Matrixmultiplikation von Strassen

Stephan Epp
hjstephan86@gmail.com

7. Juli 2025

Inhaltsverzeichnis

1 Laufzeit

Beim Algorithmus von Strassen für die Multiplikation von zwei $n \times n$ Matrizen lautet die Rekurrenzgleichung zur Ermittlung der Laufzeit $T(n)$ des Algorithmus

$$T(n) = 7\,T(\tfrac{n}{2}) + c\,n^2.$$

Der Algorithmus halbiert in jedem rekursiven Aufruf die beiden $n \times n$ Matrizen zu vier $\frac{n}{2} \times \frac{n}{2}$ Matrizen. Substituieren wir n durch $\frac{n}{2}$, erhalten wir für

$$T(\tfrac{n}{2}) = 7\,T(\tfrac{n}{4}) + c(\tfrac{n}{2})^2 = 7\,T(\tfrac{n}{4}) + \tfrac{c}{4}n^2.$$

Nach dem *ersten* rekursiven Aufruf erhalten wir mit $T(\frac{n}{2})$ eingesetzt in $T(n)$ dann

$$\begin{aligned} T(n) &= 7\,(7\,T(\tfrac{n}{4}) + \tfrac{c}{4}\,n^2) + cn^2 \\ &= 7^2\,T(\tfrac{n}{4}) + \tfrac{7}{4}\,cn^2 + cn^2. \end{aligned}$$

Mit dem *zweiten* rekursiven Aufruf werden die vier $\frac{n}{2} \times \frac{n}{2}$ Matrizen wieder halbiert zu acht $\frac{n}{4} \times \frac{n}{4}$ Matrizen. Damit ist

$$T(\tfrac{n}{4}) = 7\,T(\tfrac{n}{8}) + c(\tfrac{n}{4})^2 = 7\,T(\tfrac{n}{8}) + \tfrac{c}{16}n^2.$$

Wird $T(\frac{n}{4})$ eingesetzt in $T(n)$ ergibt sich

$$\begin{aligned} T(n) &= 7^2\,(7\,T(\tfrac{n}{8}) + \tfrac{c}{16}n^2) + \tfrac{7}{4}cn^2 + cn^2 \\ &= 7^3\,T(\tfrac{n}{2^3}) + \tfrac{7^2}{4^2}cn^2 + \tfrac{7}{4}cn^2 + cn^2. \end{aligned}$$

Betrachten wir nun den k-ten rekursiven Aufruf finden wir für

$$T(n) = 7^k\,T\left(\tfrac{n}{2^k}\right) + cn^2 \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i.$$

Zur Vereinfachung belassen wir es bei dem k-ten rekursiven Aufruf auch in $T(n)$ bei k und nicht $k+1$. Kleinere Matrizen als 1×1 Matrizen gibt es nicht, daher können die $n \times n$ Matrizen nur k mal halbiert werden. Der größte Wert, den k annehmen kann, ist $k = \log_2 n$. Damit ist

$$\begin{aligned} T(n) &= 7^{\log_2 n}\,T\Big(\frac{n}{2^{\log_2 n}}\Big) + cn^2 \sum_{i=0}^{\log_2 n - 1} \left(\tfrac{7}{4}\right)^i \\ &= n^{\log_2 7}\,T(1) + cn^2\,\frac{\left(\frac{7}{4}\right)^{\log_2 n} - 1}{\frac{7}{4} - 1} \\ &= O\left(n^{2.8074}\right) + cn^2\left(\tfrac{7}{4}\right)^{\log_2 n} \\ &= O\left(n^{2.8074}\right) + cn^2 \cdot n^{\log_2 \frac{7}{4}} \\ &= O\left(n^{2.8074}\right) + cn^{2.8074} \\ &= O\left(n^{2.8074}\right). \end{aligned}$$

2 Algorithmus

Es folgt der Algorithmus 1 von Strassen als Pseudocode. Als Eingabe erhalten wir zwei $n \times n$ Matrizen. Der Einfachheit halber wird angenommen, dass n eine Zweierpotenz ist. Zu Beginn prüfen wir, ob die Größe der Matrizen bereits den Wert 1 hat. Haben die Matrizen den Wert 1, geben wir das Produkt AB zurück. In Zeile 2 und 3 werden die Matrizen A und

Algorithmus 1 STRASSEN(A, B)

Eingabe: $\langle A, B\rangle$, mit $n \times n$ Matrizen A, B, $n = 2^k$, $k \in \mathbb{N}$
Ausgabe: $\langle C\rangle$, mit Produktmatrix $C = AB$

1: **if** $n = 1$ **then return** $C = AB$
2: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$
3: $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$
4: $P_1 = \text{STRASSEN}(A_{11} + A_{22}, B_{11} + B_{22})$
5: $P_2 = \text{STRASSEN}(A_{21} + A_{22}, B_{11})$
6: $P_3 = \text{STRASSEN}(A_{11}, B_{12} - B_{22})$
7: $P_4 = \text{STRASSEN}(A_{22}, B_{21} - B_{11})$
8: $P_5 = \text{STRASSEN}(A_{11} + A_{12}, B_{22})$
9: $P_6 = \text{STRASSEN}(A_{21} - A_{11}, B_{11} + B_{12})$
10: $P_7 = \text{STRASSEN}(A_{12} - A_{22}, B_{21} + B_{22})$
11: $C_{11} = P_1 + P_4 - P_5 + P_7$
12: $C_{12} = P_3 + P_5$
13: $C_{21} = P_2 + P_4$
14: $C_{22} = P_1 - P_2 + P_3 + P_6$
15: **return** $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

B so definiert, dass in den Zeilen 4 bis 10 die 5 Matrixmultiplikationen jeweils durchgeführt werden. In den Zeilen 11 bis 14 werden 4 Matrizen C_{ij} durch Addition und Subtraktion der Matrizen A_{ij} berechnet und in Zeile 15 Matrix C als Ergebnis zurückgegeben.

Als Idee zum Beweis der Korrektheit betrachten wir das Produkt $A \cdot B$ der zwei Matrizen A und B mit

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

Zur Vereinfachung beinhalten die Matrizen nur skalare Werte. Das Ergebnis $C = A \cdot B$ ist

$$C = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}.$$

Der Algorithmus von STRASSEN berechnet P_r mit

$$\begin{aligned}
P_1 = \text{STRASSEN}(a + d, e + h) &= ae + ah + de + dh, \\
P_2 = \text{STRASSEN}(c + d, e) &= ce + de, \\
P_3 = \text{STRASSEN}(a, f - h) &= af - ah, \\
P_4 = \text{STRASSEN}(d, g - e) &= dg - de, \\
P_5 = \text{STRASSEN}(a + b, h) &= ah + bh, \\
P_6 = \text{STRASSEN}(c - a, e + f) &= ce + cf - ae - af, \\
P_7 = \text{STRASSEN}(b - d, g + h) &= bg + bh - dg - dh.
\end{aligned}$$

Dann werden C_{ij} berechnet mit

$$\begin{aligned}
C_{11} = P_1 + P_4 - P_5 + P_7 &= ae + bg, \\
C_{12} = P_3 + P_5 &= af + bh, \\
C_{21} = P_2 + P_4 &= ce + dg, \\
C_{22} = P_1 - P_2 + P_3 + P_6 &= cf + dh,
\end{aligned}$$

wobei z.B. $C_{11} = (ae + ah + de + dh) + (dg - de) - (ah + bh) + (bg + bh - dg - dh) = ae + bg$. Für einen formalen Beweis der Korrektheit verzichten wir auf die vollständige Induktion über $n \in \mathbb{N}$ der $n \times n$ Matrizen.

