i_1 < \ldots < i_k \leq n$, und Satz 2.5 ist bewiesen.

Oftmals werden wir das einer Indikatorvariable zugrunde liegende Ereignis A nicht explizit definieren, sondern implizit durch eine textuelle Beschreibung. Im Folgenden geben wir ein weiteres Beispiel für dieses Prinzip an. Diesmal in Form eines sogenannten verteilten Algorithmus.

Beispiel 2.37. Ein *verteilter Algorithmus* ist ein Algorithmus, der von mehreren verschiedenen Prozessen gemeinsam ausgeführt wird, wobei die Prozesse untereinander ausschliesslich durch Austausch von Nachrichten kommunizieren können. Solche verteilten Algorithmen kommen in der Regel in Netzwerken von Rechnern zum Einsatz, wenn ein einzelner Rechner nicht genügend Rechenleistung zur Lösung des Problems hat, oder wenn die Eingabe selbst in verteilter Form vorliegt. Letzteres ist zum Beispiel der Fall, wenn die einzelnen Rechner die Struktur des gesamten Netzwerkes nicht kennen, sie aber dennoch eine Teilstruktur des Netzes finden sollen, das eine vorgegebene Eigenschaft hat. Im Folgenden betrachten wir eine einfache Form solche eines Problems.

Gegeben sei ein Graph $G = (V, E)$ mit $|V| = n$ und $|E| = m$. Wir möchten eine möglichst grosse Teilmenge $S \subseteq V$ bestimmen, so dass $G[S]$ keine Kante enthält (man sagt auch: Die Menge S ist *stabil* oder *unabhängig*). Dies tun wir mit einem verteilten Algorithmus, wobei wir annehmen dass für jeden Knoten von v ein eigener Prozess P_v zuständig ist. Die verschiedenen Prozesse müssen nun untereinander ausmachen, welcher Knoten Teil der Menge S ist und welcher nicht. Genauer gesagt, soll jeder Prozess eine Zahl $S_v \in \{0, 1\}$ berechnen, so dass am Ende $S = \{v \in V \mid S_v = 1\}$ eine stabile Menge ist.

Jeder Prozess P_v führt nun folgenden Algorithmus aus, wobei $p \in (0, 1)$ ein Wert ist, den wir später festlegen werden:

Algorithmus für P_v:

1: Setze $S_v \leftarrow 1$ mit Wahrscheinlichkeit p und $S_v \leftarrow 0$ sonst;
2: **for all** $u \in V$ mit $\{u, v\} \in E$ **do** tausche Nachricht mit P_u aus:
3: **if** $S_u = S_v = 1$ **then**
4: setze einen der Werte S_v und S_u auf Null.
5: **return** S_v

Wir wollen nun anschauen, wir gross die berechnete Menge S im Erwartungswert ist. Dafür definieren noch einige hilfreiche Zufallsvariabeln. Erstens bezeichnen wir mit X_v den Wert, der in Zeile 1 für S_v gewählt wird (es ist möglich, dass am Ende des Algorithmus $S_v \neq X_v$ ist, denn der Wert von S_v kann sich ja noch ändern). Ausserdem setzen wir

$$X := \sum_{v \in V} X_v.$$

Für eine Kante $e = \{u, v\}$ definieren wir die Indikatorvariable Y_e, die genau dann 1 ist falls sowohl P_u als auch P_v ihre Werte S_u bzw. S_v zunächst auf 1 gesetzt haben, falls also

$X_u = X_v = 1$ ist. Ausserdem sei noch $Y := \sum_{e \in E} Y_v$. Nun beobachten wir, dass

$$S \geq X - Y$$

gilt, denn wir konstruieren S ja dadurch, dass wir mit X Knoten anfangen, und dann noch für jede Kante e mit $Y_e = 1$ höchstens einen Knoten löschen. Wegen der Linearität des Erwartungswertes gilt nun

$$\mathbb{E}[S] \geq \mathbb{E}[X] - \mathbb{E}[Y].$$

Beide Erwartungswerte rechnen wir leicht aus, indem wir wieder die Linearität des Erwartungswertes benutzen. Zum Einen ist $\mathbb{E}[X] = \sum_{v \in V} \mathbb{E}[X_v] = np$ und zum Anderen $\mathbb{E}[Y] = \sum_{e \in E} \mathbb{E}[Y_e] = mp^2$. Daher gilt

$$\mathbb{E}[S] \geq np - mp^2.$$

Durch Ableiten sehen wir, dass dieser Erwartungswert maximiert wird, wenn $2mp = n$ ist, dass heisst $p = n/(2m)$. Nehmen wir nun noch an, dass der Graph G d-regulär ist, so gilt $2m = dn$ und daher $p = 1/d$. Damit erhalten wir:

$$\mathbb{E}[S] \geq n/d - n/(2d) = n/(2d).$$

Insbesondere hat daher jeder d-reguläre Graph eine stabile Menge der Grösse mindestens $n/(2d)$.

2.4.2 Varianz

Wenn zwei Zufallsvariablen denselben Erwartungswert besitzen, so können sie sich dennoch deutlich voneinander unterscheiden. Ein besonders wichtiges Merkmal einer Verteilung ist die Streuung um den Erwartungswert. Während bei manchen Zufallsvariablen nur Werte „in der Nähe“ des Erwartungswerts angenommen werden und das Verhalten der Variablen somit durch den Erwartungswert sehr gut charakterisiert wird, gibt es auch Zufallsvariablen, die niemals Werte in der Grössenordnung des Erwartungswerts annehmen. Man betrachte hierzu beispielsweise die Variable X mit $\Pr[X = -10^6] = \Pr[X = 10^6] = 1/2$ und $\mathbb{E}[X] = 0$. Bevor wir ein Mass für die Abweichung vom Erwartungswert einführen, betrachten wir zur Motivation ein Beispiel.

Beispiel 2.38. Wir untersuchen ein faires Roulette-Spiel, das heisst wir verzichten auf die Einführung der Null und beschränken uns somit auf die Zahlen $1, \ldots, 36$. Wir nehmen an, dass wir hinreichend viel Kapital besitzen, um sehr viele Runden in Folge spielen zu können. Nun vergleichen wir zwei Strategien:

Strategie 1: Setze immer auf Rot.

Strategie 2: Setze immer auf die Eins.

Wir setzen immer denselben Betrag, den wir der Einfachheit halber als Eins annehmen. Die Zufallsvariablen G_i geben den Gewinn pro Runde bei Strategie i ($i = 1, 2$) an. Es gilt

$\Pr[G_1 = 1] = \Pr[G_1 = -1] = \frac{1}{2}$, da in der Hälfte der Fälle „Rot“ fällt und der doppelte Einsatz ausgezahlt wird. Ferner erhalten wir $\Pr[G_2 = 35] = \frac{1}{36}$ und $\Pr[G_2 = -1] = \frac{35}{36}$, da die Eins nur mit Wahrscheinlichkeit $1/36$ fällt, dafür jedoch der 36-fache Einsatz ausgezahlt wird. Damit folgt

$$\mathbb{E}[G_1] = 1 \cdot \tfrac{1}{2} + (-1) \cdot \tfrac{1}{2} = 0 \qquad \mathbb{E}[G_2] = 35 \cdot \tfrac{1}{36} + (-1) \cdot \tfrac{35}{36} = 0.$$

Welche Strategie ist nun vorzuziehen? Beim Roulette ist dies wohl eine Charakterfrage. Überträgt man das Szenario andererseits auf ein Problem aus der Informatik, so wird der Unterschied recht schnell deutlich. Dazu nehmen wir an, dass wir ein Datenbanksystem implementieren sollen. Die Bearbeitungsstrategie dieses Systems sei teilweise zufallsgesteuert. Die Zufallsvariable $G'_i := 100 + 50 \cdot G_i$ sei die Anzahl der Anfragen, die pro Minute beantwortet werden können, wenn das System mit Strategie i arbeitet.

Bei Strategie 1 werden jeweils in der Hälfte der Fälle 50 Anfragen bzw. 150 Anfragen bearbeitet. Bei Strategie 2 werden in seltenen Fällen 1850 Anfragen pro Minute beantwortet, in $\frac{35}{36}$ der Fälle werden jedoch nur 50 Anfragen erledigt. Obwohl beide Systeme im Mittel pro Minute die selbe Anzahl von Anfragen beantworten, ist für ein reales System das Verhalten von Strategie 1 in der Regel vorzuziehen, da bei Strategie 2 Anwender meist ein deutlich geringe Performance beobachten. In anderen Worten: Strategie 1 verhält sich ausgeglichener und ist daher zu bevorzugen.

Beispiel 2.38 zeigt, dass es bei vielen Zufallsvariablen sinnvoll ist, die zu erwartende Abweichung vom Erwartungswert zu untersuchen. Eine nahe liegende Lösung wäre, $\mathbb{E}[|X - \mu|]$ zu berechnen, wobei $\mu = \mathbb{E}[X]$ sei. Dies scheitert jedoch meist an der „unhandlichen“ Betragsfunktion. Aus diesem Grund betrachtet man stattdessen $\mathbb{E}[(X - \mu)^2]$, also die quadratische Abweichung vom Erwartungswert.

Definition 2.39. Für eine Zufallsvariable X mit $\mu = \mathbb{E}[X]$ definieren wir die *Varianz* $\mathrm{Var}[X]$ durch

$$\mathrm{Var}[X] := \mathbb{E}[(X - \mu)^2] = \sum_{x \in W_X} (x - \mu)^2 \cdot \Pr[X = x].$$

Die Grösse $\sigma := \sqrt{\mathrm{Var}[X]}$ heisst *Standardabweichung* von X.

Bei Zufallsvariablen über unendlichen Wahrscheinlichkeitsräumen existiert die Varianz $\mathrm{Var}[X] = \mathbb{E}[(X - \mu)^2]$ bei manchen Zufallsvariablen nicht. Dies ist genau dann der Fall, wenn der entsprechende Erwartungswert $\mathbb{E}[(X - \mu)^2]$ nicht existiert. In dieser Vorlesung werden wir nur Zufallsvariablen betrachten, bei der die Varianz definiert ist.

Mit der folgenden Formel kann man die Varianz oft einfacher berechnen als durch direkte Anwendung der Definition.

Satz 2.40. Für eine beliebige Zufallsvariable X gilt

$$\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

Beweis. Sei $\mu := \mathbb{E}[X]$. Nach Definition gilt

$$\mathrm{Var}[X] = \mathbb{E}[(X-\mu)^2] = \mathbb{E}[X^2 - 2\mu \cdot X + \mu^2].$$

Aus der Linearität des Erwartungswertes (Satz 2.33) folgt

$$\mathbb{E}[X^2 - 2\mu \cdot X + \mu^2] = \mathbb{E}[X^2] - 2\mu \cdot \mathbb{E}[X] + \mu^2.$$

Damit erhalten wir

$$\mathrm{Var}[X] = \mathbb{E}[X^2] - 2\mu \cdot \mathbb{E}[X] + \mu^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2. \qquad \square$$

Damit zeigen wir die folgende Rechenregel.

Satz 2.41. Für eine beliebige Zufallsvariable X und $a, b \in \mathbb{R}$ gilt

$$\mathrm{Var}[a \cdot X + b] = a^2 \cdot \mathrm{Var}[X].$$

Beweis. Aus der in Satz 2.33 gezeigten Linearität des Erwartungswerts folgt $\mathbb{E}[X+b] = \mathbb{E}[X] + b$. Zusammen mit der Definition der Varianz ergibt sich damit sofort

$$\mathrm{Var}[X+b] = \mathbb{E}[(X+b-\mathbb{E}[X+b])^2] = \mathbb{E}[(X-\mathbb{E}[X])^2] = \mathrm{Var}[X].$$

Mit Hilfe von Satz 2.40 erhalten wir

$$\mathrm{Var}[a \cdot X] = \mathbb{E}[(aX)^2] - \mathbb{E}[aX]^2 = a^2\mathbb{E}[X^2] - (a\mathbb{E}[X])^2 = a^2 \cdot \mathrm{Var}[X].$$

Durch Kombination der beiden Aussagen folgt die Behauptung. $\square$

Der Erwartungswert und die Varianz gehören zu den so genannten *Momenten* einer Zufallsvariablen, die wie folgt definiert sind:

Definition 2.42. Für eine Zufallsvariable X nennen wir $\mathbb{E}[X^k]$ das k-*te Moment* und $\mathbb{E}[(X-\mathbb{E}[X])^k]$ das k-*te zentrale Moment*.

Der Erwartungswert ist also identisch zum ersten Moment, während die Varianz dem zweiten zentralen Moment entspricht.

2.5 Wichtige diskrete Verteilungen

Erinnern wir uns: Ein Wahrscheinlichkeitsraum ist gegeben durch eine Ergebnismenge Ω und eine Wahrscheinlichkeitsfunktion $\Pr\colon \Omega \to [0,1]$, die jedem Elementarereignis $\omega \in \Omega$ eine Wahrscheinlichkeit $\Pr[\omega]$ zuordnet, wobei wir verlangen, dass $\sum_{\omega\in\Omega} \Pr[\omega] = 1$ gilt. Eine Zufallsvariable X ist eine Funktion $X : \Omega \to \mathbb{R}$ von dem Ergebnisraum Ω in die reellen Zahlen. Die Wahrscheinlichkeitsfunktion $\Pr[\cdot]$ induziert die Dichte von X gemäss

$$\begin{aligned} f_X \;:\; & \mathbb{R} \to [0,1] \\ & x \mapsto \Pr[X = x] = \Pr[\{\omega \mid X(\omega) = x\}]. \end{aligned}$$

und analog die Verteilungsfunktion

$$\begin{aligned} F_X \;:\; & \mathbb{R} \to [0,1] \\ & x \mapsto \Pr[X \leq x] = \Pr[\{\omega \mid X(\omega) \leq x\}]. \end{aligned}$$

Oft ist es nicht wirklich wichtig, wie die Ergebnismenge Ω definiert ist. Was uns stattdessen interessiert, ist die Dichte oder die Verteilung einer Zufallsvariablen. Betrachten wir hierzu ein (übertriebenes) Beispiel:

Beispiel 2.43. Werfen wir eine Münze, so bezeichnet man in der deutschsprachigen Literatur die beiden möglichen Ergebnisse meist als *Kopf* und *Zahl*, abgekürzt durch K und Z. Unser Ergebnisraum ist daher $\Omega = \{K, Z\}$. In der englischsprachigen Literatur verwendet man statt dessen die beiden Begriffe *head* and *tail*, abgekürzt durch H und T. Dort würde man die Ergebnismenge daher mit $\Omega = \{H, T\}$ bezeichnen. Betrachten wir jetzt die Indikatorvariable X für *Kopf* (bzw X' für *head*), so gilt für beide Dichten:

$$f_X(x), f_{X'}(x) = \begin{cases} p, & \text{wenn } x = 1 \\ 1-p, & \text{wenn } x = 0 \\ 0, & \text{sonst,} \end{cases}$$

vorausgesetzt $\Pr[K] = \Pr[H] = p$. Das heisst, beide Ergebnismengen $\Omega = \{K, Z\}$ und $\Omega = \{H, T\}$ führen zur gleichen Dichtefunktion.

In diesem Abschnitt stellen wir einige Dichten von Zufallsvariablen vor, die sehr häufig auftreten. Genauer, handelt es sich dabei um Klassen von Zufallsvariablen, die von einem oder mehreren *Parametern* abhängen. Eigentlich betrachten wir also immer eine ganze Familie von ähnlichen Zufallsvariablen.

2.5.1 Bernoulli-Verteilung

Eine Zufallsvariable X mit $W_X = \{0, 1\}$ und der Dichte

$$f_X(x) = \begin{cases} p & \text{für } x = 1, \\ 1 - p & \text{für } x = 0, \\ 0 & \text{sonst} \end{cases}$$

heisst *Bernoulli-verteilt*. Den Parameter p nennt man die *Erfolgswahrscheinlichkeit* der Bernoulli-Verteilung. Wie wir im vorigen Beispiel gesehen haben, erhält man eine solche Verteilung zum Beispiel für die Indikatorvariable für *Kopf* bei dem Wurf einer Münze mit Wahrscheinlichkeit p für *Kopf*. Ist eine Zufallsvariable X Bernoulli-verteilt mit Parameter p, so schreibt man dies auch als

$$X \sim \text{Bernoulli}(p).$$

Für eine solche Bernoulli-verteilte Zufallsvariable X gilt

$$\mathbb{E}[X] = p \quad \text{und} \quad \text{Var}[X] = p(1 - p),$$

wobei letzteres aus Satz 2.40 und $\mathbb{E}[X^2] = p$ folgt. Der Name der Bernoulli-Verteilung geht zurück auf den Schweizer Mathematiker JAKOB BERNOULLI (1654–1705). Sein Werk *Ars Conjectandi* stellt eine der ersten Arbeiten dar, die sich mit dem Teil der Mathematik beschäftigen, den wir heute als Wahrscheinlichkeitstheorie bezeichnen.

2.5.2 Binomialverteilung

Eine Bernoulli-verteilte Zufallsvariable erhalten wir zum Beispiel als Indikator für 'Kopf', wenn wir ein Münze einmal werfen. Werfen wir die Münze statt dessen n-mal und fragen wie oft wir Kopf erhalten haben, so ist die entsprechende Zufallsvariable *binomialverteilt*.

Beispiel 2.44. Wir werfen eine Münze mit Wahrscheinlichkeit p für Kopf n-mal. X sei die Zufallsvariable, die zählt, wie oft Kopf erscheint. Dann ist der Wertebereich offenbar $W_X = \{0, 1, \ldots, n\}$ und es gilt:

$$\begin{aligned} \Pr[X = i] &= \sum_{\omega \in \{K,Z\}^n, X(\omega) = i} \Pr[\omega] = \sum_{\omega \in \{K,Z\}^n, X(\omega) = i} p^i (1-p)^{n-i} \\ &= p^i (1-p)^{n-i} \cdot |\{\omega \in \{K, Z\}^n, \ \omega \text{ enthält genau } i\text{-mal Kopf}\}|, \end{aligned}$$

denn für ein $\omega \in \{K, Z\}^n$ ist $\Pr[\omega]$ gleich p hoch Anzahl K in ω mal $1-p$ hoch Anzahl Z in ω. Für die Dichte von X gilt daher:

$$f_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x}, & x \in \{0, 1, \ldots, n\} \\ 0, & \text{sonst.} \end{cases}$$

Eine Zufallsvariable X mit $W_X = \{0, 1, \ldots, n\}$ und der Dichte wie in Beispiel 2.44 heisst *binomialverteilt* mit Parameter n und p. Man schreibt dies auch als

$$X \sim \mathrm{Bin}(n, p).$$

Für eine solche binomialverteilte Zufallsvariable X gilt

$$\mathbb{E}[X] = np \quad \text{und} \quad \mathrm{Var}[X] = np(1-p),$$

Dies kann man mit Hilfe der Definition von Erwartungswert und Varianz nachrechnen. Wir werden es aber im nächsten Abschnitt auch noch anders (und sehr viel eleganter) herleiten.

2.5.3 Geometrische Verteilung

Wir haben bereits mehrere Male Experimente kennengelernt, bei denen eine Aktion so lange wiederholt wird, bis sie erfolgreich ist (siehe Beispiel 2.31 auf Seite 113 und die Fortsetzung auf Seite 114). Wenn ein einzelner Versuch mit Wahrscheinlichkeit p gelingt, so ist die Anzahl der Versuche bis zum Erfolg *geometrisch verteilt*. Wir schreiben hierfür auch

$$X \sim \mathrm{Geo}(p).$$

Den Parameter p nennt man auch die *Erfolgswahrscheinlichkeit* der geometrischen Verteilung. Für die Dichte einer geometrisch verteilten Zufallsvariable gilt

$$f_X(i) = \begin{cases} p(1-p)^{i-1} & \text{für } i \in \mathbb{N}, \\ 0 & \text{sonst.} \end{cases}$$

Erwartungswert und Varianz sind

$$\mathbb{E}[X] = \frac{1}{p} \quad \text{und} \quad \mathrm{Var}[X] = \frac{1-p}{p^2}.$$

Für den Erwartungswert haben wir dies bereits in Beispiel 2.31 nachgerechnet, die Rechnung für die Varianz überlassen wir dem Leser als Übungsaufgabe. Für die Verteilungsfunktion gilt für alle $n \in \mathbb{N}$:

$$F_X(n) = \Pr[X \leq n] = \sum_{i=1}^{n} \Pr[X = i] = \sum_{i=1}^{n} p(1-p)^{i-1} = 1 - (1-p)^n.$$

Gedächtnislosigkeit. Eine wichtige und oft sehr nützliche Eigenschaft der geometrischen Verteilung ist, dass sie gedächtnislos ist. Was damit gemeint ist, kann man sich leicht an dem Münzbeispiel erklären: Die Wahrscheinlichkeit, dass wir gleich im ersten Versuch 'Kopf' sehen, ist identisch zu der Wahrscheinlichkeit, dass wir nach 1000 Fehlversuchen beim 1001ten Wurf 'Kopf' erhalten. Die Tatsache, dass wir die Münze schon lange (aber erfolglos) geworfen haben, hat also keinerlei Einfluss auf den Erfolg im nächsten Wurf. Formal gilt:

Satz 2.45. Ist $X \sim \mathrm{Geo}(p)$, so gilt für alle $s, t \in \mathbb{N}$:

$$\Pr[X \geq s + t \mid X > s] = \Pr[X \geq t].$$

Beweis. Wir wissen bereits, dass für die Verteilungsfunktion gilt: $F_X(n) = 1-(1-p)^n$ für alle $n \in \mathbb{N}$. Somit gilt $\Pr[X \geq n] = (1-p)^{n-1}$ für alle $n \in \mathbb{N}$. Verwenden wir nun die Definition der bedingten Wahrscheinlichkeit, so erhalten wir daher

$$\Pr[X \geq s+t \mid X > s] = \frac{\Pr[X \geq s+t]}{\Pr[X > s]} = \frac{(1-p)^{s+t-1}}{(1-p)^s} = (1-p)^{t-1} = \Pr[X \geq t],$$

wie behauptet. □

Warten auf den n-ten Erfolg. Bei der geometrischen Verteilung wird das Experiment so lange wiederholt, bis der erste Erfolg eingetreten ist. Dies kann man auf natürliche Weise dahingehend verallgemeinern, dass man auf den n-ten Erfolg wartet. Genauer sei Z die Zufallsvariable, die zählt, wie oft wir ein Experiment mit Erfolgswahrscheinlichkeit p wiederholen müssen, bis zum n-ten Mal Erfolg eintritt. Für $n = 1$ gilt offenbar $Z \sim \mathrm{Geo}(p)$. Für $n \geq 2$ nennt man Z *negativ binomialverteilt* mit Ordnung n.

Die Dichte von Z kann man mit etwas Nachdenken leicht herleiten. Dazu überlegen wir uns folgendes: Z bezeichnet die Anzahl der Versuche bis zum n-ten erfolgreichen Experiment. Wenn $Z = z$ ist, so wurden also genau n erfolgreiche und $z - n$ nicht erfolgreiche Experimente durchgeführt. Da nach Definition von Z das letzte Experiment erfolgreich sein muss, steht der Zeitpunkt des n-ten erfolgreichen Experimentes fest. Die übrigen $n-1$ erfolgreichen Experimente können beliebig auf die restlichen $z-1$ Experimente verteilt werden. Hierfür gibt es genau $\binom{z-1}{n-1}$ Möglichkeiten. Jede

dieser Möglichkeiten tritt mit Wahrscheinlichkeit $p^n(1-p)^{z-n}$ ein. Für die Dichte von Z gilt also

$$f_Z(z) = \binom{z-1}{n-1} \cdot p^n(1-p)^{z-n}.$$

Den Erwartungswert von Z kann man mit Hilfe der Definition des Erwartungswerts aus der Dichte herleiten. Einfacher geht es jedoch mit folgender Überlegung. Wir wissen, dass die Zeit bis zum ersten Erfolg geometrisch verteilt ist und daher Erwartungswert $1/p$ hat. Nach dem ersten Erfolg startet das Experiment jetzt gewissermassen neu, denn wir warten jetzt auf den zweiten Erfolg. Bezeichnen wir daher allgemein mit X_i die Anzahl Experimente *nach* dem $(i-1)$-ten Erfolg und *bis (und mit)* dem i-ten Erfolg, so ist jede der Zufallsvariablen X_i geometrisch verteilt mit Parameter p. Wegen $Z = \sum_{i=1}^{n} X_i$ folgt aus der Linearität des Erwartungswertes (Satz 2.33) daher

$$\mathbb{E}[Z] = \sum_{i=1}^{n} \mathbb{E}[X_i] = n/p.$$

Das Coupon-Collector-Problem. Zum Abschluss dieses Abschnittes betrachten wir noch ein klassisches Beispiel, das in dieser oder einer ähnlichen Form bereits so grosse Mathematiker wie DE MOIVRE, EULER und LAPLACE beschäftigt hat.

Gelegentlich werden Produkten Bilder oder ähnliche Dinge beigelegt, um den Käufer zum Sammeln anzuregen. Wenn es insgesamt n verschiedene Bilder gibt, wie viele Produkte muss man im Mittel erwerben, bis man eine vollständige Sammlung besitzt? Hierbei nehmen wir an, dass bei jedem Kauf die Bilder unabhängig voneinander mit gleicher Wahrscheinlichkeit auftreten.

Wir führen zunächst ein paar Bezeichnungen ein. X sei die Anzahl der Käufe bis zur Komplettierung der Sammlung. Ferner teilen wir die Zeit in Phasen ein: Phase i bezeichne die Schritte vom Erwerb des $(i-1)$-ten Bildes (ausschliesslich) bis zum Erwerb des i-ten Bildes (einschliesslich).

Wenn beispielsweise $n = 4$ gilt und wir die Bilder mit den Zahlen $1, 2, 3, 4$ identifizieren, so könnte ein vollständiges Experiment beispielsweise so aussehen:

$$\underbrace{2}_{1}, \underbrace{2,1}_{2}, \underbrace{2,2,3}_{3}, \underbrace{1,3,2,3,1,4}_{4},$$

wobei die Phasen jeweils durch die geschweiften Klammern gekennzeichnet sind.

X_i sei die Anzahl der Käufe in Phase i. Offensichtlich gilt $X = \sum_{i=1}^{n} X_i$. Phase i wird beendet, wenn wir eines der $n - i + 1$ Bilder erhalten, die wir noch nicht besitzen. Somit ist X_i geometrisch verteilt mit Parameter $p = \frac{n-i+1}{n}$ und es gilt $\mathbb{E}[X_i] = \frac{n}{n-i+1}$.

Mit diesem Wissen können wir $\mathbb{E}[X]$ ausrechnen, denn es folgt wegen der Linearität des Erwartungswerts, dass

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n,$$

wobei $H_n := \sum_{i=1}^{n} \frac{1}{i}$ die n-*te harmonische Zahl bezeichnet.* Aus der Analysis wissen wir, dass $H_n = \ln n + O(1)$ ist und somit folgt, dass $\mathbb{E}[X] = n \ln n + O(n)$. Diese Wartezeit ist erstaunlich kurz, wenn man bedenkt, dass man im optimalen Fall n Bilder erwerben muss. Durch völlig zufälliges Sammeln kommt also im Vergleich zum optimalen Vorgehen nur der Faktor $\ln n$ hinzu.

2.5.4 Poisson-Verteilung

Die sogennante Poisson-Verteilung, benannt nach Siméon Denis Poisson (1781–1840), ist motiviert durch die Betrachtung von Ereignissen, von denen jedes einzelne zwar mit sehr geringer Wahrscheinlichkeit eintritt, wir aber auf Grund der Vielzahl möglicher Ereignisse dennoch erwarten, dass zumindest ein paar Ereignisse eintreten. Hier denke man zum Beispiel an die Möglichkeit, dass ein Bürger in der nächsten Stunde einen Herzinfarkt bekommt. Für den Einzelnen ist dies unwahrscheinlich, schweizweit gibt es aber dennoch etwa drei bis vier pro Stunde. Formal ist eine Poisson-Verteilung (mit Parameter λ) definiert durch die Dichtefunktion

$$f_X(i) = \begin{cases} \frac{e^{-\lambda}\lambda^i}{i!} & \text{für } i \in \mathbb{N}_0 \\ 0 & \text{sonst.} \end{cases}$$

(Um nachzurechnen, dass f_X ist eine zulässige Dichte, dass also $\sum_{i \geq 0} f_X(i) = 1$ gilt, verwende man die aus der Analysis bekannte Reihenentwicklung $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.) Für den Erwartungswert und die Varianz gilt

$$\mathbb{E}[X] = \mathrm{Var}[X] = \lambda.$$

Das Nachrechnen überlassen wir dem Leser als Übungsaufgabe. Um anzugeben, dass eine Zufallsvariable X Poisson-verteilt ist mit Parameter λ schreibt man auch kurz

$$X \sim \mathrm{Po}(\lambda).$$

Poisson-Verteilung als Grenzwert der Binomialverteilung. Um zu erkennen, unter welchen Voraussetzungen die Poisson-Verteilung als sinnvolle Modellierung eines Zufallsexperiments verwendet werden kann, ist es hilfreich zu wissen, dass man die Poisson-Verteilung als Grenzwert einer Bionomialverteilung erhält. Wir betrachten hierzu zunächst ein Beispiel.

Beispiel 2.46. Wir werfen n Bälle unabhängig und gleichverteilt in n Körbe, vgl. Beispiel 2.11. Mit X bezeichnen wir die Anzahl Bälle, die im ersten Korb landen. Da jeder einzelne Ball mit Wahrscheinlichkeit $1/n$ im ersten Korb zu liegen kommt und wir n Bälle werfen, gilt $X \sim \mathrm{Bin}(n, 1/n)$. Für die Dichtefunktion gilt daher

$$f_X(i) = \binom{n}{i} \cdot \frac{1}{n^i} \cdot \left(1 - \frac{1}{n}\right)^{n-i} \qquad \text{für alle } i \in \{0, 1, \ldots, n\}.$$

Betrachten wir nun für ein festes $i \in \mathbb{N}$ den Grenzwert für $n \to \infty$, so erhalten wir

$$\lim_{n\to\infty} f_X(i) = \lim_{n\to\infty} \frac{n(n-1)\ldots(n-i+1)}{i!} \cdot \frac{1}{n^i} \cdot \left(1 - \frac{1}{n}\right)^{n-i} = \frac{1}{i!} \cdot e^{-1},$$

das heisst, für $n \to \infty$ konvergiert X, bzw. die Binomialverteilung $\mathrm{Bin}(n, 1/n)$, gegen die Poisson-Verteilung $\mathrm{Po}(1)$.

Ähnlich wie im vorigen Beispiel rechnet man allgemein nach, dass für jedes $\lambda > 0$ die Binomialverteilung $\mathrm{Bin}(n, \lambda/n)$ für $n \to \infty$ gegen die Poisson-Verteilung mit Parameter λ konvergiert.

Mit anderen Worten: gilt für eine Binomialverteilung, dass der erste Parameter n (die Anzahl Versuche) sehr gross ist und dass das Produkt der beiden Parameter np (also der Erwartungswert) eine (kleine) Konstante ist, so kann man die Binomialverteilung durch die Poisson-Verteilung approximieren, wobei der Parameter der Poisson-Verteilung durch den Erwartungswert der Binomialverteilung gegeben ist.

2.6 Mehrere Zufallsvariablen

Oftmals interessiert man sich bei einem Wahrscheinlichkeitsraum für mehrere Zufallsvariablen zugleich. Wir betrachten dazu ein Beispiel:

Beispiel 2.47. Aus einem Skatblatt mit 32 Karten ziehen wir zufällig eine Hand von zehn Karten sowie einen Skat von zwei Karten. Unter den Karten gibt es vier Buben. Die Zufallsvariable X zählt die Anzahl der Buben in der Hand, während Y die Anzahl der Buben im Skat angibt. Die Werte von X und Y hängen offensichtlich stark voneinander ab. Beispielsweise muss $Y = 0$ sein, wenn $X = 4$ gilt.

Wir beschäftigen uns mit der Frage, wie man mit mehreren Zufallsvariablen über demselben Wahrscheinlichkeitsraum rechnen kann, auch wenn sie sich wie in Beispiel 2.47 gegenseitig beeinflussen. Dazu untersuchen wir für zwei Zufallsvariablen X und Y Wahrscheinlichkeiten der Art

$$\Pr[X = x, Y = y] \quad = \quad \Pr[\{\omega \in \Omega \mid X(\omega) = x, Y(\omega) = y\}].$$

Die Schreibweise $\Pr[X = x, Y = y]$, bei der die über Zufallsvariablen definierten Ereignisse „$X = x$“ und „$Y = y$“ durch Kommata getrennt aufgelistet werden, ist hier als Abkürzung von $\Pr[\text{„}X = x\text{“} \cap \text{„}Y = y\text{“}]$ zu verstehen. Auch kompliziertere Ausdrücke wie beispielsweise $\Pr[X \leq x, Y \leq y_1, \sqrt{Y} = y_2]$ sind üblich.

Beispiel 2.47 *(Fortsetzung)* Wenn wir nur die Zufallsvariable X betrachten, so gilt für $0 \leq x \leq 4$

$$\Pr[X = x] = \frac{\binom{4}{x}\binom{28}{10-x}}{\binom{32}{10}},$$

da zehn der 32 insgesamt vorhandenen Karten für die Hand ausgewählt werden. Von diesen zehn Karten werden x aus den vier Buben gewählt und $10 - x$ aus den restlichen 28 Karten, die keine Buben sind. Für die Zufallsvariable Y erhält man analog für $0 \leq y \leq 2$ $\Pr[Y = y] = \binom{4}{y}\binom{28}{2-y}/\binom{32}{2}$. Wenn wir jedoch X und Y gleichzeitig betrachten, so besteht ein enger Zusammenhang zwischen den Werten der beiden Variablen. Beispielsweise gilt $\Pr[X = 4, Y = 1] = 0$, da insgesamt nur vier Buben vorhanden sind. Allgemein erhalten wir mit einer ähnlichen Argumentation wie zuvor

$$\Pr[X = x, Y = y] = \frac{\binom{4}{x}\binom{28}{10-x}\binom{4-x}{y}\binom{28-(10-x)}{2-y}}{\binom{32}{10}\binom{22}{2}},$$

woraus auch formal folgt, dass zum Beispiel $\Pr[X = 3, Y = 1] \neq \Pr[X = 3]\Pr[Y = 1]$.

Die Funktion

$$f_{X,Y}(x, y) := \Pr[X = x, Y = y]$$

heisst *gemeinsame Dichte* der Zufallsvariablen X und Y. Wenn man die gemeinsame Dichte gegeben hat, kann man auch wieder zu den Dichten der einzelnen Zufallsvariablen übergehen, indem man

$$f_X(x) = \sum_{y \in W_Y} f_{X,Y}(x, y) \quad \text{bzw.} \quad f_Y(y) = \sum_{x \in W_X} f_{X,Y}(x, y)$$

setzt. Die Funktionen f_X und f_Y nennt man *Randdichten*. Die Ereignisse „$Y = y$“ bilden eine disjunkte Zerlegung des Wahrscheinlichkeitsraumes und es gilt somit wegen Satz 2.3

$$\Pr[X = x] \overset{\text{Satz 2.3}}{=} \sum_{y \in W_Y} \Pr[X = x, Y = y] \overset{\text{Def.}}{=} f_X(x).$$

Die Dichten der einzelnen Zufallsvariablen entsprechen also genau den Randdichten. Auch zur Verteilung gibt es ein Analogon: Für zwei Zufallsvariablen definiert man die *gemeinsame Verteilung* als

$$\begin{aligned} F_{X,Y}(x,y) &:= \Pr[X \le x, Y \le y] = \Pr[\{\omega \in \Omega \mid X(\omega) \le x, Y(\omega) \le y\}] \\ &= \sum_{x' \le x} \sum_{y' \le y} f_{X,Y}(x', y') \end{aligned}$$

Auch hier kann man wieder zur *Randverteilung* übergehen, indem man

$$F_X(x) = \sum_{x' \le x} f_X(x') = \sum_{x' \le x} \sum_{y \in W_Y} f_{X,Y}(x', y).$$

ansetzt. Analog erhält man die Randverteilung von Y.

