Vorlesung Randomisierte Algorithmen – © Dietzfelbinger, TU Ilmenau

1 Einführung und Beispiele

Die Vorlesung „Randomisierte Algorithmen“ befasst sich mit Algorithmen, die Zufallsexperimente durchführen. Dabei stellt man sich vor, dass dem Algorithmus bzw. dem ausführenden Rechner eine Operation „Wähle eine zufällige Zahl aus $\{1, \ldots, k\}$“ zur Verfügung steht, mit k als Parameter. Während der Ausführung des Algorithmus können viele solche Experimente durchgeführt werden. Oft bestimmt dann die Eingabe x die Ausgabe $\mathcal{A}(x)$ von Algorithmus $\mathcal{A}$ auf x nicht mehr eindeutig; vielmehr ist dies eine Zufallsgröße. Von der Verwendung randomisierter Algorithmen verspricht man sich einen Effizienzgewinn, meistens bezüglich der Rechenzeit. – Wir beginnen mit *Beispielen* für den Entwurf und die Analyse solcher Verfahren.[1]

1.1 Randomisierter MinCut-Algorithmus

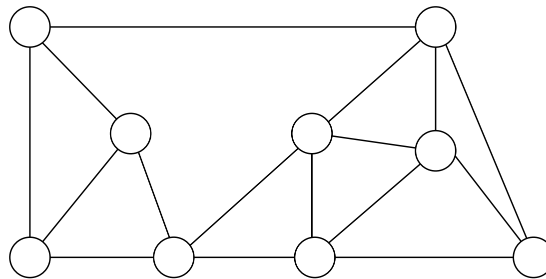

Abbildung 1.1.1: Ein zusammenhängender Graph $G = (V, E)$.

In diesem Abschnitt betrachten wir zusammenhängende (ungerichtete) Graphen, bezeichnet mit $G = (V, E)$, s. Abb. 1.1.1 für ein Beispiel. Wie üblich bezeichnen $n = |V|$

[1] In diesem einführenden Kapitel verwenden wir Konzepte und Tatsachen aus der Wahrscheinlichkeitsrechnung ohne weiteren Kommentar. Grundlagen aus der Wahrscheinlichkeitsrechnung werden in Kapitel 2 im Detail dargestellt bzw. wiederholt.

die Knotenzahl und $m = |E|$ die Kantenzahl. Ein *Schnitt* in G ist eine Menge von Kanten, deren Entfernen den Zusammenhang zerstört.[2]

Definition 1.1.1
Ein **Schnitt** in einem zusammenhängenden Graphen $G = (V, E)$ ist eine Kantenmenge $C \subseteq E$ mit der Eigenschaft, dass $(V, E - C)$ nicht zusammenhängend ist.
Ein Schnitt C heißt **minimal**, wenn es keinen Schnitt C' mit $|C'| < |C|$ gibt (Schnitt minimaler Kardinalität).

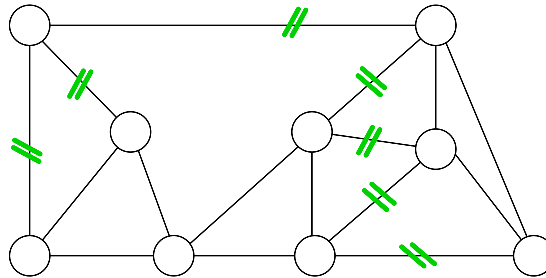

Abbildung 1.1.2: (Grün) Markierte Kanten: Schnitt C, nicht minimal.

In diesem Abschnitt betrachten wir das **MinCut-Problem**, das ist die Aufgabe, in einem gegebenen Graphen einen minimalen Schnitt zu finden. In Abb. 1.1.2 ist ein nicht minimaler Schnitt eingezeichnet, in Abb. 1.1.3 ein minimaler Schnitt.

Wir benutzen einen randomisierten Ansatz, das heißt, dass der Algorithmus „Zufallsexperimente ausführt“. Wir bauen schrittweise einen Graphen (V, R) mit $R \subseteq E$ auf. Wir beginnen mit $R = \emptyset$. Dann führen wir wiederholt den folgenden Schritt („Runde“) aus:

[2]Die technische Relevanz von Schnitten etwa für Graphen, die Kommunikationsnetzwerke darstellen, sollte offensichtlich sein. Netzwerke, die keine kleinen Schnitte haben, sind robuster gegen Verbindungsausfälle als solche, bei denen das Entfernen von wenigen Kanten dazu führt, dass das Netzwerk in Teile zerfällt. Wenn man einen sehr kleinen Schnitt findet, muss man das Netzwerk noch verbessern; wenn es keinen sehr kleinen Schnitt gibt, kann man das Netzwerk als „robust“ betrachten (gegen den Ausfall einzelner Verbindungen).

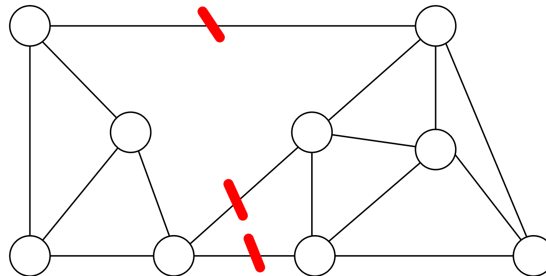

Abbildung 1.1.3: (Rot) Markierte Kanten: Minimaler Schnitt C.

> Wähle *zufällig* eine Kante e aus denjenigen Kanten aus, die zwei Zusammenhangskomponenten von (V, R) verbinden. Füge e zu R hinzu.
> (Dadurch werden zwei Zusammenhangskomponenten zu einer vereinigt.)

Dies wird iteriert, bis (V, R) genau zwei Zusammenhangskomponenten hat. Weil man mit n Zusammenhangskomponenten startet, gibt es genau $n-2$ Runden. Man sieht sofort, dass der Algorithmus die Menge R überhaupt nicht benutzt, sondern nur die Zusammenhangskomponenten. Der Algorithmus führt R daher gar nicht mit. Abb. 1.1.4 stellt einen möglichen Zwischenstand im Ablauf des Algorithmus dar.

Wenn S und $V-S$ die Knotenmengen der beiden verbleibenden Zusammenhangskomponenten sind, ist die Ausgabe

$$C = \{(v, w) \in E \mid v \in S,\, w \in V - S\}.$$

Die Idee ist als Algorithmus 1.1.2 dargestellt.

Mit einer deutlichen Anlehnung an den *Algorithmus von Kruskal* können wir diese Skizze wie folgt implementieren. Wir benutzen eine *Union-Find-Datenstruktur*.[3] Die Menge R der gewählten Kanten muss man, anders als im Algorithmus von Kruskal, nicht mitführen. (Die [...]-Teile im Algorithmus werden weggelassen.) In der Mengenvariablen F speichert man alle Kanten, die noch nicht betrachtet worden

[3] Für den Algorithmus von Kruskal und die Union-Find-Datenstruktur siehe Vorlesung „Algorithmen und Datenstrukturen" im SS 2020, `https://moodle2.tu-ilmenau.de/course/view.php?id=2465`, Abschnitte 11.3. und 11.4. Die Details der Implementierung sind für das Verständnis der Wahrscheinlichkeitsanalyse des Algorithmus aber gar nicht wichtig.

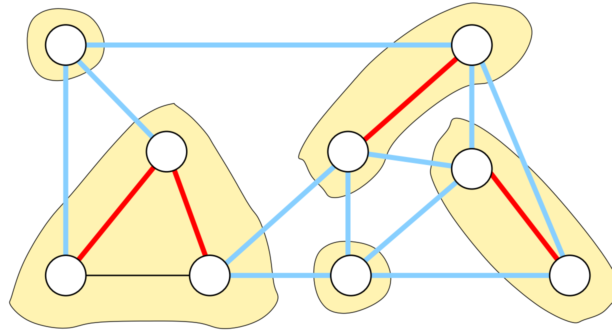

Abbildung 1.1.4: Mögliche Situation nach 4 Runden. Weiß: 9 Knoten. Rot: 4 schon gewählte Kanten. Ocker: $5 = 9 - 4$ von den roten Kanten gebildete Zusammenhangskomponenten. Blau: 11 noch verfügbare Kanten. Jede davon hat in der nächsten Runde dieselbe Wahrscheinlichkeit $\frac{1}{11}$, gewählt zu werden.

sind, und lässt sie bei der Zufallsauswahl mit zu. Wenn eine Kante gewählt wird, die innerhalb einer Komponente verläuft (mit der Union-Find-Datenstruktur leicht festzustellen, Zeilen **6** und **7**), wird sie ignoriert. Am Ende kann man die Union-Find-Datenstruktur noch benutzen, um die Kanten zu ermitteln, die zwischen den zwei verbleibenden Komponenten verlaufen (Zeilen **9**–**12**).