3 Experimentelle Ergebnisse

Um die theoretische Laufzeitanalyse von Strassens Algorithmus zu überprüfen, wurde eine Python-Implementierung des Algorithmus erstellt und deren Performance mit der einer Standard-Matrixmultiplikation verglichen. Die Experimente wurden für Matrizen unterschiedlicher Größe (n) durchgeführt, wobei n von 4 bis 256 in Schritten von 4 variiert wurde. Für jede Matrixgröße wurden 5 Messungen (Trials) durchgeführt und die durchschnittliche Laufzeit ermittelt. Ein interner Schwellwert (threshold) von 32 wurde für Strassens Algorithmus festgelegt, was bedeutet, dass für Matrizen, deren Größe kleiner oder gleich 32 ist, auf die Standard-Matrixmultiplikation umgeschaltet wird, um den zusätzliche Aufwand der Rekursion zu reduzieren. Die Systemauslastung vor Beginn der Experimente betrug 0,0% CPU-Auslastung bei 3.38 GB verwendetem RAM von insgesamt 7.01 GB. Nach Abschluss der Experimente betrug die CPU-Auslastung 3.8% und der verwendete RAM 3.43 GB, was auf eine moderate Systemauslastung während der Messungen hinweist. Die Messergebnisse sind in Tabelle 1 zusammengefasst.

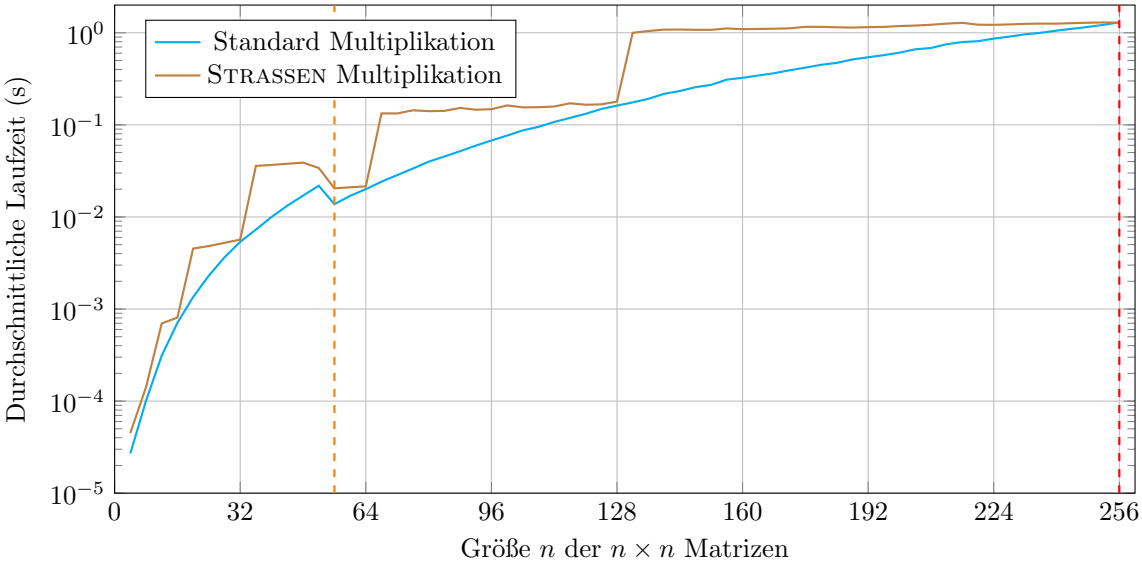
Tabelle 1: Vergleich der Matrixmultiplikationen

n	Durchschnittliche Laufzeit (s)	
	Standard	Strassen
4	0.000027	0.000045
8	0.000101	0.000142
12	0.000312	0.000698
16	0.000708	0.000808
20	0.001350	0.004538
24	0.002304	0.004827
28	0.003644	0.005236
32	0.005369	0.005674
36	0.007291	0.035827
40	0.010007	0.036722
44	0.013256	0.037815
48	0.017039	0.038888
52	0.021834	0.034057
56	0.013749	0.020460
60	0.016938	0.020990
64	0.020041	0.021484
⋮	⋮	⋮
240	1.055068	1.256394
244	1.105579	1.272495
248	1.158183	1.285980
252	1.221187	1.295652
256	1.298067	1.286444

Aus der Tabelle lässt sich ablesen, dass für kleine Matrizen (z.B. $n \leq 64$) die Standard-Matrixmultiplikation tendenziell schneller war oder eine vergleichbare Performance wie Strassens Algorithmus aufwies. Dies ist auf den zusätzlichen Aufwand der Rekursion und der zusätzlichen Matrixadditionen/-subtraktionen bei Strassens Algorithmus zurückzuführen. Insbesondere ist der Sprung in der Laufzeit des Algorithmus bei $n = 36$ (nach dem eingestellten Schwellenwert von $n = 32$) auffällig, was den Wechsel von der optimierten Basis-Multiplikation zur rekursiven Struktur widerspiegelt. Erst ab $n = 256$ übertrifft Strassen die Standardmultiplikation leicht. Die tatsächliche Laufzeit wird jedoch stark vom gewählten Schwellwert und der Effizienz der Implementierung der Basisfälle beeinflusst.

Die Abbildung 1 zeigt die Verläufe der benötigten Rechenzeit beider Matrixmultiplikationen. Für $n = 56$ waren sowohl die Standard als auch STRASSENs Multiplikation schneller als für vorherige $n < 56$.

Abbildung 1: Vergleich der Laufzeit der Matrixmultiplikationen



STRASSEN verwendet 7 Matrixmultiplikationen für zwei Matrizen der Größe $n \times n$. Lässt sich die Anzahl der benötigten Matrixmultiplikationen reduzieren?

4 Optimierung

Um die Anzahl der benötigten Matrixmultiplikationen von 7 auf 5 zu reduzieren, wird die Diagonale e und h der Matrix B betrachtet. Dadurch wird für n eine Laufzeit im Exponenten von 2.3219 statt 2.807 erreicht. Betrachtet man $P_2 = (c + d) \cdot e = ce + de$ und $P_5 = (a + b) \cdot h = ah + bh$, dann fällt auf, dass P_2 durch P_6 und P_1 ermittelt werden kann,

$$\begin{aligned} ce &= P_6 - cf + ae + af, \\ de &= P_1 - ae - ah - dh, \end{aligned}$$

und P_5 durch P_1 und P_7

$$\begin{aligned} ah &= P_1 - ae - de - dh, \\ bh &= P_7 - bg + dg + dh. \end{aligned}$$

Dazu müssen P_1, P_6 und P_7 berechnet werden bevor P_2 und P_5 ohne Multiplikation bestimmt werden können:

$$\begin{aligned} P_2 &= ce + de = (P_6 - cf + ae + af) + (P_1 - ae - ah - dh) = P_6 + P_1 - cf + af - ah + af, \\ P_5 &= ah + bh = (P_1 - ae - de - dh) + (P_7 - bg + dg + dh) = P_1 + P_7 - ae - de - bg + dg. \end{aligned}$$