Beispiel 2.48. Wir betrachten folgendes zweistufiges Experiment. Wir werfen zunächst einen Würfel. X bezeichne die geworfene Augenzahl. Danach werfen wir X-mal eine ideale Münze und bezeichnen mit Y die Anzahl Kopf. Wie gross ist der Erwartungswert für Y? – Intuitiv könnten wir folgende Rechnung anstellen: Die erwartete Augenzahl beim Würfel ist $\mathbb{E}[X] = 7/2$. Und wenn wir eine ideale Münze 3.5 mal werfen so erwarten wir (auch wenn wir nicht genau wissen, wie das ‘ein-halb-mal’ Werfen geht), dass in der Hälfte der Fälle Kopf kommt. Das heisst, wir erwarten $\mathbb{E}[Y] = 7/4$. Aber können wir dies auch formal begründen? Ja! Hierzu betrachten wir zunächst die gemeinsame Dichte von X und Y. Dies ist einfach: die Wahrscheinlichkeit für “$X = x$” ist $1/6$ für alle $x \in \{1, \ldots, 6\}$. Und wenn wir die Münze x-mal werfen, so ist die Wahrscheinlichkeit für y-mal Kopf genau $\binom{x}{y}/2^x$ für alle $y \in \{0, 1, \ldots, x\}$. Das heisst, wir haben

$$f_{X,Y}(x,y) = \begin{cases} \frac{1}{6} \cdot \frac{\binom{x}{y}}{2^x} & \text{falls } x \in \{1, \ldots, 6\} \text{ und } y \in \{0, 1, \ldots, x\} \\ 0 & \text{sonst.} \end{cases}$$

Damit erhalten wir folgende Dichte für Y:

$$f_Y(y) = \begin{cases} \sum_{x=\max\{1,y\}}^{6} \frac{1}{6} \cdot \frac{\binom{x}{y}}{2^x} & \text{falls } y \in \{0, 1, \ldots, 6\} \\ 0 & \text{sonst.} \end{cases}$$

Für den Erwartungswert von Y gilt daher in der Tat

$$\mathbb{E}[Y] = \sum_{y=0}^{6} y \cdot f_Y(y) = \sum_{y=1}^{6} y \cdot \sum_{x=y}^{6} \frac{1}{6} \frac{\binom{x}{y}}{2^x} = \frac{1}{6} \sum_{x=1}^{6} \frac{1}{2^x} \sum_{y=1}^{x} y \binom{x}{y} = \ldots = \frac{7}{4},$$

wobei wir die elementaren Rechnungen in “$\ldots$” dem Leser als Übung überlassen.

Beispiel 2.49. Wir wählen eine zufällige Folge von n Nullen und Einsen, wobei jede Folge die gleiche Wahrscheinlichkeit 2^{-n} hat. Wir bezeichnen mit X die Anzahl Einsen in der Folge. Ferner sei $0 \leq m \leq n$ und Y die Anzahl Einsen unter den ersten m Folgengliedern. Die gemeinsame Dichte von X und Y ist für $0 \leq y \leq m$ und $y \leq x \leq n - m + y$

$$f_{X,Y}(x, y) = \Pr[X = x, Y = y] = \binom{n-m}{x-y}\binom{m}{y}2^{-n},$$

denn es gibt $\binom{m}{y}$ Möglichkeiten, y Einsen auf die ersten m Folgenglieder zu verteilen und $\binom{n-m}{x-y}$ Möglichkeiten, die verbleibenden $x - y$ Einsen auf die restlichen $n - m$ Glieder zu verteilen. Die Randdichte von X erhalten wir, indem wir über alle Möglichkeiten für Y summieren:

$$f_X(x) = \Pr[X = x] = \sum_{y=0}^{m} \binom{n-m}{x-y}\binom{m}{y}2^{-n}.$$

Verwenden wir nun noch die sogenannte Vandermonde Identität für $n \geq m$

$$\binom{n}{x} = \sum_{y=0}^{m} \binom{n-m}{x-y}\binom{m}{y},$$

so erhalten wir $\Pr[X = x] = \binom{n}{x}2^{-n}$, was wir natürlich auch direkt gewusst hätten.

2.6.1 Unabhängigkeit von Zufallsvariablen

Bei Ereignissen haben wir den Begriff der Unabhängigkeit kennen gelernt, der intuitiv besagt, dass zwei Ereignisse sich gegenseitig „nicht beeinflussen“. Wie schon bei Ereignissen, so kann auch bei Zufallsvariablen dies „offensichtlich“ der Fall sein (weil die entsprechenden Zufallsvariablen „physikalisch“ unabhängig sind) oder auch nur implizit (weil wir nachrechnen können, dass keine Abhängigkeit vorliegt). Die nächsten beiden Beispiele illustrieren beide Phänomene.

Beispiel 2.50. Wir werfen eine Münze zweimal. Mit X_1 bzw. X_2 bezeichnen wir die Indikatorvariable für das Ereignis, dass der erste bzw. zweite Wurf „Kopf“ ist. Da der erste Wurf keinen Einfluss auf den zweiten Wurf hat, gilt für alle $a, b \in \{0, 1\}$, dass

$$\Pr[X_1 = a, X_2 = b] = \Pr[X_1 = a] \cdot \Pr[X_2 = b].$$

Es ist ebenfalls leicht einzusehen, dass dies *immer* der Fall sein wird, wenn die Variablen X_i Indikatorvariablen für unabhängige Ereignisse sind.

Beispiel 2.51. Wir ziehen zufällig eine Karte aus einem Skatspiel mit 32 Karten und betrachten die Indikatorvariablen X bzw. Y für die folgenden beiden Ereignisse: „die Karte ist eine Herz-Karte“ bzw. „die Karte ist ein Bube“. Dann sind die beiden Variablen offensichtlich nicht „physikalisch“ unabhängig, denn wir ziehen ja nur eine einzige Karte, es gilt aber:

$$\Pr[Y = 1 \mid X = 1] = \frac{1}{8} = \frac{4}{32} = \Pr[Y = 1].$$

Das heisst, ob wir wissen, dass die Karte eine Herz-Karte ist, hat keinen Einfluss auf die Wahrscheinlichkeit, mit der die Karte ein Bube ist.

Die Überlegungen aus Beispiel 2.51 motivieren die folgende, auf Zufallsvariablen erweiterte Definition von Unabhängigkeit.

Definition 2.52. Zufallsvariablen $X_1, \ldots, X_n$ heissen unabhängig genau dann, wenn für alle $(x_1, \ldots, x_n) \in W_{X_1} \times \ldots \times W_{X_n}$ gilt

$$\Pr[X_1 = x_1, \ldots, X_n = x_n] = \Pr[X_1 = x_1] \cdot \cdots \cdot \Pr[X_n = x_n].$$

Alternativ kann man Definition 2.52 auch über die Dichten formulieren: $X_1, \ldots, X_n$ sind genau dann unabhängig, wenn für alle $(x_1, \ldots, x_n) \in W_{X_1} \times \ldots \times W_{X_n}$ gilt

$$f_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = f_{X_1}(x_1) \cdot \cdots \cdot f_{X_n}(x_n).$$

Bei unabhängigen Zufallsvariablen ist also die gemeinsame Dichte gleich dem Produkt der Randdichten. Die Gleichung in Definition 2.52 gilt im Übrigen auch für x_i ausserhalb des Wertebereichs, sie ist dann nur nicht besonders interessant, da dann beide Seiten 0 sind.

Das folgende Lemma zeigt, dass für unabhängigen Zufallsvariablen die Produkteigenschaft nicht nur für einzelne Werte x_i gilt, sondern für beliebige Mengen.

Lemma 2.53. Sind $X_1, \ldots, X_n$ unabhängige Zufallsvariablen und sind $S_1, \ldots, S_n \subseteq \mathbb{R}$ beliebige Mengen, dann gilt

$$\Pr[X_1 \in S_1, \ldots, X_n \in S_n] = \Pr[X_1 \in S_1] \cdot \cdots \cdot \Pr[X_n \in S_n].$$

Beweis. Es genügt, die Gleichung für Teilmengen des Wertebereichs nachzurechnen, da sonstige Elemente die Wahrscheinlichkeiten nicht ändern. Insbesondere können wir annehmen, dass die S_i endlich oder abzählbar

sind. Dann rechnen wir direkt nach:

$$\begin{aligned}
&\Pr[X_1 \in S_1, \ldots, X_n \in S_n] \\
&\quad = \sum_{x_1 \in S_1} \cdots \sum_{x_n \in S_n} \Pr[X_1 = x_1, \ldots, X_n = x_n] \\
&\quad \overset{\text{Unabh.}}{=} \sum_{x_1 \in S_1} \cdots \sum_{x_n \in S_n} \Pr[X_1 = x_1] \cdot \ldots \cdot \Pr[X_n = x_n] \\
&\quad = \left(\sum_{x_1 \in S_1} \Pr[X_1 = x_1] \right) \cdot \ldots \cdot \left(\sum_{x_n \in S_n} \Pr[X_n = x_n] \right) \\
&\quad = \Pr[X_1 \in S_1] \cdot \ldots \cdot \Pr[X_n \in S_n].
\end{aligned}$$

□

Mit Hilfe von Lemma 2.53 erhalten wir insbesondere auch einen formalen Beweis für die intuitiv offensichtlich Tatsache, dass Teilmengen von unabhängigen Zufallsvariablen ebenfalls unabhängig sind.

Korollar 2.54. Sind $X_1, \ldots, X_n$ unabhängige Zufallsvariablen und ist $I = \{i_1, \ldots, i_k\} \subseteq [n]$, dann sind $X_{i_1}, \ldots, X_{i_k}$ ebenfalls unabhängig.

Beweis. Wir rechnen die Bedingung der Definition 2.52 nach. Für $x_{i_j} \in W_{X_{i_j}}$, $1 \leq j \leq k$, setzen wir

$$S_i = \begin{cases} W_{X_i} & \text{falls } i \notin I, \\ \{x_i\} & \text{falls } i \in I. \end{cases}$$

Für $i \notin I$ ist dann $X_i \in S_i$ trivialerweise erfüllt und mit Lemma 2.53 folgt daher

$$\begin{aligned}
\Pr[X_{i_1} = x_{i_1}, \ldots, X_{i_k} = x_{i_k}] &= \Pr[X_1 \in S_1, \ldots, X_n \in S_n] \\
&\overset{\text{Lemma 2.53}}{=} \Pr[X_1 \in S_1] \cdot \ldots \cdot \Pr[X_n \in S_n] \\
&= \Pr[X_{i_1} = x_{i_1}] \cdot \ldots \cdot \Pr[X_{i_k} = x_{i_k}].
\end{aligned}$$

□

Wir wissen bereits, dass die Anwendung einer Funktion f auf eine Zufallsvariable X wieder eine Zufallsvariable liefert, nämlich $f(X) := f \circ X$. Damit

können wir Zufallsvariablen der Form $-X$, $2X+10$, X^2, $(X-\mathbb{E}[X])^2$ etc. betrachten. Mit Hilfe von Lemma 2.53 werden wir nun zeigen, dass durch Anwendung von Funktionen auf Zufallsvariablen deren Unabhängigkeit nicht verloren geht.

Satz 2.55. Seien $f_1, \ldots, f_n$ reellwertige Funktionen ($f_i \colon \mathbb{R} \to \mathbb{R}$ für $i = 1, \ldots, n$). Wenn die Zufallsvariablen $X_1, \ldots, X_n$ unabhängig sind, dann gilt dies auch für $f_1(X_1), \ldots, f_n(X_n)$.

Beweis. Wir betrachten beliebige Werte $z_1, \ldots, z_n$ mit $z_i \in W_{f(X_i)}$ für $i = 1, \ldots, n$. Zu z_i definieren wir die Menge $S_i = \{x \mid f(x) = z_i\}$. Es folgt mit Hilfe von Lemma 2.53

$$\begin{aligned}
&\Pr[f_1(X_1) = z_1, \ldots, f_n(X_n) = z_n] \\
&= \Pr[X_1 \in S_1, \ldots, X_n \in S_n] \overset{\text{Unabh.}}{=} \Pr[X_1 \in S_1] \cdot \cdots \cdot \Pr[X_n \in S_n] \\
&= \Pr[f_1(X_1) = z_1] \cdot \cdots \cdot \Pr[f_n(X_n) = z_n].
\end{aligned}$$

□

Das folgende Beispiel zeigt, dass die Annahme von Satz 2.55, dass die X_i unabhängig sind, hinreichend aber nicht notwendig ist.

Beispiel 2.56. Wählen wir die konstante Funktion $f \equiv 1$, so gilt für jede Zufallsvariable X: $Y := f(X)$ ist eine Zufallsvariable mit Dichte

$$f_Y(y) = \begin{cases} 1 & y = 1 \\ 0 & \text{sonst.} \end{cases}$$

Insbesondere gilt daher: $f(X_1), \ldots, f(X_n)$ sind auch dann unabhängig, wenn $X_1, \ldots, X_n$ abhängig sind.

2.6.2 Zusammengesetzte Zufallsvariablen

Mit Hilfe einer Funktion g können mehrere Zufallsvariablen auf einem Wahrscheinlichkeitsraum miteinander kombiniert werden. Man konstruiert also aus den Zufallsvariablen $X_1, \ldots, X_n$ eine neue, zusammengesetzte Zufallsvariable Y durch $Y := g(X_1, \ldots, X_n)$. Die Wahrscheinlichkeiten der zu Y gehörenden Ereignisse „$Y = y$“ für $y \in W_Y = \{Y(\omega) \mid \omega \in \Omega\}$ berechnen wir wie gewohnt:

$$\Pr[Y = y] = \Pr[\{\omega \in \Omega \mid Y(\omega) = y\}] = \Pr[\{\omega \mid g(X_1(\omega), \ldots, X_n(\omega)) = y\}].$$

Beispiel 2.57. Ein Würfel werde zweimal geworfen. X bzw. Y bezeichne die Augenzahl im ersten bzw. zweiten Wurf. Daraus kann man zum Beispiel die Zufallsvariable $Z := X + Y$ ableiten, die der Summe der gewürfelten Augenzahlen entspricht. Für Z gilt: $\Pr[Z = 1] = \Pr[\emptyset] = 0$, $\Pr[Z = 4] = \Pr[\{(1,3),(2,2),(3,1)\}] = \frac{3}{36}$ usw.

In diesem Beispiel haben wir die Dichte von Z berechnet, indem wir zu jedem möglichen Ergebnis für X den dazu passenden Wert von Y gewählt haben. Im Allgemeinen gilt ein ganz ähnliches Prinzip:

Satz 2.58. Für zwei unabhängige Zufallsvariablen X und Y sei $Z := X + Y$. Es gilt

$$f_Z(z) = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x).$$

Beweis. Mit Hilfe des Satzes von der totalen Wahrscheinlichkeit folgt, dass

$$\begin{aligned} f_Z(z) &= \Pr[Z = z] = \sum_{x \in W_X} \Pr[X + Y = z \mid X = x] \cdot \Pr[X = x] \\ &= \sum_{x \in W_X} \Pr[Y = z - x] \cdot \Pr[X = x] = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x). \end{aligned}$$

□

Den Ausdruck $\sum_{x \in W_X} f_X(x) \cdot f_Y(z - x)$ aus Satz 2.58 nennt man in Analogie zu den entsprechenden Begriffen bei Potenzreihen *Faltung* oder *Konvolution* der Dichten f_X und f_Y.

Beispiel 2.59. Seien X und Y Poisson-verteilt mit Parametern λ_X bzw. λ_Y und sei Z die Zufallsvariable $X + Y$. Dann gilt direkt nach dem Satz 2.58:

$$\begin{aligned} f_Z(z) = \Pr[Z = z] &= \sum_{x=0}^{z} \Pr[X = x] \Pr[Y = z - x] \\ &= e^{-(\lambda_X + \lambda_Y)} \sum_{x=0}^{z} \frac{\lambda_X^x}{x!} \frac{\lambda_Y^{z-x}}{(z-x)!} \\ &= e^{-(\lambda_X + \lambda_Y)} \sum_{x=0}^{z} \frac{\lambda_X^x \lambda_Y^{z-x}}{z!} \frac{z!}{x!(z-x)!} \\ &= e^{-(\lambda_X + \lambda_Y)} \frac{1}{z!} \sum_{x=0}^{z} \lambda_X^x \lambda_Y^{z-x} \binom{z}{x} \\ &= e^{-(\lambda_X + \lambda_Y)} \frac{1}{z!} (\lambda_X + \lambda_Y)^z. \end{aligned}$$

Entsprechend sehen wir, dass Z auch Poisson-verteilt ist mit Parameter $\lambda_Z = \lambda_X + \lambda_Y$.

2.6.3 Momente zusammengesetzter Zufallsvariablen

Im Folgenden zeigen wir einige Rechenregeln für zusammengesetzte Zufallsvariablen. Die Linearität des Erwartungswertes kennen wir bereits aus Satz 2.33.

Satz 2.60. *(Linearität des Erwartungswerts)* Für Zufallsvariablen $X_1, \ldots, X_n$ und $X := a_1X_1 + \cdots + a_nX_n$ mit $a_1, \ldots, a_n \in \mathbb{R}$ gilt

$$\mathbb{E}[X] = a_1\mathbb{E}[X_1] + \cdots + a_n\mathbb{E}[X_n].$$

Dieser Satz ist äusserst nützlich, da keine Voraussetzungen an die Unabhängigkeit der beteiligten Zufallsvariablen gestellt werden. Bei Produkten von Zufallsvariablen können wir hingegen auf diese Voraussetzung nicht verzichten.

Satz 2.61. *(Multiplikativität des Erwartungswerts)* Für unabhängige Zufallsvariablen $X_1, \ldots, X_n$ gilt

$$\mathbb{E}[X_1 \cdot \cdots \cdot X_n] = \mathbb{E}[X_1] \cdot \cdots \cdot \mathbb{E}[X_n].$$

Beweis. Wir beweisen den Fall $n = 2$. Der allgemeine Fall folgt daraus dann sofort per Induktion:

$$\begin{aligned}
\mathbb{E}[X \cdot Y] &= \sum_{x \in W_X} \sum_{y \in W_Y} xy \cdot \Pr[X = x, Y = y] \\
&\overset{\text{Unabh.}}{=} \sum_{x \in W_X} \sum_{y \in W_Y} xy \cdot \Pr[X = x] \cdot \Pr[Y = y] \\
&= \sum_{x \in W_X} x \cdot \Pr[X = x] \sum_{y \in W_Y} y \cdot \Pr[Y = y] = \mathbb{E}[X] \cdot \mathbb{E}[Y].
\end{aligned}$$

□

Um einzusehen, dass für die Gültigkeit von Satz 2.61 die Unabhängigkeit der Zufallsvariablen wirklich notwendig ist, betrachte man beispielsweise den Fall $Y = X$ für eine Zufallsvariable X mit einer von Null verschiedenen Varianz. Dann gilt $\mathbb{E}[X \cdot Y] = \mathbb{E}[X^2] \neq (\mathbb{E}[X])^2 = \mathbb{E}[X] \cdot \mathbb{E}[Y]$.

Nachdem wir nun Aussagen zum Erwartungswert von Summen und Produkten gesehen haben, wollen wir untersuchen, ob ähnliche Aussagen

auch für die Varianz gelten. Der folgende Satz zeigt, dass bei unabhängigen Zufallsvariablen die Varianz der Summe auf die Varianzen der einzelnen Zufallsvariablen zurückgeführt werden kann.

Satz 2.62. Für unabhängige Zufallsvariablen $X_1, \ldots, X_n$ und $X := X_1 + \ldots + X_n$ gilt

$$\mathrm{Var}[X] = \mathrm{Var}[X_1] + \ldots + \mathrm{Var}[X_n].$$

Beweis. Wir beschränken uns auf den Fall $n = 2$ und betrachten die Zufallsvariablen X und Y. Der allgemeine Fall folgt dann wieder per Induktion. Es gilt

$$\begin{aligned}\mathbb{E}[(X+Y)^2] &= \mathbb{E}[X^2 + 2XY + Y^2] = \mathbb{E}[X^2] + 2\mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[Y^2] \\ \mathbb{E}[X+Y]^2 &= (\mathbb{E}[X] + \mathbb{E}[Y])^2 = \mathbb{E}[X]^2 + 2\mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[Y]^2,\end{aligned}$$

wobei die erste Zeile die Unabhängigkeit von X und Y benutzt. Wir ziehen die zweite Gleichung von der ersten ab und erhalten

$$\mathbb{E}[(X+Y)^2] - \mathbb{E}[X+Y]^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2.$$

Mit Hilfe von Satz 2.40 folgt die Behauptung. □

Für abhängige Zufallsvariablen $X_1, \ldots, X_n$ gilt Satz 2.62 im Allgemeinen nicht. Als Beispiel betrachte man den Fall einer Zufallsvariablen X mit von Null verschiedener Varianz. Für $Y = -X$ nimmt die Zufallsvariable $X + Y$ immer den Wert 0 an, also haben wir $\mathrm{Var}[X + Y] = 0 \neq 2 \cdot \mathrm{Var}[X] = \mathrm{Var}[X] + \mathrm{Var}[Y]$. Bei abhängigen Zufallsvariablen ist es daher meistens sehr schwer, Aussagen über die Varianz der Summe zu treffen.

Für die Varianz des Produktes von Zufallsvariablen gilt ein Analogon zu Satz 2.62 im Allgemeinen nicht einmal für unabhängige Zufallsvariablen, wie das folgende Beispiel zeigt.

Beispiel 2.63. Seinen X und Y zwei unabhängige Bernoulli-Variablen mit Parameter p. Dann ist XY eine Bernoulli-Variable mit Parameter p^2. Daher gilt $\mathrm{Var}[XY] = p^2(1-p^2)$, was für $0 < p < 1$ verschieden ist von $\mathrm{Var}[X] \cdot \mathrm{Var}[Y] = (p(1-p))^2$.

2.6.4 Waldsche Identität

Im vorherigen Abschnitt haben wir uns Summen und Produkte von Zufallsvariablen angesehen, bei denen die Anzahl der Summanden eine Konstante

ist. In vielen Anwendungen ist die Anzahl der Summanden allerdings oft ebenfalls eine Zufallsvariable. Man denke hier beispielsweise an einen Algorithmus, der ein bestimmtes Unterprogramm solange aufruft, bis eine Haltebedingung erfüllt ist. Die Laufzeit eines solchen Algorithmus können wir abschätzen, indem wir den Algorithmus in Phasen zerlegen: Jede Phase entspricht einem Aufruf des Unterprogramms. Die Anzahl Phasen ist dann in der Regel allerdings nicht konstant, sondern durch eine Zufallsvariable gegeben. Solche algorithmischen Beispiele verschieben wir auf später, hier begnügen wir uns zunächst mit einem künstlichen Beispiel.

Beispiel 2.64. Wir werfen eine Münze mit Wahrscheinlichkeit p für „Kopf“, bis wir zum ersten Mal Kopf sehen. Die Zufallsvariable N bezeichne die entsprechende Anzahl an Würfen. Anschliessend werfen wir die Münze nochmals N-mal und bezeichnen mit Z die Anzahl „Kopf“ während diesen zweiten N Versuchen. Was können wir über den Erwartungswert von Z sagen? – Aus Abschnitt und Beispiel 2.31 wissen wir bereits, dass N geometrisch verteilt ist mit Parameter p, und dass daher $\mathbb{E}[N] = 1/p$ ist. Wir erwarten daher, dass wir die Münze in der zweiten Phase $1/p$-mal werfen. Und da jeder dieser Würfe mit Wahrscheinlichkeit p „Kopf“ ist, können wir vermuten, dass $\mathbb{E}[Z] = (1/p) \cdot p = 1$ gelten sollte. Anstatt diese Vermutung durch Nachrechnen zu Beweisen, werden wir zunächst die sogenannte Waldsche Identität, benannt nach Abraham Wald (1902-1950), herleiten, die uns das vorhergesagte Ergebnis dann ohne weitere Rechnerei direkt liefert.

Satz 2.65 (Waldsche Identität). N und X seien zwei unabhängige Zufallsvariable, wobei für den Wertebereich von N gilt: $W_N \subseteq \mathbb{N}$. Weiter sei

$$Z := \sum_{i=1}^{N} X_i,$$

wobei $X_1, X_2, \ldots$ unabhängige Kopien von X seien. Dann gilt:

$$\mathbb{E}[Z] = \mathbb{E}[N] \cdot \mathbb{E}[X].$$

Beweis. Wir verwenden Satz 2.32. Dann gilt offenbar

$$\mathbb{E}[Z] = \sum_{n \in W_N} \mathbb{E}[Z \mid N = n] \cdot \Pr[N = n].$$

Um $\mathbb{E}[Z \mid N = n]$ zu berechnen, stellen wir fest, dass hier die Anzahl Iterationen nicht mehr eine Zufallsvariable, sondern eine Konstante ist, da wir auf $N = n$ bedingen. Mit der Linearität des Erwartungswertes folgt

$$\mathbb{E}[Z \mid N = n] = \mathbb{E}[X_1 + \ldots + X_n] = n \cdot \mathbb{E}[X],$$

und wir erhalten

$$\mathbb{E}[Z] = \sum_{n \in W_N} n \cdot \mathbb{E}[X] \cdot \Pr[N = n] = \mathbb{E}[X] \sum_{n \in W_N} n \cdot \Pr[N = n] = \mathbb{E}[X] \cdot \mathbb{E}[N].$$

□

2.7 Abschätzen von Wahrscheinlichkeiten

In Abschnitt 2.4.1 haben wir den Erwartungswert einer Zufallsvariablen eingeführt. Umgangssprachlich sprechen wir auch oft davon, dass wir ein bestimmtes Ergebnis eines Zufallsexperiments *erwarten*. Meinen beide Begriffe dasselbe? Dies ist leider nicht so[1]. Der Erwartungswert entspricht dem *Durchschnittswert* bei sehr vielen Wiederholungen des Experimentes. Wenn wir hingegen anschaulich von unser Erwartung sprechen, so beziehen wir uns in der Regel nur auf ein einziges Experiment. Betrachten wir ein Beispiel.

Beispiel 2.66. Wir betrachten eine Zufallsvariable X mit Dichte

$$f_X(x) = \begin{cases} 1 - \frac{1}{n} & \text{falls } x = n, \\ \frac{1}{n} & \text{falls } x = -n(n-1), \\ 0 & \text{sonst.} \end{cases}$$

Dann ist $\mathbb{E}[X] = n(1 - 1/n) - (n-1) = 0$. Dennoch nimmt aber X mit Wahrscheinlichkeit $1 - 1/n$ den Wert n an.

Wenn wir das Experiment aus Beispiel 2.66 daher nur ein einziges Mal ausführen, so 'erwarten' wir den Wert n, denn dieser tritt ja mit Wahrscheinlichkeit fast Eins ein. Wenn wir andererseits das Experiment N mal wiederholen, so wird der Durchschnitt der gesehenen Werte insbesondere für grosse Werte von N sehr nahe beim Erwartungswert sein, also nahe bei Null sein. Der Erwartungswert einer Zufallsvariablen kann also von dem erwarteten Ergebnis bei einer einzigen Wiederholung sehr stark abweichen.

Für manche Zufallsvariablen kann man jedoch zeigen, dass der Erwartungswert und der erwartete Ausgang eines einzigen Experimentes sehr nahe beieinander liegen. Formal lässt sich so eine Aussage wir folgt beschreiben: Nehmen wir an, für eine Zufallsvariable X, eine Konstante $t > 0$

[1] NB: wenn wir in Übungen von der 'erwarteten Anzahl' o.Ä. sprechen, meinen wir stets den Erwartungswert.

und eine (kleine) Konstante $\delta > 0$ gilt

$$\Pr[\,|X - \mathbb{E}[X]| \leq t] \geq 1 - \delta.$$

Dann 'erwarten' wir also, dass wir bei einer einmaligen Durchführung des Experimentes X einen Wert erhalten, der sich um höchstens t vom Erwartungswert unterscheidet. Konkret könnte so eine Aussage beispielsweise wie folgt aussehen. Sei X die Zufallsvariable, die die Anzahl Käufe bei einem Coupon-Collector-Problem mit n Bildern angibt. Dann wissen wir (siehe Seite 126), dass $\mathbb{E}[X] = n \ln n + O(n)$. In diesem Abschnitt werden wir zeigen, dass sogar gilt:

$$\Pr[\,|X - \mathbb{E}[X]| \leq n\sqrt{\ln n}] \geq 1 - \frac{\pi^2}{6 \ln n}.$$

Für eine hinreichend grosse Bilderanzahl n 'erwarten' wir daher, dass wir bei der nächsten Bilder-Aktion von Migros oder Coop höchstens $n \ln n + n\sqrt{\ln n}$ Bilder kaufen müssen, bis wir je ein Exemplar aller n Bilder besitzen.

2.7.1 Die Ungleichungen von Markov und Chebyshev

Bei der Konstruktion der Zufallsvariablen in Beispiel 2.66 haben wir den 'wahrscheinlichen' Wert n durch den negativen Wert $-n(n-1)$ ausgeglichen. Viele in der Informatik auftretende Zufallsvariablen haben jedoch die Eigenschaft, dass sie nur nicht-negative Werte annehmen. Dann ist eine Konstruktion wie die in Beispiel 2.66 nicht möglich. Und wir können sogar zeigen, dass dann sehr grosse Werte der Zufallsvariable wenig wahrscheinlich sind. Der folgende Satz, benannt nach dem russischen Mathematiker Andrei Andrejewitsch Markov (1856–1922), formalisiert dies.

Satz 2.67. *(Ungleichung von Markov)* Sei X eine Zufallsvariable, die nur nicht-negative Werte annimmt. Dann gilt für alle $t \in \mathbb{R}$ mit $t > 0$, dass

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}.$$

Oder äquivalent dazu $\Pr[X \geq t \cdot \mathbb{E}[X]] \leq 1/t$.

Beweis. Wir rechnen direkt nach, dass

$$\begin{aligned}\mathbb{E}[X] &= \sum_{x \in W_X} x \cdot \Pr[X = x] \geq \sum_{x \in W_X,\, x \geq t} x \cdot \Pr[X = x] \\ &\geq t \cdot \sum_{x \in W_X,\, x \geq t} \Pr[X = x] = t \cdot \Pr[X \geq t].\end{aligned}$$

Man erkennt, dass die Markov-Ungleichung im Wesentlichen durch Weglassen einiger Summanden aus der Definition des Erwartungswerts entsteht. □

Obwohl die Ungleichung von Markov sehr einfach zu beweisen ist, hat sie sich bei zahlreichen Anwendungen schon als sehr nützliches Hilfsmittel erwiesen. Für das Coupon-Collector-Problem erhalten wir zum Beispiel sofort die Aussage, dass wir nur mit Wahrscheinlichkeit $1/100$ viel mehr als $100n \log n$ Käufe tätigen müssen. Noch nicht unser Wunschergebnis, aber ein Anfang.

Erinnern wir uns an die Varianz. Sie misst die durchschnittliche Abweichung einer Zufallsvariablen von ihrem Erwartungswert. Definiert haben wir sie als $\mathrm{Var}[X] := \mathbb{E}[(X - \mathbb{E}[X])^2]$. Der Term $(X - \mathbb{E}[X])^2$ definiert eine nichtnegative Zufallsvariable. Und wir können daher auf diese Zufallsvariable die Markov-Ungleichung anwenden. Tun wir dies, so erhalten wir die sogenannten Chebyshev-Ungleichung, benannt nach PAFNUTI LWOWITSCH CHEBYSHEV[2] (1821–1894).

Satz 2.68. *(Ungleichung von Chebyshev)* Sei X eine Zufallsvariable und $t \in \mathbb{R}$ mit $t > 0$. Dann gilt

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{\mathrm{Var}[X]}{t^2}$$

oder äquivalent dazu $\Pr[|X - \mathbb{E}[X]| \geq t\sqrt{\mathrm{Var}[X]}] \leq 1/t^2$.

Beweis. Es gilt

$$\Pr[|X - \mathbb{E}[X]| \geq t] = \Pr[(X - \mathbb{E}[X])^2 \geq t^2].$$

[2] In der deutschsprachigen Literatur meist TSCHEBYSCHEFF; wir benutzen hier aber die im Englischen übliche Schreibweise.

Die Zufallsvariable $Y := (X - \mathbb{E}[X])^2$ ist nicht-negativ und hat nach Definition der Varianz den Erwartungswert $\mathbb{E}[Y] = \mathrm{Var}[X]$. Damit folgt die Behauptung durch Anwendung der Markov-Ungleichung:

$$\Pr[|X - \mathbb{E}[X]| \geq t] \;=\; \Pr[Y \geq t^2] \;\leq\; \frac{\mathbb{E}[Y]}{t^2} \;=\; \frac{\mathrm{Var}[X]}{t^2}.$$

□

Die Ungleichung von Chebyshev bestätigt die intuitive Bedeutung der Varianz: Je kleiner die Varianz ist, desto grösser ist die Wahrscheinlichkeit, mit der X nur Werte innerhalb eines Intervalls $[\mathbb{E}[X] - t, \mathbb{E}[X] + t]$ um den Erwartungswert $\mathbb{E}[X]$ annimmt. Die Varianz ist also in der Tat ein gutes Mass dafür, mit welcher Sicherheit wir davon ausgehen können, dass die Werte einer Zufallsvariablen „nahe“ beim Erwartungswert liegen.