Algorithmus 1.1.2 ***BasicMinCut-Skizze***
Input: Zusammenhängender Graph $G = (V, E)$ mit $n = |V|$
Methode:

1 Jeder Knoten in V bildet für sich eine Zusammenhangskomponente;
2 $R \leftarrow \emptyset$;
3 **repeat** $n - 2$ **times**
4   ***wähle*** Kante e ***zufällig*** aus den Kanten, die zwei Zsh.-Komp. von (V, R)
5   verbinden; $R \leftarrow R \cup \{e\}$ // vereinigt zwei Zsh.-Komp. zu einer neuen
// es verbleiben zwei Zusammenhangskomponenten S und $V - S$
6 **return** C = Menge der Kanten aus E zwischen S und $V - S$.

Algorithmus 1.1.3 ***BasicMinCut***
Input: Zusammenhängender Graph $G = (V, E)$ mit $n = |V|$
Methode:

1 $[R \leftarrow \emptyset;]$
2 $F \leftarrow E$;
3 Initialisiere eine Union-Find-Datenstruktur mit V als Objektmenge;
4 **while** in der Union-Find-Datenstruktur gibt es mehr als 2 Klassen **do**
5   wähle Kante (v, w) aus F zufällig; entferne (v, w) aus F;
6     $r \leftarrow \mathit{find}(v)$; $s \leftarrow \mathit{find}(w)$;
7   **if** $r \neq s$ **then** // (v, w) verbindet zwei Zsh.-Komp., neue Runde
8   [füge (v, w) zu R hinzu;] $\mathit{union}(r, s)$;
9 $C \leftarrow \emptyset$;
10 **for** Kante $(v, w) \in F$ **do**
11   **if** $\mathit{find}(v) \neq \mathit{find}(w)$ **then** füge (v, w) zu C hinzu;
12 **return** C.

Abbildung 1.1.5 zeigt eine mögliche Situation am Ende des Algorithmus mit der resultierenden Ausgabe C.

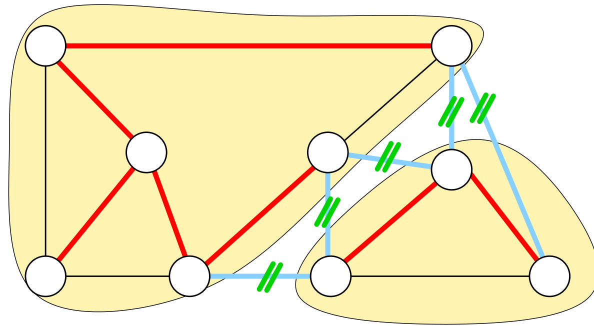

Abbildung 1.1.5: Mögliche Situation am Ende des Algorithmus BasicMinCut. Rot: Die gewählten Kanten, Menge R. Diese bilden zwei Zusammenhangskomponenten. Hellblau: Die ausgegebenen Kanten, d. h. der Schnitt C. Dies sind die Kanten, die die beiden Komponenten verbinden. Alle hellblauen Kanten sind sicher in der Menge F, schwarze Kanten können in F liegen, müssen aber nicht.

Eine kurze Bemerkung zur Rechenzeit: Wenn wir die Union-Find-Datenstruktur benutzen, die für $n-2$ Union-Operationen Zeit $O(n \log n)$ benötigt und $O(1)$ für die Find-Operationen (Array-basiert, mit Listen für die Mengen), erhalten wir Rechenzeit $O(n \log n + m)$ für den Algorithmus. Wenn die Kantenzahl nicht ganz klein ist, also $m \geq n \log n$ gilt, ist dies ein *Linearzeitalgorithmus*. Alternativ können wir Union-Find mit wurzelgerichteten Bäumen implementieren, mit Union-by-Rank und Pfadkompression, und erhalten eine Rechenzeit von $O(m \log^* n)$, auch für sehr dünne Graphen, also Kantenzahlen, die nahe bei n liegen. (Details: Vorlesung „Algorithmen und Datenstrukturen" im SS 2019.) Die Details der Rechenzeitanalyse sind hier aber nicht relevant.

Nun wenden wir uns der Frage der Korrektheit zu. Gibt es Anlass zu glauben, dass der ausgegebene Schnitt C besonders wenige Kanten hat?

Definition 1.1.4

$\mathrm{mc}(G) := \min\{|C| \mid C \text{ Schnitt in } G\}$, die Größe eines minimalen Schnitts.

Sei C_0 mit $|C_0| = k = \mathrm{mc}(G)$ ein fest gewählter minimaler Schnitt. (Für ein Beispiel siehe Abb. 1.1.3. Beachte, dass C_0 nicht eindeutig bestimmt sein muss.) Wir werden die Wahrscheinlichkeit dafür, dass Algorithmus **BasicMinCut** genau C_0 ausgibt, nach unten abschätzen.

Wir beobachten: $(V, E - C_0)$ hat genau zwei Zusammenhangskomponenten, und C_0 ist genau die Menge der Kanten zwischen diesen beiden. (Der Beweis hierfür ist eine

Übungsaufgabe. Veranschaulichung: Abb. 1.1.3.) Wir nennen S_0 die Knotenmenge der einen und $\overline{S}_0 = V - S_0$ die Knotenmenge der anderen Komponente.

Lemma 1.1.5
Die Ausgabe ist $C_0 \Leftrightarrow$ Der Algorithmus wählt nie eine Kante aus C_0.

(Der *Beweis* ist eine Übungsaufgabe.) Wir müssen nun also die Wahrscheinlichkeit dafür abschätzen, dass in keiner der Runden $i = 1, \ldots, n-2$ eine Kante aus C_0 gewählt wird. Dazu definieren wir $n-2$ *Ereignisse*[4], die ausdrücken, dass in Runde i keine Kante aus C_0 gewählt wird[5], für $i = 1, \ldots, n-2$.

$$\mathcal{E}_i := \{\text{die Kante, die in Runde } i \text{ gewählt wird, ist nicht in } C_0\}. \tag{1.1.1}$$

Lemma 1.1.6
$\mathbf{Pr}$(Algorithmus **BasicMinCut** liefert C_0 als Ausgabe) $> 2/n^2$.

Beweis: Nach Lemma 1.1.5 gibt der Algorithmus genau dann C_0 aus, wenn in keiner der $n-2$ Runden eine Kante aus C_0 gewählt wird, und das heißt, dass alle $\mathcal{E}_i$, $1 \le i \le n-2$, eintreten, formal: $\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}$ tritt ein. Wir schätzen die Wahrscheinlichkeit $\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2})$ nach unten ab.

Nach der Grundformel $\mathbf{Pr}(A \cap B) = \mathbf{Pr}(A \mid B)\mathbf{Pr}(B)$ der Wahrscheinlichkeitsrechnung gilt:

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) = \mathbf{Pr}(\mathcal{E}_1)\mathbf{Pr}(\mathcal{E}_2 \mid \mathcal{E}_1) \cdots \mathbf{Pr}(\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-3}). \tag{1.1.2}$$

(Bedingte Wahrscheinlichkeiten werden in Kapitel 2 wiederholend diskutiert.) Wir müssen also

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1})$$

abschätzen. Das heißt: Wir nehmen an, dass in Runden $1, \ldots, i-1$ keine Kante aus C_0 gewählt wurde, und fragen nach der Wahrscheinlichkeit, dass dies auch in Runde i so ist. Sei $E_{i-1} \subseteq E$ die Menge der Kanten nach Runde $i-1$, die zwei Zusammenhangskomponenten von (V, R) verbinden. Jede der Kanten in E_i hat die gleiche Wahrscheinlichkeit, in Runde i gewählt zu werden. (Für ein Beispiel siehe Abb.1.1.4.)

[4]Siehe Kapitel 2 für den Begriff des Ereignisses und die verwendete Notation.
[5]„$\mathcal{E}$“ ist ein kalligraphisches „E“ (für „Ereignis“), nicht ein zu groß geratenes „epsilon“.