Da nur 5 Matrixmultiplikationen verwendet werden, ergibt sich für die Laufzeitanalyse

$$T(n) = 5^{\log_2 n}\, T\Big(\frac{n}{2^{\log_2 n}}\Big) + cn^2 \sum_{i=0}^{\log_2 n - 1} \left(\tfrac{5}{4}\right)^i = n^{\log_2 5}\, T(1) + cn^2 \frac{\left(\frac{5}{4}\right)^{\log_2 n} - 1}{\frac{5}{4} - 1} = O\left(n^{2.3219}\right).$$

Dazu wird der Algorithmus von STRASSEN zu STRASSEN-25 so geändert, dass P_2 und P_5 in Zeile 9 und 10 ermittelt werden. Damit P_2 und P_5 ermittelt werden können, müssen die zuvor ermittelten Produkte P_r, $r \in \{1, 6, 7\}$ verwendet werden. Das bedeutet für

$$P_2 = P_6 + P_1 - A_{21}B_{12} + A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{12},$$
$$P_5 = P_1 + P_7 - A_{11}B_{11} - A_{22}B_{11} - A_{21}B_{21} + A_{22}B_{21}.$$

Dabei ist P_2 *abhängig* von P_6 und P_7 und P_5 ist *abhängig* von P_1 und P_7. Damit P_2 und P_5 ohne Matrixmultiplikation ermittelt werden können, müssen die Produkte $A_{ij}B_{ij}$ im vorherigen Rekursionsschritt jeweils gespeichert werden. Damit in jedem Rekursionsschritt

Algorithmus 2 STRASSEN-25(A, B)

Eingabe: $\langle A, B\rangle$, mit $n \times n$ Matrizen A, B, $n = 2^k$, $k \in \mathbb{N}$
Ausgabe: $\langle C, A_{ij}, B_{ij}\rangle$, mit Produktmatrix $C = AB$ und A_{ij}, B_{ij}

1: **if** $n = 1$ **then return** $C = AB$
2: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$
3: $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$
4: $P_1 =$ STRASSEN-25$(A_{11} + A_{22}, B_{11} + B_{22})$
5: $P_3 =$ STRASSEN-25$(A_{11}, B_{12} - B_{22})$
6: $P_4 =$ STRASSEN-25$(A_{22}, B_{21} - B_{11})$
7: $P_6 =$ STRASSEN-25$(A_{21} - A_{11}, B_{11} + B_{12})$
8: $P_7 =$ STRASSEN-25$(A_{12} - A_{22}, B_{21} + B_{22})$
9: $P_2 = P_6 + P_1 - A_{21}B_{12} + A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{12}$
10: $P_5 = P_1 + P_7 - A_{11}B_{11} - A_{22}B_{11} - A_{21}B_{21} + A_{22}B_{21}$
11: $C_{11} = P_1 + P_4 - P_5 + P_7$
12: $C_{12} = P_3 + P_5$
13: $C_{21} = P_2 + P_4$
14: $C_{22} = P_1 - P_2 + P_3 + P_6$
15: **return** $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

die Produkte $A_{ij}B_{ij}$ effizient gespeichert werden können, muss auf eine Datenstruktur mit einem eindeutigen Schlüssel (k, i_A, j_A, i_B, j_B) in $O(1)$ zugegriffen werden, $n = 2^k$, $k \in \mathbb{N}$.

Satz 4.1. *Es gibt keinen Teile-und-Herrsche Algorithmus zur Multiplikation von $n \times n$ Matrizen, der durch Nutzen von Abhängigkeiten weniger als 8 Matrixmultiplikationen benötigt und damit auch weniger als 5 Matrixmultiplikationen benötigt.*

Beweis. Angenommen, es gibt einen Teile-und-Herrsche Algorithmus zur Multiplikation von $n \times n$ Matrizen, der mit weniger als 5 Matrixmultiplikationen auskommt, dann muss dieser Algorithmus auch Abhängigkeiten $P_r = (P_s, P_t, A_{ij}B_{ij})$ verwenden, mit denen P_r ermittelt werden kann, $r < 5$. Da $dim(A_{ij}) = dim(B_{ij}) = \frac{n}{2}$, ist es nicht möglich $C = AB$ zu berechnen. Das gesamte Bild der *projezierten* Produktmatrix C kann nur projeziert werden, wenn mindestens mehr als die Hälfte beider Matrizen A_{ij} und B_{ij} zum Projezieren verfügbar ist. □

Satz 4.2. *Das optimale Teile-und-Herrsche Prinzip ist die effizienteste Methode, um Probleme zu lösen, wenn die Probleminstanz bei jedem rekursiven Aufruf halbiert wird.*

Beweis. In diesen drei Kriterien ist der Beweis zu führen: (1) Rekursionstiefe, (2) Anzahl der Teilprobleme, (3) Größe eines Teilproblems. Dabei wird schnell klar, dass es alles in der Halbierung zusammengefasst optimal ist. Denn die Halbierung ist nichts anderes als eine Verdoppelung im Nenner.

Zu (1): Wichtig in der Analyse der Laufzeit eines Algorithmus oder einer Lösungsvorschrift, bestehend aus einfachen Operationen und dem Aufruf von Funktionen, ist das Betrachten der gerufenen Funktionen. Weniger wichtig sind einfache Operationen und Zuweisungen. Daher ist für die Laufzeitanalyse die Rekursionstiefe von wichtigster Bedeutung (sieht man z.B. beim Algorithmus von STRASSEN oder beim Algorithmus MERGE-SORT).

Die Rekursionstiefe ist logarithmisch. Kleiner kann die Tiefe für die gesamte Lösung nicht sein, da das Problem mit der 2 im Nenner logarithmisch geteilt wird. Die Funktion, die am stärksten wächst und dabei immer noch eine Umkehrfunktion hat, ist die Exponentialfunktion. Daher ist ihre Umkehrfunktion am schnellsten in der Reduzierung der Rekursionstiefe für die Laufzeitanalyse für Teile-und-Herrsche Algorithmen. Eine Erhöhung des Nenners führt zwangsläufig zu einer Veränderung der anderen beiden Kriterien.

Zu (2), (3): Für die Anzahl der Teilprobleme und für die Größe eines Teilproblems gilt: Angenommen, der Nenner wird vergrößert. Dann wird die ursprüngliche Probleminstanz aber nicht in ihrem leichtesten Punkt geteilt. Denn ein Problem ist nur da am leichtesten zu teilen, wo sein Schwerpunkt liegt. Es gibt für jede Probleminstanz nur einen Schwerpunkt. Wenn der Nenner ganzzahlig verkleinert wird, wird nicht mehr geteilt. Es muss ganzzahlig verkleinert werden, da z.B. ein einzelnes Array-Element nicht teilbar ist. Die Zahl e scheidet damit in der Wahl der Exponentialfunktion aus. Daher muss die Exponentialfunktion 2 als Basis haben. □

Auffällig ist, dass nur das Problem, bei dem zwei Instanzen durch einen Operator verbunden sind, mit dem optimalen Teile-und-Herrsche Prinzip zu lösen ist. Das sind Probleme mit Operatoren wie z.B., $+$, $-$, $<$. Im Prinzip werden diese Probleme an ihrem Schwerpunkt gelöst, wo sie trotz ihrer Schwere am leichtesten zu tragen sind, wie bei einer Wippe mit zwei gleichen Gewichten jeweils am äußersten Ende. Die Wippe ist dann exakt horizontal ausgerichtet im Gleichgewicht.