Beispiel 2.69. Wir betrachten wieder das Coupon-Collector-Problem. Mit X bezeichnen wir die Anzahl Käufe, bis wir jedes Bild mindestens einmal besitzen. Auf Seite 126 haben wir den Erwartungswert von X ausgerechnet: $\mathbb{E}[X] = nH_n = n(1 + 1/2 + 1/3 + \cdots + 1/n)$. Um Chebyshev anzuwenden brauchen wir ausserdem die Varianz von X. Für deren Berechnung verwenden wiederum die Darstellung $X = \sum_{i=1}^{n} X_i$, wobei X_i die Anzahl Käufe ist während, wir genau $i-1$ verschiedene Bilder besitzen. Wegen der Unabhängigkeit der X_i wissen wir aus Satz 2.62, dass

$$\mathrm{Var}[X] = \mathrm{Var}\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n} \mathrm{Var}[X_i].$$

Im Abschnitt auf Seite 126 haben wir uns schon überzeugt, dass X_i geometrisch verteilt ist mit dem Parameter $p_i = 1 - \frac{i-1}{n}$. Daher gilt $\mathrm{Var}[X_i] = \frac{1-p_i}{p_i^2}$ (siehe Abschnitt 2.64), und somit

$$\mathrm{Var}[X] = \sum_{i=1}^{n} \frac{1-p_i}{p_i^2} \leq \sum_{i=1}^{n} \frac{1}{p_i^2} = \sum_{i=1}^{n} \left(\frac{n}{n-i+1}\right)^2 = n^2 \cdot \sum_{i=1}^{n} \frac{1}{i^2} \leq n^2 \cdot \frac{\pi^2}{6},$$

wobei wir die Summe $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$ verwendet haben. Mit Satz 2.68 gilt nun für eine beliebige Funktion $f(n)$

$$\Pr[|X - \mathbb{E}[X]| \geq f(n)] \leq \frac{\mathrm{Var}[X]}{(f(n))^2} \leq \frac{\pi^2 n^2}{6 \cdot (f(n))^2}.$$

Die rechte Seite geht für jede Funktion $f(n)$, die schneller als n wächst, gegen Null. Die zuvor genannte Ungleichung

$$\Pr[\,|X - \mathbb{E}[X]| \leq n\sqrt{\ln n}\,] \geq 1 - \frac{\pi^2}{6 \ln n}$$

folgt zum Beispiel für $f(n) = n\sqrt{\ln n}$.

2.7.2 Die Ungleichung von Chernoff

Wenn wir eine faire Münze n-mal werfen und mit X die Anzahl Kopf bezeichnen, so ist X binomialverteilt mit Parameter $1/2$. Insbesondere gilt daher $\mathbb{E}[X] = n/2$. Mit der Ungleichung von Markov folgt daraus $\Pr[X \geq n] \leq 1/2$. Ein etwas schärferes Resultat liefert die Chebyshev-Ungleichung: sie ergibt (Übung)

$$\Pr[X \geq n] \leq 1/n.$$

Wir wissen aber andererseits bereits, dass $\Pr[X \geq n] = 2^{-n}$ gilt. Die Markov- und Chebyshev-Ungleichungen liefern in diesem Beispiel also zwar korrekte, aber nicht sehr gute Abschätzungen. Dies liegt daran, dass diese Ungleichungen für beliebige nicht-negative Zufallsvariablen gelten – sie sind daher leicht anzuwenden, geben aber nicht immer sehr gute Schranken. Wenn wir die Verteilung der Zufallsvariable kennen, können wir oft bessere Schranken herleiten. Speziell für Summen von Bernoulli-Variabeln sind die sogenannten *Chernoff-Schranken* (benannt nach HERMAN CHERNOFF (*1923)) sehr nützlich.

Satz 2.70 (Chernoff-Schranken). Seien $X_1, \ldots, X_n$ unabhängige Bernoulliverteilte Zufallsvariablen mit $\Pr[X_i = 1] = p_i$ and $\Pr[X_i = 0] = 1 - p_i$. Dann gilt für $X := \sum_{i=1}^{n} X_i$:

(i) $\Pr[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{3}\delta^2 \mathbb{E}[X]}$ für alle $0 < \delta \leq 1$,

(ii) $\Pr[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\delta^2 \mathbb{E}[X]}$ für alle $0 < \delta \leq 1$,

(iii) $\Pr[X \geq t] \leq 2^{-t}$ für $t \geq 2e\mathbb{E}[X]$.

Beweis. Wir betrachten zunächst (iii), weil der Beweis hier etwas übersichtlicher ist. Die Idee ist, die Markov-Ungleichung auf die Zufallsvariable $Y := 4^X$ anzuwenden. Weil die Funktion 4^x streng monoton wachsend ist, ist das Ereignis „$X \geq t$“ identisch mit dem Ereignis „$4^X \geq 4^t$“. Insbesondere ist

$$\Pr[X \geq t] = \Pr[4^X \geq 4^t] \leq \frac{\mathbb{E}[4^X]}{4^t},$$

wobei wir im zweiten Schritt die Markov-Ungleichung auf $Y = 4^X$ angewandt haben. Wir müssen also nur noch $\mathbb{E}[4^X]$ ausrechnen. Dazu erinnern wir uns daran, dass die Zufallsvariablen X_i unabhängig sind, und daher die

Zufallsvariablen e^{X_i} nach Satz 2.55 ebenfalls unabhängig sind. Nach der Multiplikativität des Erwartungswertes für unabhängige Zufallsvariablen (Satz 2.61) gilt deshalb:

$$\mathbb{E}[4^X] = \mathbb{E}\left[4^{\sum_{i=1}^n X_i}\right] = \mathbb{E}\left[\prod_{i=1}^n 4^{X_i}\right] \overset{\text{Mult.}}{=} \prod_{i=1}^n \mathbb{E}[4^{X_i}].$$

Da jedes X_i eine Bernoulli-Variable mit Parameter p_i ist, nimmt der Ausdruck 4^{X_i} nur die Werte 4 oder 1 an, und zwar mit Wahrscheinlichkeit p_i bzw. $1-p_i$. Deshalb ist $\mathbb{E}[4^{X_i}] = 4p_i + (1-p_i) = 1+3p_i \leq e^{3p_i}$, wobei der letzte Schritt eine Anwendung der aus der Analysis bekannten Ungleichung $1+x \leq e^x$ ist. Damit ist also

$$\mathbb{E}[4^X] \leq \prod_{i=1}^n e^{3p_i} = e^{3\sum_{i=1}^n p_i} = e^{3\mathbb{E}[X]} \overset{t \geq 2e\mathbb{E}[X]}{\leq} e^{3t/(2e)}.$$

Mit einem Taschenrechner sehen wir schnell, dass $e^{3/(2e)} \approx 1.7364 \leq 2$ ist, und daher $e^{3t/(2e)} \leq 2^t$. Setzen wir alles zusammen, so haben wir

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[4^X]}{4^t} \leq \frac{2^t}{4^t} = 2^{-t},$$

wie behauptet. Die Ungleichungen (i) und (ii) beweist man ganz ähnlich, ausser dass wir statt $Y := 4^X$ die Zufallsvariable $Y := (1+\delta)^X$ bzw. $Y := (1-\delta)^X$ verwenden. Wir verzichten auf die Details. □

Beispiel 2.71. Wir betrachten wieder das Beispiel von vorher: Wir werfen n faire Münzen und bezeichnen mit X die Anzahl von „Kopf"-Ergebnissen. Dann gilt $\mathbb{E}[X] = n/2$. Wie gross ist nun die Wahrscheinlichkeit $\Pr[|X - n/2| > 0.1 \cdot n/2]$, dass die tatsächliche Anzahl um mindestens 10% vom Erwartungswert abweicht? Die folgende Tabelle vergleicht die Schranken, und die Chebyshev-Ungleichung und die Chernoff-Ungleichung liefern: Für

n	Chebyshev	Chernoff
1000	0.1	0.270961
2000	0.05	0.0424119
5000	0.02	0.000244096
10000	0.01	$5.77914 \cdot 10^{-8}$
100000	0.001	$4.14559 \cdot 10^{-73}$

die Chernoff-Ungleichung haben wir hier die Werte angegeben, die sich aus der Summe der Formeln in (i) und (ii) in Satz 2.70 ergeben.

2.8 Randomisierte Algorithmen

Ziel eines Algorithmus ist es, für eine Eingabe I eine Ausgabe $\mathcal{A}(I)$ zu berechnen. Bislang haben wir nur Algorithmen gesehen, die die Eigenschaft haben, dass sie für identische Eingaben auch identische Ergebnisse berechnen. Man nennt solche Algorithmen daher auch *deterministische* Algorithmen. In diesem Abschnitt wollen wir einen Schritt weiter gehen und sogenannte *randomisierte* Algorithmen betrachten. Die Idee eines randomisierten Algorithmus ist, dass der Algorithmus zusätzlich zur Eingabe I auch noch Zugriff auf Zufallsvariablen hat. Das Ergebnis des Algorithmus hängt daher nicht nur von der Eingabe I ab, sondern auch noch von den Werten der Zufallsvariablen. Wir bezeichnen die Ausgabe des Algorithmus daher mit $\mathcal{A}(I, \mathcal{R})$, wobei $\mathcal{R}$ für die Werte der Zufallsvariablen steht. Ausserdem schreiben wir $\mathcal{A}(I)$ für die Zufallsvariable, die dem Wert von $\mathcal{A}(I, \mathcal{R})$ bei einer zufälligen Wahl von $\mathcal{R}$ entspricht. Wichtig ist daher: bei randomisierten Algorithmen erhalten wir bei mehrmaligem Aufruf mit der gleichen Eingabe I im Allgemeinen verschiedene Ergebnisse.

Für eine genaue Definition eines randomisierten Algorithmus müssen wir noch präzisieren, auf welche Zufallsvariablen ein randomisierter Algorithmus Zugriff hat. In der Komplexitätstheorie nimmt man üblicherweise an, dass randomisierte Algorithmen lediglich Zugriff auf sogenannte *Zufallsbits* haben. Diese sind das Ergebnis von unabhängigen Bernoulliverteilten Zufallsvariablen mit Parameter $1/2$. Haben wir Zugriff auf solche Zufallsbits, so können wir daraus auch kompliziertere Zufallsvariablen konstruieren. Für die Binomialverteilung mit Parameter n und $1/2$ ist das sofort einsichtig. Für eine Gleichverteilung auf der Menge $\{1, 2, 3\}$ führt dies andererseits bereits zu einigen Schwierigkeiten: Wenn wir n Zufallsbits verwenden, so gibt es 2^n verschiedene 0-1-Folgen, die alle gleich wahrscheinlich sind. Jeder Folge müssen wir als Ausgabe einen der drei Werte $1, 2, 3$ zuordnen. Egal wir gross wir n wählen, so werden wir so nie eine *exakte* Gleichverteilung auf $\{1, 2, 3\}$ hinbekommen.

Um diese Probleme zu vermeiden, werden wir in dieser Vorlesung daher davon ausgehen, dass wir Zugriff auf alle Arten von Zufallsvariablen haben, wie sie von modernen Programmiersprachen bereitgestellt werden. Wie auch bei den Zufallsbits gehen wir davon aus, dass diese Aufrufe *unabhängige* Werte liefern. Realisiert wird dies in Programmiersprachen durch sogenannte Zufallszahlengeneratoren (siehe Seite 108).

Bei randomisierten Algorithmen unterscheidet man zwei verschiedene Ansätze: Algorithmen, die immer schnell sind, aber zuweilen ein falsches Ergebnis liefern (sogenannte Monte-Carlo-Algorithmen) und Algorithmen, die nie ein falsches Ergebnis liefern, aber bei denen die Laufzeit eine Zufallsvariable ist (sogenannte Las-Vegas-Algorithmen). Wir können Las-Vegas-Algorithmen auch anders betrachten: Brechen wir den Algorithmus ab, wenn eine bestimme Laufzeit überschritten ist, so haben wir einen Algorithmus, der immer schnell ist, der aber zuweilen statt einer sinnvollen Antwort nur '???' ausgibt (und umgekehrt können wir solch einen Algorithmus auch so lange wiederholen, bis wir einmal nicht '???' als Antwort kriegen – dann ist aber wieder die Laufzeit eine Zufallsvariable). Wir benutzen hier die Definition eines Las-Vegas-Algorithmus als ein Algorithmus, der manchmal '???' ausgeben kann.

Wichtig ist hier natürlich, dass Fehler oder verweigerte Antworten nur sehr selten auftreten. Ein Algorithmus, der, statt die Eingabe überhaupt nur einzulesen, sofort eine Antwort rät, ist ein Monte-Carlo-Algorithmus im obigen Sinne: die Laufzeit ist immer konstant und er macht Fehler. Aber so ein Algorithmus ist natürlich komplett nutzlos. Was wir gerne hätten, sind Algorithmen, die zumindest „fast immer" schnell sind und „fast nie" Fehler machen. Wie sich gleich zeigen wird, ist dies oft gar nicht so schwer, da sich die Fehlerwahrscheinlichkeit oft mit einigen einfachen Tricks drastisch reduzieren lässt.

2.8.1 Reduktion der Fehlerwahrscheinlichkeit

Ein Freund schlägt uns einen Münzentscheid vor für die Frage, wer die heutigen Kinokarten bezahlen soll. Auch eine Münze hat er gleich parat und sagt uns, dass wir „Zahl" nehmen sollen. Wohl wissend, dass er als Hobby-Zauberer allerlei ungewöhnliche Dinge besitzt, wollen wir zunächst sicherstellen, dass die Münze keine Spezialanfertigung ist mit „Kopf" auf beiden Seiten. Anfassen dürfen wir die Münze nicht und so bitten wir den Freund die Münze zumindest ein paar Mal zu werfen. Sobald wir „Zahl" sehen, sind wir beruhigt – und bereit für den Kinokarten-Münzentscheid. Wenn die Münze allerdings jedes Mal „Kopf" zeigt, so wird unser Misstrauen schnell so gross werden, dass wir auf den Münzentscheid verzichten. Die Betonung hier liegt auf *schnell*. Schon bei nur 10 Münzwürfen ist die Wahrscheinlichkeit bei einer fairen Münze für zehnmal Kopf kleiner als 1:1000,

bei 20-mal ist es schon weniger als $1{:}10^6$.

Die Reduktion der Fehlerwahrscheinlichkeit bei Algorithmen beruht auf genau dieser Idee: Führen wir einen randomisierten Algorithmus mehrfach aus (jedes Mal natürlich mit „neuen“ Zufallsbits), so reduziert sich die Fehlerwahrscheinlichkeit analog wie bei obigem Münzwurf. Ist die Fehlerwahrscheinlichkeit bei einem Durchlauf $1/2$, so ist die Wahrscheinlichkeit, dass 20 Durchläufe allesamt fehlerbehaftet sind kleiner als 10^{-6}.

Wir verwenden diesen Ansatz zunächst, um zu zeigen, dass es für Las-Vegas-Algorithmen genügt, dass der Algorithmus die richtige Antwort mit Wahrscheinlichkeit mindestens ε gibt, wobei $\varepsilon > 0$ beliebig klein sein darf. Durch eine genügend grosse Anzahl Wiederholungen kann man daraus dann jede beliebig kleine Fehlerwahrscheinlichkeit erzielen. Die Anzahl nötiger Wiederholungen hängt hierbei nur von der ursprünglichen Korrektheitswahrscheinlichkeit ε und der gewünschten Fehlerwahrscheinlichkeit δ ab.

Satz 2.72. Sei $\mathcal{A}$ ein randomisierter Algorithmus, der nie eine falsche Antwort gibt, aber zuweilen '???' ausgibt, wobei

$$\Pr[\mathcal{A}(I) \text{ korrekt}] \geq \varepsilon.$$

Dann gilt für alle $\delta > 0$: bezeichnet man mit $\mathcal{A}_\delta$ den Algorithmus, der $\mathcal{A}$ solange aufruft, bis entweder ein Wert verschieden von '???' ausgegeben wird (und $\mathcal{A}_\delta$ diesen Wert dann ebenfalls ausgibt) oder bis $N = \varepsilon^{-1} \ln \delta^{-1}$-mal '???' ausgegeben wurde (und $\mathcal{A}_\delta$ dann ebenfalls '???' ausgibt), so gilt für den Algorithmus $\mathcal{A}_\delta$, dass

$$\Pr[\mathcal{A}_\delta(I) \text{ korrekt}] \geq 1 - \delta.$$

Beweis. Die Wahrscheinlichkeit, dass $\mathcal{A}$ bei N Aufrufen immer den Wert '???' ausgibt, ist nach Annahme höchstens $(1-\varepsilon)^N$. Da $1 - x \leq e^{-x}$ für alle $x \in \mathbb{R}$, folgt daher für $N = \varepsilon^{-1} \ln \delta^{-1}$, dass die Wahrscheinlichkeit, dass $\mathcal{A}_\delta$ den Wert '???' ausgibt, höchstens $(1-\varepsilon)^N \leq e^{-\varepsilon N} = e^{\ln \delta} = \delta$ ist. □

Mit Hilfe von Satz 2.72 können wir uns jetzt auch leicht ganz formal davon überzeugen, wie man aus 0-1-Zufallsbits einen Zufallsgenerator für eine Gleichverteilung auf den Zahlen $\{0, \ldots, n\}$ erzeugt.

Beispiel 2.73. Wir betrachten der Einfachheit halber hier nur den Fall $n = 2$. Mit Hilfe von zwei Zufallsbits können wir einen exakt uniformen Sampler wie folgt konstruieren: Bei 00 gebe 0 aus, bei 01 gebe 1 aus, bei 10, gebe 2 aus, bei 11 gebe '???' aus. Dieser

Algorithmus gibt dann jede Zahl 0,1,2 mit der gleichen Wahrscheinlichkeit aus, und '???' mit Wahrscheinlichkeit 1/4. Die Wahrscheinlichkeit, dass wir nach 30 Aufrufen dieses einfachen Samplers noch immer keine Zahl ausgegeben haben, ist $4^{-30} \leq 10^{-18}$. Wenn wir in diesem Fall dann zum Beispiel Null ausgeben, haben wir zwar keine genaue Gleichverteilung mehr, aber dies spielt für die Praxis keine Rolle. Alternativ kann man den einfachen Sampler auch so lange aufrufen, bis er eine Antwort liefert. Dann kann die Laufzeit des Algorithmus zwar im Prinzip beliebig gross werden, aber da die Wahrscheinlichkeit hierfür so gering ist, werden in der Praxis exorbitant grosse Laufzeiten nicht auftreten.

Für Monte-Carlo-Algorithmen gilt ebenfalls, dass die Fehlerwahrscheinlichkeit sehr schnell abnimmt. Allerdings ist nicht immer offensichtlich, wie man das ausnutzen kann. Denn a priori ist nicht klar, wie man einer Antwort ansehen kann, ob sie korrekt ist oder nicht. In dem Spezialfall, dass ein Monte-Carlo-Algorithmus nur zwei Werte ausgibt *und* wir wissen, dass einer dieser Werte immer korrekt ist, können wir jedoch ein ähnliches Prinzip anwenden.

Satz 2.74. Sei $\mathcal{A}$ ein randomisierter Algorithmus, der immer eine der beiden Antworten JA oder NEIN ausgibt, wobei

$$\Pr[\mathcal{A}(I) = \text{JA}] = 1 \quad \text{falls I eine JA-Instanz ist,}$$

und

$$\Pr[\mathcal{A}(I) = \text{NEIN}] \geq \varepsilon \quad \text{falls I eine NEIN-Instanz ist.}$$

Dann gilt für alle $\delta > 0$: bezeichnet man mit $\mathcal{A}_\delta$ den Algorithmus, der $\mathcal{A}$ solange aufruft, bis entweder der Wert NEIN ausgegeben wird (und $\mathcal{A}_\delta$ dann ebenfalls NEIN ausgibt) oder bis $N = \varepsilon^{-1} \ln \delta^{-1}$-mal JA ausgegeben wurde (und $\mathcal{A}_\delta$ dann ebenfalls JA ausgibt), so gilt für alle Instanzen I

$$\Pr[\mathcal{A}_\delta(I) \text{ korrekt}] \geq 1 - \delta.$$

Beweis. Falls I eine JA-Instanz ist, wird $\mathcal{A}(I) = \text{JA}$ mit Wahrscheinlichkeit 1 gelten, d.h. insbesondere $\Pr[A_\delta(I) \text{ korrekt}] = 1$. Ist I eine NEIN-Instanz, dann gibt jeder Aufruf von $\mathcal{A}$ mit Wahrscheinlichkeit mindestens ε die Antwort NEIN. Die Wahrscheinlichkeit, dass unter $N = \varepsilon^{-1} \ln \delta^{-1}$ unabhängigen Aufrufen kein einziges NEIN ist, ist höchstens

$$(1 - \varepsilon)^N \leq e^{-\varepsilon N} = e^{-\ln \delta^{-1}} = \delta,$$

das heisst $\Pr[A_\delta(I) \text{ korrekt}] \geq 1 - \delta$. □

Monte-Carlo-Algorithmen, die die Bedingung aus Satz 2.74 erfüllen, nennt man Algorithmen mit *einseitigem* Fehler, denn sie machen ja nur bei einer der beiden Antworten einen Fehler. Bei Monte-Carlo-Algorithmen mit *zweiseitigem Fehler*, können wir den Fehler ebenfalls reduzieren, aber nur wenn die Fehlerwahrscheinlichkeit des Algorithmus von Anfang an *strikt kleiner* als $1/2$ ist.

Satz 2.75. Sei $\varepsilon > 0$ und $\mathcal{A}$ ein randomisierter Algorithmus, der immer eine der beiden Antworten JA oder NEIN ausgibt, wobei

$$\Pr[\mathcal{A}(I) \text{ korrekt}] \geq 1/2 + \varepsilon.$$

Dann gilt für alle $\delta > 0$: bezeichnet man mit $\mathcal{A}_\delta$ den Algorithmus, der $N = 4\varepsilon^{-2} \ln \delta^{-1}$ unabhängige Aufrufe von $\mathcal{A}$ macht und dann die Mehrheit der erhaltenen Antworten ausgibt, so gilt für den Algorithmus $\mathcal{A}_\delta$, dass

$$\Pr[\mathcal{A}_\delta(I) \text{ korrekt}] \geq 1 - \delta.$$

Beweis. Wir können annehmen, dass $\varepsilon \leq 1/2$ ist. Sei X die Anzahl korrekter Antworten unter N unabhängigen Aufrufen von $\mathcal{A}(I)$. Setzen wir $p := \Pr[\mathcal{A}(I) \text{ ist korrekt}]$, so ist X binomialverteilt mit Parametern N und p. Offenbar ist

$$\Pr[\mathcal{A}_\delta(I) \text{ korrekt}] \geq \Pr[X > N/2] = 1 - \Pr[X \leq N/2].$$

Zu beachten ist, dass $N/2$ viel kleiner ist als der Erwartungswert von X, nämlich ist $\mathbb{E}[X] = pN \geq N/2 + \varepsilon N$, und somit

$$N/2 \leq (1 - \varepsilon)(N/2 + \varepsilon N) \leq (1 - \varepsilon)\mathbb{E}[X]$$

(die erste Ungleichung gilt für alle $0 \leq \varepsilon \leq 1/2$). Daher können wir die Chernoff-Schranken anwenden. Aus $\mathbb{E}[X] \geq N/2 = 2\varepsilon^{-2} \ln \delta^{-1}$ und Satz 2.70 (ii) folgt

$$\Pr[X \leq N/2] \leq \Pr[X \leq (1 - \varepsilon)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\varepsilon^2 \mathbb{E}[X]} \leq \delta,$$

und somit $\Pr[\mathcal{A}_\delta(I) \text{ korrekt}] \geq 1 - \delta$. □

Bei randomisierten Algorithmen für Optimierungsprobleme (wie beispielsweise beim Berechnen einer möglichst grossen stabilen Menge wie in

Beispiel 2.37) sprechen wir üblicherweise nicht von Fehlerwahrscheinlichkeiten, sondern davon, ob sie das gewünschte Ergebnis erzielen, also beispielsweise einen Wert ausgeben der mindestens so gross ist wie der Erwartungswert. Auch in diesem Fall funktioniert das Prinzip der unabhängigen Wiederholungen analog:

Satz 2.76. Sei $\varepsilon > 0$ und $\mathcal{A}$ ein randomisierter Algorithmus für ein Maximierungsproblem, wobei gelte:

$$\Pr[\mathcal{A}(I) \geq f(I)] \geq \varepsilon.$$

Dann gilt für alle $\delta > 0$: bezeichnet man mit $\mathcal{A}_\delta$ den Algorithmus, der $N = \varepsilon^{-1} \ln \delta^{-1}$ unabhängige Aufrufe von $\mathcal{A}$ macht und die *beste* der erhaltenen Antworten ausgibt, so gilt für den Algorithmus $\mathcal{A}_\delta$, dass

$$\Pr[\mathcal{A}_\delta(I) \geq f(I)] \geq 1 - \delta.$$

(Für Minimierungsprobleme gilt eine analoge Aussage wenn wir „$\geq f(I)$“ durch „$\leq f(I)$“ ersetzen.)

Beweis. Die Wahrscheinlichkeit, dass bei N unabhängigen Aufrufen von $\mathcal{A}(I)$ kein einziges Mal ein Wert von mindestens $f(I)$ erzielt wird, ist höchstens

$$(1 - \varepsilon)^N \leq \exp(-\varepsilon N) = \exp(-\ln \delta^{-1}) = \delta.$$

Die Aussage für Minimierungsprobleme folgt analog. □

In den folgenden Abschnitten geben wir einige Anwendungsbeispiele für randomisierte Algorithmen.

2.8.2 Sortieren und Selektieren

Ein klassisches Beispiel für einen Las-Vegas-Algorithmus ist der QUICKSORT Algorithmus, bei dem wir in jedem Schritt das Pivot-Element zufällig wählen. Dieser Algorithmus sortiert immer richtig, aber die konkrete Laufzeit hängt von der (zufälligen) Wahl der Pivot-Elemente ab. Wenn wir hierbei viel Pech haben, kann die Laufzeit quadratisch in der Anzahl zu sortierender Elemente sein. Wir wollen in diesem Abschnitt zeigen, dass die Laufzeit in der Regel aber durch $O(n \ln n)$ beschränkt sein wird.

Wir wiederholen hier kurz den QUICKSORT-Algorithmus zur Sortierung einer Folge $(A[1], \ldots, A[n])$. Mit PARTITION(A, ℓ, r, p) bezeichnen wir eine Prozedur, die für ein $\ell \leq p \leq r$ die Elemente $A[\ell], \ldots, A[r]$ mit Hilfe von $r - \ell$ vielen Vergleichen so umsortiert, dass auf der linken Seite von $A[p]$ die Elemente kleiner als $A[p]$ stehen und auf der rechten Seite die Elemente grösser als $A[p]$. Das Element $A[p]$ selbst wird dazwischen einsortiert und der Rückgabewert t gibt die entsprechende Position an. Zur Einfachheit nehmen wir hier an, dass die Elemente von A paarweise verschieden sind, sodass dieses t eindeutig definiert ist.

QUICKSORT(A, ℓ, r)

1: **if** $\ell < r$ **then**
2: $\quad p \leftarrow \text{Uniform}(\{\ell, \ell+1, \ldots, r\})$ ▷ wähle Pivotelement zufällig
3: $\quad t \leftarrow$ PARTITION(A, ℓ, r, p)
4: $\quad$ QUICKSORT$(A, \ell, t-1)$
5: $\quad$ QUICKSORT$(A, t+1, r)$

Die Sortierung von $(A[1], \ldots, A[n])$ erfolgt durch den Aufruf QUICKSORT$(A, 1, n)$. Für eine genauere Beschreibung des Algorithmus verweisen wir auf den ersten Teil der Vorlesung.

Wir bezeichnen nun mit $T_{\ell,r}$ die (zufällige) Anzahl an Vergleichen, die bei einem Aufruf QUICKSORT(A, ℓ, r) ausgeführt werden. Wir wollen zeigen, dass gilt:

$$\mathbb{E}[T_{1,n}] \leq 2(n+1)\ln n + O(n). \tag{2.6}$$

Für $\ell \geq r$ ist $T_{\ell,r} = 0$. Für $\ell < r$ gilt mit Satz 2.32 und der Linearität des Erwartungswerts

$$\begin{aligned}\mathbb{E}(T_{\ell,r}) &= \sum_{i=\ell}^{r} \Pr[t = i] \cdot (r - \ell + \mathbb{E}[T_{\ell,i-1}] + \mathbb{E}[T_{i+1,r}]) \\ &= \frac{1}{r-\ell+1} \cdot \sum_{i=0}^{r-\ell} (r - \ell + \mathbb{E}[T_{\ell,\ell+i-1}] + \mathbb{E}[T_{\ell+i+1,r}]).\end{aligned}$$

Hierbei haben wir benutzt, dass t gleichverteilt auf $\{\ell, \ldots, r\}$ ist – dies folgt direkt aus der Annahme, dass die Elemente von A verschieden sind.

Betrachten wir die eben gezeigte Rekursion nochmals etwas genauer, so sehen wir, dass der Erwartungswert $\mathbb{E}[T_{\ell,r}]$ gar nicht von r und ℓ abhängt,

sondern nur von der Anzahl $r - \ell + 1$ der zu sortierenden Elemente. Diese Beobachtung motiviert die folgende rekursive Definition von *Zahlen* t_n:

$$t_n = \begin{cases} 0, & \text{falls } n \leq 1, \text{und} \\ \frac{1}{n} \sum_{i=0}^{n-1} (n - 1 + t_i + t_{n-i-1}), & \text{falls } n \geq 2. \end{cases}$$

Durch Induktion über $r - \ell$ sieht man dann leicht, dass für alle ℓ, r die Gleichung $\mathbb{E}[T_{\ell,r}] = t_{r-\ell+1}$ gilt. Im Folgenden leiten wir jetzt noch eine obere Schranke für t_n her. Für alle $n \geq 3$ gilt nach Definition:

$$n \cdot t_n = \sum_{i=0}^{n-1} (n - 1 + t_i + t_{n-i-1})$$

und ebenso

$$(n-1) \cdot t_{n-1} = \sum_{i=0}^{n-2} (n - 2 + t_i + t_{n-i-2}).$$

Ziehen wir nun die zweite diese Gleichungen von der ersten ab, so erhalten wir:

$$nt_n - (n-1) \cdot t_{n-1} = 2(n-1) + 2t_{n-1}$$

und daher

$$t_n = \frac{n+1}{n} \cdot t_{n-1} + \frac{2(n-1)}{n} \leq \frac{n+1}{n} \cdot t_{n-1} + 2.$$

Nun verwenden wir nochmals Induktion (diesmal über n) um zu zeigen, dass für alle $n \geq 2$ gilt:

$$t_n \leq 2 \sum_{i=3}^{n+1} \frac{n+1}{i}.$$

Für $n = 2$ folgt aus der Definition von t_n, dass $t_2 = 1 \leq 2 = \sum_{i=3}^{3} 1$. Für $n \geq 3$ können wir die oben hergeleitete Ungleichung $t_n \leq \frac{n+1}{n} \cdot t_{n-1} + 2$ zusammen mit der Induktionsannahme verwenden, um die Gültigkeit der Ungleichung für alle $n \geq 3$ nachzurechnen. Wegen $\sum_{i=1}^{n} \frac{1}{i} = H_n = \ln n + O(1)$ gilt somit insbesondere

$$\mathbb{E}[T_{1,n}] = t_n \leq 2(n+1) \ln n + O(n),$$

wie in (2.6) behauptet.

Zum Abschluss dieses Abschnitts betrachten wir nun noch das sogenannte Selektionsproblem. Hierbei geht es darum, in einer Folge $(A[1], \ldots,$

$A[n]$) von paarweise verschiedenen Zahlen den k-te kleinsten Wert zu ermitteln. Eine Möglichkeit dies zu erreichen, wäre die Folge zunächst zu sortieren und dann das k-te Element der sortierten Folge auszugeben. Da wir in Zeit $O(n \log n)$ sortieren können, können wir das Selektionsproblem also sicherlich ebenfalls in Zeit $O(n \log n)$ lösen. Der QUICKSELECT-Algorithmus, den wir nun vorstellen werden, ist aber noch etwas schneller: mit ihm können wir das Problem in erwartet $O(n)$ Schritten lösen.

```
QUICKSELECT(A, ℓ, r, k)
1: p ← Uniform({ℓ, ℓ + 1, ..., r})          ▷ wähle Pivotelement zufällig
2: t ← PARTITION(A, ℓ, r, p)
3: if t = ℓ + k − 1 then
4:     return A[t]                           ▷ gesuchtes Element ist gefunden
5: else if t > ℓ + k − 1 then
6:     return QUICKSELECT(A, ℓ, t − 1, k)    ▷ gesuchtes Element ist links
7: else
8:     return QUICKSELECT(A, t + 1, r, k − t)  ▷ gesuchtes Element ist rechts
```

Die Korrektheit des Algorithmus ist leicht einzusehen (Übung!). Wie aber steht es um die erwartete Laufzeit? Wenn wir QUICKSELECT$(A, 1, n, k)$ ausführen, dann führt das zu einer zufälligen Zahl N an Aufrufen der Form PARTITION(A, ℓ_i, r_i, p) für eine (ebenfalls zufällige) Folge

$$(\ell_0, r_0, k_0), (\ell_2, r_2, k_2), \ldots, (\ell_N, r_N, k_N),$$

mit $(\ell_0, r_0, k_0) = (1, n, k)$. Hierbei ist immer entweder $(\ell_{i+1}, r_{i+1}) = (\ell_i, t-1)$ oder $(\ell_{i+1}, r_{i+1}) = (t + 1, r_i)$, wobei t eine auf $\{\ell_i, \ldots, r_i\}$ gleichverteilte Variable ist (weil die Element von A paarweise verschieden sind). Die Laufzeit bei einem Aufruf von QUICKSELECT$(A, 1, n, k)$, gemessen an der Zahl an Vergleichen von Elementen in A, ist nun

$$T = \sum_{i=1}^{N} (r_i - \ell_i),$$

weil jeder Aufruf PARTITION(A, ℓ_i, r_i, p) genau $r_i - \ell_i$ Vergleiche braucht. Wir definieren nun N_j als die Anzahl der Aufrufe von QUICKSELECT für die $(3/4)^j n < r_i - \ell_i + 1 \leq (3/4)^{j-1} n$ gilt. Dann gilt sicherlich

$$T \leq \sum_{j=1}^{\infty} N_j \cdot (3/4)^{j-1} n$$

und wegen der Linearität des Erwartungswertes somit

$$\mathbb{E}[T] \leq n \cdot \sum_{j=1}^{\infty} \mathbb{E}[N_j] \cdot (3/4)^{j-1}.$$

Was können wir nun über $\mathbb{E}[N_j]$ sagen? Dazu überlegen wir uns, dass die Anzahl der zu betrachtenden Elemente sicherlich immer dann von $r_i - \ell_i + 1$ auf höchstens $\frac{3}{4}(r_i - \ell_i + 1)$ reduziert wird, wenn wir als Pivot-Element eines der mittleren $\frac{1}{2}(r_i - \ell_i + 1)$ Elemente wählen. Mit anderen Worten: bei jeder Wahl eines Pivot-Elementes haben wir eine Wahrscheinlichkeit von mindestens $1/2$, dass die Anzahl Elemente um mindestens einen Faktor $3/4$ reduziert wird. Also gilt $\mathbb{E}[N_j] \leq 2$ für alle $j \geq 1$ und somit

$$\mathbb{E}[T] \leq 2n \sum_{j=1}^{\infty} (3/4)^{j-1} = 8n.$$

Die erwartete Laufzeit von QUICKSELECT$(A, 1, n, k)$ ist also in der Tat linear, wie oben behauptet.

2.8.3 Primzahltest

In der Kryptographie benötigt man oft sehr grosse Primzahlen. So erfordert zum Beispiel die Generierung eines RSA-Schlüsselpaares die Wahl zweier Primzahlen mit mehreren Tausend Bits. Solche Primzahlen generiert man üblicherweise, indem man zufällig eine Zahl der gegebenen Länge auswählt, und dann überprüft, ob diese tatsächlich prim ist – und die Prozedur, wenn nötig, so lange wiederholt, bis eine Primzahl gefunden ist.