Wir beobachten: (1) Aus jeder Zusammenhangskomponente von (V, R) führen mindestens k Kanten hinaus. (Grund: Diese Kanten bilden einen Schnitt.) (2) Es gibt $n - i + 1$ Zusammenhangskomponenten in (V, R). (Grund: Am Anfang waren es n viele; in jeder der bisherigen $i - 1$ Runden hat sich die Anzahl um 1 verringert.) Daraus folgt:

$$|E_{i-1}| \geq k(n - i + 1)/2.$$

(Man muss durch 2 dividieren, da jede Kante zu zwei Zusammenhangskomponenten gehört.) Die Wahrscheinlichkeit, in Runde i eine Kante aus C_0 zu wählen (d. h. dass $\overline{\mathcal{E}_i}$ eintritt), ist also höchstens

$$\frac{|C_0|}{k(n - i + 1)/2} = \frac{2k}{k(n - i + 1)} = \frac{2}{n - i + 1}.$$

(Man beachte den „magischen Schritt“ im Beweis: Der Wert k kürzt sich weg.) Übergang zur Wahrscheinlichkeit für das Komplementereignis liefert:

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1}) \geq 1 - \frac{2}{n - i + 1} = \frac{n - i - 1}{n - i + 1}.$$

Nach (1.1.2) folgt

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}. \tag{1.1.3}$$

Durch Kürzen verschwinden im Zähler und im Nenner die Faktoren $n - 2$, $n - 3$, ..., 4, 3, und wir erhalten

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{2}{n(n-1)} > \frac{2}{n^2}.$$

Das ist die Behauptung. □

Die Wahrscheinlichkeit, dass die Ausgabe C von Algorithmus 1.1.3 *nicht* gleich C_0 ist, ist also höchstens $1 - 2/n^2$, eine Zahl, die leider sehr nahe an 1 liegt.

Um die Irrtumswahrscheinlichkeit zu verringern, *wiederholen* wir Algorithmus **Ba-**

sicMinCut ℓ-mal und geben den kleinsten dabei produzierten Schnitt aus.[6]

Algorithmus 1.1.7 ***MinCut***
Input: Zusammenhängender Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$, $\ell \geq 1$.
Methode:

1 $\mathtt{C} \leftarrow \{(1, j) \mid (1, j) \in E\}$; // diese Kanten bilden einen Schnitt
2 $\mathtt{h} \leftarrow |\mathtt{C}|$;
3 **repeat** ℓ **times**
4     $\mathtt{CC} \leftarrow$ **BasicMinCut**(G);
5   **if** $|\mathtt{CC}| < \mathtt{h}$ **then**
6         $\mathtt{C} \leftarrow \mathtt{CC}$; $\mathtt{h} \leftarrow |\mathtt{CC}|$;
7 **return** $\mathtt{C}$.

Die Rechenzeit dieses Algorithmus ist ℓ mal die Rechenzeit von **BasicMinCut**. (Wir diskutieren weiter unten, wie ℓ zu wählen ist.) Es seien $C_1, \ldots, C_\ell$ die Ausgaben der ℓ Aufrufe, und es sei C^* die Ausgabe von **MinCut**. (Alle diese Kantenmengen sind Zufallsgrößen.) Wir analysieren: Wenn C^* kein minimaler Schnitt ist, dann kann C_0 nicht in $\{C_1, \ldots, C_\ell\}$ vorkommen. Also gilt:

$$\begin{aligned} &\mathbf{Pr}(C^* \text{ ist kein minimaler Schnitt}) \\ &\quad \leq \mathbf{Pr}(C_0 \notin \{C_1, \ldots, C_\ell\}) \\ &\quad = \mathbf{Pr}(C_1 \neq C_0) \cdots \mathbf{Pr}(C_\ell \neq C_0) \\ &\quad \leq \left(1 - \frac{2}{n^2}\right)^\ell . \end{aligned} \tag{1.1.4}$$

(Beim Übergang von der zweiten zur dritten Zeile wird die *Unabhängigkeit* der Zufallsexperimente in den verschiedenen Aufrufen von **BasicMinCut** benutzt, d. h., dass bei jedem Aufruf neue Zufallsexperimente durchgeführt werden. Das wahrscheinlichkeitstheoretische Konzept „Unabhängigkeit" wird in Kap. 2 diskutiert.) Wenn wir z. B. $\ell = \lceil n^2/2 \rceil$ setzen, ergibt sich

$$\mathbf{Pr}(|C^*| > k) \leq \left(1 - \frac{2}{n^2}\right)^{n^2/2} < e^{-1} \approx 0.3679, \tag{1.1.5}$$

[6] Dem Trick, Algorithmen unabhängig mehrfach zu wiederholen, um die Wahrscheinlichkeit für eine falsche Ausgabe zu verringern, werden wir wieder und wieder begegnen.

also ist

$$\mathbf{Pr}(C^* \text{ ist minimaler Schnitt}) > 1 - e^{-1} > 0.632.$$

Wir haben hier die folgende wichtige Ungleichung benutzt:

$$1 + z \le e^z, \text{ für alle } z \in \mathbb{R}, \text{ mit Gleichheit genau für } z = 0.$$

Daraus folgt mit den Potenz-Rechenregeln

$$(1 + z)^y \le e^{zy}, \text{ für alle } z \ge -1,\ y > 0.$$

Dies wurde für $z = -2/n^2$ und $y = n^2/2$ angewandt.
Diese und weitere Ungleichungen werden am Ende von Kapitel 2 diskutiert.

Wenn wir $\ell = \lceil c \cdot n^2 \cdot \ln(n)/2 \rceil$ setzen, für eine Konstante c, dann ergibt sich

$$\mathbf{Pr}(|C^*| > k) \le \left(1 - \frac{2}{n^2}\right)^{c \cdot n^2 \cdot \ln n/2} < e^{-c \ln n} = n^{-c}. \tag{1.1.6}$$

Mit einer relativ kleinen Wiederholungszahl lässt sich die Irrtumswahrscheinlichkeit also sogar auf einen „polynomiell kleinen“ Wert drücken!

Man beachte, dass die Gesamtrechenzeit des Algorithmus **MinCut** auf G mit n Knoten und m Kanten und mit Wiederholungszahl $\ell = \lceil c \cdot n^2 \cdot (\ln n)/2 \rceil$ durch $O((n \log n + m) \cdot c \cdot n^2 \cdot \ln n) = O(n^3 (\log n)^2 + mn^2 \log n)$ beschränkt ist. Auf jeden Fall ist die Rechenzeit polynomiell.

Satz 1.1.8
Algorithmus 1.1.7 **MinCut** mit $\ell = \frac{1}{2}n^2$ $[\ell = \frac{1}{2}n^2 \ln n;\ \ell = \frac{c}{2}n^2 \ln n]$ liefert mit Wahrscheinlichkeit größer als $1 - 1/e > 0.632$ $[1 - \frac{1}{n};\ 1 - \frac{1}{n^c}]$ einen Schnitt minimaler Kardinalität. Der Algorithmus hat in jedem Fall polynomielle Rechenzeit.

Mitteilung: Man kann den Algorithmus **MinCut** so modifizieren und implementieren, dass die Rechenzeit geringer wird als für sämtliche bekannten deterministischen Algorithmen. Dies wurde von Karger und Stein 1993 geleistet. Ihr Algorithmus hat eine Rechenzeit von $O(n^2 (\log n)^4)$. Karger zeigte 1996, dass man mit $O(m(\log n)^3)$ Zeit auskommt. Dies wurde erst im Jahr 2020 auf $O(m(\log n)^2)$ verbessert (Pawel Gawrychowski, Shay Mozes, Oren Weimann: Minimum Cut in $O((m \log^2 n)$ Time. ICALP 2020: 57:1-57:15).