Lemma 4.1. *Schneller als mit $T(n) = O(n^{2.3219})$ können zwei $n \times n$ Matrizen nicht multipliziert werden.*

Beweis. Der Beweis folgt daraus, dass STRASSEN-25(A, B) zwei $n \times n$ Matrizen nach dem optimalen Teile-und-Herrsche Prinzip multipliziert. □

Satz 4.3. *Wenn es bei einer Lösung erforderlich ist, jedes Element der Eingabe während der Abarbeitung der Lösungsvorschrift einmal zu betrachten, dann gibt es keine bessere Laufzeit als die logarithmische.*

Beweis. Wenn Laufzeit so aufgefasst wird, dass in jedem Schritt ein Register besucht wird, dann kann die Anzahl der unterschiedlich besuchten Register nicht weniger als logarithmisch sein. Die Sonnenblumenkerne in einer Sonnenblume sind logarithmisch angeordnet und damit optimal untergebracht mit minimal wenig Platz. Sie sind im goldenen Winkel angebracht. Würden sie besser angebracht werden können, dann wäre der goldene Winkel kein goldener Winkel. □

Lemma 4.2. *Das optimale Teile-und-Herrsche Prinzip ist optimal unter allen Lösungsprinzipien, wo es während der Abarbeitung der Lösungsvorschrift erforderlich ist, jedes Element der Eingabe während der Lösung einmal zu betrachten.*

Beweis. Der Beweis folgt daraus, dass Algorithmen, die nach dem optimalen Teile-und-Herrsche Prinzip Lösungsvorschriften abarbeiten, logarithmische Laufzeit haben müssen. □

4.1 Komplexitätsklassen

Algorithmen folgen einem Lösungsprinzip nach dem sie ein Problem lösen. Es gibt zum Beispiel das optimale Teile-und-Herrsche Prinzip oder das Prinzip der dynamischen Programmierung. Es kann sich lohnen, die Klasse **P** in Komplexitätsklassen einzuteilen, bei

der eine Klasse durch ein Lösungsprinzip und ihre charakteristische Laufzeit beschrieben wird. Zum Beispiel kann man sich die Klasse vorstellen, in der alle Probleme enthalten sind, die nach dem optimalen Teile-und-Herrsche Prinzip zu lösen sind und nicht besser. Da die charakteristische Laufzeit dieser Klasse die logarithmische ist, könnte sie den Namen **L** haben. Man kann sich auch die Klasse $\mathbf{P_2}$ vorstellen, in der alle Probleme enthalten sind, die nach dem Prinzip der dynamischen Programmierung zu lösen sind und für deren Lösung quadratische Laufzeit benötigt wird.

Definiton 4.1. *Die Klasse* **P** *enthält die Menge aller Probleme, für die es jeweils einen Algorithmus gibt, der das Problem in polynomieller Laufzeit löst.*

Definiton 4.2. *Die Klasse* **NP** *enthält die Menge aller Probleme, von denen bekannt ist, dass es für sie jeweils keinen Algorithmus gibt, der das Problem noch in polynomieller Laufzeit lösen kann.*

Bisher hat man vermutet, dass zwischen **P** und **NP** eine Beziehung derart existiert, dass $\mathbf{P} \subseteq \mathbf{NP}$. Diese Vermutung ist aber nicht richtig.

Satz 4.4. *Die Klassen* **P** *und* **NP** *sind gleich, d.h.,* $\mathbf{P} = \mathbf{NP}$.

Beweis. Betrachte das Problem **S**, die eindeutige Schneeflocke. Gegeben ist eine Schneeflocke mit 5 gleichlangen Armen. Wie wird diese Schneeflocke eindeutig unter allen bereits vorhandenen Schneeflocken? Finde einen Algorithmus, der in polynomieller Zeit immer eine neue Schneeflocke erzeugt. Beweis durch Induktion:
IB: Sei S die Menge aller eindeutigen Schneeflocken, o.B.d.A.
IA: $|S| = 0$ trivial, $|S| = 1$ trivial
IS: $|S| \to |S| + 1$, $|S| = n, n \in \mathbb{N}$
Ziel: Mache aus 8 Knoten wieder 6 Knoten mit gleicher Kantenlänge in polynomieller Zeit. Berechne dazu die Restklasse 6. Jeder Knoten berechnet eine Funktion $f(b_1, \ldots, b_n) = \{0, \ldots, 9\}$, $b_i \in \{0, 1\}$. Bei Hinzufügen zwei neuer Knoten erhöht sich die Wertemenge des vorhandenen Graphen um maximal $18 \equiv 0 \mod 6$.
Die Vermutung liegt nahe, das Problem **S** als NP-vollständig einzuschätzen. Da dieses Problem aber in polynomieller Zeit lösbar ist, ist es echt in **P**. □

Da $\mathbf{P} = \mathbf{NP}$, lohnt es sich gerade auch die Probleme genauer zu betrachten, von denen man vorher vermutete, dass es für sie keinen Algorithmus gibt, der das Problem in polynomieller Laufzeit lösen kann.

Vielleicht kann man in Zukunft ein Problem der Klasse $\mathbf{P_2}$ auch effizienter lösen und damit der Klasse **L** zuordnen. Man kann sich die Klasse $\mathbf{P_3}$ vorstellen, in der alle Probleme enthalten sind, die nach dem Prinzip der dynamischen Programmierung zu lösen sind und für deren Lösung kubische Laufzeit benötigt wird (die Lösungstabelle hat eine dritte Dimension). Auch hier kann es sein, dass für ein Problem dieser Klasse eine effizientere Lösung gefunden wird und sie damit einer leichteren Komplexitätsklasse zugeordnet werden kann. Da ein Problem auch immer weniger effizient, d.h., langsamer, gelöst werden kann, ist es nicht so, dass z.B. ein Problem der Klasse **L** auch der Klasse $\mathbf{P_2}$ zuzuordnen ist, da man Probleme in der Praxis nicht langsamer löst. Alle Klassen sind in **P** enthalten aber untereinander nicht:

$$\mathbf{L} \subset \mathbf{P}, \quad \mathbf{P_2} \subset \mathbf{P}, \quad \mathbf{P_3} \subset \mathbf{P}, \quad \mathbf{L} \not\subset \mathbf{P_2}, \quad \mathbf{P_2} \not\subset \mathbf{P_3}, \quad \mathbf{L} \not\subset \mathbf{P_3}.$$

Beispiele für Probleme, die in **L** enthalten sind, sind die $n \times n$ Matrixmultiplikation und das Sortieren von ganzen Zahlen. Beide Probleme lassen sich mit dem optimalen Teile-und-Herrsche Prinzip lösen.

4.2 Experimentelle Ergebnisse

Bei der Analyse der zu messenden Laufzeit von Algorithmus Strassen-25 ist zu erwarten, dass dieser mit nur 5 Matrixmultiplikationen deutlich weniger Rechenzeit benötigt als der Algorithmus von Strassen mit 7 Matrixmultiplikationen.