Wir wenden uns nun also dem Problem zu, für eine gegebene Zahl n zu überprüfen, ob sie prim ist. Eine naheliegende Methode, dies zu überprüfen, ist alle möglichen Teiler bis $\sqrt{n}$ durchzuprobieren. Dieser Algorithmus ist bei Zahlen mit mehreren Tausend Bits allerdings extrem ineffizient. Wir wollen stattdessen einen Algorithmus, dessen Laufzeit polynomiell in der Länge der Eingabe ist, also polynomiell in $\ln n$.

Es gibt mittlerweile deterministische Algorithmen, die in polynomieller Zeit überprüfen, ob die Zahl n prim ist, so zum Beispiel den AKS-PRIMZAHLTEST. In der Praxis verwendet man aber ausschliesslich randomisierte Algorithmen, und zwar aus zwei Gründen: Einerseits sind diese Algorithmen immer noch weitaus effizienter, andererseits sind sie viel einfacher zu implementieren.

Im Folgenden wollen wir die wichtigsten Ideen hinter solch randomisierten Primzahltests vorstellen. Sei n die Zahl, von der wir wissen wollen, ob sie prim ist. Eine einfache Idee für einen randomisierten Primzahltest wäre, eine zufällige Zahl $a \in \{2, \ldots, \sqrt{n}\}$ zu wählen, und zu schauen, ob a ein Teiler von n ist (oder, was ein bisschen cleverer ist, zu schauen, ob der ggT von a und n grösser als 1 ist, was mit dem Euklidschen Algorithmus ebenfalls leicht machbar ist). Ist dies der Fall, so ist n sicher nicht prim – wir können die Zahl a also als *Zertifikat* dafür ansehen, dass n zusammengesetzt ist. Leider ist die Erfolgswahrscheinlichkeit für diesen Algorithmus sehr klein, wenn n eine zusammengesetzte Zahl mit nur wenigen Primfaktoren ist. Ist zum Beispiel $n = pq$ für zwei ähnlich grosse Primzahlen p und q, so erkennt der Algorithmus dies nur mit Wahrscheinlichkeit $O(1/\sqrt{n})$, und um die Fehlerwahrscheinlichkeit genügend zu reduzieren, wären etwa $\sqrt{n}$ Wiederholungen nötig – das ist nicht besser als die deterministische Brute-Force-Lösung.

Jeder randomisierte Primzahltest muss dieses Problem irgendwie umgehen. Fast alle benutzen dafür folgendes Resultat der elementaren Zahlentheorie (bekannt aus der Vorlesung Diskrete Mathematik).

Satz 2.77 (Kleiner fermatscher Satz). Ist $n \in \mathbb{N}$ prim, so gilt für alle Zahlen $0 < a < n$

$$a^{n-1} \equiv 1 \mod n.$$

Falls wir für ein gegebenes n also eine Zahl $1 \leq a \leq n$ mit der Eigenschaft $a^{n-1} \not\equiv 1 \pmod{n}$ finden, so ist n sicher nicht prim – ein solches a ist also ebenfalls ein Zertifikat dafür, dass n zusammengesetzt ist. Hierbei ist es noch wichtig, dass wir $a^{n-1} \bmod n$ auch effizient berechnen können. Dies können wir mit der Methode der binären Exponentiation tun. Genauer: hat die Zahl k Bits, so kann die Berechnung von $a^{n-1} \bmod n$ in Zeit $O(k^3)$ ausgeführt werden. Wie dies geht haben Sie in der Vorlesung Diskrete Mathematik im letzten Semester gesehen.

Nun gibt es zusammengesetzte Zahlen n, die sogenannten *Carmichael-Zahlen*, die die Eigenschaft haben, dass $a^{n-1} \equiv 1 \pmod{n}$ für alle mit n teilerfremde Zahlen a gilt (die kleinste Carmichael-Zahl ist $561 = 3 \cdot 11 \cdot 17$). Für solche Zahlen n liefert Satz 2.77 also keine Zertifikate, die wir nicht auch durch Ausrechnen des ggT von a und n finden würden. Solche Carmichael-Zahlen sind eher selten, aber es gibt davon unendlich viele, und um einen

effizienten randomisierten Primzahltest zu erhalten, müssen wir unsere Idee eines Zertifikats daher noch ein bisschen verbessern.

Die Idee, die wir dazu benutzen, ist folgende: Wenn n eine Primzahl ist, dann bilden die Zahlen $0 \leq a < n$ bezüglich der Addition und Multiplikation Modulo n bekanntlich einen Körper (siehe die Vorlesung Diskrete Mathematik). Das heisst insbesondere, dass die Kongruenz $x^2 \equiv 1 \pmod{n}$ für $0 \leq x < n$ genau die zwei Lösungen $x = 1$ und $x = n - 1$ hat.

Diese Idee können wir mit der vorherigen Idee kombinieren. Dazu schreiben wir zuerst $n - 1 = d2^k$, wobei d ungerade ist. Ist n prim, dann muss nach Satz 2.77 für jedes beliebige $a \in \{1, \ldots, n-1\}$ gelten: $a^{n-1} = (a^d)^{2^k} \equiv 1 \pmod{n}$. Dann ist aber wegen der vorhergehenden Beobachtung entweder $(a^d)^{2^{k-1}} \equiv 1 \pmod{n}$ oder $(a^d)^{2^{k-1}} \equiv n-1 \pmod{n}$. Durch Iterieren sieht man leicht: entweder gilt für alle $0 \leq i \leq k$ die Kongruenz $(a^d)^{2^i} \equiv 1 \pmod{n}$, oder aber es gibt ein $0 \leq i \leq k-1$ mit $(a^d)^{2^i} \equiv n - 1 \pmod{n}$. Sind beide Bedingungen verletzt, so sagen wir, dass a ein Zertifikat für die Zusammengesetztheit von n ist.

Der folgende Algorithmus folgt genau dieser Idee: er wählt eine zufällige Zahl $a \in \{2, 3, \ldots, n-1\}$ und schaut, ob a ein Zertifikat für die Zusammengesetztheit von n ist.

MILLER-RABIN-PRIMZAHLTEST(n)

if $n = 2$ **then**
    return 'Primzahl'
else if n gerade oder $n = 1$ **then**
    return 'keine Primzahl'
Wähle $a \in \{2, 3, \ldots, n-1\}$ zufällig und
berechne $k, d \in \mathbb{Z}$ mit $n - 1 = d2^k$ und d ungerade.
$x \leftarrow a^d \pmod{n}$
if $x = 1$ or $x = n - 1$ **then**
    return 'Primzahl'
repeat $k - 1$ mal
    $x \leftarrow x^2 \pmod{n}$
    if $x = 1$ **then**
        return 'keine Primzahl'
    if $x = n - 1$ **then**
        return 'Primzahl'
return 'keine Primzahl'

Die Laufzeit dieses Algorithmus ist in $O(\ln n)$. Ist n prim, so gibt der Algorithmus immer die Antwort 'Primzahl'. Falls n zusammengesetzt ist, so gibt der Algorithmus die Antwort 'keine Primzahl' mit Wahrscheinlichkeit mindestens $3/4$.[3] Wir können die Fehlerwahrscheinlichkeit mit Satz 2.74 beliebig klein machen.

2.8.4 Target-Shooting

In diesem Abschnitt betrachten wir folgendes Problem: Gegeben eine Menge U und eine Untermenge $S \subseteq U$ unbekannter Grösse, wie gross ist der Quotient $|S|/|U|$? Wir nehmen an, dass wir für S die Indikatorfunktion $I_S \colon U \to \{0,1\}$ haben, sodass $I_S(u) = 1$ genau dann gilt, wenn $u \in S$.

Wir betrachten nun folgenden Algorithmus zur Schätzung von $|S|/|U|$: Für eine geeignete Wahl von $N > 0$, wähle N Elemente aus U zufällig (d.h. unabhängig und gleichverteilt), und gebe das Verhältnis der gefundenen Elemente in S zu N aus. Man bezeichnet dieses Prinzip auch als TARGET-SHOOTING.

TARGET-SHOOTING

1: Wähle $u_1, \ldots, u_N \in U$ zufällig, gleichverteilt und unabhängig
2: **return** $N^{-1} \cdot \sum_{i=1}^{N} I_S(u_i)$

Dieser Algorithmus beruht auf zwei Annahmen: Erstens muss die Indikatorfunktion I_S effizient berechenbar sein, das heisst, für ein Element $u \in U$ müssen wir schnell entscheiden können, ob $u \in S$ ist oder nicht. Zweitens brauchen wir eine effiziente Prozedur, die uns ein uniform zufälliges Element aus U gibt.

Beispiel 2.78. Sei $U = [-1,1]^2$ und sei $S = \{(x,y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$. Dann ist bekanntlich

$$\pi = 4|S|/|U|.$$

Es ist möglich, effizient ein zufälliges Paar $(x,y) \in [-1,1]^2$ zu wählen und darauf zu überprüfen, ob es in S liegt (wobei wir vernachlässigen, dass wir uns wegen der endlichen Präzision auf Fliesskommazahlen beschränken müssen). Wir können also versuchen, die Zahl π mittels Target-Shooting zu approximieren.

[3] Rabin, Michael O., Probabilistic algorithm for testing primality, Journal of Number Theory, 12 (1) (1980) 128–138.

Wir wollen den Target-Shooting-Algorithmus genauer analysieren. Sei dazu für $i \in [N]$

$$Y_i := I_S(u_i).$$

Nun sind wegen der unabhängigen und uniformen Wahl der u_i die Variablen $Y_1, \ldots, Y_N$ unabhängige Bernoulli-Variablen mit $\Pr[Y_i = 1] = |S|/|U|$ für jedes $i = 1, \ldots, N$. Für die Zufallsvariable

$$Y := \frac{1}{N} \sum_{i=1}^{N} Y_i = \frac{1}{N} \sum_{i=1}^{N} I_S(u_i)$$

erhalten wir demnach $\mathbb{E}[Y] = |S|/|U|$, unabhängig von der Wahl von N. Die Varianz

$$\mathrm{Var}[Y] = \frac{1}{N} \left(\frac{|S|}{|U|} - \left(\frac{|S|}{|U|} \right)^2 \right)$$

ist ihrerseits durchaus abhängig von N: Sie ist streng monoton fallend in N. Wir erwarten also für grosse N, dass die Ausgabe des Algorithmus mit grosser Wahrscheinlichkeit sehr nah an $|S|/|U|$ ist. Im Folgenden wollen wir dies genauer untersuchen.

Sei $\varepsilon > 0$ beliebig klein. Wie gross muss N sein, damit der Algorithmus mit Wahrscheinlichkeit mindestens $1/2$ eine Antwort im Intervall $[(1-\varepsilon)|S|/|U|, (1+\varepsilon)|S|/|U|]$ ausgibt? Sicherlich muss N von ε abhängen. Andererseits hängt N auch von $|S|$ und $|U|$ ab. Wir betrachten dazu zwei extreme Fälle. Ist U eine sehr grosse endliche Menge und $|S| = 1$, dann suchen wir sozusagen die Nadel im Heuhaufen: Die Abweichung $|Y - \mathbb{E}[Y]|$ kann nur kleiner als $\varepsilon\mathbb{E}[Y]$ sein, wenn $Y > 0$ ist, d.h. wenn wir mindestens einmal das Element in S auswählen. Damit dies mit Wahrscheinlichkeit mindestens $1/2$ geschieht, müssen wir augenscheinlich etwa $N = \Omega(|U|)$ Versuche machen. Ein anderer extremer Fall tritt ein, wenn $|S| = |U| - 1$ ist. In diesem Fall besteht unser Heuhaufen fast nur aus Nadeln, und es gilt $\Pr[|Y - \mathbb{E}[Y]| \leq \varepsilon\mathbb{E}[Y]] \geq 1/2$ schon nach einer konstanten Anzahl an Versuchen. Es scheint also, dass N nur von dem Quotienten $|S|/|U|$ abhängt, und zwar müssen wir N grösser wählen, je kleiner dieser Quotient ist. Dies zeigen wir in folgendem Satz:

Satz 2.79. Seien $\delta, \varepsilon > 0$. Falls $N \geq 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \ln(2/\delta)$, so ist die Ausgabe des Algorithmus TARGET-SHOOTING mit Wahrscheinlichkeit

mindestens $1-\delta$ im Intervall $\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$.

Beweis. Wegen $\mathbb{E}[Y] = |S|/|U|$ reicht es zu zeigen, dass

$$\Pr\Big[|Y - \mathbb{E}[Y]| \geq \varepsilon \cdot \mathbb{E}[Y]\Big] \leq \delta$$

gilt. Schreiben wir $Z := \sum_{i=1}^{N} Y_i = NY$, ist dies äquivalent zu

$$\Pr\Big[|Z - \mathbb{E}[Z]| \geq \varepsilon \cdot \mathbb{E}[Z]\Big] \leq \delta.$$

Da die Y_i unabhängige Bernoulli-Variablen sind, können wir die Chernoff-Schranken benutzen. Mit Satz 2.70 (i) und (ii) folgt

$$\Pr\Big[|Z - \mathbb{E}[Z]| \geq \varepsilon \cdot \mathbb{E}[Z]\Big] \leq 2e^{-\varepsilon^2 \mathbb{E}[Z]/3} = 2e^{-\varepsilon^2 N|S|/(3|U|)}.$$

Nun ist wegen der Wahl $N = 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \ln(2/\delta)$ diese Wahrscheinlichkeit höchstens δ. □

Beispiel 2.78 *(Fortsetzung)* Wir wollen π mit 99%iger Wahrscheinlichkeit bis auf die 10te Nachkommastelle genau bestimmen. Wir wählen dazu also $\delta = 0.01$ und $\varepsilon = 10^{-10}$ und sehen, dass wir nach Satz 2.79 mindestens $N = 3\pi^{-1} \cdot 10^{20} \cdot \ln 200$ viele Versuche machen sollen! Es gibt durchaus bessere Methoden, π zu approximieren.

2.8.5 Finden von Duplikaten

In diesem Abschnitt betrachten wir verschiedene Lösungsverfahren für folgendes Problem: Gegeben ein Datensatz $\mathcal{S} := \{s_1, \ldots s_n\}$, finde Duplikate, d.h. finde ein bzw. alle Paare $1 \leq i < j \leq n$ mit $s_i = s_j$. Wenn die Datensätze zum Beispiel in einem Array gespeichert sind und wir dieses verändern dürfen, so könnten wir das Problem dadurch lösen, dass wir das Array sortieren und dann aufeinanderfolgende Elemente auf Gleichheit vergleichen. In diesem Abschnitt gehen wir jedoch davon aus, dass wir zwar Zugriff auf den Datensatz haben, wir diesen aber nicht verändern dürfen. Solch eine Annahme ist beispielsweise dann sinnvoll, wenn jeder einzelne Datensatz s_i sehr gross ist und ein Umsortieren daher mit sehr vielen zeitaufwändigen Speicherzugriffen verbunden wäre.

Wir lösen dieses Problem auf zwei Arten, jeweils unter Verwendung von Hashfunktionen. Dafür erinnern wir uns zunächst, was eine Hashfunktion ist. Gilt $\mathcal{S} \subseteq U$, d.h. unser Datensatz ist eine Teilmenge eines sogenannten

Universums U, so ist eine *Hashfunktion* eine Abbildung $h : U \to [m]$, wobei $[m] = \{1, \ldots, m\}$ und m die Anzahl der zusätzlich verfügbaren Speicherzellen ist. Wie üblich gehen wir dabei davon aus, dass die Hashfunktion h zum einen *effizient berechenbar* ist und zum anderen die Elemente *zufällig verteilt*, also für jedes Element $u \in U$ gilt: $\Pr[h(u) = i] = 1/m$, für alle $i \in [m]$, wie dies beispielsweise der Fall ist, wenn wir h zufällig aus einer universellen Hashfamilie wählen.

Mit solch einer Hashfunktion können wir unser Problem dann zum Beispiel mit einer Hashmap lösen. In dieser speichern wir die Elemente $(h(s_i), i)$ für alle $i \in \{1, \ldots, n\}$ und sortieren die Elemente dann bezüglich der ersten Koordinate. Anschliessend müssen wir dann nur noch diejenigen Datensätze auf Gleichheit vergleichen, die identische Einträge in der ersten Koordinate haben. Um die Güte dieses Verfahrens abzuschätzen, müssen wir die erwartete Anzahl Kollisionen bestimmen. Dieser Erwartungswert setzt sich zusammen aus der Anzahl tatsächlicher Duplikate zuzüglich der Anzahl der unbeabsichtigten Kollisionen. Um letztere abzuschätzen, definieren wir für jeden Datensatz s_i eine Indikatorvariable X_i, die genau dann Eins ist, wenn es eine Kollision mit einem Datensatz s_j mit $s_j \neq s_i$ gibt. Um $\Pr[X_i = 1]$ abzuschätzen, gehen wir wie folgt vor. Nach Annahme an die Hashfunktion wird jeder Datensatz $s_j \neq s_i$ auf einen zufälligen Wert aus $[m]$ abgebildet. Daher gilt:

$$\Pr[X_i = 1] = 1 - (1 - \tfrac{1}{m})^{n-1}$$

und somit

$$\begin{aligned} \mathbb{E}[\text{Anzahl Kollisionen}] &\leq \text{Anzahl Duplikate} + \sum_{i=1}^{n} \mathbb{E}[X_i] \\ &\leq \text{Anzahl Duplikate} + n \cdot (1 - (1 - \tfrac{1}{m})^{n-1}). \end{aligned}$$

Wenn wir daher m so wählen, dass $1 - (1 - \frac{1}{m})^{n-1} = O(1/n)$ gilt, so erwarten wir nur konstant viele zusätzliche Kollisionen. Dies ist zum Beispiel dann der Fall, wenn wir $m = n^2$ setzen. In diesem Fall können wir daher alle Kollisionen nach Sortieren der Hashmap durch $O(\text{Anzahl Duplikate})$ Vergleiche der Datensätze bestimmen. Diese Lösung unseres Problems benötigt $\Theta(n(\log n + \log m)) = \Theta(n \log n)$ zusätzlichen Speicher (für das Abspeichern der Hashmap) und Laufzeit $\Theta(n \log n)$ (für das Sortieren der Hashmap).

Eine Alternative zur Hashmap sind sogenannte *Bloomfilter*, die immer dann zum Einsatz kommen, wenn n sehr gross ist und man sich daher die zusätzlichen logarithmischen Faktoren bei Speicher und Laufzeit einsparen möchte. Grundlegende Idee ist hier, nicht nur eine, sondern k viele Hashfunktionen $h_1, \ldots, h_k : U \to [m]$ zu verwenden. Als Speicher M verwenden wir m *Bits*, die wir zu Beginn alle auf Null setzen. Zusätzlich initialisieren wir eine Liste $\mathcal{L}$ von möglichen Wiederholungen mit $\mathcal{L} := \emptyset$: $i \in [n]$ heisst *Wiederholung* in S falls $s_i \in \{s_1, \ldots, s_{i-1}\}$. Für jeden Datensatz s_i, $1 \leq i \leq n$, gehen wir dann wie folgt vor:

(1) Berechne $(x_1, \ldots, x_k) := (h_1(s_i), \ldots, h_k(s_i))$,

(2) gilt $(M[x_1], \ldots, M[x_k]) = (1, \ldots, 1)$, so fügen wir i zu $\mathcal{L}$ hinzu, ansonsten setzen wir $(M[x_1], \ldots, M[x_k]) := (1, \ldots, 1)$.

Man kann sich leicht überlegen, dass tatsächlich jede Wiederholung in $\mathcal{L}$ landet, wir aber auch falsche Einträge machen. Nachdem wir alle Datensätze s_i auf diese Weise behandelt haben, gehen wir nochmals alle Datensätze s_i durch und prüfen, ob $s_i = s_j$ für ein $j \in \mathcal{L}$, $j \neq i$, gilt. Unser Ziel ist es m und k so zu wählen, dass $\mathcal{L}$ mit hoher Wahrscheinlichkeit nur wenige Elemente enthält, sodass sich dann auch dieser zweite Durchlauf effizient durchführen lässt.

Um die Güte dieses Verfahrens abzuschätzen, müssen wir ähnlich zu vorhin die erwartete Länge der Liste $\mathcal{L}$ bestimmen. Dieser Erwartungswert setzt sich zusammen aus der Anzahl tatsächlicher Wiederholungen zuzüglich der sogenannten "*false positives*", nennen wir sie *falsche $\mathcal{L}$-Einträge*. Dies sind Elemente $i \in \mathcal{L}$, für die beim Abarbeiten von s_i bereits $M[h_j(s_i)] = 1$ für alle $1 \leq j \leq k$ gilt, obwohl s_i noch nicht in $(s_1, \ldots, s_{i-1})$ aufgetreten ist. Um die Anzahl der falschen $\mathcal{L}$-Einträge abzuschätzen, definieren wir wieder für jeden Datensatz s_i eine Indikatorvariable X_i, die genau dann Eins ist, wenn $M[h_j(s_i)] = 1$ für alle $1 \leq j \leq k$ gilt. Um $\Pr[X_i = 1]$ abzuschätzen, überlegen wir uns zunächst, was die Wahrscheinlichkeit ist, dass ein Bit $M[x]$ noch Null ist, nachdem wir alle von s_i verschiedenen $n-1$ Datensätze s_j eingefügt haben. Hierfür müssen wir genau $n-1$ mal k viele Hashfunktionen auswerten – und jede von ihnen schreibt mit Wahrscheinlichkeit $1/m$ eine Eins an die Stelle $M[x]$. Daher gilt:

$$\Pr[M[x] = 0] \leq (1 - \tfrac{1}{m})^{k(n-1)}$$

und wir erhalten

$$\Pr[X_i = 1] = \Pr[M[h_j(s_i)] = 1 \text{ für alle } 1 \leq j \leq k] \leq (1 - (1 - \tfrac{1}{m})^{k(n-1)})^k.$$

Obige Ungleichung gilt streng genommen nur, wenn die einzelnen Bits $M[x]$ *unabhängig* voneinander Eins wären. Dies sind sie aber nicht. Denn wenn eine Hashfunktion für einen Datensatz s_j an der Stelle x eine Eins schreibt, dann kann diese Hashfunktion nicht auch noch an der Stelle $x' \neq x$ eine Eins schreiben. Man kann zeigen, dass aus dieser Eigenschaft (man sagt hierzu auch: die Bits $M[x]$ sind *negativ korreliert*) folgt, dass die Ungleichheit in obiger Abschätzung trotz fehlender Unabhängigkeit dennoch gilt. Somit erhalten wir dann:

$$\begin{aligned} \mathbb{E}[\#\text{falsche } \mathcal{L}\text{-Einträge}] &\leq \sum_{i=1}^{n} \mathbb{E}[X_i] \\ &\leq n \cdot (1 - (1 - \tfrac{1}{m})^{k(n-1)})^k. \end{aligned}$$

Wenn wir daher k und m so wählen, dass $(1 - (1 - \frac{1}{m})^{k(n-1)})^k = O(1/n)$ gilt, so erwarten wir nur konstant viele "false positives". Etwas Rechnung zeigt, dass wir für $k = \ln n$ bereits für $m = n \ln n$ nur konstant viele "false positives" erwarten. Das folgende Beispiel verdeutlicht den Vorteil der Bloomfilter gegenüber der Hashmap für eine konkrete Anwendung.

Beispiel 2.80. Wir nehmen an, dass unser Datensatz aus $n = 10^7$ Einträgen besteht und nur ein einziges Duplikat enthält. Dieses möchten wir effizient bestimmen. Verwenden wir eine Hashmap, so können wir die erwartete Anzahl Kollisionen durch $n \cdot (1 - 1/m)^{n-1}$ abschätzen. Um diesen Wert für $n = 10^7$ auf Eins zu reduzieren müssen wir $m \approx 10^{14}$ wählen. Da jeder Eintrag in der Hashmap aus $\lceil \log_2 n \rceil + \lceil \log_2 m \rceil$ vielen Bits besteht, benötigen wir daher etwa $7.1 \cdot 10^8$ viele Bits. Wählen wir stattdessen einen Bloomfilter mit $k = \lceil \ln n \rceil$, so benötigen wir lediglich $m = 3.5 \cdot 10^8$ viele Bits an Speicher, um die erwartete Anzahl Kollisionen ebenfalls auf 1 zu reduzieren. Beachte: die Laufzeit ist beim Bloomfilter dann, wie auch wie auch bei der Hashmap, $\theta(n \log n)$, da wir pro Datensatz $k = \Theta(\log n)$ viele Hashfunktionen auswerten müssen. Wollen wir die Laufzeit auf $O(n)$ reduzieren, können wir beispielsweise $k = 3$ wählen. Der Speicherbedarf ist dann mit etwa $6.4 \cdot 10^9$ jedoch deutlich grösser.

Wir beenden diesen Abschnitt mit einer Variante des Problems, die zwar in dieser Form keine unmittelbare praktische Bedeutung hat – aber zeigt, was für überraschende Dinge man mit einer guten algorithmischen Herangehensweise erzielen kann. Der Algorithmus den wir hier vorstellen werden, ein nach Robert Floyd benannter Algorithmus für das Auffinden von Kreisen in gerichteten Graphen, kommt in verschiedenen Varianten in

ganz unterschiedlichen Bereichen der Informatik zum Einsatz. Wir schauen uns hier eine idealisierte Version an:

Gegeben ein Array $a[1,\ldots,n]$*, wobei* $a[i] \in \{1,\ldots,n-1\}$*. Entwerfe Sie einen Algorithmus, der in Laufzeit* $O(n)$ *zwei Indizes* $i \neq j$ *findet mit* $a[i] = a[j]$*. Der Algorithmus darf dabei das Array* a *nicht verändern, er kann aber beliebig oft auf die Elemente* $a[i]$ *zugreifen. Weiterhin darf der Algorithmus nur* $O(1)$ *zusätzliche Speicherzellen benutzen.*

Wir empfehlen dem Leser an dieser Stelle erst einmal nicht weiter zu lesen, sondern zu versuchen selbst eine Lösung zu finden. – Für diejenigen, die jetzt neugierig geworden sind, kommt hier nun die Lösung. Zunächst formulieren wir die Aufgabenstellung etwas um. Dazu definieren wir uns einen gerichteten Graphen $D = (V, A)$ wie folgt. Als Knotenmenge wählen wir $V = [n]$ und als Kantenmenge $A = \{(i, a[i]) \mid 1 \leq i \leq n\}$. Was können wir über diesen Graphen sagen? Aus der Definition folgt unmittelbar, dass jeder Knoten genau eine ausgehende Kante hat. Aus der Aufgabenstellung folgt zusätzlich, dass der Knoten n keine eingehende Kante hat. Wir betrachten jetzt den Subgraphen, den wir erhalten in dem wir ausgehend von Knoten n immer die ausgehende Kante weiterverfolgen. Da n keine eingehende Kante besitzt, besteht dieser Subgraph aus einem Pfad und einem anschliessenden Kreis. Nehmen wir an, der Pfad besteht aus $k \geq 1$ Kanten und der Kreis aus $\ell \geq 3$ Kanten. Wobei sicherlich $k+\ell \leq n$ gilt. Wir wählen nun ein $0 \leq r < \ell$ und $s \geq 1$ mit $k + r = s\ell$. (Beachte: für $k \leq \ell$ können wir $s = 1$ wählen, für $k > \ell$ nehmen wir $s = \lceil k/\ell \rceil$.) Definieren wir jetzt eine Folge von Knoten wie folgt: $x_0 := n$ und $x_i := a[x_{i-1}]$ für alle $i \geq 1$, so überlegt man sich schnell, dass $x_{k+r} = x_{2(k+r)}$ gelten muss. Insbesondere gibt es also ein $i \leq n$ mit $x_i = x_{2i}$. Ein solches können wir mit einem sogenannten Hase-und-Igel Ansatz in linearer Zeit unter Verwendung von nur

drei Variablen berechnen:

```
igel = a[n]; hase := a[a[n]]; i := 1
while (igel ≠ hase)
    igel = a[igel]; hase := a[a[hase]]; i := i + 1
```

Jetzt haben wir also ein $1 \leq i < n$ mit $x_i = x_{2i}$. Es ist auch nicht schwer, sich zu überlegen, dass $i = k + r$ gelten muss, mit r wie oben definiert. Daraus folgt dann sehr einfach, dass $x_k = x_{i+k}$ gelten muss. Wir können also das k bestimmen in dem wir den Igel weiterlaufen lassen und den Hasen zurück auf den Ausgangspunkt n setzen und ihn dieses Mal nur mit der Geschwindigkeit des Igels laufen lassen. Wenn wir uns dabei noch die jeweiligen Vorgängerknoten merken, so haben wir die gesuchten Indizes $i \neq j$ mit $a[i] = a[j]$ gefunden, sobald Hase und Igel wieder auf dem gleichen Knoten stehen:

```
hase = n;
while (igel ≠ hase)
    i := igel; j := hase;
    igel = a[igel]; hase := a[hase];
return i, j
```

Wir können also in der Tat die gesuchten Indizes in linearer Zeit unter Verwendung von nur vier zusätzlichen Speicherplätzen bestimmen.

Kapitel 3

Algorithmen - Highlights

3.1 Graphenalgorithmen

3.1.1 Lange Pfade

Neben kürzesten Wegen (oder Pfaden) ist es natürlich, auch nach einem längsten Pfad zwischen zwei Knoten in einem Graph zu fragen, oder einfach nach einem längsten Pfad in einem Graph. Wir betrachten hier die Entscheidungsvariante des Problems.

Das Problem. Gegeben (G, B), G ein Graph und $B \in \mathbb{N}_0$, stelle fest ob es einen Pfad der Länge B in G gibt. Wir nennen das das *LONG-PATH Problem*.

Zur Erinnerung: Ein Pfad der Länge ℓ in einem Graph $G = (V, E)$ ist eine Folge $\langle v_0, v_1, \ldots, v_\ell \rangle$ von paarweise verschiedenen Knoten, mit $\{v_{i-1}, v_i\} \in E$ für $i = 1, 2, \ldots \ell$. Beachte, dass $\ell + 1$ Knoten auf einem Pfad der Länge ℓ liegen.

Das Problem ist vermutlich schwer

Wir haben schon von einer Theorie ($\mathcal{NP}$-Vollständigkeit) gehört, die vermuten lässt, dass es keinen Polynomialzeit-Algorithmus gibt, der entscheidet, ob in einem gegebenen Graph ein Hamiltonkreis existiert. Wir wollen nun folgende Aussage zeigen: wenn das Hamiltonkreisproblem schwer ist (was zumindest plausibel ist), dann ist auch das LONG-PATH Problem schwer.

Dazu zeigen wir, dass wir für jeden Graph G mit n Knoten effizient einen Graph G′ mit $n' \leq 2n-2$ Knoten konstruieren können, sodass G einen Hamiltonkreis hat gdw.[1] G′ einen Pfad der Länge n hat.

Wähle dazu in G einen beliebigen Knoten v aus, entferne erst v und ersetze die zu v inzidenten Kanten $\{v, w_1\}, \{v, w_2\}, \ldots \{v, w_{\deg(v)}\}$ durch Kanten $\{\widehat{w_1}, w_1\}, \{\widehat{w_2}, w_2\}, \ldots \{\widehat{w_{\deg(v)}}, w_{\deg(v)}\}$, wobei $\widehat{w_1}, \widehat{w_2}, \ldots, \widehat{w_{\deg(v)}}$ neue Knoten sind. Das ist nun der obengenannte Graph G′, offensichtlich mit $(n-1) + \deg(v) \leq 2n-2$ Knoten; die Knoten $\widehat{w_1}, \widehat{w_2}, \ldots, \widehat{w_{\deg(v)}}$ haben alle Grad 1 in G′. Die Behauptung ist nun, dass G einen Hamiltonkreis hat gdw. G′ einen Pfad der Länge n hat.

(i) Sei $\langle v_1, v_2, v_3, \ldots, v_n, v_1\rangle$ ein Hamiltonkreis in G. O.B.d.A.[2] sei $v_1 = v$, der in unserer Konstruktion entfernte Knoten („O.B.d.A.“, weil wir jeden Kreis beim Knoten unserer Wahl in der Darstellung beginnen lassen können). Dann ist aber

$$\langle \widehat{v_2}, v_2, v_3, \ldots, v_n, \widehat{v_n}\rangle$$

ein Pfad der Länge n in G′.

(ii) Sei nun $\langle u_0, u_1, \ldots, u_n\rangle$ ein Pfad der Länge n in G′. Man beachte, dass alle Knoten $u_1, u_2, \ldots, u_{n-1}$ Grad mindestens Grad 2 in G′ haben, das müssen also genau die $n-1$ überlebenden Knoten von G sein, und folglich sind $u_0 = \widehat{w_i}$ und $u_n = \widehat{w_j}$ zwei verschiedene der neuen Knoten mit Grad 1 in G′. Daher muss $u_1 = w_i$ und $u_{n-1} = w_j$ sein und

$$\langle v, u_1, \ldots, u_{n-1}, v\rangle$$

ist ein Hamiltonkreis in G.

Wir können G′ aus G sicherlich in $O(n^2)$ Schritten erzeugen. Es gilt daher:

Satz 3.1. Falls wir LONG-PATH für Graphen mit n Knoten in $t(n)$ Zeit entscheiden können, dann können wir in $t(2n-2) + O(n^2)$ Zeit entscheiden, ob ein Graph mit n Knoten einen Hamiltonkreis hat.

Mit anderen Worten: Können wir LONG-PATH in polynomieller Zeit lösen ist, ein Brief an das Clay Mathematics Institute[3] fällig.

[1] genau dann wenn

[2] Ohne Beschränkung der Allgemeinheit

[3] Das Clay Mathematics Institute bietet eine Million US$ für einen polynomiellen Al-

Kurze lange Pfade

In einer biologischen Anwendung geht es genau darum, in einem Graph lange Pfade zu finden (Knoten sind Proteine und Kanten stellen Wechselwirkungen dar[4]). Dabei gibt es aber typischerweise im Vergleich zur Grösse des Graphs keine wirklich langen Pfade. Wir könnten uns also fragen, ob das LONG-PATH Problem bei Eingabe (G, B), mit B klein im Vergleich zu n, immer noch schwer ist oder eventuell in polynomieller Zeit lösbar ist.

Diese Frage war für $B = \log n$ einige Zeit offen, bis tatsächlich ein polynomieller Algorithmus mit einer sehr eleganten randomisierten Methode – *Color Coding* – gefunden wurde[5]. Wir wollen dieses Verfahren jetzt kennenlernen.

Hilfsmittel. Als erstes werden wir aber ein paar Notationen und einfache Hilfsmittel rekapitulieren, damit wir sie später griffbereit haben, wenn anderes unsere Aufmerksamkeit erfordert.

- $[n] := \{1, 2, \ldots, n\}$; $[n]^k$ ist die Menge der Folgen über $[n]$ der Länge k, es gilt $\left|[n]^k\right| = n^k$; $\binom{[n]}{k}$ ist die Menge der k-elementigen Teilmengen von $[n]$ und es gilt $\left|\binom{[n]}{k}\right| = \binom{n}{k}$.
- Für jeden Graph $G = (V, E)$ gilt $\sum_{v \in V} \deg(v) = 2|E|$.
- Die k Knoten auf einem Pfad der Länge $k-1$ kann man mit $[k]$ auf genau k^k Arten färben[6], $k!$ dieser Färbungen nutzen jede Farbe genau einmal; (das ist natürlich für jede Menge von k Knoten so, unabhängig davon, ob sie einen Pfad bilden).
- Für $c, n \in \mathbb{R}^+$, gilt $c^{\log n} = n^{\log c}$. Also, z.B. $2^{\log n} = n^{\log 2} = n$ und $2^{O(\log n)} = n^{O(1)}$ ist immer polynomiell in n.