Bemerkung: Das MinCut-Problem besitzt effiziente Algorithmen, die ohne Randomisierung auskommen. Manche dieser Algorithmen beruhen auf „Flussberechnungen“ (siehe Veranstaltung „Effiziente Algorithmen“ im Masterstudiengang oder [Cormen, Leiserson, Rivest und Stein, Introduction to Algorithms, Kapitel „Maximum Flow“]) und haben Rechenzeiten der Größenordnung $O(nm \log(n^2/m)) = O(nm \log n)$. Diese Algorithmen funktionieren sogar, wenn die Kanten mit Gewichten ≥ 0 versehen sind. Andere deterministische Algorithmen kommen ohne Flussberechnungen aus und haben Rechenzeiten von $O(nm)$ [Stoer/Wagner 1997]. In [Brinkmeier 2007] wird ein Algorithmus vorgestellt, der Rechenzeit $O(n^2\delta_G)$ hat, wobei δ_G der kleinste Knotengrad in G ist. Beachte: $m \geq n\delta_G/2$, also ist dies immer mindestens so gut wie $O(nm)$.
Wir werden im weiteren Verlauf oft der Situation begegnen, dass unsere randomisierten Algorithmen Berechnungsprobleme lösen, für die es *auch* recht effiziente deterministische Algorithmen gibt. Der Vorteil der randomisierten Verfahren ist oft, dass die Algorithmen einfacher sind als deterministische, oder dass sie etwas schneller laufen, auch in der praktischen Anwendung.

Bemerkung: Das MinCut-Problem hat also deterministische Polynomialzeitalgorithmen und einen etwas schnelleren randomisierten Algorithmus. In der Vorlesung „Automaten, Sprachen und Komplexität“ lernt man NP-vollständige Entscheidungsprobleme kennen, zusammen mit der Vermutung, dass es für diese *keinen* deterministischen Polynomialzeitalgorithmus gibt. Zu diesen Entscheidungsproblemen gehören „NP-schwere“ Optimierungsprobleme, zum Beispiel das Problem, in einem vollständigen Graphen mit Kantengewichten eine TSP-Tour mit kleinsten Kosten zu finden oder die Knoten eines Graphen mit möglichst wenigen Farben „legal“ zu färben (d. h. so dass die Endpunkte jeder Kante verschiedene Farben haben), oder auch der, zu einem zusammenhängenden Graphen einen Schnitt *maximaler* Kardinalität zu finden. Kann Randomisierung hier helfen? Leider kennt man kein einziges NP-schweres Optimierungsproblem, das einen randomisierten Polynomialzeitalgorithmus zulässt. Aufgrund von Resultaten in der Komplexitätstheorie gibt es sogar Anlass zu der Vermutung, dass es solche Probleme nicht gibt. Der Ansatz „Randomisierung“ kann also effizientere (d. h. etwas schnellere) Algorithmen liefern; vermutlich wird er aber nicht dazu führen, dass man NP-schwere Probleme effizient lösen kann. In der Vorlesung „Approximationsalgorithmen“ kann man lernen (Master-Studium!), wie Randomisierung in Verbindung mit anderen Techniken dabei hilft, effizient *näherungsweise* optimale Lösungen für NP-schwere Probleme zu berechnen.

1.2 Gleichheit von Polynomprodukten

Beispiel: Gilt

$$(X^4+4X^3+6X^2+4X+1)(X^4-4X^3+6X^2-4X+1) = (X^2-1)(X^2-1)(X^2-1)(X^2-1)\ ? \quad (1.2.7)$$

Dabei ist natürlich die Gleichheit zwischen Polynomen gemeint. Allgemeiner könnte man $(\sum_{1\le i\le n}\binom{n}{i}X^i)(\sum_{1\le i\le n}(-1)^i\binom{n}{i}X^i)$ mit dem Produkt von n Kopien von (X^2-1) vergleichen. (Erst nachher lesen: Lösung unter[7].)

Allgemein betrachten wir die folgende Aufgabenstellung: Gegeben seien Polynome $f_1(X),\dots,f_s(X)$ und $g_1(X),\dots,g_t(X)$ über einem Körper $\mathbb{F}$ (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ oder $\mathbb{Z}_p=\mathbb{Z}/p\mathbb{Z}$ für eine Primzahl p).

Frage: Gilt

$$f_1(X)\cdot\ldots\cdot f_s(X) = g_1(X)\cdot\ldots\cdot g_t(X)\ ? \quad (1.2.8)$$

Die naheliegende Methode ist, die Polynome auf beiden Seiten *auszumultiplizieren*, zu vereinfachen und dann zu vergleichen. Dabei beobachtet man Folgendes: Die Multiplikation zweier Polynome vom Grad d benötigt bei naivem Vorgehen $\Theta(d^2)$ Körperoperationen. Mit der FFT-Methode (siehe Vorlesung „Algorithmen und Datenstrukturen“) erreicht man für manche Grundbereiche (z. B. $\mathbb{R}$ oder $\mathbb{C}$) Kosten von $O(d\log d)$. Ähnliche Kosten ergeben sich bei der Multiplikation von d Kopien eines Polynoms vom Grad 2. Es ist kein deterministischer Algorithmus bekannt, der für solche Multiplikationsaufgaben mit linearem Aufwand auskommt.

Wir versuchen es mit folgendem Trick: Wir wählen ein zufälliges Argument, setzen es in die einzelnen Polynome ein und werten diese aus, multiplizieren dann die Werte und vergleichen die Produkte. (Damit werden zu keiner Zeit die Koeffizienten eines Produktpolynoms berechnet.)

Dies heißt genauer: Sei z. B. $\mathbb{F}$ gleich $\mathbb{Q}$ (oder $\mathbb{R}$ oder $\mathbb{C}$). Bestimme eine Zahl k aufgrund der Eingabe und der angestrebten Irrtumswahrscheinlichkeit. Wähle eine Menge $S\subseteq\mathbb{F}$ der Größe k, zum Beispiel die Menge $\{1,\dots,k\}$. Dann wähle a aus S rein zufällig. Berechne $b_1=f_1(a),\dots,b_s=f_s(a),c_1=g_1(a),\dots,c_t=g_t(a)$ durch Einsetzen und Auswerten, und berechne dann $u=b_1\cdot\ldots\cdot b_s$ und $v=c_1\cdot\ldots\cdot c_t$. Falls $u\ne v$, weiß man sicher, dass die Produktpolynome verschieden sind. Falls $u=v$, muss man mit der Interpretation des Ergebnisses etwas vorsichtiger sein.

---

[7] Mit der binomischen Formel $(Y+Z)^n=\sum_i\binom{n}{i}Y^iZ^{n-i}$ sieht man, durch Einsetzen von 1 für Y und X bzw. $(-X)$ für Z, dass dabei $(1+X)^n(1-X)^n$ mit $(1-X^2)^n=((1+X)(1-X))^n$ verglichen wird. Es herrscht also Gleichheit.

Wesentlich ist die Betrachtung der Grade. Sei

$$d := \max\Big\{\sum_{1\le i\le s} \deg(f_i(X)), \sum_{1\le j\le t} \deg(g_j(X))\Big\}.$$

Dann hat das Polynom

$$h(X) := f_1(X)\cdot\ldots\cdot f_s(X) - g_1(X)\cdot\ldots\cdot g_t(X) \tag{1.2.9}$$

auf keinen Fall einen Grad größer als d. Offensichtlich gilt für alle $a \in \mathbb{F}$:

$$h(a) = 0 \quad\Leftrightarrow\quad f_1(a)\cdot\ldots\cdot f_s(a) = g_1(a)\cdot\ldots\cdot g_t(a). \tag{1.2.10}$$

Bei der angedeuteten Berechnung erhalten wir also $u = v$ genau dann, wenn $h(a) = 0$ ist. Wenn die Polynomprodukte gleich sind, ist $h(X)$ das Nullpolynom, und es gilt $h(a) = 0$ für alle $a \in \mathbb{F}$. Andernfalls ist $h(X)$ nicht das Nullpolynom.

Fakt 1.2.1

Ein Polynom vom Grad höchstens d über einem beliebigen Körper (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}_p$ für eine Primzahl p), das nicht das Nullpolynom ist, hat in diesem Körper nicht mehr als d Nullstellen.[8]

Daher hat $h(X)$ nicht mehr als d Nullstellen in $\mathbb{F}$. Wenn wir $k \ge d$ wählen, erhalten wir für zufällig aus S gewähltes a:

$$\mathbf{Pr}(h(a) = 0) = \frac{|\{a \in S \mid h(a) = 0\}|}{k} \le \frac{d}{k}.$$

Durch geeignete Wahl von $k = |S|$ können wir diese Wahrscheinlichkeit klein machen.