Die Tabelle 2 zeigt die theoretische Anzahl benötigter Operationen für die Matrixmultiplikation in Abhängigkeit von n. Die Operationen sind elementare arithmetische Operationen wie Multiplikationen und Additionen/Subtraktionen von Skalaren. Die Anzahl der Operationen in der Tabelle sind asymptotisch und ignorieren konstante Faktoren, um die Skalierungseffekte zu betonen. Die Wahl des optimalen Algorithmus zur Matrixmultiplikation hängt

Tabelle 2: Vergleich der theoretischen Anzahl an Operationen

n	Standard	Strassen	Strassen-25	Matrixgröße $(n \times n)$
10	$10^3 = 1.000$	$10^{2.8074} \approx 642$	$10^{2.3219} \approx 209$	100
100	$100^3 = 10^6$	$100^{2.8074} \approx 6.4 \times 10^5$	$100^{2.3219} \approx 2.09 \times 10^4$	10^4
1.000	$1.000^3 = 10^9$	$1.000^{2.8074} \approx 6.4 \times 10^8$	$1.000^{2.3219} \approx 2.09 \times 10^7$	10^6
10.000	$10.000^3 = 10^{12}$	$10.000^{2.8074} \approx 6.4 \times 10^{11}$	$10.000^{2.3219} \approx 2.09 \times 10^9$	10^8
100.000	$100.000^3 = 10^{15}$	$100.000^{2.8074} \approx 6.4 \times 10^{14}$	$100.000^{2.3219} \approx 2.09 \times 10^{11}$	10^{10}
1.000.000	$1.000.000^3 = 10^{18}$	$1.000.000^{2.8074} \approx 6.4 \times 10^{17}$	$1.000.000^{2.3219} \approx 2.09 \times 10^{13}$	10^{12}
10^7	10^{21}	6.4×10^{20}	2.09×10^{15}	10^{14}
10^8	10^{24}	6.4×10^{23}	2.09×10^{17}	10^{16}

stark von der Größe der Matrizen und dem spezifischen Anwendungsbereich ab. Interessant ist, dass schon für $n = 100$ nur 10^4 Operationen benötigt werden von Strassen-25 und dagegen 10^5 Operationen von Strassen. Für $n \geq 10.000$ unterscheiden sich Strassen-25 und Strassen um 2 Zehnerpotenzen und für $n \geq 10^8$ unterscheiden sie sich um 6 Zehnerpotenzen. Praktische Anwendungsbereiche für die $n \times n$ Matrixmultiplikation sind die folgenden:

1. **Kleine Matrizen** ($n < 200$): Für kleinere Matrizen, wie sie in der Computergrafik oder bei einfachen linearen Gleichungssystemen vorkommen, ist der Standardalgorithmus mit $O(n^3)$ aufgrund seines geringen zusätzlichen Aufwands für die Rekursion und optimaler Cache-Nutzung die erste Wahl.

2. **Mittlere bis große Matrizen** ($n \approx 200 - 10.000$): Im wissenschaftlichen Rechnen, bei Optimierungsproblemen oder im maschinellen Lernen dominieren weiterhin optimierte Implementierungen des Standardalgorithmus.

3. **Sehr große Matrizen** ($n \geq 10.000$): Bei großen Matrizen, wie sie in Big Data Analysen oder beim Training umfangreicher Deep-Learning-Modelle auftreten, wird der Bedarf an Alternativen zu optimierten Standardalgorithmen deutlich. Damit gewinnt der Algorithmus von Strassen an Attraktivität und Relevanz für die Bewältigung rechenintensiver Multiplikationen.

5 Signalübertragung

Für die Signalübertragung ist es wichtig, das Signal vom Sender zum Empfänger sicher zu übertragen. Welche kleinste Einheit eignet sich dabei als die optimale? Bisher betrachtet wurde: Es gibt die 2 zum Teilen beim optimalen Teile-und-Herrsche Prinzip. Es gibt die Exponentialfunktion zur Basis 2. Es gibt die logarithmische Laufzeit. Darüber hinaus gibt es das Binärsystem, bei dem die kleinste Einheit durch 2 Zustände repräsentiert wird.

Satz 5.1. *Das Binärsystem ist optimal für die Signalübertragung, bei der elektronische Signale genutzt werden: für an (1) und für aus (0).*

Beweis. Mit einer 2 Bitfolge können 4 Werte übertragen werden, mit einer 3 Bitfolge können 8 Werte übertragen werden. Die Anzahl der übertragbaren Werte ist exponentiell zur Basis 2 in der Anzahl der Bits in der zu übertragenden Bitfolge. Das heißt, die Anzahl an Bits ist logarithmisch zur Wertigkeit, die übertragbar ist. Warum ist 3 oder 4 als Basis nicht besser? Es ist deshalb nicht besser, da man vom Querschnitt der Leitung aus betrachtet so einen maximalen Abstand zwischen (0) und (1) hat zur sicheren, lesbaren Signalübertragung und Störungen (wie durch Rauschen) so maximal entgegengewirkt wird. In der Mitte des Leitungsquerschnitts ist der Schwerpunkt der Leitung. Nur bei 2 Zuständen (0) und (1) mit gleichem Abstand zum Schwerpunkt der Leitung ist der Abstand aller Signale voneinander (maximal) und zum Schwerpunkt der Leitung (minimal) optimal. □

Das heißt, der Leitungsquerschnitt wird somit so klein wie möglich gehalten und damit ist so wenig Leitungsmaterial wie nötig erforderlich. Es ist aus wirtschaftlicher und fairer Sicht nicht vertretbar, städtische Haushalte mit einem anderen als dem Binärsystem für die Signalübertragung zu versorgen. Zugleich verfügen ländliche Gebiete oft nicht über dieselbe Internetbandbreite wie Städte. Die Debatte um die Nutzung von Quantenphysik für die Signalübertragung erübrigt sich somit unter wirtschaftlichen und gerechten Gesichtspunkten. Solange die Quantenphysik nicht für die Signalübertragung eingesetzt wird, ist auch die Verarbeitung von Signalen bzw. Daten mittels Quantenphysik nicht sinnvoll.

Die *Shannon-Hartley-Gleichung* beschreibt die maximale theoretische Datenrate, d.h., *Kanalkapazität C*, die über einen Kommunikationskanal fehlerfrei übertragen werden kann, unter Berücksichtigung von Bandbreite und Rauschen. Sie lautet:

$$\mathrm{C} = \mathrm{B} \cdot \log_2\left(1 + \frac{\mathrm{S}}{\mathrm{N}}\right), \quad \text{dabei ist}$$

C Kanalkapazität in Bit pro Sekunde (bit/s)

B Bandbreite des Kanals in Hertz (Hz)

S Durchschnittliche Signalleistung über den Kanal

N Durchschnittliche Rauschleistung über den Kanal

S/N Signal-Rausch-Verhältnis (SNR)

Das Signal-Rausch-Verhältnis (SNR) ist dabei der Quotient aus der durchschnittlichen Leistung zur Signalübertragung und der durchschnittlichen Rauschleistung im Kanal:

$$\mathrm{S/N} = 10 \cdot \log_{10}\left(\frac{\mathrm{P_{Sign}}}{\mathrm{P_{Nois}}}\right).$$