---

gorithmus für das Hamiltonkreis Problem, oder für einen Beweis, dass dies nicht möglich ist („P vs NP Problem“).

[4] J. Scott, T. Ideker, R.M. Karp, R. Sharan, Efficient algorithms for detecting signaling pathways in protein interaction networks Proceedings of RECOMB (2005) 1-13; B. Kelley, R. Sharan, R. Karp, et al.: Conserved pathways within bacteria and yeast as revealed by global protein network alignment. Proc. Natl. Acad. Sci. USA 100 (2003) 11394-11399.

[5] N. Alon, R. Yuster, U. Zwick, Color-coding, Journal of the ACM, 42 (4) (1995) 844-856.

[6] Wir sprechen hier von beliebigen Färbungen (nicht nur in irgendeiner Weise zulässigen), z.B. können alle Knoten gleich gefärbt werden!

- Für $n \in \mathbb{N}_0$ gilt $\sum_{i=0}^{n} \binom{n}{i} = 2^n$, eine Anwendung des Binomialsatzes: $\sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} = (x+y)^n$.
- Für $n \in \mathbb{N}_0$ gilt $\frac{n!}{n^n} \geq e^{-n}$, was man leicht der Potenzreihenentwicklung der Exponentialfunktion ablesen kann:

$$e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!} \geq \frac{n^n}{n!} \quad \text{(der Term in der Summe für } i = n\text{)}$$

- Wiederholt man ein Experiment mit Erfolgswahrscheinlichkeit p solange, bis man Erfolg hat, dann ist der Erwartungswert der Anzahl der Versuche $\frac{1}{p}$ (geometrische Verteilung $\mathrm{Geo}(p)$). Das haben wir u.a. gerade bei der Diskussion von Monte Carlo Algorithmen wieder gesehen.

Bunte Pfade

Wir gehen einen Umweg[7] und untersuchen erst eine Variante des Problems.

Dazu betrachten wir einen mit k Farben gefärbten Graphen, d.h. $G = (V, E)$ mit einer Abbildung $\gamma\colon V \to [k]$. Ein Pfad in G heisst nun *bunt*, wenn alle Knoten auf dem Pfad verschiedene Farben haben. Wir beschreiben nun einen Algorithmus, der entscheidet, ob es in G einen bunten Pfad der Länge $k-1$ gibt (es müssen auf dem Pfad also alle k Farben genau einmal zu finden sein).

Dazu definieren wir für $v \in V$ und $i \in \mathbb{N}_0$ die Menge

$$P_i(v) := \left\{ S \in \binom{[k]}{i+1} \,\middle|\, \exists \text{ in } v \text{ endender genau mit } S \text{ gefärbter bunter Pfad} \right\};$$

$P_i(v)$ enthält also eine Menge S von $i+1$ Farben gdw. es einen bunten Pfad mit Endknoten v gibt, dessen Farben genau die Farben in S sind. Beachte, ein solcher Pfad muss immer die Länge genau i haben. Auch muss jedes $S \in P_i(v)$ natürlich die Farbe $\gamma(v)$ von v enthalten. Können wir die Menge $P_{k-1}(v)$ für jedes $v \in V$ berechnen, so ist unser Problem gelöst weil

$$\exists \text{ bunter Pfad der Länge } k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset\ .$$

[7] Sehr passend beim Thema lange Pfade.

Offensichtlich gilt $P_0(v) = \{\{\gamma(v)\}\}$ und

$$P_1(v) = \{\{\gamma(x), \gamma(v)\} \mid x \in N(v) \text{ und } \gamma(x) \neq \gamma(v)\}.$$

Allgemein bekommen wir eine Menge $S = R \dot\cup \{\gamma(v)\}$ in $P_i(v)$ gdw. wir einen genau mit R gefärbten bunten Pfad (der Länge $|R|-1$) zu einem Nachbarn x von v finden. Das können wir als Formel schreiben:

$$P_i(v) = \bigcup_{x \in N(v)} \left\{R \cup \{\gamma(v)\} \mid R \in P_{i-1}(x) \text{ und } \gamma(v) \notin R\right\}$$

Als Algorithmus liest sich diese Relation wie folgt:

BUNT(G, i)

```
1: for all v ∈ V do
2:     P_i(v) ← ∅
3:     for all x ∈ N(v) do
4:         for all R ∈ P_{i-1}(x) mit γ(v) ∉ R do
5:             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}}
```

Wir berechnen also, ausgehend von den Mengen $P_0(v)$ (für alle $v \in V$), sukzessive in $k-1$ Runden die Mengen $P_1(v)$, $v \in V$, bis zu den Mengen $P_{k-1}(v)$, $v \in V$.

Jede Menge in $P_{i-1}(v)$ hat genau i Elemente und wegen $P_{i-1}(v) \subseteq \binom{[k]}{i}$ gilt sicherlich $|P_{i-1}(v)| \leq \binom{k}{i}$. Folglich können wir die i-te Runde (d.h. BUNT(G, i), wo wir die P_i's aus den P_{i-1}'s berechnen) in Zeit

$$O\left(\overbrace{\sum_{v \in V} \deg(v)}^{2m} \cdot \binom{k}{i} \cdot i\right) = O\left(\binom{k}{i} \cdot i \cdot m\right)$$

bewältigen (wobei m die Anzahl der Kanten in G ist; der Faktor „i“ berücksichtigt, dass wir für jedes $R \in P_{i-1}(x)$ prüfen müssen, ob $\gamma(v) \notin R$ gilt). Insgesamt, über alle $k-1$ Runden, ergibt das

$$O\left(\sum_{i=1}^{k-1} \binom{k}{i} im\right) = O(2^k km).$$

Hier haben wir erst den Faktor i durch k abgeschätzt, dann den Faktor km herausgehoben, und schliesslich $\sum_{i=1}^{k-1} \binom{k}{i} \leq \sum_{i=0}^{k} \binom{k}{i} = 2^k$ abgeschätzt.

Wir haben es tatsächlich geschafft für $k = O(\log n)$ einen polynomiellen Algorithmus zu entwerfen und zu analysieren, allerdings nur für bunte Pfade. Wie wenden wir das aber jetzt für unser ursprüngliches farbloses LONG-PATH Problem an?

Auf gut Glück – Zufallsfärbungen

Wir haben also jetzt einen Graph G mit $B \in \mathbb{N}_0$ gegeben und wollen entscheiden, ob es in G einen Pfad der Länge B gibt. Dazu färben wir die Knoten zufällig mit den Farben $[k]$, wobei $k := B + 1$, und prüfen, ob es einen bunten Pfad der Länge $k-1$ gibt. Natürlich, wenn es dann einen bunten Pfad der Länge $k-1$ gibt, dann haben wir definitiv einen Pfad der Länge $B = k-1$ und wir können das so behaupten. Andererseits, wenn es in G einen Pfad der Länge $k-1$ gibt, dann können wir Glück haben, ein solcher Pfad wird bunt gefärbt und unser Algorithmus wird diesen dann auch finden. Oder wir können Pech haben, und kein Pfad der Länge $k-1$ wird bunt gefärbt. Was ist die Wahrscheinlichkeit, dass wir Glück haben?

Zum Zwecke der Analyse nehmen wir an, G enthält einen Pfad der Länge $k-1$. Sei P ein solcher Pfad. Dann gibt es k^k mögliche Färbungen von P mit k Farben, in $k!$ davon ist der Pfad bunt. Folglich gilt für die Erfolgswahrscheinlichkeit

$$\begin{aligned} p_{\text{Erfolg}} &:= \Pr[\exists \text{ bunter Pfad der Länge } k-1] \\ &\geq \Pr[P \text{ ist bunt}] = \frac{k!}{k^k} \geq e^{-k}; \end{aligned}$$

hier setzen wir ein weiteres der bereit gestellten Hilfsmittel ein.

Wir erhalten nun leicht (mit Blick auf die geometrische Verteilung):

Satz 3.2. Sei G ein Graph mit einem Pfad der Länge $k-1$.

(1) Eine zufällige Färbung mit k Farben erzeugt einen bunten Pfad der Länge $k-1$ mit Wahrscheinlichkeit $p_{\text{Erfolg}} \geq e^{-k}$.

(2) Bei wiederholten Färbungen mit k Farben ist der Erwartungswert der Anzahl Versuche, bis man einen bunten Pfad der Länge $k-1$ erhält $\frac{1}{p_{\text{Erfolg}}} \leq e^k$.

Wir können jetzt leicht einen Monte Carlo Algorithmus bauen, der unser Problem in polynomieller Zeit löst. Dazu wählen wir ein $\lambda \in \mathbb{R}$, $\lambda > 1$, und

wiederholen unseren Test höchstens $\lceil \lambda e^k \rceil$ mal, bis wir eine Bestätigung haben, dass es einen Pfad der Länge $k-1$ gibt. Gelingt dies, antworten wir JA; scheitern wir in allen Versuchen, antworten wir NEIN.

Satz 3.3.

(1) Der Algorithmus hat eine Laufzeit von $O(\lambda(2e)^k km)$.

(2) Antwortet der Algorithmus mit JA, dann hat der Graph einen Pfad der Länge $k-1$.

(3) Hat der Graph einen Pfad der Länge $k-1$, dann ist die Wahrscheinlichkeit, dass der Algorithmus mit NEIN antwortet, höchstens $e^{-\lambda}$.

Teil (3) folgt dabei aus der uns bekannten Ungleichung $\forall x \in \mathbb{R} : 1+x \leq e^x$, denn die Wahrscheinlichkeit, dass der Algorithmus mit NEIN antwortet, ist (mit $x = e^{-k}$) höchstens

$$\left(1-e^{-k}\right)^{\lceil \lambda e^k \rceil} \leq \left(e^{-e^{-k}}\right)^{\lceil \lambda e^k \rceil} \leq e^{-\lambda}.$$

Es handelt sich also um einen Monte Carlo Algorithmus, der mit Wahrscheinlichkeit von mindestens $1-e^{-\lambda}$ die korrekte Antwort liefert (mit einseitigem Fehler). Man beachte, dass wir hier das Argument von Satz 2.74 wiederholt haben, bzw. diesen Satz mit $\varepsilon = e^{-k}$ und $\delta = e^{-\lambda}$ hätten verwenden können.

Anmerkungen.

(1) Es gibt eine deterministische Variante, die das Problem für $B = O(\log n)$ in Polynomialzeit löst.

(2) Es gibt eine randomisierte Verbesserung,[8] die in Zeit $O(2^k \text{poly}(n))$ läuft (statt $O((2e)^k \text{poly}(n))$.

(3) Wenn man neben der Existenz eines Pfades der Länge $k-1$ einen solchen Pfad tatsächlich finden will, so kann man den Algorithmus oben leicht in die Richtung adaptieren. Man merkt sich dazu einfach zu jedem $S \in P_i(v)$ eine

[8] R. Williams, Finding paths of length k in $O^*(2^k)$ time, Information Processing Letters 109:6 (2009) 315-318.

Realisierung, d.h. einen genau mit S gefärbten bunten Pfad $\langle u_0, u_1, \ldots, u_i \rangle$, $u_i = v$, der Länge i nach v.

(4) Das Problem wird einfach in gerichteten azyklischen Graphen: Man gewichtet alle Kanten einfach mit -1 und berechnet kürzeste Pfade (siehe z.B. die Algorithmen und Datenstruktur-Vorlesung im Herbst).

3.1.2 Flüsse in Netzwerken

Maximale Flüsse in Netzwerken sind ein klassisches Beispiel der Algorithmik und Diskreten Mathematik mit zahlreichen Anwendungen in Wissenschaft und Praxis, z.B. in der Verkehrsplanung, beim *data mining*, in der Bildverarbeitung, etc. Die Berechnung eines maximalen Flusses in einem Netzwerk ist eines der schwierigsten Probleme, die „noch" effizient (in polynomieller Zeit) lösbar sind. In diesem Abschnitt konzentrieren wir uns auf die Modellierung des Problems, den Zusammenhang mit dem Problem minimaler Schnitte sowie zwei Beispielen, wie man andere Probleme mit Hilfe von Netzwerkflüssen lösen kann.

Modellierung

Wir stellen uns ein verzweigtes Netzwerk von Röhren unterschiedlicher Dicke vor, in das man an einer Stelle Flüssigkeit zuführen kann, die an einer anderen Stelle entweichen kann. Der Elektrotechniker denkt vielleicht eher an ein System von Leitungen unterschiedlichen Widerstands, durch die Strom fliesst. Und der Verkehrsplaner denkt an ein System von Strassen verschiedener Breite, über die Verkehr fliesst (es kann auch so etwas wie Einbahnen geben). Uns interessiert, wie viel Wasser (Strom, Verkehr) pro Zeiteinheit von einem gegebenen Startpunkt zu einem gegebenen Endpunkt fliessen kann.

Ein solches System abstrahieren (bzw. modellieren)[9] wir wie folgt.

Definition 3.4. Ein *Netzwerk* ist ein Tupel $N = (V, A, c, s, t)$, wobei gilt:

[9] Wir abstrahieren, um das Problem des Elektrotechnikers, Verkehrsplaners, etc. in einem Modell lösen zu können, wir modellieren, um die Probleme exakt (und in einem falsifizierbaren Kontext) lösen zu können.

(V, A) ist ein gerichteter Graph,

$s \in V$, die *Quelle* (engl.: source),

$t \in V \setminus \{s\}$, die *Senke* (engl.: target), und

$c : A \to \mathbb{R}_0^+$, die *Kapazitätsfunktion* (engl.: capacity function).

Die Kapazitätsfunktion beschränkt, wie viel durch eine Kante fliessen kann (und wir sehen, dass wir schon Fluss nur in eine Richtung vorgesehen haben). Für einen Fluss gibt es eine einfache allgemeine Bedingung, die sich aus der Tatsache ergibt, dass das System abgesehen von Quelle und Senke abgeschlossen ist und deswegen in einem inneren Knoten weder Fluss entstehen noch verschwinden kann. In der Elektrotechnik kennt man diese Eigenschaft als erstes Kirchhoff'sches Gesetz (Gustav Robert Kirchhoff, 1824, Königsberg, –1887, Berlin): *„Die Summe der zufliessenden Ströme in einem elektrischen Knotenpunkt ist gleich der Summe der abfliessenden Ströme."*

Diese Eigenschaft lassen wir in die Definition des Flusses in einem Netzwerk eingehen.

Definition 3.5. Gegeben sei ein Netzwerk $N = (V, A, c, s, t)$. Ein Fluss in N ist eine Funktion $f : A \to \mathbb{R}$ mit den Bedingungen

$0 \leq f(e) \leq c(e)$ für alle $e \in A$, die *Zulässigkeit*, und

für alle $v \in V \setminus \{s, t\}$ gilt

$$\sum_{u \in V:\,(u,v) \in A} f(u, v) = \sum_{u \in V:\,(v,u) \in A} f(v, u)$$

die *Flusserhaltung*.

Der *Wert* (engl.: value) eines Flusses f ist durch

$$\mathrm{val}(f) := \mathrm{netoutflow}(s) := \sum_{u \in V:\,(s,u) \in A} f(s, u) - \sum_{u \in V:\,(u,s) \in A} f(u, s)$$

definiert. Wir nennen f *ganzzahlig*, wenn $f(e) \in \mathbb{Z}\ \forall e \in A$.

In unserem Modell besagt der Wert eines Flusses also, wie viel Flüssig-

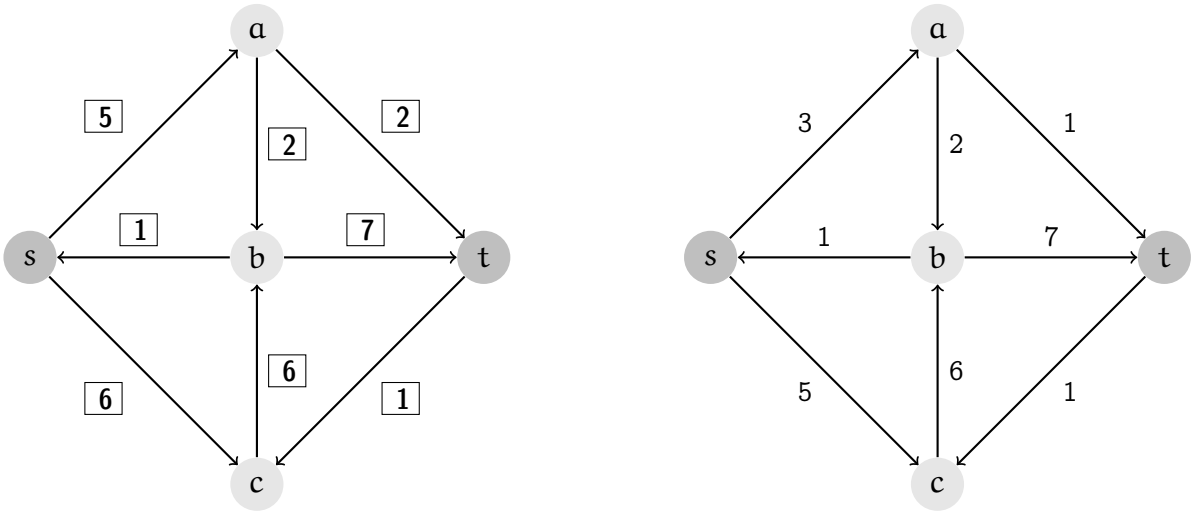


Abbildung 3.1: Ein Netzwerk und ein Fluss mit Wert $3 - 1 + 5 = 7$.

keit „netto“ von der Quelle in das System fliesst. Wenn unsere Modellierung vernünftig ist, sollte folglich dieselbe Menge an Flüssigkeit an der Senke ankommen und das System verlassen. Das lässt sich nun auch mathematisch beweisen (was unsere Modellierung bekräftigt).

Lemma 3.6. Der Nettozufluss der Senke gleicht dem Wert des Flusses, d.h.

$$\mathsf{netinflow}(t) := \sum_{u \in V:\,(u,t)\in A} f(u,t) - \sum_{u \in V:\,(t,u)\in A} f(t,u) \quad = \quad \mathsf{val}(f)\ .$$

Beweis. Es gilt

$$\begin{aligned} 0 &= \sum_{(v,u)\in A} f(v,u) - \sum_{(u,v)\in A} f(u,v) && (3.1)\\ &= \sum_{v\in V} \underbrace{\left(\sum_{u\in V:\,(v,u)\in A} f(v,u) - \sum_{u\in V:\,(u,v)\in A} f(u,v) \right)}_{=0 \ \text{für} \ v\notin\{s,t\}} && (3.2)\\ &= \underbrace{\left(\sum_{u\in V:\,(s,u)\in A} f(s,u) - \sum_{u\in V:\,(u,s)\in A} f(u,s) \right)}_{=\mathsf{val}(f)} + \underbrace{\left(\sum_{u\in V:\,(t,u)\in A} f(t,u) - \sum_{u\in V:\,(u,t)\in A} f(u,t) \right)}_{=-\mathsf{netinflow}(t)}. \end{aligned}$$

woraus die Aussage unmittelbar folgt. (Die Summe in (3.2) ist nur eine Umordnung der Terme in (3.1).) □

Das algorithmische Problem ist nun, für ein Netzwerk effizient einen maximalen Fluss zu berechnen, d.h. einen Fluss grösstmöglichen Werts. Schon bei der Problemstellung ist Vorsicht geboten, da es nicht klar ist, dass ein solcher maximaler Fluss existiert: Ähnlich wie das offene Intervall $(0,1)$ keine grösste Zahl hat, könnte es eine steigende Folge von Werten von Flüssen geben, deren Grenzwert aber nicht dem Wert eines Flusses entspricht. Offensichtlich gibt es im Allgemeinen unendlich viele Flüsse und so ist selbst bei Existenz eines maximalen Flusses dessen Berechenbarkeit nicht offensichtlich. Und schliesslich ist es nicht klar mit welchem Argument man belegen sollte, dass ein Fluss maximal ist, selbst wenn man einen solchen in der Hand haben sollte.

Etwas Licht auf diese Fragen wirft die Betrachtung minimaler Schnitte in Netzwerken.

Schnitte

Teilt man die Knotenmenge eines Netzwerks in zwei Teile S (mit der Quelle) und T (mit der Senke), so muss jeder Fluss von s nach t durch Kanten aus $S \times T$ fliessen. Die Kapazität dieser Kanten sollte also eine Beschränkung für einen Fluss im Netzwerk darstellen. Diese Intuition wollen wir nun modellieren und beweisen.

Definition 3.7. Ein s-t-*Schnitt* für ein Netzwerk (V, A, c, s, t) ist eine Partition (S, T) von V (d.h. $S \cup T = V$ und $S \cap T = \emptyset$) mit $s \in S$ und $t \in T$. Die *Kapazität* eines s-t-Schnitts (S, T) ist durch

$$\mathrm{cap}(S,T) := \sum_{(u,w)\in(S\times T)\cap A} c(u,w)$$

definiert (siehe Abbildung 3.2).

Lemma 3.8. Ist f ein Fluss und (S, T) ein s-t-Schnitt in einem Netzwerk

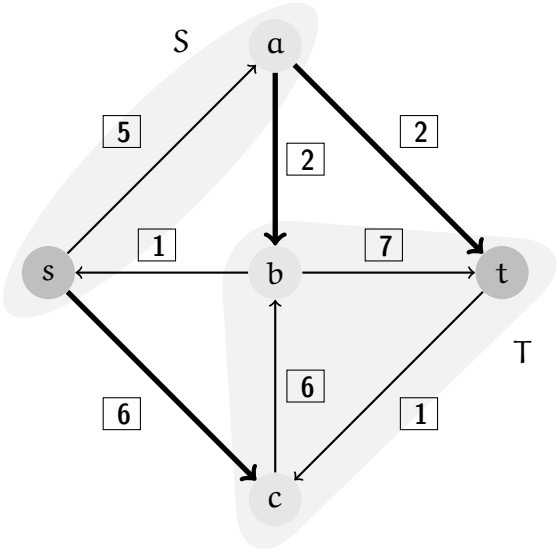


Abbildung 3.2: Schnitt mit Kapaziät $6+2+2=10$.

(V, A, c, s, t), so gilt

$$\mathsf{val}(f) \leq \mathsf{cap}(S, T) \ .$$

Beweis. Für eine Partition (U, W) von V setzen wir

$$f(U, W) := \sum_{(u,w)\in(U\times W)\cap A} f(u, w) \ .$$

Nun behaupten wir

$$\mathsf{val}(f) \overset{(i)}{=} f(S, T) - f(T, S) \overset{(ii)}{\leq} f(S, T) \overset{(iii)}{\leq} \mathsf{cap}(S, T),$$

was ja genau die Aussage des Lemmas ergibt. Die Relation (ii) folgt aus der Nichtnegativität des Flusses auf jeder Kante, die Relation (iii) ergibt sich aus der Beschränkung „$f(u, w) \leq c(u, w)$“ für Flüsse. Für den Beweis von (i) kommt uns die Erfahrung aus dem Beweis zu Lemma 3.6 zugute:

$$\begin{aligned}\mathsf{val}(f) &= \sum_{u\in V:\,(s,u)\in A} f(s, u) - \sum_{u\in V:\,(u,s)\in A} f(u, s) \\ &= \sum_{v\in S} \underbrace{\left(\sum_{u\in V:\,(v,u)\in A} f(v, u) - \sum_{u\in V:\,(u,v)\in A} f(u, v) \right)}_{=0 \ \text{für} \ v\neq s}\end{aligned}$$

In dieser Summe heben sich die Beiträge aller Kanten (u, w) auf, die beide Endpunkte in S haben. Es bleiben nur die Kanten, die nur ihren Anfangspunkt in S haben (und die ihren Fluss positiv beitragen) und die Kanten, die ihren Endpunkt in S haben (und ihren Fluss negativ beitragen). Daher können wir wie folgt fortsetzen:

$$\begin{aligned} &= \sum_{(u,w)\in(S\times T)\cap A} f(u,w) - \sum_{(u,w)\in(T\times S)\cap A} f(u,w) \\ &= f(S,T) - f(T,S) \end{aligned}$$

□

Das Problem der Berechnung eines minimalen Schnitts scheint fürs erste einfacher, als maximale Flüsse zu bestimmen. Jedes Netzwerk hat nur endlich viele s-t-Schnitte (wenn auch exponentiell viele: $2^{|V|-2}$), aber so ist auf jeden Fall die Existenz eines minimalen Schnitts gesichert und wir können diesen prinzipiell auch berechnen (wenn auch noch nicht klar ist, wie man das effizient machen sollte). Überraschenderweise sind die beiden betrachteten Probleme äquivalent, wie folgende klassische Dualität zeigt.

Satz 3.9 („Maxflow-Mincut Theorem"). Jedes Netzwerk $N = (V, A, c, s, t)$ erfüllt

$$\max_{f \text{ Fluss in } N} \mathsf{val}(f) = \min_{(S,T) \text{ s-t-Schnitt in } N} \mathsf{cap}(S,T)$$

Den Beweis dieses Satzes können wir später nur in abgeschwächter Form liefern. Er geht Hand in Hand mit der algorithmischen Erschliessung des Problems.

Augmentierende Pfade

Die Grundidee fast aller Algorithmen ist, mit einem beliebigen Fluss zu beginnen (f konstant 0 eignet sich immer), und diesen sukzessive zu verbessern. Dazu ein erster Ansatz. Angenommen, wir finden einen gerichteten Pfad P von Quelle zu Senke, wo der Fluss auf allen Kanten die Kapazität noch nicht erschöpft hat, d.h. $f(e) < c(e)$ für alle e auf P. Sei nun $\delta := \min_{e\in P} c(e) - f(e)$. Wenn wir nun auf allen Kanten auf P den Fluss um δ erhöhen, verletzen wir an keiner Stelle die Flusseigenschaft, und der Wert des Flusses hat sich um δ erhöht.

So weit, so gut. Allerdings gibt es Flüsse, die sich nach diesem Schema nicht verbessern lassen, obwohl sie nicht optimal sind. Man kann nämlich die Erhöhung eines Flusses an einer eingehenden Kante eines Knotens nicht nur durch Erhöhung des Flusses an einer ausgehenden Kante kompensieren, sondern auch durch Verringerung des Flusses an einer anderen eingehenden Kante. In diesem Sinne kann man nun Pfade von Quelle zu Senke betrachten, auf denen Kanten vorwärts und rückwärts gerichtet sein können. Nehmen wir an, dass wir auf jeder vorwärts gerichteten Kante den Fluss um δ erhöhen können, ohne die Zulässigkeit zu verletzen, und auf jeder rückwärts gerichteten Kante den Fluss um δ verkleinern können, ohne dass dieser negativ wird. Dann können wir entlang dieses Pfades den Fluss entsprechend modifizieren, ohne die Flusserhaltung zu verletzen. Verbessert man einen Fluss sukzessive mittels solcher sogenannter *augmentierender* Pfade, so kann man nur in maximalen Flüssen stecken bleiben (was man beweisen muss).

Das heisst aber nicht, dass man so einen maximalen Fluss in endlich vielen Schritten erreicht (dies gilt interessanterweise nur, wenn die Kapazitäten rational sind).

Das Restnetzwerk

Unser nächstes Ziel ist es nun, die Idee der augmentierenden Pfade algorithmisch konkreter zu fassen und einen ersten einfachen Algorithmus zur Berechnung eines maximalen Flusses zu beschreiben. Dazu machen wir eine vereinfachende Annahme: Wir beschränken uns auf Netzwerke *ohne entgegen gerichtete Kanten*, d.h. zwischen zwei Knoten gibt es nie zugleich die beiden Kanten beider Richtungen. Dies dient hier nur zur Vereinfachung der Darstellung, man kann das auf verschiedene Arten umgehen, ohne schlechtere Ergebnisse zu bekommen.

Die folgende Definition beschreibt für ein Netzwerk mit Fluss f, was relativ zu f der verbleibende Spielraum an einer Kante e ist, d.h. wie viel mehr oder weniger (als $f(e)$) Fluss man durch diese Kante schicken kann. Dabei bezeichnen wir für eine Kante $e = (u, v)$ die entgegen gerichtete Kante (v, u) mit e^{opp}.

Definition 3.10. Sei $N = (V, A, c, s, t)$ ein Netzwerk ohne entgegen ge-

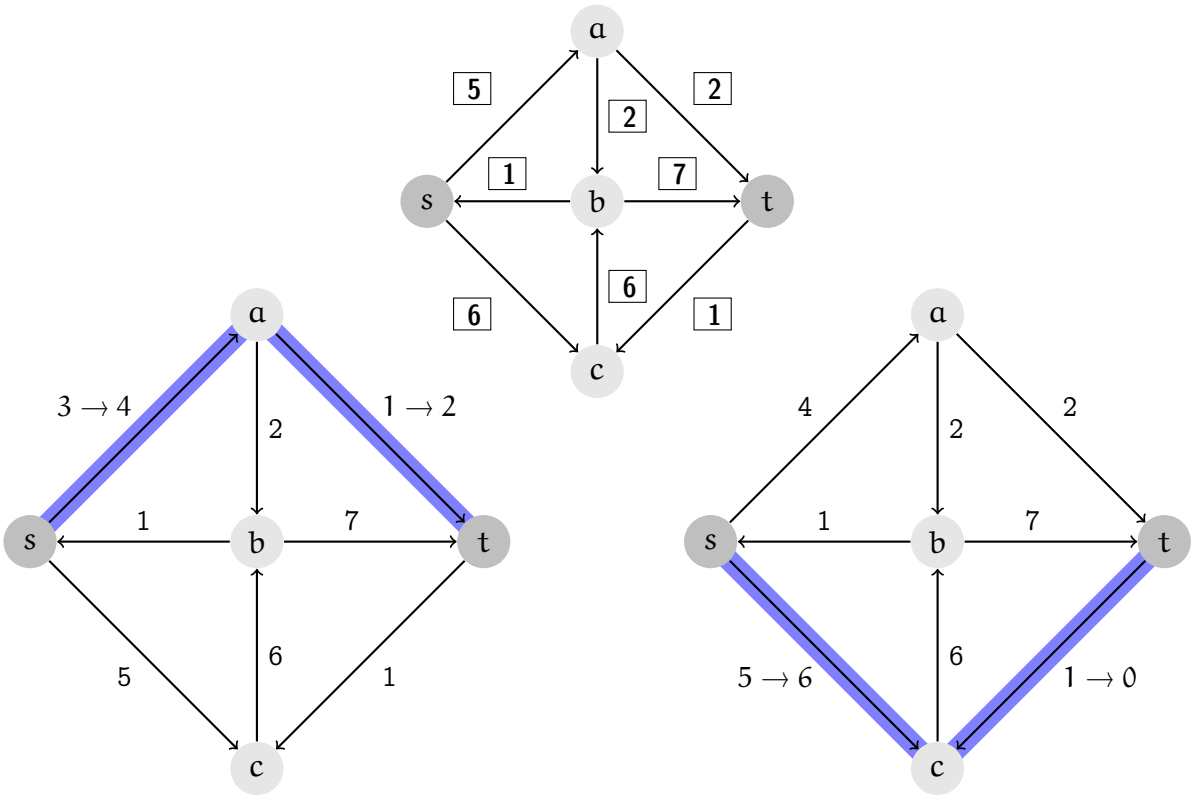


Abbildung 3.3: Augmentierende Pfade, der rechte mit rückwärts gerichteter Kante.

richtete Kanten und sei f ein Fluss in N. Das Restnetzwerk $N_f := (V, A_f, r_f, s, t)$ ist wie folgt definiert:

(1) Ist $e \in A$ mit $f(e) < c(e)$, dann ist e auch eine Kante in A_f, mit $r_f(e) := c(e) - f(e)$.

(2) Ist $e \in A$ mit $f(e) > 0$, dann ist e^{opp} in A_f, mit $r_f(e^{opp}) = f(e)$.

(3) Nur Kanten wie in (1) und (2) beschrieben finden sich in A_f.

$r_f(e)$, $e \in A_f$, nennen wir die *Restkapazität* der Kante e.

Wir bemerken, dass wir zwar annehmen, dass es in N keine entgegen gerichteten Kanten gibt, diese dann im Restnetzwerk typischerweise sehr wohl auftreten werden; konkret existiert für eine Kante $e \in A$ sowohl e als auch e^{opp} in A_f, ausser $f(e) \in \{0, c(e)\}$, und dann gilt $r_f(e) + r_f(e^{opp}) = c(e)$. Die Restkapazitäten aller Kanten in A_f sind strikt positiv.

Satz 3.11. Ein Fluss f in einem Netzwerk N ist ein maximaler Fluss gdw. es im Restnetzwerk N_f keinen gerichteten Pfad von der Quelle s zur Senke t gibt. Für jeden solchen maximalen Fluss gibt es einen s-t-Schnitt (S, T) mit $val(f) = cap(S, T)$.

Beweis. Sei $N = (V, A, c, s, t)$. Angenommen, es gibt einen s-t-Pfad P in N_f mit ε der kleinsten auf diesem Pfad auftretenden Restkapazität (es gilt also $\varepsilon > 0$). Dann können wir den Fluss f zu einem Fluss f' wie folgt erhöhen.

$$f'(e) := \begin{cases} f(e) + \varepsilon & e \text{ auf } P, \\ f(e) - \varepsilon & e^{opp} \text{ auf } P, \text{ und} \\ f(e) & \text{sonst.} \end{cases}$$

Auf diese Weise kann f' nie negativ werden und respektiert immer die Kapazitätsgrenzen. Auch addieren oder substrahieren wir an jedem inneren Knoten den gleichen Wert ε vom Fluss f, so gilt auch die Flusserhaltung weiter. Wir haben so einen neuen Fluss f' mit $val(f') = val(f) + \varepsilon$ und f kann also kein maximaler Fluss gewesen sein.

Nun gehen wir davon aus, dass N_f keinen Pfad von s nach t erlaubt; wir zeigen, dass es dann einen s-t-Schnitt (S, T) mit $cap(S, T) = val(f)$ gibt. Da wir schon wissen (Lemma 3.8), dass kein Fluss die Kapazität eines Schnitts übersteigen kann, muss dann der Fluss f maximal sein.

Sei S die Menge der in (V, A_f) von s aus auf gerichteten Pfaden erreichbaren Knoten und sei $T := V \setminus S$. Es gilt $s \in S$, und nach Annahme $t \notin S$, d.h. (S, T) ist ein s-t-Schnitt. Ist $e = (u, v) \in A$, mit $u \in S$ und $v \notin S$, dann gibt es keine Kante in A_f von u nach v, was ja heisst, dass $f(e) = c(e)$. Ist $e = (v, u) \in A$, mit $u \in S$ und $v \notin S$, dann gibt es keine Kante in A_f von u nach v, was ja heisst, dass $f(e) = 0$, sonst wäre (u, v) eine Kante im Restnetzwerk und wir könnten über $u \in S$ auch v erreichen. Zusammenfassend, es gilt $f(S, T) = \mathsf{cap}(S)$ und $f(T, S) = 0$. Es folgt (siehe (i) im Beweis von Lemma 3.8)

$$\mathsf{val}(f) = f(S, T) - f(T, S) = \mathsf{cap}(S, T) - 0 = \mathsf{cap}(S, T) \ .$$

□

Algorithmen

Wie bereits besprochen, liegt nun ein algorithmischer Ansatz auf der Hand: Suche mit Hilfe des Restnetzwerks augmentierende Pfade, solange es solche gibt.