[8] Die Beweisidee ist folgende: Wenn das Polynom $f(X)$ die verschiedenen Nullstellen $a_1, \ldots, a_r$ hat, dann kann man $f(X) = (X - a_1)\ldots(X - a_r)\cdot h(X)$ schreiben, für ein Polynom $h(X)$, man kann also für jede Nullstelle einen linearen Faktor abspalten. Daraus sieht man sofort, dass $f(X)$ Grad mindestens r haben muss.

Algorithmus 1.2.2 ***PPV – Polynomproduktvergleich***
Input: Listen $f_1(X), \ldots, f_s(X)$ und $g_1(X), \ldots, g_t(X)$ von Polynomen über $\mathbb{F}$
Methode:
1 $\mathtt{d} \leftarrow \max\{\sum_{1 \le i \le s} \deg(f_i(X)), \sum_{1 \le j \le t} \deg(g_j(X))\}$;
2 $\mathtt{k} \leftarrow 2 \cdot \mathtt{d}$;
3 Wähle eine Teilmenge S von $\mathbb{F}$ mit $|S| = \mathtt{k}$;
4 Wähle $\mathtt{a}$ aus S *zufällig* // (uniforme Verteilung)
5 $\mathtt{u} \leftarrow f_1(\mathtt{a}) \cdot \ldots \cdot f_s(\mathtt{a})$; // gerechnet in $\mathbb{F}$
6 $\mathtt{v} \leftarrow g_1(\mathtt{a}) \cdot \ldots \cdot g_t(\mathtt{a})$;
7 **return** $[\mathtt{u} \neq \mathtt{v}]$.

Die zentralen Eigenschaften von Algorithmus 1.2.2 sind folgende:

- Die Anzahl der Multiplikationen von Körperelementen ist höchstens $4d + s + t$, die Rechenzeit ist $O(d)$, wenn man die einzelnen Polynome mit Hilfe des Horner-Schemas auswertet.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) = g_1(X) \cdot \ldots \cdot g_t(X)$, ist die Ausgabe sicher 0.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) \neq g_1(X) \cdot \ldots \cdot g_t(X)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \le \frac{1}{2}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \le d/k$).

Wenn man höhere Zuverlässigkeit möchte, könnte man viel größere k wählen. Für Körper, bei denen eine Rechenoperation tatsächlich mit Zeit $O(1)$ zu veranschlagen ist, ist dies ohne große Rechenzeitveränderung möglich. Schwierigkeiten entstehen, wenn der Körper nur endlich viele Elemente hat oder wenn, wie im Fall $K = \mathbb{Q}$, Rechnen mit größeren Zahlen größeren Aufwand hat. Alternativ führt man Algorithmus 1.2.2 mehrfach (ℓ-mal) durch, mit Ergebnissen $b_1, \ldots, b_\ell$, und liefert $\max\{b_1, \ldots, b_\ell\}$ als Gesamtergebnis. Für diesen Algorithmus gilt:

- Die Anzahl der Multiplikationen von reellen Zahlen ist höchstens $(4d + s + t)\ell$.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) = g_1(X) \cdot \ldots \cdot g_t(X)$, ist die Ausgabe sicher 0.
- Wenn $f_1(X) \cdot \ldots \cdot f_s(X) \neq g_1(X) \cdot \ldots \cdot g_t(X)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \le \frac{1}{2^\ell}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \le (d/k)^\ell$).

Um zu sehen, dass Rechenzeitunterschiede drastisch sein können, betrachten wir abschließend noch folgendes Beispiel: $\mathbb{F} = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ für eine Primzahl p, mit Addition und Multiplikation modulo p. Es sei n ein Maß für die Eingabegröße. Die Frage lautet: Ist $(1+X)(1+X^2)(1+X^4)\ldots(1+X^{2^{n-1}})(1-X)$ gleich $1-X^{2^n}$?

Der naive Algorithmus berechnet das Produkt der linken Seite durch Multiplikation von links nach rechts. Da

$$\prod_{0 \le i < n} (1+X^{2^i}) = \sum_{0 \le j < 2^n} X^j$$

gilt (Hinweis: Binärdarstellung der Exponenten j), hat das vorletzte Zwischenergebnis 2^n viele Terme, bevor sich bei der letzten Multiplikation fast alles wegkürzt. Mit geschickter Auswertung kann man das Produkt für ein Körperelement a aber viel schneller ausrechnen: Mit $b_0 = a$, $b_1 = b_0^2 = a^2$, $b_2 = b_1^2 = a^4$, $b_3 = b_2^2 = a^8$, $\ldots$ (immer modulo p) erhält man die benötigten Potenzen durch nur n Multiplikationen modulo p.

1.3 Verifikation von Matrixprodukten

Gegeben seien drei $n \times n$-Matrizen A, B und C über einem Körper $\mathbb{F}$ (zum Beispiel $\mathbb{F} = \mathbb{Q}$ oder $\mathbb{F} = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ mit Operationen modulo p, wobei p eine Primzahl ist). Jemand behauptet, dass $A \cdot B = C$ ist. Wir wollen dies überprüfen.

Eine naheliegende Methode ist, das Produkt $A \cdot B$ zu berechnen und das Resultat mit C zu vergleichen. Mit der Schulmethode für die Multiplikation dauert dies Zeit $O(n^3)$, mit der Strassen-Methode (Vorlesung „Algorithmen und Datenstrukturen“) Zeit $O(n^{\log 7})$, wobei $\log 7 \approx 2.81$.[9] Wir zeigen hier, dass ein sehr einfacher randomisierter Algorithmus mit Zeit $O(n^2)$ auskommt.

[9] Schnellere Methoden: Coppersmith und Winograd (1987/90), Zeit $O(n^{2.376})$; Stothers (2010), Williams (2011), Le Gall (2014): Zeit $O(n^{2.3728639})$. (Diese Algorithmen gelten allerdings als rein theoretisch wichtig; für realistisch große n sind andere Verfahren schneller.)

Algorithmus 1.3.1 ***VerifyMatrixProduct***
Input: $n \times n$-Matrizen A, B, C über Körper $\mathbb{F}$.
Methode:
1 Wähle endliche Menge $S \subseteq \mathbb{F}$ mit $0, 1 \in S$;
2 Wähle Vektor $\mathtt{v} = (v_1, \ldots, v_n) \in S^n$ *zufällig* // (uniforme Verteilung)
3 $\mathtt{u} \leftarrow B \cdot \mathtt{v}$;
4 $\mathtt{w} \leftarrow A \cdot \mathtt{u}$;
5 $\mathtt{x} \leftarrow C \cdot \mathtt{v}$;
6 **return** $[\mathtt{w} \neq \mathtt{x}]$.

Die Rechenzeit dieses Algorithmus ist $O(n^2)$, weil drei Matrix-Vektor-Multiplikationen auszuführen sind. Da die Größe der Eingabe $\Theta(n^2)$ ist, ist dies lineare Rechenzeit.

Wie steht es mit dem Ein-Ausgabe-Verhalten? Wir bezeichnen die Vektoren in `v`, `w`, `x` mit v, w, x. Offensichtlich gilt Folgendes: Wenn $A \cdot B = C$ ist, dann gilt

$$x = C \cdot v = (A \cdot B) \cdot v = A \cdot (B \cdot v) = A \cdot u = w,$$

also ist die Ausgabe 0. (In der Rechnung haben wir die Assoziativität im Produkt $A \cdot B \cdot v$ benutzt. Der Trick ist, dass wir auf diese Weise Information über $A \cdot B$ erhalten können, ohne dieses Matrixprodukt auszurechnen!)

Ab hier gelte $A \cdot B \neq C$. Wir definieren

$$D := A \cdot B - C,$$

mit $D = (d_{ij})_{1 \le i,j \le n}$. Weil $A \cdot B \neq C$, gibt es einen Eintrag $d_{i_0 j_0} \neq 0$. Wir überlegen: Wenn $v_1, \ldots, v_{j_0-1}, v_{j_0+1}, \ldots, v_n$ irgendwelche Elemente von S sind, dann gibt es in S entweder ein oder kein Element v_{j_0} mit

$$d_{i_0 1} v_1 + \cdots + d_{i_0, j_0-1} v_{j_0-1} + d_{i_0 j_0} v_{j_0} + d_{i_0, j_0+1} v_{j_0+1} + \cdots + d_{i_0 n} v_n = 0. \tag{1.3.11}$$

Dies liegt daran, dass man wegen $d_{i_0 j_0} \neq 0$ die Gleichung (1.3.11) nach v_{j_0} auflösen kann. Eine Lösung gibt es, wenn das Ergebnis in S liegt, keine Lösung sonst. Damit erhalten wir: Wenn v aus S^n zufällig gewählt wird, ist die Wahrscheinlichkeit dafür, dass (1.3.11) gilt, höchstens $1/|S|$.