Beträgt die Signalleistung zum Beispiel $\mathrm{P_{Sign}} = 100\mathrm{mW}$ und die Rauschleistung $\mathrm{P_{Sign}} = 1\mathrm{mW}$, dann ist

$$\mathrm{S/N} = 10 \cdot \log_{10}\left(\frac{100\mathrm{mW}}{1\mathrm{mW}}\right) = 20.$$

Das Signal-Rausch-Verhältnis wird nicht in dB angegeben, da es nur ein Verhältnis beschreibt, in dem zwei Leistungen, die Signalleistung und die Rauschleistung, zueinander stehen. Tabelle 3 zeigt typische SNR-Wertebereiche. Bei einem SNR, das größer als 40 ist, ist eine optimale Signalübertragung möglich. Hochpräzise Messsysteme und Spezialanwen-

Tabelle 3: Typische SNR-Wertebereiche und deren Bedeutung für die Signalqualität

SNR	Qualitäts-einstufung	Beschreibung / Typische Auswirkungen	Anwendungsbeispiele (Tendenz)
< 0	Unbrauchbar	Rauschleistung überwiegt Signalleistung deutlich. Keine Kommunikation möglich oder extrem gestört.	Sehr schlechte/keine Mobilfunkverbindung, gestörtes WLAN am Rand.
$0 - 5$	Sehr schlecht	Signal kaum von Rauschen zu unterscheiden. Häufige Abbrüche, extrem langsame/unzuverlässige Datenübertragung.	Schwaches WLAN, sehr schlechter Radioempfang.
$5 - 10$	Schlecht	Verbindung instabil, Störungen. Datenraten stark eingeschränkt, viele Wiederholungen nötig.	Grundlegende Internetnutzung (E-Mails), VoIP mit Aussetzern.
$10 - 15$	Akzeptabel	Brauchbare Verbindung. Leichte Qualitätseinbußen, gelegentliche Verzögerungen. Für Grundaufgaben ausreichend.	Web-Browse, normale Videoanrufe, nicht-kritisches Streaming.
$15 - 25$	Gut	Stabile, zuverlässige Verbindung mit guter Qualität. Für die meisten Anwendungen ausreichend.	HD-Streaming, Online-Gaming (Casual), allgemeine Netzwerkaktivitäten.
$25 - 40$	Sehr Gut	Exzellente Bedingungen, kaum Störeinflüsse. Hohe Datenraten und sehr stabile Verbindungen.	4K-Streaming, anspruchsvolles Online-Gaming, professionelle Audio/Video, Unternehmensnetzwerke.
> 40	Optimal	Außergewöhnlich klare, störungsfreie Signale. Ideal für kritische Anwendungen, höchste Signalintegrität.	Hochpräzise Messsysteme, Laborequipment, Spezialanwendungen.

dungen nutzen dieses Verhältnis. Bei einem SNR, das zwischen 25 und 40 liegt, ist die Qualität für die Signalübertragung sehr gut und ermöglicht z.B. 4K-Streaming. Bei einem SNR zwischen 10 und 15 ist die Qualität akzeptabel für normale Videoanrufe oder nicht-kritisches Streaming.

6 Der BMM-Witnesses-Algorithmus: Wege finden in Graphen

Mit dem Algorithmus von Strassen-25 lassen sich $n \times n$ Matrizen effizient multiplizieren. Matrizen sind nicht nur für arithmetische Berechnungen nützlich; sie spielen auch eine zentrale Rolle in der Graphentheorie. Insbesondere können damit Fragen zur Erreichbarkeit in Graphen beantworten. Man kann sich vorstellen, man hat eine Karte (einen Graphen) und möchten wissen, ob man von Punkt A nach Punkt B gelangen kann. Der BMM-

Witnesses-Algorithmus (Boolean Matrix Multiplication with Witnesses) hilft dabei, solche Verbindungen zu finden und sogar zu identifizieren, über welchen Zwischenpunkt eine solche Verbindung verläuft.

Im Gegensatz zur üblichen Matrixmultiplikation, bei der wir Produkte addieren, verwenden wir hier logische Operationen. Anstelle der Multiplikation ($\times$) wird die logische AND-Operation ($\wedge$) und anstelle der Addition ($+$) die logische OR-Operation ($\vee$) genutzt. Die Elemente der Matrizen sind dabei binäre Werte, also entweder 0 (aus) oder 1 (an).

Seien A und B zwei $n \times n$ Boolesche Matrizen. Das Produkt $C = A \cdot B$ ist dann eine $n \times n$ Matrix, wobei jedes Element C_{ij} wie folgt berechnet wird:

$$C_{ij} = \bigvee_{k=1}^{n} (A_{ik} \wedge B_{kj}). \tag{6.1}$$

Ein Wert $C_{ij} = 1$ bedeutet, dass es einen Pfad von Knoten i zu Knoten j der Länge 2 gibt. Das liegt daran, dass $A_{ik} = 1$ bedeutet, es gibt eine Kante von i nach k, und $B_{kj} = 1$ bedeutet, es gibt eine Kante von k nach j. Wenn beides wahr ist ($A_{ik} \wedge B_{kj} = 1$), dann existiert ein Pfad von $i \to k \to j$. Die OR-Operation ($\bigvee$) sorgt dafür, dass $C_{ij} = 1$ ist, wenn es mindestens einen solchen Zwischenknoten k gibt.

6.1 Die Rolle des "Witness"(Zeugen)

Der BMM-Witnesses-Algorithmus geht über die reine Feststellung der Erreichbarkeit hinaus. Er liefert zusätzlich einen *Zeugen* für jede gefundene Verbindung. Ein Zeuge für einen Pfad $i \to k \to j$ ist der Zwischenknoten k. Der Algorithmus erstellt eine dritte Matrix, die *Witness-Matrix* W, in der W_{ij} den Index des ersten gefundenen Zwischenknotens k speichert, für den $A_{ik} \wedge B_{kj} = 1$ ist. Wenn kein solcher Pfad existiert ($C_{ij} = 0$), dann ist $W_{ij} = 0$.

Dies ist besonders nützlich, um nicht nur zu wissen, ob ein Pfad existiert, sondern auch wie man dorthin gelangt. Im Kontext von Graphen kann man so die direkten Nachfolger auf einem 2-Schritt-Pfad identifizieren. Der Algorithmus zur Berechnung des Booleschen Matrixprodukts C und der Witness-Matrix W ist relativ einfach und ähnelt der Standard-Matrixmultiplikation, nur mit angepassten Operationen.

Algorithmus 3 BMM-Witnesses(A, B)

Eingabe: $\langle A, B \rangle$, mit $n \times m$ Matrix A und $m \times p$ Matrix B. Die Elemente sind 0 oder 1.
Ausgabe: $\langle C, W \rangle$, mit Produktmatrix C ($n \times p$) und Witness-Matrix W ($n \times p$)

```
1: Initialisiere C als n × p Matrix mit Nullen
2: Initialisiere W als n × p Matrix mit Nullen
3: for i = 1 to n do
4:     for j = 1 to p do
5:         for k = 1 to m do
6:             if A_ik = 1 ∧ B_kj = 1 then
7:                 C_ij = 1
8:                 if W_ij = 0 then          ▷ Speichere den ersten gefundenen Zeugen
9:                     W_ij = k + 1          ▷ Oft k + 1 um 0 als kein Zeuge zu nutzen
10: return ⟨C, W⟩
```

Der Algorithmus durchläuft alle möglichen Tripel (i, j, k). Wenn eine Kante von i nach k und eine Kante von k nach j existiert ($A_{ik} = 1$ und $B_{kj} = 1$), dann wird C_{ij} auf 1 gesetzt. Gleichzeitig wird der Index k (hier als $k + 1$ gespeichert, um 0 als Default-Wert für "kein Zeugeßu reservieren) in W_{ij} eingetragen, sofern noch kein Zeuge für diese Position gefunden wurde. Die Laufzeit dieses Algorithmus ist $O(n \cdot m \cdot p)$, also $O(n^3)$ für quadratische Matrizen, ähnlich der Standard-Matrixmultiplikation.