FORD-FULKERSON(V, A, c, s, t)	
1: $f \leftarrow \mathbf{0}$	▷ Fluss konstant 0
2: **while** $\exists$ s-t-Pfad P in (V, A_f) **do**	▷ augmentierender Pfad
3: Erhöhe den Fluss entlang P	▷ wie in Beweis zu Satz 3.11
4: **return** f	▷ maximaler Fluss

Man ist versucht, zu behaupten, dass der Algorithmus nun laut Satz 3.11 unser Problem löst. Allerdings haben wir nicht sichergestellt, dass der Algorithmus jemals zum Ende kommt. Es ist tatsächlich möglich, dass der Algorithmus bei irrationalen Kapazitäten nicht terminiert. Andererseits ist das bei ganzzahligen Kapazitäten nicht möglich: In jedem Schritt erhöhen wir in diesem Fall den Wert des Flusses um einen ganzzahligen Wert, und da der Wert des Flusses nach oben beschränkt ist (z.B. durch $\mathsf{cap}(\{s\}, V \setminus \{s\})$), ist eine Ende sichergestellt.

In einem ganzzahligen Netzwerk (ohne entgegen gerichtete Kanten) sei $U \in \mathbb{N}$ eine obere Schranke für die auftretenden Kapazitäten, sei n die Anzahl der Knoten und m die Anzahl der Kanten. Dann sieht man leicht, dass kein Fluss Wert gösser als nU haben kann ($\mathsf{val}(f) \leq \mathsf{cap}(\{s\}, V \setminus \{s\}) \leq$

$nU)$, d.h. der Ford-Fulkerson terminiert nach höchstens nU Runden. In einer Runde müssen wir das Restnetzwerk[10] konstruieren und einen Pfad von Quelle zu Senke suchen, das können wir in $O(m)$ Zeit erledigen.

Satz 3.12. Sind in einem Netzwerk ohne entgegen gerichtete Kanten alle Kapazitäten ganzzahlig und höchstens U, so gibt es einen ganzzahligen maximalen Fluss, der in Zeit $O(mnU)$ berechnet werden kann (m ist die Anzahl Kanten, n die Anzahl Knoten im Netzwerk).

Man beachte, dass wir jetzt tatsächlich (und erst jetzt!) das Maxflow-Mincut Theorem (Satz 3.9)

$$\max_{\text{f Fluss in N}} \mathsf{val}(f) = \min_{\text{(S,T) s-t-Schnitt in N}} \mathsf{cap}(S,T)$$

für Netzwerke ohne entgegen gesetzte Kanten und ganzzahligen Kapazitäten bewiesen haben.

- Lemma 3.8 zeigt, dass $\mathsf{val}(f) \leq \mathsf{cap}(S,T)$ für jeden Fluss f und jeden Schnitt (S,T) gilt.
- Lemma 3.11 zeigt, dass, wenn f ein maximaler Fluss ist, dann gibt es einen Schnitt (S,T) mit $\mathsf{cap}(S,T) = \mathsf{val}(f)$.
- Der Ford-Fulkerson Algorithmus zeigt, dass es unter den gegebenen Umständen (Ganzzahligkeit, keine entgegen gerichtete Kanten) einen maximalen Fluss gibt.

Weiterführende Algorithmen. Es gibt eine reichhaltige Literatur, aus der wir zwei Ergebnisse zitieren: Einerseits kann der U-Faktor bei ganzzahligen Schnitten durch einen Faktor logarithmisch in U ersetzt werden. Andererseits kann man im Einheitskostenmodell einen maximalen Fluss auch unabhängig von der Grösse der Zahlen, selbst bei reelen Zahlen, schnell berechnen.

[10] Tatsaächlich werden wir das Restnetzwerk nicht immer neu konstruieren, sondern schrittweise entlang des gewählten augmentierenden Pfades ändern.

Proposition 3.13 (*Capacity-Scaling*, Dinitz-Gabow, 1973). Sind in einem Netzwerk alle Kapazitäten ganzzahlig und höchstens U, so gibt es einen ganzzahligen maximalen Fluss, der in Zeit $O(mn(1+\log U))$ berechnet werden kann (m Anzahl Kanten, n Anzahl Knoten).

Proposition 3.14 (*Dynamic Trees*, Sleator-Tarjan, 1983). Der maximale Fluss eines Netzwerks kann in Zeit $O(mn \log n)$ berechnet werden (m Anzahl Kanten, n Anzahl Knoten).

Offensichtlich ist die erste Schranke besser, wenn U klein ist ($U = o(n)$). Wir werden gleich eine Anwendung sehen, in der alle Kapazitäten 1 sind (also $U = 1$). Man beachte, dass nach Satz 3.9 die entsprechenden Schranken auch für die Berechnung eines minimalen s-t-Schnitts gelten.

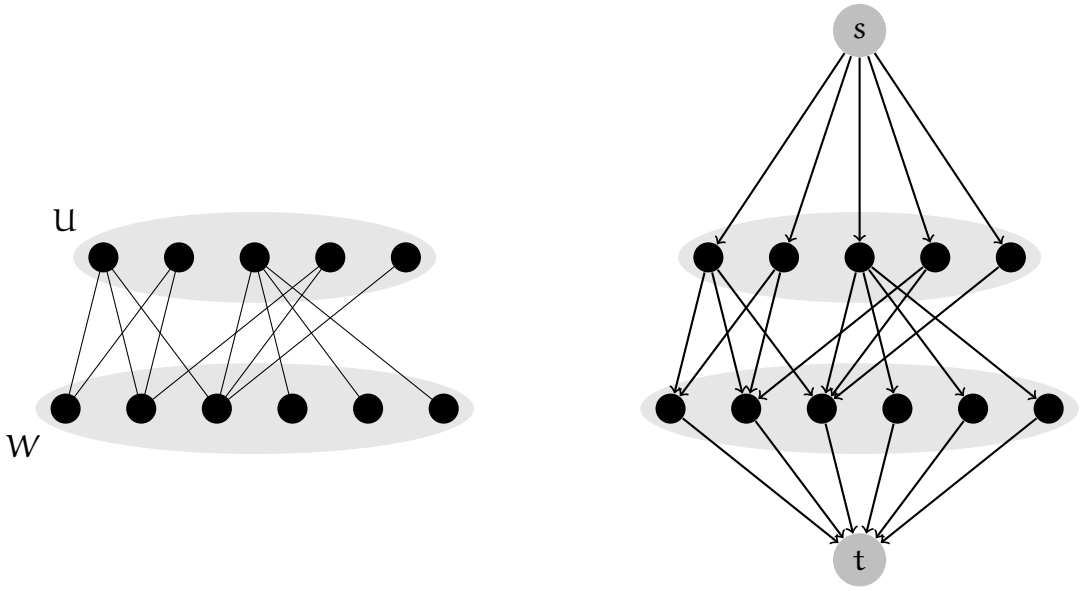
Bipartites Matching als Flussproblem

Sei $G = (V, E)$ ein bipartiter Graph, d.h. es gibt eine Partition (U, W) von V, sodass $E \subseteq \{\{u, w\} | u \in U, w \in W\}$. Wir interessieren uns für ein Matching maximaler Grösse in G.

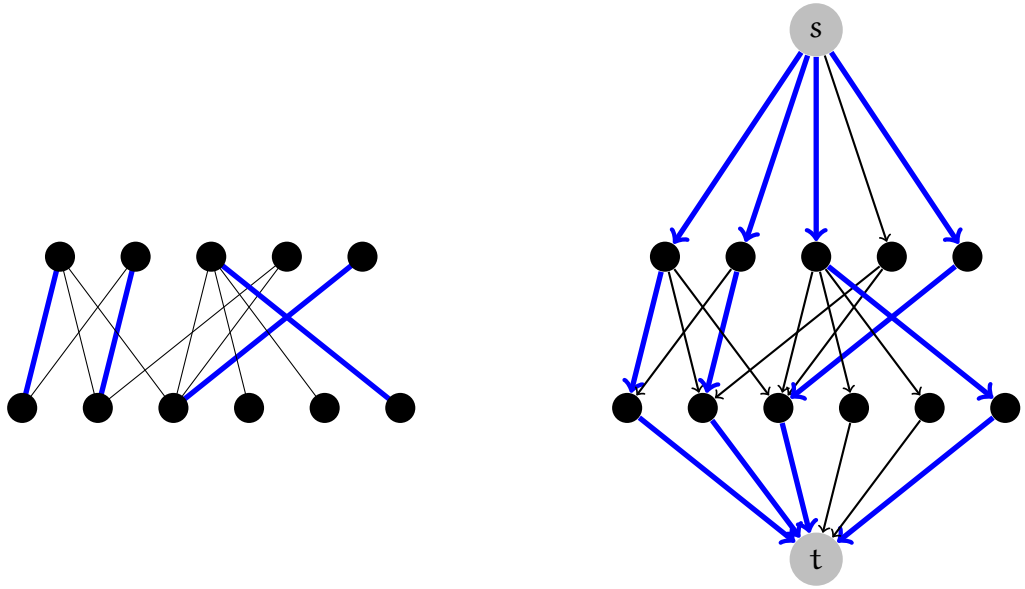
Dazu definieren wir ein Netzwerk $N = (V \dot\cup \{s, t\}, A, c, s, t)$. Die Knotenmenge besteht also aus den Knoten V von G und zwei neuen Knoten s und t, die die Rolle von Quelle bzw. Senke in N spielen. Die Kapazitätsfunktion c ist konstant 1. Die Kanten aus E werden übernommen, immer von U nach W gerichtet. Ausserdem hat s Kanten zu allen Knoten in U; zu t fügen wir Kanten von allen Knoten in W ein. Formal setzen wir

$$A := (\{s\} \times U) \quad \cup \quad \{(u, w) \in U \times W \,|\, \{u, w\} \in E\} \quad \cup \quad (W \times \{t\}) \,.$$

Wir betrachten nun einen ganzzahligen Fluss f auf diesem Netzwerk und beachten, dass alle Kapazitäten 1 sind. Für jeden Knoten u aus U gibt es nur eine eingehende Kante, also fliesst höchstens Fluss 1 in diesen Knoten und folglich kann aus u auch höchstens Fluss 1 zu Knoten aus W fliessen, d.h. höchstens eine ausgehende Kante kann positiven Fluss haben. Ähnlich kann in jeden Knoten aus W höchstens Fluss 1 fliessen, weil es nur eine ausgehende Kante (nach t) gibt. Wir haben uns gerade überzeugt, dass die Kanten mit Fluss 1 zwischen U und W ein Matching auf $U \cup W$



bilden müssen. Die Grösse dieses Matchings ist genau der Wert $\mathrm{val}(f)$ des Flusses.



Analog kann man aus einem Matching zwischen U und W mit k Kanten einen Fluss mit Wert k konstruieren. Es gilt also, da es in einem ganzzahligen Netzwerk immer einen ganzzahligen maximalen Fluss gibt:

Lemma 3.15. Die maximale Grösse eines Matchings im bipartiten Graph G ist gleich dem Wert eines maximalen Flusses im Netzwerk N.

Kanten- und knotendisjunkte Pfade

Im Abschnitt 1.4 haben wir gesehen, dass wir den Grad des Zusammenhangs eines Graphen (wieviele Knoten bzw Kanten wir entfernen müssen, bis wir einen unzusammenhängenden Graphen erhalten) mit Hilfe des Satzes von Menger auch dadurch bestimmen können, dass wir bestimmen, wie viele knoten- bzw kantendisjunkte Pfade es zwischen je zwei Knoten gibt. Damals hatten wir offen gelassen wie man diese Anzahl algorithmisch bestimmen kann. Jetzt, wo wir wissen, wie man maximale Flüsse in Netzwerken bestimmt, scheint das ein viel einfacheres Problem.

Betrachten wir einen (ungerichteten) Graphen $G = (V, E)$ und zwei Knoten $u, v \in V$, $u \neq v$. Wir wollen bestimmen wieviele intern kanten- (oder knoten-)disjunkte u-v-Pfade es in G gibt. Um dieses Problem mit einem Flussalgorithmus zu lösen, müssen wir zunächst aus dem Graphen G ein Netzwerk N machen. Und hier ergibt sich schon ein erstes Problem: unser Graph G ist ungerichtet, ein Netzwerk N enthält aber gerichteten Kanten. Die Lösung dieses Problems ist aber einfach: Wir ersetzen jede ungerichtete Kante $\{x, y\} \in E$ durch *zwei* gerichtete Kanten (x, y) und (y, x) und geben beiden Kanten eine Kapazität von Eins. Wie können wir sicherstellen, dass ein Fluss wirklich nur *eine* der beiden Kanten verwendet? Nun ja, a priori können wir das nicht. Aber das ist auch nicht schlimm: Stellen wir uns vor, für einen Fluss f würde gelten $f(x, y) = f(y, x) = 1$. Anschaulich gesprochen fliesst dann das, was aus x in Richtung y fliesst, anschliessend wieder zurück zu x. Wir können daher den Fluss f so zu einem Fluss f' abändern, dass wir $f'(x, y) = f'(y, x) = 0$ setzen und ihn auf allen übrigen Kanten unverändert lassen. Dieser Ansatz funktioniert analog, wenn $f(x, y)$ und $f(y, x)$ beide einen positiven Wert annehmen, der nicht unbedingt Eins ist: Erniedrigen wir den Fluss auf beiden Kanten um das Minimum aus $f(x, y)$ und $f(y, x)$ so ist danach einer der beiden Werte Null. Wir sehen daher: wir dürfen immer annehmen, dass ein Fluss höchstens auf einer der beiden Kanten einen positiven Wert annimmt.[11]

[11] Wir haben hier einfach den Fluss entlang eines (gerichteten) Kreises der Länge zwei reduziert. Dies funktioniert analog auch für gerichtete Kreise beliebiger Länge, in der

Als letztes fügen wir zu unserem Netzwerk noch zwei neue Knoten s und t hinzu – und verbinden s mit u und v mit t, d.h., wir fügen die beiden gerichteten Kanten (s, u) und (v, t) zu unserem Netzwerk hinzu. Diesen beiden Kanten geben wir eine Kapazität von $n = |V|$. Abbildung 3.4 zeigt diese Konstruktion an einem einfachen Beispiel.

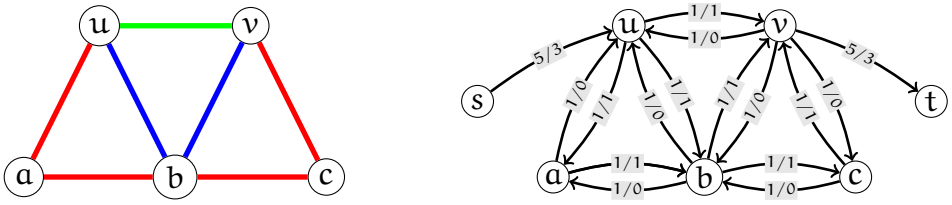


Abbildung 3.4: Links ein Graph $G = (V, E)$ mit drei intern kantendisjunkten u-v Pfaden (farbig). Rechts das daraus konstruierte Netzwerk $N = (V \dot\cup \{s, t\}, A, c, s, t)$; die Zahlen an den Kanten beschreiben wie üblich Kapazität (erste Zahl) und Fluss (zweite Zahl). Der Fluss entspricht genau den drei u-v-Pfaden und entsprechend ist der Wert des Flusses drei.

Aus der Konstruktion unseres Netzwerkes folgt unmittelbar, dass die Anzahl intern kantendisjunkter u-v-Pfade genau dem Wert eines maximalen Flusses entspricht, vgl. Abbildung 3.4. Wie aber sieht es aus, wenn wir nicht *kanten*disjunkte, sondern *knoten*disjunkte Pfade finden wollen? Hierfür ist unser Netzwerk in der jetzigen Form noch nicht geeignet. Aber auch hier hilft wieder ein einfacher Trick: Wir ersetzen jeden Knoten $x \in V\backslash\{u, v\}$ im Netzwerk durch *zwei* Knoten x_{in} und x_{out} und verbinden alle Eingangskanten von x mit x_{in}, alle Ausgangskanten von x mit x_{out} und fügen eine zusätzliche Kante von x_{in} zu x_{out} mit Kapazität Eins zum Netzwerk hinzu.

Abbildung 3.5 zeigt das resultierende Netzwerk für obiges Beispiel. Wieder überzeugt man sich leicht davon, dass der maximale Fluss in diesem Netzwerk genau der Anzahl intern knotendisjunkter u-v-Pfade entspricht.

Bildsegmentierung als Schnittproblem

Wir wollen ein Bild bestehend aus Pixeln (typischerweise in einem Gitter angeordnet) in Vordergrund und Hintergrund unterteilen—diese Operation

englischsprachigen Literatur nennt man diesen Ansatz ganz anschaulich *cycle canceling*. Cycle Canceling ist ein wichtiges Hilfsmittel, wenn man einen maximalen Fluss mit minimalen Kosten bestimmen wird, aber dies überlassen wir weiterführenden Vorlesungen.

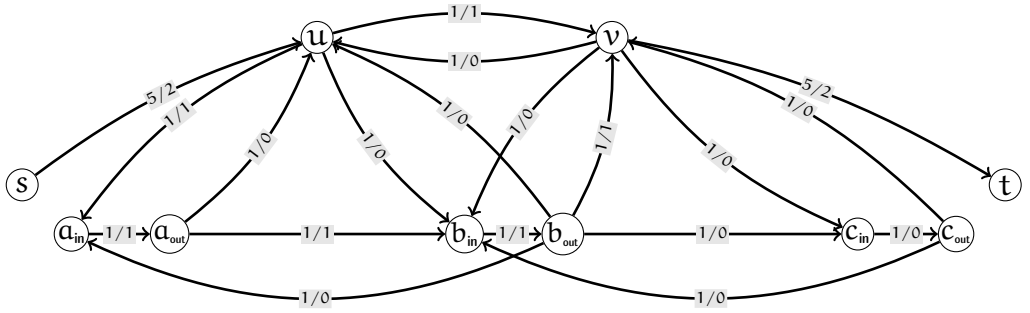


Abbildung 3.5: Das Netzwerk, um für den Graphen aus der vorigen Abbildung die maximale Anzahl *knoten*disjunkter u-v Pfade zu bestimmen. Der Fluss ist maximal und entspricht den beiden Pfaden u-v und u-a-b-v.

nennt man auch *Segmentierung*.

Für unsere Betrachtungen ist es wichtig, dass die Pixelmenge P mit einer Nachbarschaftsrelation assoziiert ist. In einem Gitter etwa ist jedes Pixel (ausser am Rand) zu vier anderen Pixeln benachbart (darüber, darunter, links und rechts). Dies ergibt eine Kantenmenge E und somit bildet (P, E) einen ungerichteten Graph.

Für uns ist es hier nicht wesentlich, ob die Pixel Graustufen oder Farbwerte darstellen. Was auch immer diese Werte sind, wir nehmen an, dass wir daraus für jedes Pixel p zwei nichtnegative Zahlen α_p und β_p extrahieren: Je grösser α_p ist, desto mehr erwarten wir, dass p zum Vordergrund gehört; analog drückt β_p unsere Erwartung aus, dass p im Hintergrund liegt. Mit diesen Zahlen scheint die Partition von P in Vordergrund A und Hintergrund B einfach:

$$A := \{p \in P \mid \alpha_p > \beta_p\} \quad \text{und} \quad B := P \setminus A \ .$$

Allerdings wollen wir tendenziell keine zu feinkörnige Unterteilung, d.h. liegen viele Nachbarn eines Pixels p im Vordergrund, so sollte p eher auch im Vordergrund liegen. Diese Präferenz modellieren wir, indem wir jeder Kante e in E eine nichtnegative Zahl γ_e zuordnen, mit der Interpretation, dass je grösser γ_e ist, wir umso mehr erwarten, dass die beiden beteiligten Pixel im gleichen Teil der Segementierung landen (γ_e kann man als Strafzoll betrachten, wenn man in der Unterteilung die Endpunkte trennt).

Mit diesen Zahlen bewerten wir nun eine Partition (A, B) mittels der

Qualitätsfunktion[12]

$$q(A,B) := \sum_{p\in A} \alpha_p + \sum_{p\in B} \beta_p - \sum_{e\in E,\,|e\cap A|=1} \gamma_e \ .$$

Uns interessiert hier nicht so sehr, wie man die Parameter α, β, und γ günstig wählt, so dass man gute Ergebnisse bekommt. Stattdessen fragen wir uns, wie wir eine Unterteilung von P in A und B finden, sodass $q(A,B)$ maximal ist.

Wir merken, dass die Fragestellung zumindest in einigen Aspekten einem Schnittproblem gleicht. Tatsächlich gelingt es uns wieder, wie beim Matchingproblem, die Fragestellung in ein Netzwerkproblem (hier minimaler Schnitt) überzuführen.

Zuerst ändern wir die Qualitätsfunktion, sodass wir es—wie bei s-t-Schnitten gewohnt—mit einem Minimierungsproblem zu tun haben. Dazu definieren wir $Q := \sum_{p\in P}(\alpha_p + \beta_p)$ womit wir $q(A,B)$ in

$$q(A,B) = Q - \sum_{p\in A} \beta_p - \sum_{p\in B} \alpha_p - \sum_{e\in E,|e\cap A|=1} \gamma_e$$

umschreiben können. Daraus wird klar, dass $q(A,B)$ zu maximieren äquivalent zur Minimierung von

$$q'(A,B) := \sum_{p\in A} \beta_p + \sum_{p\in B} \alpha_p + \sum_{e\in E,|e\cap A|=1} \gamma_e$$

ist.

In einem nächsten Schritt erzeugen wir ein Netzwerk wie folgt. Wieder führen wir neue Knoten s und t ein, die für die Quelle und Senke im Netzwerk stehen. s hat gerichtete Kanten zu allen Pixeln in P, die Kapazität der Kante zu p sei α_p. Analog hat dann jedes Pixel p eine gerichtete Kante zu t mit Kapazität β_p. Schliesslich ersetzen wir jede ungerichtete Kante $e = \{p,p'\}$ in E durch zwei gerichtete Kanten (p,p') und (p',p), je mit Kapazität γ_e. Ein Netzwerk N mit Knotenmenge $P\cup\{s,t\}$ ist geschaffen.

Was ist nun die Kapazität eines s-t-Schnitts (S,T) in N? Dazu sehen wir uns an, welche Kanten des Netzwerks in $S\times T$ liegen, wobei wir $A := S\setminus\{s\}$ und $B := T\setminus\{t\}$ verwenden.

[12]Nochmals, wir kümmern uns hier nicht um berechtigten Fragen, wo die Werte α_p, β_p, und γ_e herkommen, und auch nicht darum, ob die Qualitätsfunktion gut ist und den Zweck erfüllt. Tatsächlich ist das aber ein Ansatz, der in der Praxis so zum Einsatz kommt.

- Kanten (s, p) mit $p \in B$. Ihr Beitrag zur Kapazität $\mathrm{cap}(S, T)$ ist $\sum_{p \in B} \alpha_p$.
- Kanten (p, t) mit $p \in A$, die $\sum_{p \in A} \beta_p$ zur Kapazität des Schnitts beitragen.
- Kanten (p, p') des Netzwerks in $A \times B$ mit Beitrag $\sum_{(p,p')} \gamma_{(p,p')}$, wobei die Summe über alle Kanten des Netzwerks aus $A \times B$ geht.

Wir sehen, dass für die aus (S, T) abgeleitete Unterteilung (A, B) von P tatsächlich

$$q'(A, B) = \mathrm{cap}(S, T)$$

gilt. Eine optimale Segmentierung entspricht somit einem minimalen s-t-Schnitt im generierten Netzwerk N und kann so mit einem Fluss- oder Schnittalgorithmus für Netzwerke berechnet werden.

Eine Auswahl weiterer Anwendungen findet man unter anderem in *Algorithm Design* von Jon Kleinberg und Éva Tardos (Pearson – Addison Wesley).

Flüsse und konvexe Mengen

Wir schliessen diesen Abschnitt ab, indem wir versuchen die Menge aller Flüsse in einem Netzwerk besser zu verstehen, und einen Bezug zu einem zentralen Begriff der Geometrie herstellen: *konvexe Mengen*.

So wie wir Netzwerke und Flüsse eingeführt haben, hat jedes Netzwerk mindestens einen Fluss: die Funktion konstant 0; bezeichnen wir diesen Fluss mit $\mathbf{0}$. Nehmen wir an, es gibt mindestens einen zweiten Fluss (ungleich $\mathbf{0}$), nennen wir den f_1. Aber nun gibt es sofort einen weiteren Fluss f_1^* indem wir den Fluss f_1 einfach halbieren, d.h.

$$\forall e \in A : f_1^*(e) := \frac{1}{2} f_1(e) \ .$$

Dazu müssen wir sicherstellen, dass die Bedingungen für einen Fluss erfüllt sind: $0 \leq f_1^*(e) \leq c(e)$ für alle Kanten $e \in A$ und die Flusserhaltung – beides lässt sich leicht zeigen. Allgemeiner gilt

Lemma 3.16. Sind f_0 und f_1 Flüsse in einem Netzwerk N und $\lambda \in \mathbb{R}$, $0 < \lambda < 1$, dann ist der Fluss f_λ, definiert durch

$$\forall e \in A : f_\lambda(e) := (1-\lambda)f_0(e) + \lambda f_1(e) ,$$

ebenfalls ein Fluss in N. Es gilt

$$\mathsf{val}(f_\lambda) = (1-\lambda) \cdot \mathsf{val}(f_0) + \lambda \cdot \mathsf{val}(f_1) .$$

Beweis. Wir können aus der Zulässigkeit von f_0 und f_1 leicht die Zulässigkeit von f_λ herleiten (man beachte, dass $\lambda \geq 0$ und $1-\lambda \geq 0$):

$$\begin{aligned} f_\lambda(e) &= (1-\lambda)\overbrace{f_0(e)}^{\geq 0} + \lambda \overbrace{f_1(e)}^{\geq 0} \geq (1-\lambda)\cdot 0 + \lambda \cdot 0 = 0 \\ f_\lambda(e) &= (1-\lambda)\underbrace{f_0(e)}_{\leq c(e)} + \lambda \underbrace{f_1(e)}_{\leq c(e)} \leq (1-\lambda)\cdot c(e) + \lambda \cdot c(e) = c(e) \end{aligned}$$

Ähnlich einfach lassen sich Flusserhaltung und die Behauptung zum Wert des Flusses verifizieren. □

Korollar 3.17. Es gilt:

(i) Ein Netzwerk N hat entweder genau einen Fluss (den Fluss $\mathbf{0}$) oder unendlich viele Flüsse.

(ii) Ein Netzwerk hat entweder genau einen maximalen Fluss, oder unendlich viele maximale Flüsse.

Konvexe Mengen. Für eine geometrische Interpretation wählen wir eine Numerierung (Ordnung) $(e_1, e_2, \ldots, e_m)$, $m := |A|$, der Kanten in A. Jede Funktion $f\colon A \to \mathbb{R}$ induziert so einen Vektor $v_f := (f(e_1), f(e_2), \ldots, f(e_m))$ in $\mathbb{R}^m$. Das heisst, die Menge der Flüsse (oder die Menge der maximalen Flüsse) kann man so als Teilmengen des $\mathbb{R}^m$ interpretieren.

Definition 3.18. Sei $m \in \mathbb{N}$.

(i) Für $v_0, v_1 \in \mathbb{R}^m$ sei $\overline{v_0 v_1} := \{(1-\lambda)v_0 + \lambda v_1 \mid \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1\}$,

das v_0 und v_1 verbindende *Liniensegment*.

(ii) Eine Menge $C \subseteq \mathbb{R}^m$ heisst *konvex*, falls für alle $v_0, v_1 \in C$ das ganze Liniensegement $\overline{v_0 v_1}$ in C enthalten ist.

Beispiele konvexer Mengen sind Kugeln oder konvexe Polytope (z.B. Tetraeder oder Würfel im $\mathbb{R}^3$). Konvexe Polygone (im $\mathbb{R}^2$) wurden in der Vorlesung Diskrete Mathematik erwähnt, konvexe Mengen in der Analysis (oder auch kurz in der Linearen Algebra). Aus Lemma 3.16 folgt jetzt unmittelbar folgende Eigenschaft.

Satz 3.19. Die Menge der Flüsse eines Netzwerks mit m Kanten, interpretiert als Vektoren, ist eine konvexe Teilmenge des $\mathbb{R}^m$. Ebenso bildet die Menge der maximalen Flüsse eine konvexe Teilmenge des $\mathbb{R}^m$.

Man kann zeigen, dass diese Flüssen entsprechenden Mengen konvexe Polytope sind, wir gehen darauf nicht weiter ein. Die Tatsache, dass es immer einen maximalen Fluss gibt, folgt leicht aus der Theorie konvexer Mengen (dazu muss man zeigen, dass die Menge der Flüsse eine konvexe kompakte Menge bildet).

Der Bezug (oder besser die Äquivalenz) von Funktionen und Vektoren ist wichtig, und erlaubt die Anwendung von Konzepten und Ergebnissen aus der Geometrie und Linearen Algebra auf Mengen von Funktionen.

Als Illustration ist empfohlen sich die Menge der Flüsse für das einfache Netzwerk $N = (\{s, t, q\}, \{e_1, e_2\}, c, s, t)$ mit $e_1 = (s, q)$, $e_2 = (q, t)$, $c(e_1) = 1$ und $c(e_2) = 2$ im $\mathbb{R}^2$ zu veranschaulichen (jede Funktion $f : \{e_1, e_2\} \to \mathbb{R}$ wird als Vektor $(f(e_1), f(e_2)) \in \mathbb{R}^2$ interpretiert). Dazu zeichne man erst das Rechteck aller zulässigen Funktionen/Vektoren f mit $0 \leq f(e_1) \leq 1$ und $0 \leq f(e_2) \leq 2$, und schneide dieses Rechteck mit der Geraden $f(e_1) = f(e_2)$ (die in diesem einfachen Beispiel einzige Flusserhaltungsbedingung). Man sieht dann leicht, dass die Menge der Flüsse ein Liniensegment ist.

3.1.3 Minimale Schnitte in Graphen

In diesem Abschnitt betrachten wir ungerichtete ungewichtete Graphen $G = (V, E)$ ohne Schleifen, wobei wir mehrere Kanten zwischen demselben Knotenpaar erlauben – sogenannte *Multigraphen*. Wir könnten statt

Mehrfachkanten auch ganzzahlige Kantengewichte erlauben, aber die Abhandlung wird mit Mehrfachkanten einfacher.

Wir interessieren uns in einem solchen Multigraph $G = (V, E)$ für eine kleinste Menge C von Kanten, deren Entfernung einen unzusammenhängenden Multigraph erzeugt. Ist G selber nicht zusammenhängend, so kann man hier offensichtlich $C = \emptyset$ wählen. Eine Menge $C \subseteq E$ für die $(V, E \setminus C)$ unzusammenhängend ist, nennen wir *Kantenschnitt* in G. Mit $\mu(G)$ bezeichnen wir die Kardinalität eines kleinstmöglichen Kantenschnitts in G.

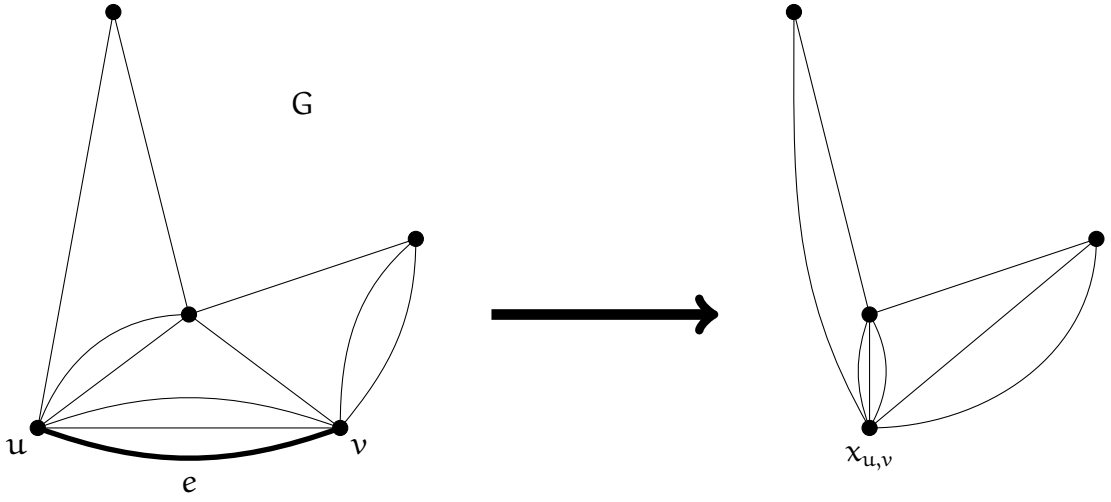
Das Problem. Gegeben ein Multigraph G, bestimme $\mu(G)$ (die Kardinalität eines *minimalen Schnitts*). Wir nennen das das *MIN-CUT Problem*.

Ein ähnliches Problem hatten wir ja gerade bei den Flüssen kennengelernt. Wo liegt der Unterschied oder haben wir das Problem schon gelöst? Wir skizzieren den Zusammenhang: Statt Mehrfachkanten könnten wir, wie erwähnt, einfach Kantengewichte verwenden. Ungerichtete Kanten $\{u, v\}$ können wir durch Paare gerichteter Kanten (u, v), (v, u) ersetzen. Und dann hatten wir es im letzten Kapitel mit s-t-Schnitten zu tun, hier jetzt sind s und t nicht spezifiziert. Um jetzt MIN-CUT für einen Multigraph $G = (V, E)$ zu lösen, transformieren wir ihn wie beschrieben zu einem gerichteten Graph (V, A) (mit Kanten-Gewichtsfunktion w), fixieren einen Knoten $s \in V$ und berechnen minimale s-t-Schnitte, für alle $t \in V \setminus \{s\}$, im Netzwerk (V, A, w, s, t). Der kleinste dieser s-t-Schnitte zeigt uns den minimalen Schnitt im Ausgangs-Multigraph G. Wir hatten erwähnt, dass man minimale s-t-Schnitte in $O(nm \log n) = O(n^3 \log n)$ Zeit berechnen kann. Das macht dann insgesamt $O(n^4 \log n)$ für unser MIN-CUT Problem (weil wir $n - 1$ s-t-Schnitte berechnen müssen).

Kantenkontraktion

Sei G ein Multigraph und $e = \{u, v\}$ eine Kante in G. Die *Kontraktion von* e verschmilzt die beiden Knoten u und v zu einem neuen Knoten $x_{u,v}$, der nun zu allen Kanten inzident ist, zu denen u oder v inzident war, ausser den Kanten zwischen u und v – diese Kanten verschwinden. Den entstehenden Graph bezeichnen wir mit G/e. Ist k die Anzahl der Kanten zwischen u und v, dann gilt

$$\deg_{G/e} x_{u,v} = \deg_G(u) + \deg_G(v) - 2k \ .$$



(Wichtig ist hier, dass wir in einem Multigraphen G mit $\deg_G(u)$ die Anzahl der zu u inzidenten Kanten bezeichnen und nicht die Anzahl der Nachbarn.)