Wir können also abschätzen:

$$\begin{aligned}
&\mathbf{Pr}(\text{Ausgabe ist } 0)\\
&= \mathbf{Pr}(A \cdot B \cdot v = C \cdot v) \qquad (\text{für den Zufallsvektor } v)\\
&= \mathbf{Pr}(D \cdot v = 0)\\
&\le \mathbf{Pr}(\text{Gleichung (1.3.11) gilt})\\
&\le \frac{1}{|S|} \le \frac{1}{2}. \qquad (1.3.12)
\end{aligned}$$

Wir fassen zusammen: Algorithmus 1.3.1 hat folgendes Verhalten: Wenn $A \cdot B = C$ ist, ist die Ausgabe 0; wenn $A \cdot B \ne C$ ist, entsteht die (unerwünschte) Ausgabe 0 höchstens mit Wahrscheinlichkeit $1/|S|$.

Spezialfall: Wenn $\mathbb{F} = \mathbb{Z}_p$ für eine Primzahl p, wird man $S = \mathbb{Z}_p$ wählen; die Irrtumswahrscheinlichkeit ist dann (genau) $1/p$.

Wenn die Irrtumswahrscheinlichkeit $1/|S|$ zu groß ist, können wir ebenso wie in Abschnitt 1.1 durch ℓ-fache Wiederholung die Irrtumswahrscheinlichkeit auf $(1/|S|)^\ell$ drücken. Beispielsweise genügt $(c\lceil(\log n)/\log|S|\rceil)$-fache Wiederholung, um eine Irrtumswahrscheinlichkeit von höchstens $2^{-c\log n} = n^{-c}$ zu erhalten. Dann hat der Algorithmus eine Rechenzeit von $O(n^2 \log n)$, immer noch viel weniger als die deterministischen Verfahren.

1.4 Randomisiertes Quicksort

Der Algorithmus „Quicksort“ wurde schon in mehreren Vorlesungen betrachtet: „Algorithmen und Programmierung“ und „Algorithmen und Datenstrukturen“. Hier konzentrieren wir uns auf die „randomisierte“ Variante und auf eine clevere Analysemethode, die einige interessante Konzepte aus der Wahrscheinlichkeitsrechnung benutzt.

Wir gehen von folgender idealisierter Situation aus: In einem Array `A[`$1..n$`]` stehen verschiedene Zahlen[10] $a_1, \ldots, a_n$, in dieser Reihenfolge. Die Aufgabe ist, diese Zahlen umzusortieren, in eine Reihenfolge $b_1 < \cdots < b_n$, wobei b_i in `A[`i`]` steht, für $1 \le i \le n$.

[10] In tatsächlichen Anwendungen hat man Datensätze mit Schlüsseln aus einem total geordneten Universum, nicht unbedingt Zahlen. Dabei kann es natürlich auch identische Schlüssel geben, und man muss entsprechende Vorkehrungen treffen. Eine Möglichkeit ist 3-Wege-Partitionierung, siehe weiter unten. Wir betrachten die Situation mit verschiedenen Zahlen, um bei der Analyse den Überblick zu behalten.

Randomisiertes Quicksort geht so vor: Wenn $n = 1$ ist, passiert nichts. Wenn $n > 1$ ist, wird eine Prozedur **rqsort**$(1, n)$ aufgerufen. Dabei ist **rqsort**(l, r) so angelegt, dass gilt: Die Einträge in `A[`$l..r$`]` werden sortiert, die Einträge in `A[`$1..l-1$`]` und `A[`$r+1..n$`]` bleiben unverändert. Die Prozedur **rqsort**(l, r) geht wie folgt vor: Wenn $l \geq r$ gilt, passiert nichts. Wenn $l < r$ gilt, wird ein Index $t \in \{l, \ldots, r\}$ ***zufällig*** gewählt. Der Eintrag x in `A[`t`]` wird das „Pivotelement“ oder „partitionierende Element“. Durch eine Prozedur „**partition**“ werden die Einträge in `A[`$l..r$`]` wie folgt verschoben: Der Eintrag x landet in einer Position `A[`p`]` mit $l \leq p \leq r$, alle Einträge in `A[`$l..p-1$`]` sind kleiner als x, alle Einträge in `A[`$p+1..r$`]` sind größer als x.[11] Dieser Vorgang erfordert genau $r - l$ Vergleiche und $\Theta(r - l)$ Zeit. Anschließend wird rekursiv **rqsort**$(l, p-1)$ aufgerufen (falls $l < p - 1$ ist) und **rqsort**$(p + 1, r)$ (falls $p + 1 < r$ ist).

Durch einen einfachen Induktionsbeweis über die Größe $k = r - l$ zeigt man, dass **rqsort**(l, r) tatsächlich das Teilarray `A[`$l..r$`]` sortiert. Die Frage ist, wie groß die erwartete Anzahl von Schlüsselvergleichen und wie groß die erwartete Rechenzeit ist. Hierbei wird der Erwartungswert über die Zufallsentscheidungen des Algorithmus (bei der Wahl der Pivotelemente) gebildet, nicht etwa über verschiedene Inputs.

[11] 3-Wege-Partitionierung erzeugt drei Segmente in `A[`$l..r$`]`: In `A[`$l..p_1 - 1$`]` stehen Einträge $< x$, in `A[`$p_1..p_2$`]` stehen Einträge mit Schlüssel x, in `A[`$p_2 + 1..r$`]` stehen Einträge $> x$. Der erste Teil `A[`$l..p_1 - 1$`]` und der dritte Teil `A[`$p_2 + 1..r$`]` werden rekursiv sortiert, wenn sie mehr als einen Eintrag haben. Man kann relativ leicht zeigen, dass die für den Spezialfall ermittelte mittlere Anzahl von Vergleichen eine obere Schranke für die Anzahl von Vergleichen im allgemeinen Fall ist.

Wir geben den Algorithmus nochmals im Zusammenhang an:

Algorithmus 1.4.1 ***RandomizedQuicksort***

Input: Ein Array `A[`$1..n$`]` mit Einträgen $a_1, \ldots, a_n$, alle verschieden.

Aufruf: if $n > 1$ **then rqsort**$(1, n)$.

Dabei ist **rqsort** die folgende rekursive, randomisierte Prozedur:

Prozedur rqsort(ℓ, r) // Vorbedingung: $1 \le \ell < r \le n$

1. wähle *zufällig* ein $t \in \{\ell, \ell+1, \ldots, r\}$ // (uniforme Verteilung)
2. $x \leftarrow$ `A`$[t]$; // x heißt *Pivotelement* oder *partitionierendes Element*
3. Aufruf der Prozedur **partition**(ℓ, r, t, p), mit Ausgabeparameter p:
   Verschiebe die `A[`i`]`, $\ell \le i \le r$, mit `A[`i`]` $< x$ nach `A[`$\ell..p-1$`]`,
   verschiebe x nach `A[`p`]`,
   verschiebe die `A[`i`]`, $\ell \le i \le r$, mit `A[`i`]` $> x$ nach `A[`$p+1..r$`]`;
4. **if** $\ell < p - 1$ **then rqsort**$(\ell, p-1)$;
5. **if** $p + 1 < r$ **then rqsort**$(p+1, r)$;

Zur Analyse beobachten wir:

- Ein Aufruf **rqsort**(l, r) kostet $r - l$ Vergleiche und Zeit $\Theta(r - l)$, wenn man die rekursiven Aufrufe nicht mitzählt.
- Es gibt insgesamt maximal $n - 1$ Aufrufe von **rqsort**. (Dies liegt daran, dass ein Eintrag, der im Aufruf **rqsort**(l, r) Pivotelement war, nicht in davon ausgelösten rekursiven Aufrufen vorkommen kann, und die Mindestlänge des Arrays in einem Aufruf 2 ist. Es kann tatsächlich $n - 1$ Aufrufe geben.)

Zur Veranschaulichung betrachte man Abb. 1.4.6.