Der BMM-Witnesses-Algorithmus findet breite Anwendung in Bereichen wie:

- Graphentheorie: Bestimmung der Transitivität (Existenz von Pfaden) in Graphen und Auffinden von Gliedern in diesen Pfaden.

- Soziale Netzwerkanalyse: Identifizierung von Personen, die indirekt miteinander verbunden sind, und über welche gemeinsamen Kontakte diese Verbindung zustande kommt.

- Routenplanung: In einfachen Fällen kann er helfen, Zwischenstationen auf Routen zu finden.

- Datenbanken: Abfragen von indirekten Beziehungen zwischen Entitäten.

Während Strassens Algorithmus die Laufzeit für die arithmetische Matrixmultiplikation verbessert, ist der BMM-Witnesses-Algorithmus ein fundamentales Werkzeug, das die Mächtigkeit von Matrizen für logische Schlussfolgerungen und die Analyse von Beziehungen in komplexen Netzwerken demonstriert.

6.2 Zeugen für Boolesche Matrixmultiplikation

Seien A und B zwei $n \times n$ Boolesche Matrizen, deren Einträge entweder 0 oder 1 sind. Das Boolesche Produkt P von A und B ist eine Matrix, deren Einträge p_{ij} wie folgt definiert sind:

$$p_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}).$$

Hierbei repräsentieren $\bigvee$ und $\wedge$ die Booleschen Operatoren ODER und UND. Dies bedeutet, dass $p_{ij} = 1$ genau dann ist, wenn es mindestens ein k gibt, für das sowohl $a_{ik} = 1$ als auch $b_{kj} = 1$ gilt.

6.2.1 Berechnung des Booleschen Produkts

Das Boolesche Produkt $P = AB$ kann effizient berechnet werden. Man kann die 0/1-Einträge als ganze Zahlen behandeln und ein herkömmliches Matrixprodukt M (mit ganzen Zahlen) berechnen. Dann ist $p_{ij} = 1$ genau dann, wenn der entsprechende Eintrag m_{ij} in der Matrix M größer als 0 ist. Die Zeitkomplexität für diese Berechnung liegt bei $O(M(n))$, wobei $M(n)$ die Zeit ist, die zur Multiplikation zweier $n \times n$ Matrizen benötigt wird, was wiederum $o(n^3)$ ist, wenn man Strassens Algorithmus oder schnellere Methoden verwendet.

6.2.2 Die Notwendigkeit von Zeugen

In einigen Anwendungen ist es nicht ausreichend zu wissen, dass $p_{ij} = 1$. Man möchte zusätzlich einen Index k finden, für den $a_{ik} = 1$ und $b_{kj} = 1$ gilt. Dieser Index k wird als *Zeuge* bezeichnet. Eine Zeugenmatrix W für $P = AB$ ist eine Matrix, deren Einträge w_{ij} wie folgt definiert sind:

$$w_{ij} = \begin{cases} 0 & \text{falls } p_{ij} = 0 \\ k \text{ mit } a_{ik} = b_{kj} = 1 & \text{falls } p_{ij} = 1 \end{cases}$$

Die Schwierigkeit besteht darin, eine solche Zeugenmatrix in subkubischer Zeit zu berechnen, da ein naiver Ansatz, der jedes k für jedes Paar (i, j) überprüft, zu einer Laufzeit von $O(n^3)$ führen würde.

6.2.3 Ansatz bei eindeutigem Zeugen

Angenommen, für ein Paar (i, j) mit $p_{ij} = 1$ gibt es einen eindeutigen Zeugen k_{ij}. In diesem speziellen Fall könnte man eine modifizierte Matrix $\hat{A}$ definieren, indem man $\hat{a}_{ik} = k \cdot a_{ik}$ setzt. Das (i, j)-Element des Produkts $\hat{A}B$ wäre dann k_{ij}, also der korrekte Zeuge. Wenn der Zeuge jedoch nicht eindeutig ist, würde der Eintrag im Produkt $\hat{A}B$ "Müllënthalten, und man könnte den Index nicht direkt bestimmen. Ein einfacher Ansatz, $\hat{a}_{ik} = 2^k a_{ik}$ zu verwenden, um alle Zeugen zu identifizieren, würde zu Problemen mit der Größe der Zahlen führen, da dann n-Bit-Zahlen verwendet würden und die Annahme von $O(1)$ Zeit für Ganzzahloperationen nicht mehr gültig wäre.

6.2.4 Randomisierter Ansatz für allgemeine Zeugen

Eine allgemeine Lösung für das Finden von Zeugen kann mithilfe von Randomisierung erreicht werden. Sei w_{ij} die Anzahl der Zeugen für $p_{ij} = 1$. Wir definieren eine modifizierte Matrix $\hat{A}$ mit $\hat{a}_{ik} = r_k \cdot a_{ik}$, wobei r_k eine Zufallsvariable ist. Genauer gesagt, $r_k = 1$ mit einer Wahrscheinlichkeit π und $r_k = 0$ mit Wahrscheinlichkeit $1 - \pi$. Die Wahrscheinlichkeit π wird so gewählt, dass $1/(2w_{ij}) \leq \pi < 1/w_{ij}$ ist.

6.2.5 Wahrscheinlichkeit eines Zeugen

Der Kern des randomisierten Ansatzes ist die Behauptung, dass die Wahrscheinlichkeit, dass die Summe $\sum_{k=1}^{n} \hat{a}_{ik} b_{kj}$ ein Zeuge für $p_{ij} = 1$ ist, mindestens $1/(2e)$ beträgt, wobei e die Eulersche Zahl ist. Dies kann wie folgt bewiesen werden: Angenommen, es gibt w "weiße Kugeln"(Zeugen) und $n - w$ ßchwarze Kugeln"(keine Zeugen). Wenn jede der n Kugeln unabhängig voneinander mit Wahrscheinlichkeit π ausgewählt wird, ist die Wahrscheinlichkeit ρ, dass genau eine weiße Kugel gewählt wird, gegeben durch:

$$\rho = w \cdot \pi \cdot (1 - \pi)^{w-1}$$

Unter Verwendung der Grenzen für π erhalten wir für $w > 1$: $\rho > (1/2)(1 - 1/w)^{w-1} \geq 1/(2e)$. Für $w = 1$ gilt $\rho \geq 1/(2e)$ ebenfalls.