Es gibt eine natürliche Bijektion

$$\{\text{Kanten in } G \text{ ausser denen zwischen } u \text{ und } v\} \longleftrightarrow \{\text{Kanten in } G/e\}:$$

Entweder bleibt eine Kante in G bei der Kontraktion gleich, oder $\{w,u\}$ oder $\{w,v\}$ wird zu einer Kante $\{w,x_{u,v}\}$. Man beachte, dass es so für jeden Kantenschnitt C in G/e einen entsprechenden Kantenschnitt in G mit gleicher Kardinalität gibt. Umgekehrt gibt es für jeden Kantenschnitt *ohne* e in G einen entsprechenden in G/e, wieder mit gleicher Kardinalität. Es ergibt sich folgendes Lemma.

Lemma 3.20. Sei G ein Graph und e eine Kante in G. Dann gilt $\mu(G/e) \geq \mu(G)$ und falls es in G einen minimalen Schnitt C mit $e \notin C$ gibt, dann gilt $\mu(G/e) = \mu(G)$.

Auf gut Glück – zufällige Kantenkontraktionen

Wir können nun den einfachen randomisierten Algorithmus dieses Abschnitts beschreiben.

CUT(G)	G zusammenhängender Multigraph

1: **while** $|V(G)| > 2$ **do**
2: $e \leftarrow$ gleichverteilt zufällige Kante in G
3: $G \leftarrow G/e$
4: **return** Grösse des eindeutigen Schnitts in G

Sei n Anzahl der Konten in G. Für die Implementierung setzen wir folgendes voraus:

- Eine Kantenkontraktion kann in $O(n)$ Zeit durchgeführt werden.
- Eine gleichverteilt zufällige Kante in G kann in $O(n)$ gewählt werden.

(Das ist nicht ganz offensichtlich, und erfordert u.a., dass Mehrfachkanten mittels Kantengewichten dargestellt werden.) Mit dieser Voraussetzung können wir CUT(G) mit einer Laufzeit von $O(n^2)$ implementieren.

Als nächstes stellen wir sicher, dass eine zufällige Wahl von e mit guter Wahrscheinlichkeit die Grösse des minimalen Schnitts nicht verändert.

Lemma 3.21. Sei $G = (V, E)$ ein Multigraph mit n Knoten. Falls e gleichverteilt zufällig unter den Kanten in G gewählt wird, dann gilt

$$\Pr[\mu(G) = \mu(G/e)] \geq 1 - \frac{2}{n} .$$

Beweis. Sei C ein minimaler Schnitt in G und sei $k := |C| = \mu(G)$. Sicherlich ist der Grad jedes Knotens in G mindestens k, da die zu einem Knoten inzidenten Kanten immer einen Schnitt bilden. Es gilt daher $|E| = \frac{1}{2}\sum_{v \in V} \deg(v) \geq \frac{kn}{2}$. Wir erinnern uns an $e \notin C \Rightarrow \mu(G/e) = \mu(G)$ und somit

$$\Pr[\mu(G) = \mu(G/e)] \geq \Pr[e \notin C] = 1 - \frac{|C|}{|E|} \geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n},$$

was zu zeigen war. □

Nun interessieren wir uns dafür, dass der Algorithmus CUT(G) insgesamt den richtigen Wert ausgibt, d.h.

$\hat{p}(G) :=$ Wahrscheinlichkeit, dass CUT(G) den Wert $\mu(G)$ ausgibt

und

$$\hat{p}(n) := \inf_{G=(V,E),\,|V|=n} \hat{p}(G) \ .$$

Man beachte, dass $\hat{p}(2) = 1$.

Lemma 3.22. Es gilt für alle $n \geq 3$

$$\hat{p}(n) \geq (1 - 2/n) \cdot \hat{p}(n-1).$$

Beweis. Sei G ein Multigraph mit n Knoten. Damit CUT(G) tatsächlich $\mu(G)$ ausgibt, müssen die beiden folgenden Ereignisse eintreten:

- E_1 := Ereignis $\mu(G) = \mu(G/e)$.
- E_2 := Ereignis, dass CUT(G/e) den Wert $\mu(G/e)$ ausgibt.

Es gilt nun

$$\hat{p}(G) = \Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2 \mid E_1] \geq (1 - 2/n) \cdot \hat{p}(n-1)$$

und da dies für jeden Multigraph G mit n Knoten gilt, folgt auch die Aussage $\hat{p}(n) \geq (1 - 2/n) \cdot \hat{p}(n-1)$. □

Wir erhalten so

$$\hat{p}(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \cdot \underbrace{\hat{p}(2)}_{=1} = \frac{2}{n(n-1)}$$

Lemma 3.23. Es gilt $\hat{p}(n) \geq \frac{2}{n(n-1)} = 1/\binom{n}{2}$ für alle $n \geq 2$.

Wir wiederholen nun den Algorithmus CUT(G) $\lambda\binom{n}{2}$ mal, für ein $\lambda > 0$, und geben dann den kleinsten je erhaltenen Wert aus. Dann erhalten wir folgendes Ergebnis (wir erinnern uns, dass ein Aufruf von CUT(G) Zeit $O(n^2)$ benötigt).

Satz 3.24. Für den Algorithmus der $\lambda\binom{n}{2}$-maligen Wiederholung von CUT(G) gilt:

(1) Der Algorithmus hat eine Laufzeit von $O(\lambda n^4)$.

(2) Der kleinste angetroffene Wert ist mit einer Wahrscheinlichkeit von mindestens $1 - e^{-\lambda}$ gleich $\mu(G)$.

Für die Fehlerabschätzung gehen wir nach dem bekannten Muster vor: Die Wahrscheinlichkeit, dass wir $\lambda\binom{n}{2}$ Mal nicht einen minimalen Schnitt finden ist höchstens

$$(1 - \hat{p}(n))^{\lambda\binom{n}{2}} \leq \left(1 - 1/\binom{n}{2}\right)^{\lambda\binom{n}{2}} \leq e^{-\lambda}$$

(mit der bewährten Ungleichung $\forall x \in \mathbb{R} : 1 + x \leq e^x$).

Wählen wir $\lambda = \ln n$, gibt das eine Laufzeit von $O(n^4 \log n)$ mit Fehlerwahrscheinlichkeit höchstens $1/n$. Diese Laufzeit haben wir aber mittels Flussalgorithmen auch schon deterministisch und ohne Fehler erreicht. Aber der Ansatz hat noch Potential, wie wir gleich sehen werden.

Bootstrapping Sollte man den Algorithmus implementieren, wird man schnell folgende Beobachtung machen. Hat unser Multigraph (im Zuge der Rekursion) nur mehr 3 Knoten und machen wir eine zufällige Kantenkontraktion, dann haben wir nur mehr eine Erfolgswahrscheinlichkeit von $1/3$. Dabei können wir doch ganz einfach den minimalen Schnitt schnell bestimmen, es besteht gar kein Grund mehr im letzten Schritt noch dieses Risiko einzugehen. Das könnten wir eigentlich schon bei 4 Knoten machen, etc.

Was passiert, wenn wir die Rekursion in $\textsc{Cut}(G)$ abbrechen, sobald wir bei einem Multigraph mit t Knoten angelangt sind (für einen Parameter t, den wir noch bestimmen), um dann das Problem für diesen kleineren Multigraph mit einem anderen Algorithmus in Zeit $z(t)$ und Erfolgswahrscheinlichkeit $\geq p^*(t)$ zu lösen: z.B. deterministisch mittels Flüssen in $z(t) = O(t^4 \log t)$ Zeit mit $p^*(t) = 1$, oder nach Satz 3.24 (mit $\lambda = 1$) in $z(t) = O(t^4)$ Zeit mit $p^*(t) = 1 - e^{-1}$).

$\textsc{Cut1}(G)$	G zusammenhängender Multigraph
1: **while** $\lvert V(G)\rvert > t$ **do**	
2: $\quad e \leftarrow$ gleichverteilt zufällige Kante in G	
3: $\quad G \leftarrow G/e$	
4: **return** Grösse des eindeutigen Schnitts in G	$\triangleright$ in Zeit $O(z(t))$

Dieser Algorithmus benötigt nun Zeit $O(n(n-t)+z(t))$, er macht Fehler in den ersten $n-t$ Schritten, und schliesslich noch einen Fehler mit Wahrscheinlichkeit $\leq 1-p^*(t)$ im letzten Schritt (auf dem Multigraph der Grösse t), d.h. die Erfolgswahrscheinlichkeit ist

$$\geq \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{t+1}{t+3}\cdot\frac{t}{t+2}\cdot\frac{t-1}{t+1}\cdot p^*(t)=\frac{t(t-1)}{n(n-1)}\cdot p^*(t)\ .$$

Legen wir uns (vorerst) auf $z(t)=O(t^4)$ und $p^*(t)=1-e^{-1}=\frac{e-1}{e}$ fest. Wir müssen den revidierten Algorithmus CUT1(G) nun $\lambda\frac{n(n-1)}{t(t-1)}\frac{e}{e-1}$ Mal wiederholen, um eine Erfolgswahrscheinlichkeit von $1-e^{-\lambda}$ zu erreichen. Die Laufzeit ist dann

$$\lambda\frac{n(n-1)}{t(t-1)}\frac{e}{e-1}\cdot O(n(n-t)+t^4)=O\left(\lambda\left(\frac{n^4}{t^2}+n^2t^2\right)\right)\ .$$

Wählen wir t klein, dann dominiert $\frac{n^4}{t^2}$, bei t gross, dominiert n^2t^2. Wir entscheiden uns für $t=n^{1/2}$ (sodass sich die beiden Terme ausbalancieren) und erhalten so eine Laufzeit von $O(\lambda n^3)$, was nun eine Verbesserung gegenüber dem ursprünglichen Flussansatz ergibt. Das würde an dieser Stelle einen Satz rechtfertigen, könnten wir diesen schnelleren Algorithmus nicht gleich wieder verwenden (uns steht jetzt $z(t)=O(t^3)$ zur Verfügung), um das Problem für den Multigraphen mit t Knoten zu lösen. Das führt dann zu einer anderen Wahl von t und einem noch schnelleren Algorithmus,[13] den wir natürlich gleich verwenden, um den Algorithmus wieder schneller zu machen, ... Dieses in sich selbst einsetzen eines Algorithmus, um diesen immer schneller zu machen, nennt man *Bootstrapping*.

Wo führt das hin? Es konvergiert zu einem Verfahren mit einer Laufzeit von $O(n^2\text{poly}(\log n))$, man kann sich das wie einen „Grenzwertalgorithmus“ vorstellen. Natürlich muss man eine sorgfältige Buchhaltung über die Fehlerwahrscheinlichkeit machen. Wir verweisen für diese weiteren Schritte auf die Literatur.

3.2 Geometrische Algorithmen

3.2.1 Kleinster umschliessender Kreis

In diesem Abschnitt betrachten wir folgendes Problem. Zu einer gegebenen Menge P von n Punkten in der Ebene möchten wir einen Kreis $C(P)$ be-

[13] Bestimmen Sie als Übung einen guten Parameter t und die resultierende Laufzeit.

stimmen, sodass $C(P)$ alle Punkte aus P umschliesst und der *Radius* von $C(P)$ so klein wie möglich ist. Für einen Kreis C verwenden wir $C^\bullet$ für die von C umschlossene Kreisscheibe (abgeschlossen, also inklusive C).

Hierbei erlauben wir, dass die Punkte in P auch auf $C(P)$ liegen dürfen – die Punkte müssen also nicht *strikt* innerhalb des Kreises liegen. Dann ist es möglich zu zeigen, dass es immer einen kleinsten Kreis gibt, der alle Punkte enthält. Bevor wir uns der Entwicklung eines Algorithmus für dieses Problem zuwenden, leiten wir zunächst zwei Eigenschaften eines solchen kleinsten umschliessenden Kreises her. Zunächst zeigen wir, dass der kleinste umschliessende Kreis eindeutig bestimmt ist, es also insbesondere Sinn macht von *dem* Kreis $C(P)$ zu sprechen.

Lemma 3.25. Für jede (endliche) Punktemenge P im $\mathbb{R}^2$ gibt es einen eindeutigen kleinsten umschliessenden Kreis $C(P)$.

Beweis. Den Beweis für die Existenz überspringen wir. Wir zeigen die Eindeutigkeit durch einen Widerspruchsbeweis. Nehmen wir also an, es gäbe eine Menge P, für die es (mindestens) zwei verschiedene kleinste umschliessende Kreise C_1 und C_2 gibt. Da C_1 und C_2 beides kleinste Kreise sind, haben sie den gleichen Radius, sagen wir r. Und da C_1 und C_2 verschieden sind, müssen sie verschiedene Mittelpunkte z_1 und z_2 haben. Da beide Kreise die Punktemenge P enthalten, gilt auch: $P \subseteq C_1^\bullet \cap C_2^\bullet$. Wir konstruieren nun einen neuen Kreis C wie folgt: als Mittelpunkt nehmen wir den Mittelpunkt z der Strecke von z_1 zu z_2 ($z = \frac{1}{2}(z_1 + z_2)$). Als Radius $\hat{r}$ wählen wir den Abstand von z zu einem (der beiden) Schnittpunkte von C_1 und C_1. (Man überlegt sich leicht, dass z zu beiden Schnittpunkten den gleichen Abstand hat.) Dann gilt $P \subseteq C_1^\bullet \cap C_2^\bullet \subseteq C^\bullet$ und $\hat{r} = \sqrt{r^2 - (\frac{1}{2}|z_1 z_2|)^2}$, wobei $|z_1 z_2|$ die Länge der Strecke von z_1 nach z_2 ist. Insbesondere gilt also $\hat{r} < r$, im Widerspruch zu unserer Annahme, dass C_1 und C_2 kleinste umschliessende Kreise sind. □

Das nächste Lemma wird die Basis unseres Algorithmus werden. Es zeigt, dass es immer drei Punkte aus P gibt, die den kleinsten umschliessenden Kreis bereits eindeutig bestimmen.

Lemma 3.26. Für jede (endliche) Punktemenge P im $\mathbb{R}^2$ mit $|P| \geq 3$ gibt es eine Teilmenge $Q \subseteq P$, so dass $|Q| = 3$ und $C(Q) = C(P)$.

Beweis. Sei P eine beliebige Punktemenge mit mindestens drei Punkten. Aus Lemma 3.25 wissen wir, dass $C := C(P)$ eindeutig ist. Mit B bezeichnen wir die Menge derjenigen Punkte aus P, die auf dem *Rand* von C liegen. Wir zeigen zunächst folgende Hilfseigenschaft:

($\star$) Ist g eine beliebige Gerade durch den Mittelpunkt von C, so kann es nicht sein, dass alle Punkte von B (strikt) auf einer der beiden Seiten von g liegen.

Dies gilt, da wir ansonsten den Mittelpunkt von C ganz leicht senkrecht zu g *weg* von der Seite ohne Punkte auf dem Rand verschieben könnten und gleichzeitig den Radius etwas verkleinern, so dass dennoch noch immer alle Punkte aus P innerhalb des nun kleineren Kreises liegen, im Widerspruch zu unserer Annahme.

Aus dieser Beobachtung folgt bereits, dass $|B| \geq 2$ gelten muss, und dass im Falle $|B| \leq 3$ der Kreis C durch $C(B)$ gegeben ist. Falls $|B| \geq 4$ so wählen wir zwei beliebige Punkte p und q von B, die bei Durchlaufen der Kreislinie von C *nicht* unmittelbar aufeinander folgen. Wenn sie auf einer Geraden durch den Mittelpunkt liegen, so gilt $C = C(\{p, q\})$ und das Lemma ist bewiesen. Falls nicht, so wählen wir einen beliebigen Punkt r aus B aus dem *kürzeren* der beiden durch p und q gegebenen Kreissegmente. Dann gilt die Eigenschaft ($\star$) auch noch für die Menge $B \setminus \{r\}$ – und das Lemma folgt damit durch Induktion über $|B|$. □

Aus Lemma 3.26 können wir sofort einen $O(n^4)$ Algorithmus für die Berechnung von $C(P)$ ableiten:

CompleteEnumeration(P)

```
1: for all Q ⊆ P mit |Q| = 3 do
2:     bestimme C(Q)
3:     if P ⊆ C•(Q) then
4:         return C(Q)
```

Die Laufzeit sieht man wie folgt ein: wir müssen höchstens $\binom{n}{3}$ viele Mengen Q testen. Für jede solche Menge können wir $C(Q)$ in konstanter

Zeit bestimmen (da Q ja nur drei Punkte enthält) und müssen dann für jeden Punkt in P testen, ob er in $C^\bullet(Q)$ enthalten ist. Für jeden Punkt geht dies in konstanter Zeit, jede Iteration der for-Schleife benötigt daher $O(n)$ Zeit.

Unser Ziel ist ein *randomisierter* Algorithmus mit (erwarteter) Laufzeit $O(n \ln n)$.

Dazu betrachten wir als erstes eine randomisierte Version von obigem Algorithmus:

RANDOMISED_PRIMITIVEVERSION(P)

```
1: repeat forever
2:     wähle Q ⊆ P mit |Q| = 3 zufällig und gleichverteilt
3:     bestimme C(Q)
4:     if P ⊆ C•(Q) then
5:         return C(Q)
```

Kurzes Nachdenken zeigt uns, dass das noch keine sehr clevere Idee ist: Für Punktemengen P, in denen der kleinste einschliessende Kreis durch genau drei Punkte gegeben ist, müssen wir die repeat-Schleife ausführen, bis wir *genau diese* dreielementige Menge ziehen. Dies geschieht in jeder Iteration mit Wahrscheinlichkeit $1/\binom{n}{3}$. Die Anzahl Iterationen ist daher geometrisch verteilt mit Parameter $1/\binom{n}{3}$ – und die erwartete Anzahl Iterationen ist somit $\binom{n}{3}$. Die erwartete Laufzeit unseres randomisierten Algorithmus ist daher ebenfalls $O(n^4)$.

Was können wir tun, um den Algorithmus zu beschleunigen? Eine erste Idee ist, in jeder Runde mehr als 3 Punkte zu ziehen. Wir könnten beispielsweise 11 Punkte ziehen. Dann können wir $C(Q)$ noch immer in konstanter Zeit berechnen, aber wir haben eine höhere Chance, dass die Menge Q die drei definierenden Punkte von $C(P)$ enthält. Eine kurze Rechnung zeigt dann allerdings schnell, dass dies an der asymptotischen Laufzeit von $O(n^4)$ nichts ändert.

Um die Laufzeit wirklich zu verbessern, müssen wir noch eine zusätzliche Idee hinzufügen: wir verdoppeln in jeder Iteration die Punkte ausserhalb von $C(Q)$:

RANDOMISED_CLEVERVERSION(P)

1: **repeat forever**
2: wähle $Q \subseteq P$ mit $|Q| = 11$ zufällig und gleichverteilt
3: bestimme $C(Q)$
4: **if** $P \subseteq C^{\bullet}(Q)$ **then**
5: **return** $C(Q)$
6: verdoppele alle Punkte von P ausserhalb von $C(Q)$

Für die Implementierung dieses Algorithmus müssen wir uns zunächst überlegen, wie wir die "Verdopplung" der Punkte realisieren. Dies ist recht einfach: Wir initialisieren ein Array num der Länge n mit konstant Eins. Die Idee ist, dass num[i] die Anzahl Kopien des i-ten Punktes angibt. Liegt der i-te Punkt dann ausserhalb von $C(Q)$, so können wir die Verdopplung aller Kopien dieses Punktes einfach dadurch realisieren, indem wir num[i] durch $2 \cdot$ num[i] ersetzen.

Als nächstes müssen wir uns noch überlegen, wie wir in Zeit $O(n)$ die 11 Punkte $q_1, \ldots, q_{11} \in P$ zufällig und gleichverteilt wählen können. Hierfür zeigen wir folgendes Lemma:

Lemma 3.27. Seien $n_1, \ldots, n_t$ natürliche Zahlen und $N := \sum_{i=1}^{t} n_i$. Wir erzeugen $X \in \{1, \ldots, t\}$ zufällig wie folgt:

$k \leftarrow$ UNIFORMINT$(1, N)$
$x \leftarrow 1$
while $\sum_{i=1}^{x} n_i < k$ **do**
$x \leftarrow x + 1$
return x

Dann gilt $\Pr[X = i] = n_i/N$ für alle $i = 1, \ldots, t$.

Beweis. Der Beweis folgt sofort aus der Tatsache, dass $x = i$ genau für $k \in \{1 + \sum_{i=1}^{x-1} n_i, \ldots, \sum_{i=1}^{x} n_i\}$ ausgegeben wird, also für n_i der möglichen N Werte für k. □

Mit Hilfe dieses Lemmas und des Arrays num[·] können wir somit Punkte der Menge P so ziehen, dass die Wahrscheinlichkeit eines jeden Punktes proportional zu der Anzahl seiner Kopien ist. Wählen wir dann den nächsten Punkt bezüglich eines leicht korrigierten Arrays num[·] (in dem wir berücksichtigen, dass wir einen Punkt schon gezogen haben), so erhalten

wir auf diese Weise in Zeit $O(n)$ die gewünschte Teilmenge Q, die aus genau 11 Punkten besteht.

Für die Analyse des Algorithmus benötigen wir jetzt noch das folgende Lemma:

Lemma 3.28. Sei P eine Menge von n (nicht unbedingt verschiedenen) Punkten und für $r \in \mathbb{N}$, R zufällig gleichverteilt aus $\binom{P}{r}$. Dann ist die erwartete Anzahl Punkte von P, die ausserhalb von $C(R)$ liegen, höchstens $3\frac{n-r}{r+1} \leq 3\frac{n}{r+1}$.

Beweis. Für den Beweis definieren wir uns zwei Hilfsfunktionen, für $p \in P$, $R, Q \subseteq P$:

$$\mathrm{out}(p, R) := \begin{cases} 1 & \text{falls } p \notin C^\bullet(R) \\ 0 & \text{sonst} \end{cases}$$

(beachte, dass $\sum_{p \in P \setminus R} \mathrm{out}(p, R)$ die Anzahl der Punkte ausserhalb $C(R)$ ist) und

$$\mathit{essential}(p, Q) := \begin{cases} 1 & \text{falls } C(Q \setminus \{p\}) \neq C(Q) \\ 0 & \text{sonst.} \end{cases}$$

Leicht überzeugt man sich davon, dass beide Funktionen in folgender Beziehung stehen. Für $p \notin R$,

$$\mathrm{out}(p, R) = 1 \qquad \Longleftrightarrow \qquad \mathit{essential}(p, R \cup \{p\}) = 1.$$

Damit erhalten wir für die Anzahl X der Punkte ausserhalb von $C(R)$:

$$\begin{aligned} \mathbb{E}[X] &= \frac{1}{\binom{n}{r}} \sum_{R \in \binom{P}{r}} \sum_{s \in P \setminus R} \mathrm{out}(s, R) \\ &= \frac{1}{\binom{n}{r}} \sum_{R \in \binom{P}{r}} \sum_{s \in P \setminus R} \mathit{essential}(s, R \cup \{s\}) \\ &= \frac{1}{\binom{n}{r}} \sum_{Q \in \binom{P}{r+1}} \underbrace{\sum_{p \in Q} \mathit{essential}(p, Q)}_{\leq 3} \\ &\leq \frac{1}{\binom{n}{r}} \sum_{Q \in \binom{P}{r+1}} 3 \;=\; 3 \cdot \frac{\binom{n}{r+1}}{\binom{n}{r}} \;=\; 3\frac{n-r}{r+1}, \end{aligned}$$

wobei die erste Ungleichung in der letzten Zeile aus Lemma 3.26 folgt. □

Mit diesem Lemma können wir nun die Laufzeit und Korrektheit unseres Algorithmus beweisen.

Satz 3.29. Algorithmus RANDOMIZED_CLEVERVERSION berechnet den kleinsten umschliessenden Kreis von P in erwarteter Zeit $O(n \log n)$.

Beweis. Wir führen zunächst einige Zufallsvariablen ein: mit T bezeichnen wir die Anzahl Iterationen des Algorithmus und mit X_k die Anzahl Punkte nach k Iterationen; $X_0 = n$.

Den Erwartungswert von X_k können wir mit Hilfe von Lemma 3.28 leicht abschätzen. Es gilt:

$$\begin{aligned}
\mathbb{E}[X_k] &= \sum_{t=0}^{\infty} \mathbb{E}[X_k \mid X_{k-1} = t] \cdot \Pr[X_{k-1} = t] \\
&\overset{\text{Lemma 3.28}}{\leq} \sum_{t=0}^{\infty} \left(1 + \frac{3}{r+1}\right) t \cdot \Pr[X_{k-1} = t] \\
&= \left(1 + \frac{3}{r+1}\right) \cdot \sum_{t=0}^{\infty} t \cdot \Pr[X_{k-1} = t] \\
&= \left(1 + \frac{3}{r+1}\right) \cdot \mathbb{E}[X_{k-1}].
\end{aligned}$$

Per Induktion folgt somit (da $X_0 \equiv n$), dass $\mathbb{E}[X_k] \leq (1 + \frac{3}{r+1})^k \cdot n$.

Aus Lemma 3.26 wissen wir, dass es eine Menge $Q_0 \subseteq P$ der Grösse 3 gibt, mit $C(Q_0) = C(P)$. Sobald daher Q_0 in der Menge Q, die der Algorithmus in einer Iteration wählt, enthalten ist, wird der Algorithmus stoppen. Falls der Algorithmus daher nach k Runden noch nicht terminiert hat, muss mindestens einer der Punkte in Q_0 in $k/3$ der Runden ausserhalb von $C(Q)$ gelegen haben – was zu eine Verdopplung seiner Punkte geführt hat. Mit anderen Worten: wenn der Algorithmus nach k Runden noch nicht terminiert hat, so gibt es von mindestens einem der Punkte in Q_0 mindestens $2^{k/3}$ viele Kopien. Insbesondere gilt daher:

$$\mathbb{E}[X_k] = \underbrace{\mathbb{E}[X_k \mid T \geq k]}_{\geq 2^{k/3}} \cdot \Pr[T \geq k] + \underbrace{\mathbb{E}[X_k \mid T < k]}_{\geq 0} \cdot \Pr[T < k] \geq 2^{k/3} \cdot \Pr[T \geq k].$$

Vergleichen wir beide Ungleichungen, die wir für $\mathbb{E}[X_k]$ erhalten haben, so folgt

$$2^{k/3} \cdot \Pr[T \geq k] \leq \mathbb{E}[X_k] \leq (1 + \frac{3}{r+1})^k \cdot n$$

bzw. für $r = 11$ gilt $\Pr[T \geq k] \leq \min\{1, [(1+3/12)/2^{1/3}]^k n\} \leq \min\{1, 0.995^k n\}$ und somit für $k_0 = -\log_{0.995} n$:

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{k\geq 1} \Pr[T \geq k] \\
&\leq \sum_{k=1}^{k_0} 1 + \sum_{k>k_0} 0.995^k n \\
&\overset{k=k_0+k'}{=} \underbrace{\sum_{k=1}^{k_0} 1}_{=k_0} + \sum_{k'\geq 1} 0.995^{k'} \cdot \underbrace{0.995^{k_0} n}_{=1} \\
&= k_0 + O(1) \;\leq\; 200 \ln n + O(1).
\end{aligned}$$

Dies ist die erwartete Anzahl Runden; für die Laufzeit müssen wir noch mit n multiplizieren. □

Der Algorithmus, den wir in diesem Abschnitt vorgestellt haben, ist eine Variation eines Algorithmus von Clarkson, siehe Beitrag im *Taschenbuch der Algorithmen*[14]. Man sollte hier auch erwähnen, dass es randomisierte Algorithmen gibt, die $C(P)$ in optimal linearer Zeit berechnen.

Das Sampling Lemma

Hinter dem Lemma 3.28 steckt ein einfaches allgemeines Prinzip, welches wir noch ergründen wollen. Dazu beginnen wir mit einer sehr allgemeinen Definition.

Definition 3.30. Gegeben sei eine endliche Menge S, $n := |S|$, und ϕ eine beliebige Funktion auf 2^S in einen beliebigen Wertebereich. Wir definieren

$$\begin{aligned}
V(R) = V_\phi(R) &:= \{s \in S \mid \phi(R \cup \{s\}) \neq \phi(R)\} \\
X(R) = X_\phi(R) &:= \{s \in R \mid \phi(R \setminus \{s\}) \neq \phi(R)\}
\end{aligned}$$

[14] E. Welzl, Kleinster umschliessender Kreis (Ein Demokratiebeitrag aus der Schweiz?), Taschenbuch der Algorithmen, (B. Vöcking et al., Eds.), Springer-Verlag Berlin Heidelberg, eXamen.press, (2008) 385-388

Elemente in $V(R)$ nennen wir *Verletzer von* R, Elemente in $X(R)$ nennen wir *extrem in* R.

Wir sehen, dass $s \in V(R) \Leftrightarrow s \in X(R \cup \{s\})$, eine Beziehung die im nächsten Beweis wesentlich sein wird. Unser Ziel ist zu verstehen, wie gross $V(R)$ ist, wenn R zufällig mit gegebener Grösse r gewählt wird.

Beispiel Kleinste Zahl. Sei $A \subseteq \mathbb{N}$, endlich, und für $R \subseteq A$ sei $\phi(R) := \min(R)$, mit $\min(\emptyset) := \infty$. Es gilt $X(R) = \{\min(R)\}$ (für $R \neq \emptyset$) und $V(R) = \{a \in A \mid a < \min(R)\}$. Offensichtlich gilt für $R \neq \emptyset$, dass $|X(R)| = 1$ und $|V(R)|$ ist die Anzahl der Zahlen in A, die kleiner als $\min(R)$ sind, d.h. $|V(R)|+1$ ist der *Rang* von $\min(R)$ in A (der Rang von $a \in A$ ist der Index von a in der aufsteigend sortierten Reihenfolge von A).

Beispiel kleinster umschliessender Kreis. Sei P eine endliche Punktemenge und für $R \subseteq P$ sei $\phi(R) := C(R)$ der kleinste umschliessende Kreis von R (mit $\phi(\emptyset) := \emptyset$). $V(R)$ ist genau die Menge der Punkte ausserhalb $C(R)$ und wir haben gesehen, dass $|X(R)| \leq 3$.

Das Sampling Lemma setzt die erwartete Anzahl verletzender Elemente in Bezug zur Anzahl extremer Elemente. Wir verwenden im Beweis folgende einfache Tatsache: Sei $G = (A \cup B, E)$ ein bipartiter Graph mit $E \subseteq \{\{a, b\} \mid a \in A, b \in B\}$. Dann ist der Durchschnittsgrad[15] der Knoten in A mal $|A|$ genau $|E|$.

Lemma 3.31 (Sampling Lemma). Sei $k \in \mathbb{N}$, $0 \leq k \leq n$. Wir setzen $\nu_k := \mathbb{E}[|V(R)|]$ und $x_k := \mathbb{E}[|X(R)|]$, wobei R eine k-elementige Teilmenge von S ist, zufällig gleichverteilt aus $\binom{S}{k}$. Dann gilt für $r \in \mathbb{N}$, $0 \leq r < n$,

$$\frac{\nu_r}{n-r} = \frac{x_{r+1}}{r+1} .$$

Beweis. Wir definieren einen bipartiten Graph wie folgt: Die Knoten sind $\binom{S}{r} \dot\cup \binom{S}{r+1}$ mit den Kanten $\{R, R \cup \{s\}\}$, $R \in \binom{S}{r}$, s verletzt R (d.h. $\phi(R) \neq \phi(R \cup \{s\})$). Folglich ist der Grad des Knoten $R \in \binom{S}{r}$ in dem Graph genau $|V(R)|$ und die Anzahl der Kanten des Graphen ist genau $\binom{n}{r}\nu_r$.

[15] Das ist $\frac{1}{|A|}\sum_{a \in A} \deg(a)$.

Beachte nun, dass die Kanten dieses Graphen genau die Paare $\{Q\backslash\{s\}, Q\}$ sind mit $Q \in \binom{S}{r+1}$ und s extrem in Q (d.h. $\phi(Q \setminus \{s\}) \neq \phi(Q)$). Der Grad von $Q \in \binom{S}{r+1}$ ist folglich $|X(Q)|$ und die Anzahl der Kanten des Graphen ist genau $\binom{n}{r+1}\chi_{r+1}$.

Wir haben $\binom{n}{r}\nu_r = \binom{n}{r+1}\chi_r$ gezeigt und das Lemma folgt aus der einfachen Identiät $\binom{n}{r+1}/\binom{n}{r} = \frac{n-r}{r+1}$. □

Korollar 3.32. Wählen wir r Elemente R aus einer Menge A von n Zahlen zufällig, dann ist der erwartete Rang des Minimums von R in A genau $\frac{n-r}{r+1} + 1 = \frac{n+1}{r+1}$.

Wählen wir r Punkte R aus einer Menge P von n Punkten in der Ebene zufällig, dann ist die erwartete Anzahl von Punkten aus P ausserhalb von $C(R)$ höchstens $3\frac{n-r}{r+1}$.

3.2.2 Konvexe Hülle

Wir haben bereits bei den Flüssen konvexe Mengen kennengelernt. Ähnlich wie beim kleinsten umschliessenden Kreis kann man auch nach der kleinsten konvexen Menge fragen, die eine gegebene Teilmenge S des $\mathbb{R}^d$ enthält.

Definition 3.33. Sei $S \subseteq \mathbb{R}^d$, $d \in \mathbb{N}$. Die *konvexe Hülle,* $\mathsf{conv}(S)$*, von* S ist der Schnitt aller konvexen Mengen, die S enthalten, d.h.

$$\mathsf{conv}(S) := \bigcap_{S \subseteq C \subseteq \mathbb{R}^d,\, C \text{ konvex}} C\ .$$

Man kann sich leicht davon überzeugen, dass die konvexe Hülle immer selbst wieder konvex ist, also tatsächlich die kleinste konvexe Menge ist, die S enthält.

Wir beschränken uns auf die Ebene ($\mathbb{R}^2$) und die Berechnung der konvexen Hülle einer endlichen Menge P, $n := |P| \geq 3$. Der Einfachheit halber nehmen wir an, dass P in allgemeiner Lage ist, d.h. keine drei Punkte auf einer Geraden liegen und dass keine zwei Punkte die gleiche x-Koordinate haben. Wir gehen später noch auf diese wesentliche Einschränkung ein,

vorerst wollen wir uns aber nicht von Spezialfällen ablenken lassen und uns auf die algorithmischen Gesichtspunkte konzentrieren.

Für eine endliche Punktemenge P in der Ebene wird die konvexe Hülle durch ein Polygon, der Rand von conv(P), bestimmt, dessen Ecken Punkte aus P sind. Wenn wir von der Berechnung von conv(P) sprechen, so meinen wir die Bestimmung der Folge

$$(q_0, q_1, \ldots, q_{h-1}), \quad h \leq n, \tag{3.3}$$

der Ecken dieses Polygons, beginnend bei einer beliebiger Ecke q_0 und dann entgegen dem Uhrzeigersinn entlang dieses Polygons.

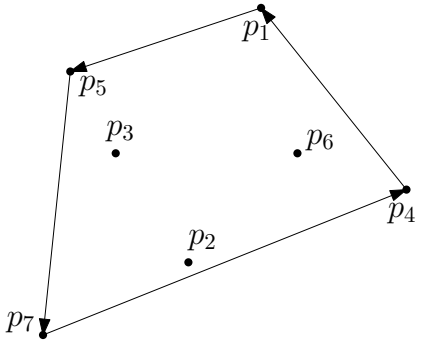


Abbildung 3.6: Punktemenge $P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ mit dem Polygon (p_4, p_1, p_5, p_7), welches conv(P) umrandet.