Den gesamten Zeitaufwand erhalten wir durch Summation über alle Aufrufe. Die fixen Kosten aller Aufrufe liefern Zeitaufwand $O(n)$, die variablen Kosten sind

$$\sum_{\substack{1 \le l < r \le n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} \Theta(r - l) = \Theta\Bigg(\underbrace{\sum_{\substack{1 \le l < r \le n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} (r - l)}_{=:\, C} \Bigg)$$

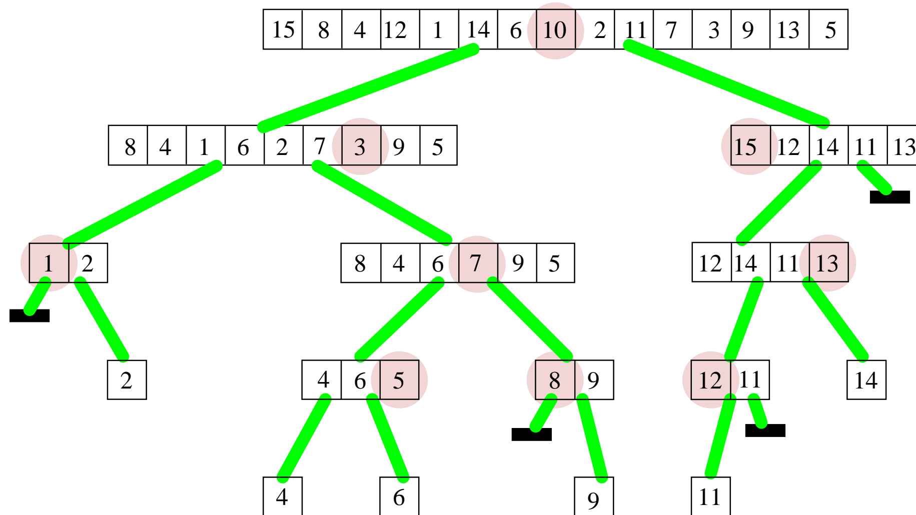

Abbildung 1.4.6: Ablauf von Quicksort als Baum. Die zu sortierenden Schlüssel sind $1, \ldots, 15$ in irgendeiner Reihenfolge. Die aufgrund der zufallsgesteuerten Pivotwahl entstehenden Aufrufe von **rqsort** sind als Baum dargestellt, wobei in einem Knoten das Teilarray $\mathtt{A}[\ell..r]$ steht, für das ein solcher Aufruf erfolgt, in den Kindern entsprechend die Teilarrays, die den beiden rekursiven Aufrufen entsprechen. Man beachte, dass der Baum vom Zufall abhängt. (Die Aufrufreihenfolge entspricht einem Präorderdurchlauf durch den Baum.) Pivots sind rosa dargestellt. Kosten: Der Knoten für $\mathtt{A}[\ell..r]$ kostet Zeit $O(1)$ plus Zeit proportional zu $r-\ell$; zu ihm gehören $r-\ell$ Vergleiche in der Partitionierungsprozedur.

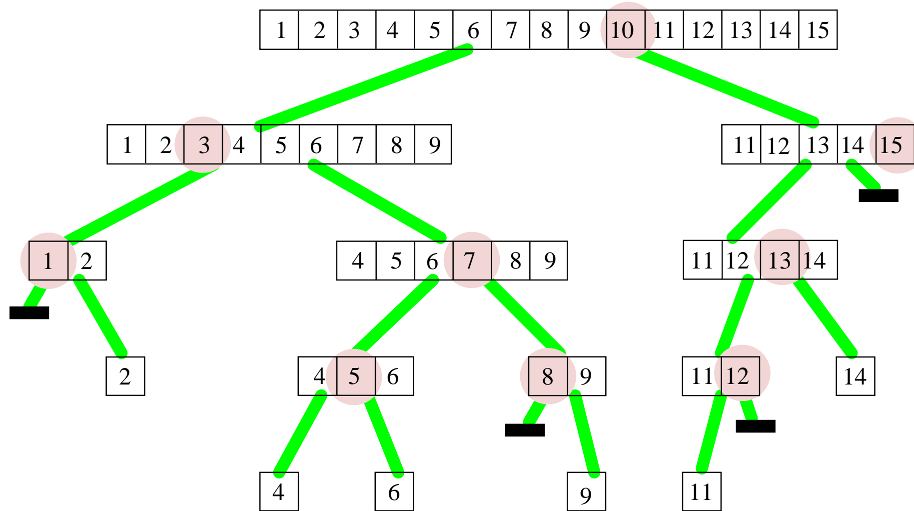

Abbildung 1.4.7: Ablauf von Quicksort als Baum. Wir betrachten die ideale Situation, wo der Input schon aufsteigend sortiert ist (also $(a_1,\ldots,a_{15}) = (b_1,\ldots,b_{15}) = (1,\ldots,15)$ gilt). Die Wahrscheinlichkeiten für die Pivotwahlen sind exakt dieselben wie in Abb. 1.4.6, weil jeder Eintrag in $\texttt{A}[\ell..r]$ genau dieselbe Wahrscheinlichkeit $1/(r-\ell+1)$ hat, als Pivot gewählt zu werden, ganz gleich wie die Zahlen angeordnet sind. Wenn der Ablauf wie in dieser Abbildung ist, gilt $C = 14+8+4+1+5+3+2+1+1 = 39$ und $X_{4,10} = 1$, $X_{5,11} = 0$, $X_{4,7} = 1$, $X_{3,8} = 1$, $X_{4,9} = 0$, für die im Text definierten Indikatorvariablen X_{ij}. Es sei nochmals betont, dass der Baum von den zufällig gewählten Pivots bestimmt wird. Andere Pivots führen zu einem anderen Baum und zu anderen Werten für C und für die X_{ij}.

Dabei gibt die Zufallsvariable C die Anzahl aller ausgeführten Vergleiche an. Die erwartete Anzahl von Vergleichen ist der Erwartungswert $\mathbf{E}(C)$. Diesen werden wir im Folgenden berechnen. Die erwartete Rechenzeit ist $\Theta(n + \mathbf{E}(C))$.

Erinnerung: $b_1 < \cdots < b_n$ ist die Folge der Eingabezahlen *in sortierter Reihenfolge*. Wir definieren:[12]

$$X_{ij} := [b_i \text{ und } b_j \text{ werden verglichen}] := \begin{cases} 1 & \text{, falls } b_i \text{ und } b_j \text{ verglichen werden,} \\ 0 & \text{, andernfalls.} \end{cases}$$

Die X_{ij} sind sogenannte ***Indikator-Zufallsvariablen***. Offensichtlich gilt

$$C = \sum_{1 \le i < j \le n} X_{ij} \text{ , also } \mathbf{E}(C) = \sum_{1 \le i < j \le n} \mathbf{E}(X_{ij}), \tag{1.4.13}$$

wegen der Linearität des Erwartungswertes (siehe Kapitel 2). Zur Veranschaulichung siehe Abb. 1.4.7.

Da X_{ij} nur die Werte 0 und 1 annimmt, gilt

$$\mathbf{E}(X_{ij}) = \mathbf{Pr}(X_{ij} = 1) = \mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}).$$

Wir müssen also nur die letztgenannten Wahrscheinlichkeiten bestimmen und dann summieren.

Betrachte $1 \le i < j \le n$ und dazu die Menge $I_{ij} = \{b_i, b_{i+1}, \ldots, b_j\}$ der Eingabezahlen zwischen b_i und b_j. Beobachte den Ablauf des Algorithmus, wie er sich durch die rekursiven Aufrufe hindurch entwickelt. Solange kein Element von I_{ij} als Pivotelement gewählt wird, werden bei der Partitionierung stets alle Elemente von I_{ij} im gleichen Teilarray landen. Genau in dem Moment, in dem zum ersten Mal ein Element von I_{ij} Pivotelement wird, fällt die Entscheidung darüber, ob b_i und b_j verglichen werden oder nicht. Wenn entweder b_i oder b_j dieses Pivotelement ist, erfolgt der Vergleich, wenn einer der Einträge $\{b_{i+1}, \ldots, b_{j-1}\}$ das erste ist, landen b_i und b_j bei der Partitionierung in verschiedenen Teilarrays und werden nie verglichen.