6.2.6 Wiederholung des Experiments

Ein einzelnes Experiment, das mit Wahrscheinlichkeit $\geq 1/(2e)$ einen Zeugen liefert, ist möglicherweise nicht ausreichend. Durch mehrfache Wiederholung des Experiments"kann die Wahrscheinlichkeit, genau einen Zeugen zu erhalten, erhöht werden. Bei N Versuchen ist die Wahrscheinlichkeit des Scheiterns in allen Versuchen kleiner als $(1-1/(2e))^N \leq e^{-N/(2e)}$. Wählt man $N = 2ec \log n$, so ist die Fehlerwahrscheinlichkeit kleiner als $1/n^c$. Dies bedeutet, dass nur für wenige Einträge (i, j) ein Zeuge nicht gefunden wird.

Da w_{ij} für jedes (i, j) unterschiedlich sein kann und uns nicht bekannt ist, versucht der Algorithmus verschiedene Wahrscheinlichkeiten $\pi_s = 1/2^s$ für $s = 0, \ldots, \lceil \log n \rceil$. Dies wird erreicht, indem man ein Startset $R = \{1, \ldots, n\}$ hat und iterativ ein $(1/2)$-Sample aus R zieht, wobei jedes Element unabhängig mit Wahrscheinlichkeit $1/2$ ausgewählt wird. Im i-ten Iterationsschritt ist k dann mit Wahrscheinlichkeit $1/2^i$ in R.

6.2.7 Algorithmus BMM-WITNESS

Der Algorithmus `BMM-WITNESS` findet Zeugen für die Boolesche Matrixmultiplikation:

6.2.8 Laufzeitanalyse

Die Schleife in den Zeilen 4-10 gewährleistet, dass für jedes Paar (i, j) eine Wahrscheinlichkeit nahe $1/w_{ij}$ ausprobiert wird. Dies geschieht $2ec \log n$ Mal, sodass die Wahrscheinlichkeit, einen Zeugen in dieser Schleife nicht zu finden, höchstens $1/n^c$ beträgt. Da es n^2 Paare (i, j)

Algorithmus 4 BMM-WITNESS(A, B)

1: $W \leftarrow$ AB ▷ Initialisiere W (mit negativen Werten für fehlende Zeugen)
2: **for** $t \leftarrow 1 \rightarrow 2ec \log n$ **do**
3: $R \leftarrow \{1, \ldots, n\}$
4: **for** $s \leftarrow 0 \rightarrow \lceil \log n \rceil$ **do**
5: Compute A^R: $a^R_{ik} \leftarrow [k \in R] \cdot k \cdot a_{ik}$ ▷ $[k \in R]$ ist ein Indikator, 1 falls $k \in R$
6: $Z \leftarrow A^R B$
7: **for** all i, j **do**
8: **if** $W_{ij} < 0$ AND Z_{ij} is a witness **then** ▷ Überprüfe, ob $a_{i,Z_{ij}} = 1$ und $b_{Z_{ij},j} = 1$
9: $W_{ij} \leftarrow Z_{ij}$
10: $R \leftarrow (1/2)$-sample from R ▷ Jedes Element in R wird mit Wahrscheinlichkeit 1/2 ausgewählt
11: **for** all i, j **do**
12: **if** $W_{ij} < 0$ **then**
13: Finde Zeugen für i, j mittels Brute Force ▷ Überprüfe jedes k, Zeit $O(n)$ pro Paar
return W

gibt, ist die erwartete Anzahl fehlgeschlagener Paare höchstens $1/n^{c-2}$. Setzt man $c = 1$, ist dies höchstens n, und die erwartete Zeit der Schleife in Zeile 11-13 (Brute Force) beträgt höchstens $O(n^2)$. Die dominierende Operation in der Schleife 4-10 ist die Matrixmultiplikation in Zeile 6. Daher beträgt die erwartete Gesamtlaufzeit $O(M(n) \log^2 n)$. Die Korrektheit des Algorithmus wird dadurch sichergestellt, dass Zeugen nur dann gesetzt werden, wenn sie direkt überprüft wurden.

6.3 Nachfolger für kürzeste Pfade

6.3.1 Anwendung von Boolescher Matrixmultiplikation

Wir nutzen nun die Techniken der Zeugensuche für die Boolesche Matrixmultiplikation, um eine Nachfolgermarix S für das All-Paar-kürzeste-Pfade-Problem (APSP) zu erhalten. Für ein Paar von Knoten (i, j) ist ein Knoten s ein Nachfolger von i auf einem kürzesten Pfad von i nach j genau dann, wenn die Distanz von s nach j genau eins kleiner ist als die Distanz von i nach j, also $d_{sj} = d_{ij} - 1$.

6.3.2 Herausforderung bei der Definition von F

Intuitiv könnte man eine Boolesche Matrix F definieren, wobei $f_{sj} = 1$ ist, wenn $d_{sj} = d_{ij} - 1$. Dann könnte ein Nachfolger für (i, j) als der (i, j)-Eintrag einer Zeugenmatrix für das Boolesche Produkt AF gefunden werden. Das Problem hierbei ist jedoch, dass die Definition von d_{sj} von i abhängt, was bedeuten würde, dass wir alle n Möglichkeiten durchprobieren müssten, was ineffizient wäre.

6.3.3 Lösung mittels Modulo-3-Arithmetik

Glücklicherweise kann man das Problem vereinfachen, indem man nur die Distanzen modulo 3 unterscheidet. Genauer gesagt, es ist ausreichend, drei verschiedene Matrizen $F^{(c)}$ für $c = 0, 1, 2$ zu definieren:

$$f^{(c)}_{sj} = 1 \quad \text{falls} \quad d_{sj} \equiv (c - 1) \pmod 3$$

Obwohl diese Definition immer noch von i abhängt, gibt es nur drei verschiedene Fälle, was die Berechnung vereinfacht.

6.3.4 Algorithmus MM-APSP

Der Algorithmus `MM-APSP` zur Berechnung der Nachfolgermarix für kürzeste Pfade ist wie folgt strukturiert:

Algorithmus 5 MM-APSP(A)

1: $D \leftarrow$ MM-APSD(A) ▷ Berechne alle kürzesten Distanzen
2: **for** $c \leftarrow 0, 1, 2$ **do**
3: Definiere $F^{(c)}$: $f_{sj}^{(c)} = 1$ falls $d_{sj} \equiv (c-1) \pmod 3$
4: $W^{(c)} \leftarrow$ BMM-WITNESS($A, F^{(c)}$) ▷ Verwende den Algorithmus aus VI.4
5: Initialisiere S als $n \times n$ Matrix
6: **for** all i, j **do**
7: $s_{ij} \leftarrow w_{ij}^{(d_{ij} \pmod 3)}$ ▷ Wähle den Zeugen aus der passenden $W^{(c)}$ Matrix
return S

6.3.5 Laufzeitanalyse

Der Algorithmus `MM-APSD(A)` (All-Paar-kürzeste-Distanzen für Einheitsgewichte) hat eine Laufzeit von $O(M(n) \log n)$. Der Hauptteil des `MM-APSP`-Algorithmus beinhaltet drei Aufrufe von `BMM-WITNESS`, einmal für jedes $c \in \{0, 1, 2\}$. Jeder Aufruf von `BMM-WITNESS` hat eine erwartete Laufzeit von $O(M(n) \log^2 n)$. Daher ist die erwartete Gesamtlaufzeit des `MM-APSP`-Algorithmus $O(M(n) \log^2 n)$.