Sei ein geordnetes Paar qr, $q, r \in P$ mit $q \neq r$, *Randkante von* P, falls alle Punkte in $P \backslash \{q, r\}$ links von qr, d.h. auf der linken Seite der gerichteten Geraden durch q und r, gerichtet von q nach r, liegen.[16] Offensichtlich müssen in der Folge (3.3) oben alle Paare $q_{i-1}q_i$, $i = 1, 2, \ldots, h$, Randkanten von P sein (wir verstehen Indizes modh). Tatsächlich charakterisiert das schon die gültigen Ergebnisse für unser Problem.

Lemma 3.34. $(q_0, q_1, \ldots, q_{h-1})$ ist die Eckenfolge des conv(P) umschliessenden Polygons gegen den Uhrzeigersinn genau dann, wenn alle Paare (q_{i-1}, q_i), $i = 1, 2, \ldots, h$, Randkanten von P sind.

[16] Falls ein Punkt s links (rechts) von qr liegt, so nennen wir (q, r, s) einen *leftturn* (*rightturn*).

Offensichtlich ist die Frage, ob ein Punkt p links (oder rechts) von qr liegt für uns wichtig. Mit Hilfe der linearen Algebra und insbesondere Determinanten ist diese Frage algorithmisch aus den Koordinaten der Punkte leicht zu beantworten (hier ohne Beweis).

Lemma 3.35. Seien $p = (p_x, p_y)$, $q = (q_x, q_y)$, und $r = (r_x, r_y)$ Punkte in $\mathbb{R}^2$. Es gilt $q \neq r$ und p liegt links von qr genau dann, wenn

$$\begin{aligned} \det(p,q,r) &:= \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - p_x & r_y - p_y \end{vmatrix} > 0 \\ &\Leftrightarrow \quad (q_x - p_x)(r_y - p_y) > (q_y - p_y)(r_x - p_x) \end{aligned}$$

Genauer gesagt, ist $\frac{1}{2}|\det(p,q,r)|$ die Fläche des Dreiecks pqr, wobei das Vorzeichen bestimmt, wie die Punkte p, q, r den Rand dieses Dreiecks durchlaufen. Falls $\det(p,q,r) > 0$, dann passiert das gegen den Uhrzeigersinn[17] und p ist links von qr. $\det(p,q,r) = 0$ zeigt an, dass die drei Punkte auf einer gemeinsamen Gerade liegen[18].

Basierend auf Lemma 3.34 können wir einen ersten Algorithmus zur Bestimmung der konvexen Hülle angeben: Gehe durch jedes der $n(n-1)$ geordneten Paare qr, und prüfe, ob dies eine Randkante ist, indem man für alle $n-2$ Punkte p in $P \setminus \{q, r\}$ feststellt, ob p links von qr liegt. So haben wir die Randkanten in $O(n^3)$ gefunden, die wir nur mehr richtig aneinanderreihen müssen.

Einwickeln

Den ersten Ansatz wollen wir nun verbessern. Sei q_0 der Punkt mit kleinster x-Koordinate in P (der ist eindeutig auf Grund unserer Annahme zur allgemeinen Lage von P). q_0 ist sicher eine Ecke der konvexen Hülle, also Teil der gesuchten Folge und wir können insbesondere die Folge auch mit q_0 beginnen. Was ist nun der Punkt q_1, der mit q_0 eine Randkante q_0q_1 bildet? Dazu rufen wir die Funktion FINDNEXT(q_0) auf, die uns genau dieses q_1 liefert.

[17] Gegen den Uhrzeigersinn gilt in der Mathematik als die positive, bzw. "normale" Orientierung, man denke etwa an die Nummerierung der Quadranten in der Ebene.

[18] Das beinhaltet den Fall, dass zwei der Punkte, oder alle drei gleich sind.

FINDNEXT(q)

1: Wähle $p_0 \in P \setminus \{q\}$ beliebig
2: $q_{next} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: **if** p rechts von qq_{next} **then** $q_{next} \leftarrow p$
5: **return** q_{next}

Wir sollten uns bewusst machen, warum das funktioniert. Der Algorithmus erinnert uns an das Finden des Minimums einer Menge von Zahlen. Tatsächlich können wir, gegeben $q \in P$, eine Relation $\prec_q$ auf $P \setminus \{q\}$ mittels

$$p_1 \prec_q p_2 \quad :\Leftrightarrow p_1 \text{ rechts von } qp_2$$

definieren und es gilt:

Lemma 3.36. Ist q eine Ecke der konvexen Hülle von P, so ist die Relation $\prec_q$ eine totale Ordnung auf $P \setminus \{q\}$. Für das Minimum p_{min} dieser Ordnung gilt, dass qp_{min} eine Randkante ist.

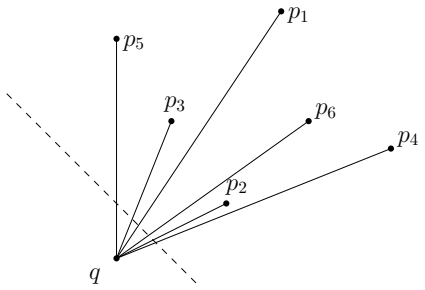


Abbildung 3.7: Totale Ordnung $p_4 \prec_q p_2 \prec_q p_6 \prec_q p_1 \prec_q p_3 \prec_q p_5$. qp_4 ist Randkante.

Die wesentliche Eigenschaft, die sich aus "q eine Ecke der konvexen Hülle von P" ergibt, ist, dass es eine Gerade gibt, die q von $P \setminus \{q\}$ trennt. Ist dies nicht möglich, so ist die Relation $\prec_q$ zyklisch, also definitiv keine totale Ordnung und der Algorithmus FINDNEXT liefert einen beliebigen Punkt in $P \setminus \{q\}$.

Nun können wir mittels FINDNEXT(q_1) den Punkt q_2 finden, der die Randkante q_1q_2 bildet, usw., bis wir wieder bei q_0 angelangt sind. Wir

haben den sogenannten Jarvis' Wrap Algorithmus (Einwickelalgorithmus) gefunden.

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{now} \leftarrow$ Punkt in P mit kleinster x-Koordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{now}$
5: $\quad p_{now} \leftarrow$ FindNext(q_h)
6: $\quad h \leftarrow h + 1$
7: **until** $p_{now} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

Jeder Aufruf der Funktion FindNext benötigt $O(n)$ Zeit, und wir rufen diese genau h-mal auf. Daraus ergibt sich sofort folgendes Ergebnis.

Satz 3.37. Gegeben eine Menge P von n Punkten in allgemeiner Lage in $\mathbb{R}^2$, berechnet der Algorithmus JarvisWrap die konvexe Hülle in Zeit $O(nh)$, wobei h die Anzahl der Ecken der konvexen Hülle von P ist.

Dieser Algorithmus ist sehr schnell für h klein (z.B. wenn die konvexe Hülle ein Dreieck ist, ist der Algorithmus linear), und im schlimmsten Fall $h = n$ haben wir immerhin die erste $O(n^3)$ Schranke auf $O(n^2)$ verbessert. Bevor wir diese quadratische Schranke verbessern, widmen wir uns kurz den Spezialfällen und Fragen der Implementierung.

Spezialfälle und Implementierung. Geometrie ist anschaulich, der JarvisWrap Algorithmus ist wirklich einfach und man ist vielleicht versucht, den Algorithmus sofort zu implementieren. Dabei tritt sofort die Frage nach der "allgemeine Lage" Annahme auf. Haben wir es mit einer beliebigen Menge P zu tun, also mit Kollinearitäten (mehr als zwei Punkte auf einer gemeinsamen Geraden) oder mehreren Punkten gleicher x-Koordinate, ist folgendes zu beachten.

1. Der Anfangspunkt q_0: Wir wählen q_0 als den Punkt mit kleinster x-Koordinate, und wenn es mehrere Punkte mit gleicher kleinster x-Koordinate gibt, den unter denen mit kleinster y-Koordinate, d.h.

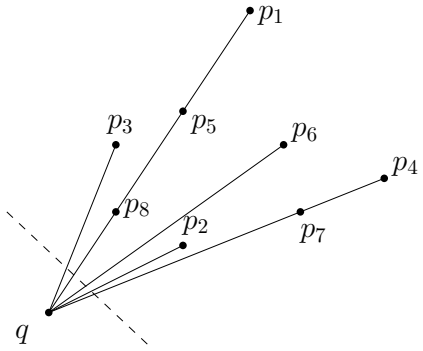


Abbildung 3.8: Totale Ordnung bei möglichen Kollinearitäten: $p_4 \prec_q p_7 \prec_q p_2 \prec_q p_6 \prec_q p_1 \prec_q p_5 \prec_q p_8 \prec_q p_3$. qp_4 ist Randkante.

den lexikographisch kleinsten Punkt in P. Dieser Punkt ist immer Ecke der konvexen Hülle.

2. Der Test "p rechts von qq_{next}" muss ersetzt werden durch (p rechts von qq_{next}) oder (p auf der Geraden durch qq_{next} und $|qp| > |qq_{next}|$); kompakt:

$$(\det(p, q, q_{next}) < 0) \ \vee \ (\det(p, q, q_{next}) = 0 \wedge |qp| > |qq_{next}|) \ .$$

3. Wie bekommen wir P gegeben? In der Regel als Array, wobei wir keine Garantie haben, dass alle Punkte verschieden sind. Wir könnten natürlich in einem ersten Schritt Doubletten entfernen. Dazu brauchen wir aber $\Theta(n \log n)$ Zeit (indem wir die Folge lexikographisch sortieren und dann die sortierte Folge auf Wiederholungen prüfen). Dann verlieren wir aber die Linearzeit des JarvisWrap bei $h = O(1)$. Es genügt aber in der Funktion FindNext bei der Wahl von p_0 auf $p_0 \neq q$ zu achten. Und beim Test "$p_{now} = q_0$" muss man achten, dass man nicht die Array-Indizes von p_{now} und q_0 vergleicht, sondern tatsächlich die Punkte[19]. Sonst läuft man möglicherweise in eine unendliche Schleife.

Neben den gerade diskutierten Spezialfällen treten auch numerische Probleme auf. Der Test "$(q_x - p_x)(r_y - p_y) > (q_y - p_y)(r_x - p_x)$" ist in einer Implementierung mit floating point nicht exakt, insbesondere auch, wenn

[19] Alternativ kann man am Anfang einmal das Array nach Doubletten von q_0 durchsuchen und entfernen.

man auf Gleichheit abfragt. Der Gleichheitstest macht in floating point nach arithmetischen Operationen eigentlich schlichtweg keinen Sinn. Dabei geht es weniger darum, dass unser Ergebnis absolut exakt ist. Oft ist die Eingabe schon mit numerischen Ungenauigkeiten behaftet, weshalb wir auch in der Ausgabe mit kleinen Ungenauigkeiten leben können. Aber das Problem ist, dass wir wegen dieser Ungenauigkeiten beim JarvisWrap am Startpunkt "vorbeilaufen" können.

Es empfiehlt sich daher, einen exakten Datentyp zu verwenden, wie er in verschiedenen Programmbibliotheken angeboten wird.

Lokal Verbessern

Wir kehren zurück zu unserer Annahme, dass unsere Punktemenge in allgemeiner Lage ist. Unser nächster Ansatz ist mit einem Polygon

$$(r_0, r_1, \ldots, r_{k-1})$$

mit Ecken aus P zu beginnen und dieses Polygon sukzessive zu "korrigieren", wann immer wir ein Problem sehen. Sollte diese Folge tatsächlich eine konvexe Menge gegen den Uhrzeigersinn umranden, dann muss sicher gelten, dass r_i rechts von $r_{i-1}r_{i+1}$ liegt, für alle $i = 0, 1, \ldots, k-1$ (Indizes mod k). Ist r_i links von $r_{i-1}r_{i+1}$, entfernen wir r_i einfach aus der Folge, bis wir keinen solchen Defekt mehr entdecken.

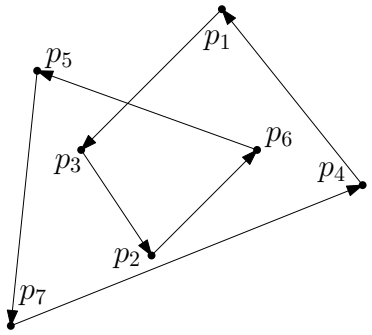


Abbildung 3.9: Eine lokal konvexes Polygon durch alle Punkte von P.

Als kleine Warnung: Ist in der Folge kein solcher Defekt zu entdecken, heisst das noch lange nicht, dass wir das richtige Polygon gefunden haben. Offensichtlich wissen wir nicht, dass das Polygon die Menge P umschliesst. Aber selbst wenn etwa jeder Punkt in P in der Folge auftritt, können wir

nicht sicher sein. Es kann sein, dass wir einen geschlossenen lokal konvexen Polygonzug haben, der sich selbst schneidet, siehe Abb. 3.9. Richtig eingesetzt wird uns die Grundidee aber gute Dienste leisten.

Als erstes sortieren wir P aufsteigend nach x-Koordinate; sei $(p_1, p_2, \ldots, p_n)$ die sich ergebende Reihenfolge. Betrachten wir das Polygon

$$(p_1, p_2, \ldots, p_{n-1}, p_n, p_{n-1}, \ldots, p_2) \ ,$$

d.h. wir durchlaufen P einmal von links nach rechts und dann wieder zurück von rechts nach links. Das Polygon hat $2(n-1)$ gerichtete Kanten. Nun beginnen wir mit dem lokalen Verbessern dieses Polygons, siehe Abb. 3.10. Dabei wird sowohl p_1 wie auch p_n sicher nie entfernt, und zu jedem Zeitpunkt teilt sich die Folge in einen Teil der x-monoton von p_1 nach p_n läuft, und einen zweiten Teil der dann von p_n zurück nach p_1 läuft (man erinnere sich, dass wir implizit immer das erste Element der Folge auch am Ende der Folge sehen, so dass sich das Polygon schliesst).

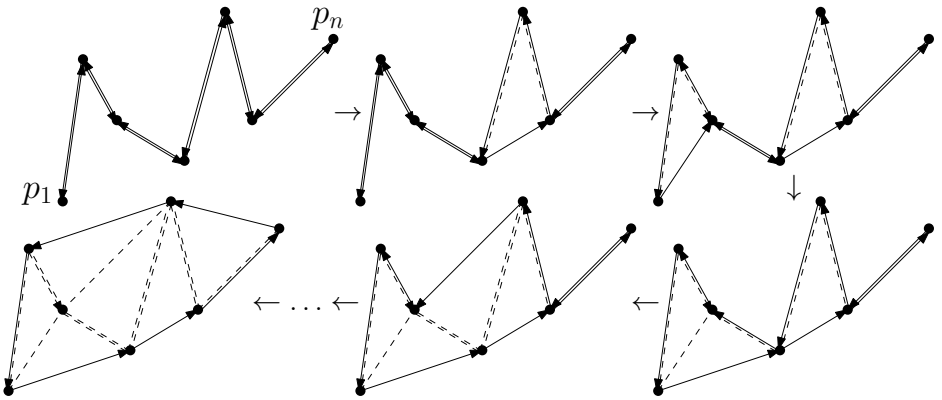


Abbildung 3.10: Das links-rechts-links Polygon durch P, und lokale Verbesserungsschritte (in unorganisierter Reihenfolge) bis zum Polygon, welches conv(P) umrandet.

Kümmern wir uns vorerst nicht um die konkrete Implementierung unseres Algorithmus, z.B. in welcher Reihenfolge wir die Tests durchführen, sondern sehen wir das sukzessive lokale Verbessern einfach als nichtdeterministischen Prozess. Ein *Verbesserungsschritt* sei definiert als die Operation, bei der wir für drei aufeinanderfolgende Punkte p, p', p'' feststellen, dass p' links von pp'' liegt und wir p' deshalb aus der Folge entfernen. Dabei vergrössert sich das vom Polygonzug umrandete Gebiet genau

um das Dreieck $pp'p''$. Behalten wir alte Kanten in der Zeichung, wie in Abb. 3.10, bekommen wir am Ende eine sogenannte Triangulierung[20] der Punktemenge P.

Bevor wir eine konkrete Organisation der Verbesserungsschritte präsentieren, widmen wir uns diesen Triangulierungen.

Ebene Graphen und Triangulierungen

Definition 3.38. Ein Graph $G = (P, E)$ auf P heisst *eben*, wenn sich die Segmente $pq := \mathrm{conv}(\{p, q\})$ der Kanten $\{p, q\} \in E$ höchstens in ihren jeweils gemeinsamen Endpunkten schneiden. Entfernen wir die Segmente pq, $\{p, q\} \in E$, aus $\mathbb{R}^2$, so nennen wir die entstehenden zusammenhängenden Gebiete die *Gebiete von* G. Die beschränkten Gebiete nennen wir *innere Gebiete*, das unbeschränkte (unendliche) Gebiet das *äussere Gebiet*.

Ein Graph $T = (P, E)$ auf P heisst *Triangulierung von* P, falls T eben ist und maximal mit dieser Eigenschaft ist, d.h. das Hinzufügen jeder Kante $\binom{P}{2} \setminus E$ zu T verletzt die Eigenschaft "eben".

Wir beobachten, dass die Gebiete eines ebenen Graphen G auf P bis auf genau eines beschränkt sind, siehe Abb. 3.11. Die inneren Gebiete einer Triangulierung sind alle Dreiecke[21]; das einzige äussere Gebiet ist das Komplement von $\mathrm{conv}(P)$ in $\mathbb{R}^2$ und liegt genau an den Randkanten von P an.

Lemma 3.39. Sei h die Anzahl der Ecken der konvexen Hülle von P. Der lokale Verbesserungsprozess macht genau $2n - 2 - h$ Verbesserungsschritte und erzeugt eine Triangulierung mit $2n-2-h$ inneren Dreiecken und $3n - 3 - h$ Kanten.

[20] Triangulierungen spielen eine wichtige Rolle in Computergraphik, geographischen Informationssystemen, Numerik, Mathematik, etc. Will man etwa ein Gelände aus Messpunkten $(x, y, z) \in \mathbb{R}^3$ rekonstruieren, so projiziert man die Punkte in die x-y-Ebene ($(x, y, z) \mapsto (x, y)$), bestimmt eine Triangulierung der Punkte in der Ebene, und "liftet" die entstandenen Dreiecke wieder in den $\mathbb{R}^3$.

[21] Jedes andere Gebiet kann man durch Segmente weiter unterteilen, was der Maximalitätseigenschaft widerspricht.

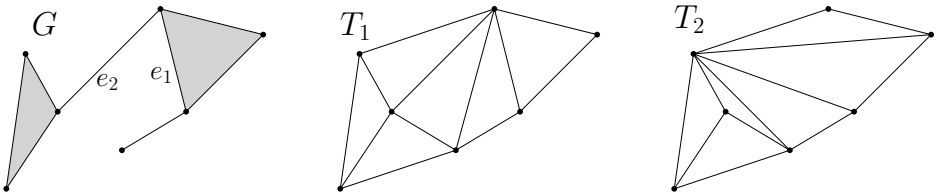


Abbildung 3.11: Ein ebener Graph G und zwei Triangulierungen T_1 und T_2 einer Punktemenge, wobei T_1 die sich aus dem Prozess in Abb. 3.10 ergebende Triangulierung ist. Der ebene Graph G hat zwei innere Gebiete (schattiert) und ein äusseres Gebiet (der nicht schattierte Rest). Beide Triangulierungen haben 6 innere Gebiete, alles Dreiecke.

Beweis. (i) Anzahl Schritte: Wir beginnen mit einem Polygon mit $2n-2$ gerichteten Kanten. In jedem Verbesserungsschritt werden zwei alte gerichtete Kanten durch eine neue ersetzt, d.h. es gibt genau eine gerichtete Kante weniger. Da wir am Ende h gerichtete Kanten haben, sind wir genau $2n-2-h$ Schritte durchlaufen.

(ii) Anzahl Dreiecke der Triangulierung: Wir beginnen mit 0 Dreiecken, jeder Verbesserungsschritt erzeugt ein neues Dreieck. Da wir $2n-2-h$ Schritte durchlaufen, haben wir am Schluss genau $2n-2-h$ Dreiecke.

(iii) Anzahl ungerichteter Kanten der Triangulierung: Wir beginnen für die Triangulierung mit $n-1$ ungerichteten Kanten (hier zählen wir die Kanten nicht doppelt für die beiden Richtungen) und fügen in jedem Schritt eine Kante hinzu, also haben wir am Ende $(n-1)+(2n-2-h) = 3n-3-h$ Kanten.

(iii') Für die Anzahl der Kanten können wir uns alternativ überlegen, dass jedes der $2n-2-h$ Dreiecke drei Kanten hat, die – abgesehen von den h Randkanten – an jeweils zwei Dreiecken anliegen. Berücksichtigen wir noch das äussere Gebiet $\mathbb{R}^2 \setminus \mathrm{conv}(P)$ mit seinen h Kanten, haben wir jede Kante genau zweimal gezählt. Folglich ist die Anzahl der Kanten

$$\frac{1}{2}(3\underbrace{(2n-2-h)}_{\text{Anzahl Dreiecke}}+\underbrace{h}_{\text{unendl. Gebiet}}) = 3n-3-h\ .$$

□

An der Stelle fragen wir uns, ob jede Triangulierung T von P genau $2n-2-h$ innere Dreiecke und genau $3n-3-h$ Kanten hat, oder ob man dies nur

für Triangulierungen behaupten kann, die aus unserem lokalen Verbesserungsprozess entstehen. Dazu zeigen wir erst eine wichtige Identität für alle ebenen Graphen.

Lemma 3.40. (Euler Relation) Sei $G = (P, E)$ ein ebener Graph auf P mit $v := |P|$ Knoten, $e := |E|$ Kanten, f Gebieten (inkl. des äusseren Gebiets), und c Zusammenhangskomponenten. Dann gilt

$$v - e + f = 1 + c \; . \tag{3.4}$$

Beweis. Die Aussage gilt sicher, wenn $E = \emptyset$, da es dann v Zusammenhangskomponenten und genau ein Gebiet gibt, es gilt also $v-e+f = v-0+1$ und $1 + c = 1 + v$.

Nehmen wir an, dass $E \neq \emptyset$ und entfernen wir eine Kante $g \in E$, was einen Graph $G' := (P, E \setminus \{g\})$ ergibt, der sicher wieder eben ist. Unser Ziel ist zu zeigen, dass die Gültigkeit der Gleichheit (3.4) durch diese Operation nicht betroffen ist. Seien v', e', f', c' die den v, e, f, c entsprechenden Werte für G'. Wir wollen also

$$v' - e' + f' - (1 + c') = v - e + f - (1 + c) \tag{3.5}$$

beweisen. Sicher gilt $v' = v$ und $e' = e - 1$. Was passiert mit f' und c'?

Wir wählen die Kante g nicht beliebig. Wenn möglich, sei g Teil eines Kreises in G. Ist dies nicht möglich, sind alle Zusammenhangskomponenten Bäume und wir wählen g so, dass es zu einem Knoten vom Grad 1 (ein Blatt) inzident ist[22] (was möglich[23] ist, weil wir $E \neq \emptyset$ voraussetzen).

Fall 1. g *ist Kante auf einem Kreis in* G. Dann ändert sich die Anzahl der Zusammenhangskomponenten nicht, d.h. $c' = c$. Und der Kreis trennt die beiden Seiten von g, daher liegt g an seinen beiden Seiten[24] zwei verschiedenen Gebieten von G an, die in G' zu einem Gebiet verschmelzen, d.h. $f' = f-1$. Es folgt (3.5), weil $v'-e'+f'-(1+c') = v-(e-1)+(f-1)-(1+c)$.

[22] Diese Einschränkung auf zu Blättern inzidenten Kanten ist nicht wesentlich, macht aber unsere Argument einfacher.

[23] Zur Erinnerung: Jeder nichtleere Baum hat mindestens zwei Blätter.

[24] Wir identifizieren hier die *kombinatorische* Kante $\{p, q\} \in \binom{P}{2}$ bisweilen mit dem *geometrischen* Liniensegment pq.

Fall 2. g *ist zu einem Knoten* p *vom Grad 1 in* G *inzident.* Dann gilt sicher $c' = c+1$. Und die beiden Seiten von g liegen den gleichen Gebieten von G an, weil wir von der einen Seite des Segments g entlang von g und dann um p herum auf die andere Seite gelangen, ohne eines der Kanten von G zu kreuzen. Das heisst, es gilt $f' = f$. (3.5) folgt leicht.

Durch sukzessives Entfernen von Kanten folgt nun, dass (3.4) für G gilt gdw. es für den leeren Graph $(P, \emptyset)$ gilt, wovon wir uns schon überzeugt haben. □

Korollar 3.41. Sei P eine Menge von $n \geq 3$ Punkten in allgemeiner Lage in $\mathbb{R}^2$ und sei h die Anzahl der Kanten von $\mathrm{conv}(P)$. (i) Jede Triangulierung T von P hat genau $3n-3-h$ Kanten und genau $2n-2-h$ innere Gebiete. (ii) Jeder ebene Graph G auf P hat höchstens $3n-3-h \leq 3n-6$ Kanten und höchstens $2n-2-h \leq 2n-5$ innere Gebiete.

Beweis. (i) Sei f die Anzahl der Gebiete von T. $f-1$ der Gebiete sind innere Gebiete, Dreiecke. Das äussere Gebiet liegt an h Kanten an. Die Anzahl der Kanten ist folglich $\frac{1}{2}(3(f-1)+h)$. Da T zusammenhängend ist, gilt wegen (3.4), dass

$$n - \frac{1}{2}(3(f-1)+h) + f = 2 \quad \Leftrightarrow f = 2n-1-h$$

und die Anzahl der inneren Gebiete ist $2n-2-h$. Daraus ergibt sich auch die Anzahl der Kanten mit

$$\frac{1}{2}(3(f-1)+h) = \frac{1}{2}(3(2n-1-h-1)+h) = 3n-3-h\ .$$

Ausserdem gilt wegen $n \geq 3$ und allgemeiner Lage, dass $h \geq 3$.

(ii) Die Schranken für G ergeben sich, weil Triangulierungen maximale ebene Graphen sind, und durch Hinzufügen von Kanten in einem ebenen Graph nicht nur die Anzahl der Kanten steigt, sondern auch die Anzahl der Gebiete nicht reduziert werden kann. □

Triangulierungen (und ebene Graphen im Allgemeinen) sind also dünne Graphen mit $O(n)$ Kanten.

Planare Graphen. Die Definitionen wie oben werden in der Graphentheorie allgemeiner betrachtet: Ein Graph heisst *planar*, wenn man ihn in der Ebene so zeichnen[25] kann (wobei Kanten als ihre Endpunkte verbindende Kurven realisiert werden können), dass sich keine zwei Kanten kreuzen. Ein *ebener Graph* ist ein planarer Graph zusammen mit einer Zeichnung ohne Kreuzung.

Unsere Ergebnisse oben (obere Schranken von $3n-6$ für die Anzahl von Kanten bzw. Euler Relation) gelten für planare bzw. ebene Graphen genau so. Eine wichtige Familie planarer Graphen erhält man aus den Kantengraphen konvexer Polytope in $\mathbb{R}^3$ (Tetraeder, Würfel, Oktaeder, etc.). Deshalb heisst die Euler Relation auch *Eulersche Polyederformel*: In einem konvexen Polytop in $\mathbb{R}^3$ gilt immer: #Ecken minus #Kanten plus #Seiten ist immer 2; z.B. $4-6+4=2$ für den Tetraeder oder $8-12+6=2$ für den Würfel. Tatsächlich sind Begriffe wie z.B. "Kanten" in der Graphentheorie aus der Geometrie motiviert.

Lokales Verbessern – Organisiert

Wir kehren nun zum lokalen Verbessern zurück und beschreiben eine konkrete Realisierung, wobei wir die Verwendung verketterter Listen vermeiden wollen, wie auch die anfängliche doppelte Speicherung der Punkte.

Betrachten wir die nach x-Koordinate sortierte Folge $(p_1, p_2, \ldots, p_n)$ (gespeichert in einem Array) und unser Ziel ist die Berechnung der konvexen Hülle $(q_0, q_1, \ldots, q_{h-1})$ (wieder in einem Array). Für $i = 2, 3, \ldots, n$ bestimmen wir die untere konvexe Hülle von $\{p_1, p_2, \ldots, p_i\}$ und speichern sie in $(q_0, q_1, \ldots, q_h)$, wobei $q_h = p_i$ gelten muss. Dann erweitern wir diese Folge um p_{i+1} und verbessern sie lokal, indem wir die Folge $(q_0, q_1, \ldots, q_h)$ beginnend mit q_h zurücklaufen, bis wir die Tangente von p_{i+1} an diese Folge gefunden haben; jeder Schritt zurück ist eines unserer Verbesserungsschritte. Sind wir bei $i = n$ angelangt, wiederholen wir das ganze von rechts nach links für die obere konvexe Hülle. Wir fassen das Vorgehen im Algorithmus LocalRepair zusammen, siehe auch Abb. 3.12.

[25] Man sagt auch: "in der Ebene einbetten".

LocalRepair$(p_1, p_2, \ldots, p_n)$

▷ setzt $(p_1, p_2, \ldots, p_n)$, $n \geq 3$, nach x-Koordinate sortiert, voraus

1: $q_0 \leftarrow p_1$
2: $h \leftarrow 0$
3: **for** $i \leftarrow 2$ **to** n **do** ▷ untere konvexe Hülle, links nach rechts
4: **while** $h > 0$ und q_h links von $q_{h-1}p_i$ **do**
5: $h \leftarrow h - 1$
6: $h \leftarrow h + 1$
7: $q_h \leftarrow p_i$ ▷ $(q_0, \ldots, q_h)$ untere konvexe Hülle von $\{p_1, \ldots, p_i\}$
8: $h' \leftarrow h$
9: **for** $i \leftarrow n - 1$ **downto** 1 **do** ▷ obere konvexe Hülle, rechts nach links
10: **while** $h > h'$ und q_h links von $q_{h-1}p_i$ **do**
11: $h \leftarrow h - 1$
12: $h \leftarrow h + 1$
13: $q_h \leftarrow p_i$
14: **return** $(q_0, q_1, \ldots, q_{h-1})$ ▷ Ecken der konvexen Hülle, gg. Uhrzeigersinn

Jeder erfolgreiche Test "$h > 0$ und q_h links von $q_{h-1}p_i$" erzeugt ein neues Dreieck, d.h. wir wissen, dass es genau $2n - 2 - h$ davon gibt. Dazu gibt es genau einen erfolglosen Test für jedes neue p_i, $i = 2, 3, \ldots, n$, bei der unteren konvexen Hülle, und jedes neue neue p_i, $i = n-1, n-2, \ldots, 1$, bei der oberen konvexen Hülle. Das macht zusätzliche $2(n-1)$ solche Tests. Insgesamt also $4n-4-h$ Tests. Wir erhalten so also eine sehr genau lineare Schranke für den LocalRepair Algorithmus.

Satz 3.42. Gegeben eine Folge $p_1, p_2, \ldots, p_n$ nach x-Koordinate sortierter Punkte in allgemeiner Lage in $\mathbb{R}^2$, berechnet der Algorithmus LocalRepair die konvexe Hülle von $\{p_1, p_2, \ldots, p_n\}$ in Zeit $O(n)$.

Das heisst, inklusive vorbereitendes Sortieren haben wir einen $O(n \log n)$ Algorithmus, der asymptotisch schneller als JarvisWrap ist, es sei denn $h = o(\log n)$. Auch ist es relativ leicht, Punkte gleicher x-Koordinate (sortiere lexikographisch) und Kollinearitäten miteinzubeziehen. Wiederholungen von Punkten kann man nach dem Sortieren entfernen. Numerische Probleme können Ungenauigkeiten hervorrufen, aber man kann deswegen nicht in eine unendliche Schleife zu laufen.

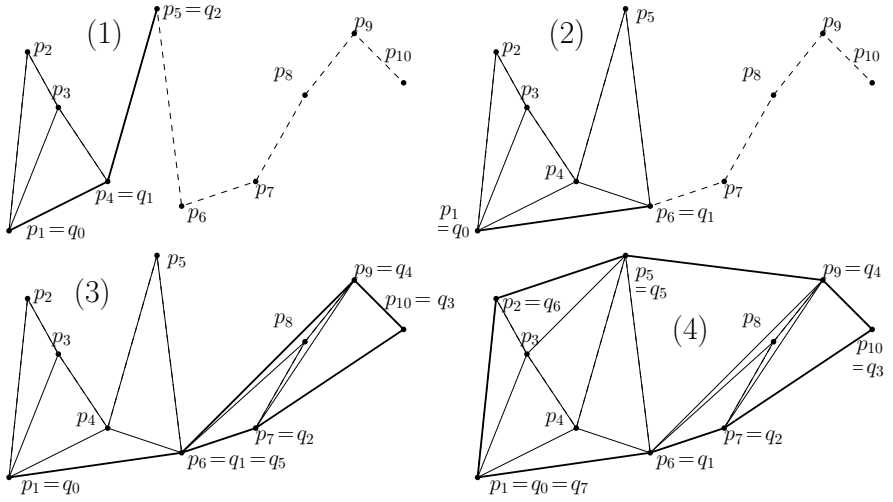


Abbildung 3.12: Illustration des LocalRepair Algorithmus. (1) (q_0, q_1, q_2) nach $i = 5$ bei links nach rechts. (2) (q_0, q_1) nach Erweiterung auf $i = 6$. (3) $(q_0, \ldots, q_5)$ nach $i = 6$ bei rechts nach links. (4) $(q_0, \ldots, q_7)$ nach Beendigung des Algorithmus, mit Ausgabe $(q_0, \ldots, q_6)$.

Geht es besser? Konvexe Hülle kann nicht schneller als Sortieren gelöst werden. Dazu betrachte man eine Folge $(x_1, x_2, \ldots, x_n)$ von Zahlen in $\mathbb{R}$. Wir setzen $p_i = (x_i, x_i^2)$, $i = 1, 2, \ldots, n$. Das heisst, wir betten die x_i's auf der x-Achse im $\mathbb{R}^2$ ein und projizieren diese Punkte vertikal auf die Einheitsparabel $y = x^2$. Berechnen wir jetzt die konvexe Hülle von $P := \{p_1, p_2, \ldots, p_n\}$ in unserem Sinn, d.h. als sortierte Folge der Ecken der konvexen Hülle so ergibt sich daraus (in linearer Zeit) auch die aufsteigend sortierte Reihenfolge der x_i's. Wir haben eine sogenannte Reduktion gezeigt: Kann man konvexe Hülle in $t(n)$ berechnen, so kann man in $t(n) + O(n)$ Zeit sortieren. Trotz-

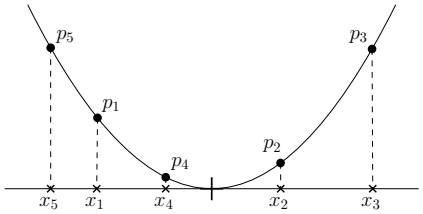


Abbildung 3.13: Vertikale Projektion auf die Einheitsparabel $y = x^2$.

dem haben wir gesehen, dass es für spezielle Eingaben (z.B. Punktemengen mit konvexen Hüllen mit wenig Ecken) schnellere Algorithmen geben kann.