Weil Pivotelemente uniform zufällig gewählt werden, hat jedes der $j-i+1$ Elemente von I_{ij} die gleiche Wahrscheinlichkeit, das erste Pivotelement in dieser Menge zu sein. Damit gilt:[13]

$$\mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}) = \frac{|\{i,j\}|}{|\{i, i+1, \ldots, j\}|} = \frac{2}{j-i+1}. \tag{1.4.14}$$

[12] Wir benutzen hier und später die sogenannten „Iverson-Klammern“ $[\ldots]$, die Wahrheitswerte in Zahlen übersetzen. Wenn φ eine Aussage ist, dann ist $[\varphi] = 1$, wenn φ zutrifft und $[\varphi] = 0$ sonst.

[13] In Abb. 1.4.8 wird diese Überlegung anhand von Illustrationen anschaulich begründet.

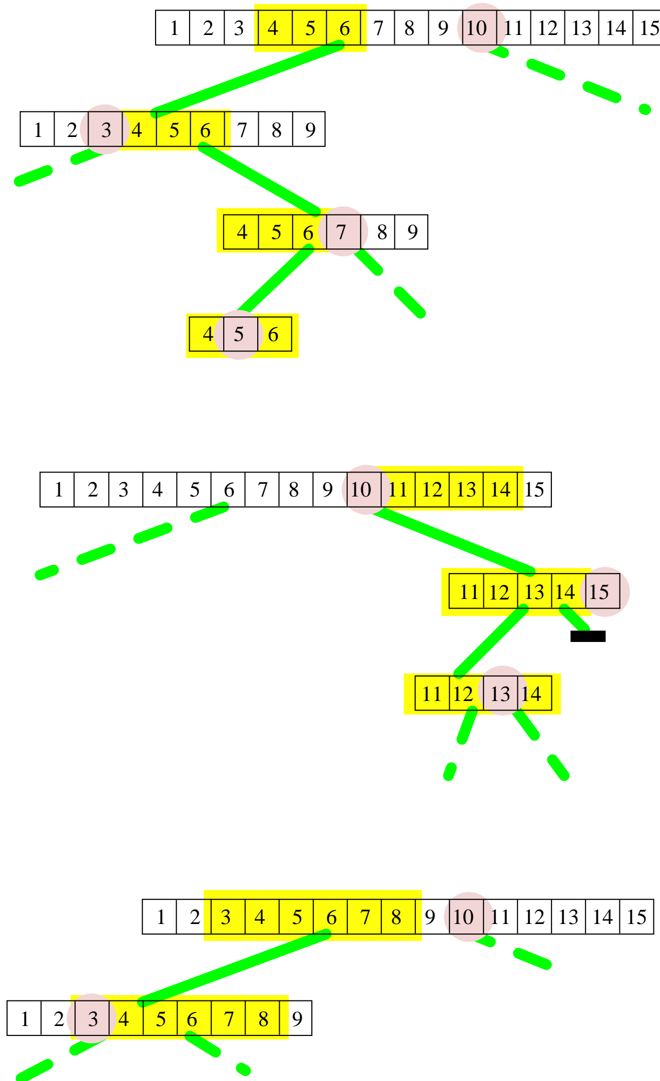

Abbildung 1.4.8: Verläufe für verschiedene Intervalle I_{ij}. Input ist (idealisiert) gleich $\{1, \ldots, 15\}$, also ist $I_{ij} = \{i, \ldots, j\}$.
Oben: $I_{4,6} = \{4, 5, 6\}$. $\mathbf{Pr}$(4 und 6 werden verglichen) $= \frac{2}{|\{4,5,6\}|} = \frac{2}{3}$. In diesem Beispiel (Pivot 5) passiert dies nicht, $X_{4,6} = 0$.
Mitte: $I_{11,14} = \{11, 12, 13, 14\}$. $\mathbf{Pr}$(11 und 14 werden verglichen) $= \frac{2}{|\{11,12,13,14\}|} = \frac{2}{4}$. In diesem Beispiel (Pivot 13) passiert dies nicht, $X_{11,14} = 0$.
Unten: $I_{3,8} = \{3, 4, 5, 6, 7, 8\}$. $\mathbf{Pr}$(3 und 8 werden verglichen) $= \frac{2}{|\{3,4,5,6,7,8\}|} = \frac{2}{6}$. In diesem Beispiel (Pivot 3) passiert dies, $X_{3,8} = 1$.

Wenn wir die „2“ auf Paare $i < j$ und $j < i$ verteilen, erhalten wir

$$\mathbf{E}(C) = \sum_{1 \le i \ne j \le n} \frac{1}{j-i+1}, \tag{1.4.15}$$

also ist $\mathbf{E}(C)$ die Summe aller Einträge in der folgenden Matrix:

$$\begin{pmatrix}
0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n-1} & \frac{1}{n} \\
\frac{1}{2} & 0 & \frac{1}{2} & \frac{1}{3} & \ddots & \cdots & \frac{1}{n-1} \\
\frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} & \ddots & & \vdots \\
\vdots & \frac{1}{3} & \frac{1}{2} & 0 & \ddots & & \frac{1}{4} \\
\vdots & & \ddots & \ddots & \ddots & & \frac{1}{3} \\
\frac{1}{n-1} & & & \ddots & \ddots & \ddots & \frac{1}{2} \\
\frac{1}{n} & \frac{1}{n-1} & \cdots & \cdots & \frac{1}{3} & \frac{1}{2} & 0
\end{pmatrix}.$$

Den Wert der Summe kann man leicht direkt hinschreiben, indem man entlang der (Neben-)Diagonalen addiert und die Symmetrie beachtet. Summand $\frac{1}{2}$ etwa kommt $2(n-1)$-mal vor, Summand $\frac{1}{3}$ kommt $2(n-2)$-mal vor, usw. Allgemein gilt: Summand $\frac{1}{k}$ kommt exakt $2(n-k+1)$-mal vor, für $2 \le k \le n$. Also gilt:

$$\mathbf{E}(C) = \left(\sum_{2 \le k \le n} \frac{2(n-k+1)}{k} \right) = 2(n+1) \cdot \sum_{2 \le k \le n} \frac{1}{k} - 2(n-1).$$

Wir setzen

$$H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}, \text{ für } n \ge 1 \quad (n\textit{-te harmonische Zahl}),$$

und erhalten

$$\mathbf{E}(C) = 2(n+1)(H_n - 1) - 2(n-1) = 2(n+1)H_n - 4n.$$

Man weiß, dass $\ln n < H_n < 1 + \ln n$ gilt.[14] Damit erhalten wir $\mathbf{E}(C) = 2n \ln n - O(n)$ oder $\mathbf{E}(C) = (2 \ln 2) n \log_2 n - O(n)$, wobei $2 \ln 2 = 1.386\ldots$ die „Quicksort-Konstante“ ist.

[14] Genaueres über Abschätzungen von H_n findet man z. B. in `https://www.tu-ilmenau.de/fileadmin/Bereiche/IA/al/AuD/AuD2021-komplett.pdf`, Kapitel 4, Folien 33/34. (Vorsicht: etwa 6MB.)

Man kann sogar den linearen Term genau bestimmen: Mit dem Wissen

$$H_n = \ln n + \gamma + \frac{1}{2n} - O(1/n^2),$$

wobei $\gamma = 0.5772156649\ldots$ die „Euler-Mascheroni-Konstante“ ist, erhält man

$$\mathbf{E}(C) = (2\ln 2)n\log_2 n - (4-2\gamma)n + 2\ln n + O(1),$$

mit $4 - 2\gamma \geq 2.845$.

Bemerkung: Was ist der Unterschied zwischen der Analyse von Quicksort mit deterministischer Pivotauswahl („immer $x = A[l + \lfloor (r-l)/2 \rfloor]$ wählen“) und der randomisierten Variante? Rein rechnerisch ergibt sich genau die gleiche Formel, wenn man annimmt, dass jede Anordnung der Eingabe die gleiche Wahrscheinlichkeit hat. Bei deterministischem Quicksort gibt es aber immer (worst-case-)Inputs, auf denen sich das Verfahren schlecht verhält, das heißt, Zeit $\Omega(n^2)$ benötigt. Bei der randomisierten Variante gilt dies nicht mehr: Auf jedem beliebigen Input $(a_1, \ldots, a_n)$ ist die *erwartete* Rechenzeit $O(n \log n)$. Quadratische Rechenzeiten treten auch hier auf, aber mit verschwindend kleiner Wahrscheinlichkeit, *für jeden beliebigen festen Input*. Die Randomisierung führt also zur Eliminierung von worst-case-Inputs.