

Boolean Matrixmultiplikation in $O(n^2)$

Anwendung des Signatur-Verfahrens aus dem Subgraph Algorithmus

Basierend auf der Arbeit von Stephan Epp

16. Januar 2026

Zusammenfassung

Dieses Dokument beschreibt eine Anwendung der Signatur-Technik aus dem Subgraph Algorithmus auf die Boolean Matrixmultiplikation. Während die allgemeine Matrixmultiplikation eine Komplexität von mindestens $\Omega(n^{2.37})$ hat, zeigen wir, dass Boolean Matrixmultiplikation mit der Signatur-Methode in $O(n^2)$ Zeit berechnet werden kann. Dies wird durch geschickte Nutzung von Bitoperationen und polynomialer Hash-Kodierung erreicht.

Inhaltsverzeichnis

1 Einführung	2
2 Definitionen	2
3 Implementierung	4
3.1 Beispiel	6
4 Vergleich herkömmlicher Methoden	7
5 Zusammenfassung	7
5.1 Anwendungen	7
5.2 Implementierung	8

1 Einführung

Die klassische Matrixmultiplikation zweier $n \times n$ Matrizen A und B berechnet eine Ergebnismatrix C mit:

$$C_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj} \quad (1)$$

Die naive Implementierung hat eine Laufzeit von $O(n^3)$. Fortgeschrittene Algorithmen wie STRASSEN (1969) oder COPPERSMITH-WINOGRAD (1990) erreichen $O(n^{2.807})$ bzw. $O(n^{2.376})$.

Für **Boolean Matrizen**, bei denen alle Einträge $\{0, 1\}$ sind und die Operationen durch logische Operationen ersetzt werden, kann die Signatur-Technik aus dem Subgraph Algorithmus genutzt werden, um eine Laufzeit von $O(n^2)$ zu erreichen.

2 Definitionen

Definition 1 (Boolean Matrixmultiplikation). Für zwei Boolean Matrizen $A, B \in \{0, 1\}^{n \times n}$ ist die Boolean Matrixmultiplikation definiert als:

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

wobei \vee das logische ODER und \wedge das logische UND bezeichnet.

Mit anderen Worten: $C_{ij} = 1$ genau dann, wenn es mindestens ein k gibt mit $A_{ik} = 1$ und $B_{kj} = 1$.

Die zentrale Idee aus dem Subgraph Algorithmus ist die Verwendung einer polynomialen Hash-Funktion zur Kodierung von Binärvektoren:

Definition 2 (Signatur-Funktion). Für einen Binärvektor $v = (v_0, v_1, \dots, v_{n-1}) \in \{0, 1\}^n$ ist die Signatur definiert als:

$$\sigma(v) = \sum_{i=0}^{n-1} v_i \cdot 2^i \quad (2)$$

Beispiel 1. Für $v = (1, 0, 1, 1)$ ergibt sich:

$$\sigma(v) = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 1 + 4 + 8 = 13$$

Lemma 1 (Eindeutigkeit). Die Signatur-Funktion $\sigma : \{0, 1\}^n \rightarrow \mathbb{N}$ ist injektiv.

Beweis. Die Signatur entspricht der Interpretation des Binärvektors als Dezimalzahl in Binärdarstellung. Verschiedene Binärvektoren ergeben verschiedene Dezimalzahlen im Bereich $[0, 2^n - 1]$. \square

Die Signatur hat eine wichtige algebraische Eigenschaft bezüglich der bitweisen UND-Operation:

Lemma 2 (Bitweise UND-Operation). Seien $v, w \in \{0, 1\}^n$ zwei Binärvektoren mit Signaturen $\sigma(v)$ und $\sigma(w)$. Dann gilt:

$$\sigma(v) \& \sigma(w) = \sigma(v \wedge w)$$

wobei $\&$ die bitweise UND-Operation auf den Dezimalzahlen bezeichnet und $v \wedge w$ die komponentenweise UND-Operation auf den Vektoren.

Beweis. Die bitweise UND-Operation auf Dezimalzahlen entspricht genau der komponentenweisen UND-Operation auf den Binärdarstellungen. Da die Signatur die Binärdarstellung ist, folgt die Behauptung direkt. \square

Der Kerngedanke für die Arbeitsweise des Algorithmus zur Boolean Matrixmultiplikation mit Signaturen ist:

1. Kodiere jede **Zeile** von A als Signatur
2. Kodiere jede **Spalte** von B als Signatur
3. Für jedes Element C_{ij} :
 - Berechne bitweise UND der Signaturen: $\sigma(\text{row}_i(A)) \& \sigma(\text{col}_j(B))$
 - Setze $C_{ij} = 1$ falls Ergebnis $\neq 0$, sonst $C_{ij} = 0$

Algorithmus 2.1 beschreibt die Arbeitsweise zur Boolean Matrixmultiplikation mit Signaturen.

Algorithm 2.1 Boolean Matrixmultiplikation mit Signaturen

Eingabe: Zwei Boolean Matrizen $A, B \in \{0, 1\}^{n \times n}$

Ausgabe: Boolean Matrix $C \in \{0, 1\}^{n \times n}$ mit $C_{ij} = \bigvee_{k=1}^n (A_{ik} \wedge B_{kj})$

```

1:  $n \leftarrow$  Dimension von  $A$ 
2:  $\text{rowSig} \leftarrow$  leeres Array der Länge  $n$ 
3:  $\text{colSig} \leftarrow$  leeres Array der Länge  $n$ 
4: for  $i = 0$  to  $n - 1$  do
5:    $\text{rowSig}[i] \leftarrow \sum_{k=0}^{n-1} A_{ik} \cdot 2^k$   $\triangleright O(n)$ 
6: end for
7: for  $j = 0$  to  $n - 1$  do
8:    $\text{colSig}[j] \leftarrow \sum_{k=0}^{n-1} B_{kj} \cdot 2^k$   $\triangleright O(n)$ 
9: end for
10:  $C \leftarrow n \times n$  Nullmatrix
11: for  $i = 0$  to  $n - 1$  do
12:   for  $j = 0$  to  $n - 1$  do
13:      $\text{andResult} \leftarrow \text{rowSig}[i] \& \text{colSig}[j]$   $\triangleright O(1)$  Bitoperation
14:     if  $\text{andResult} \neq 0$  then
15:        $C_{ij} \leftarrow 1$ 
16:     end if
17:   end for
18: end for
19: return  $C$ 

```

Satz 1 (Korrektheit der Signatur-Methode). Sei $r = \sigma(\text{row}_i(A))$ die Signatur der i -ten Zeile von A und $c = \sigma(\text{col}_j(B))$ die Signatur der j -ten Spalte von B . Dann gilt:

$$C_{ij} = 1 \Leftrightarrow (r \& c) \neq 0 \quad (3)$$

Beweis. Nach Definition ist:

$$C_{ij} = 1 \Leftrightarrow \exists k : A_{ik} = 1 \text{ und } B_{kj} = 1 \quad (4)$$

Die bitweise UND-Operation $r \& c$ berechnet:

$$r \& c = \sum_{k=0}^{n-1} (A_{ik} \wedge B_{kj}) \cdot 2^k \quad (5)$$

Dieses Ergebnis ist genau dann $\neq 0$, wenn mindestens ein Term $(A_{ik} \wedge B_{kj}) \neq 0$ ist, was äquivalent ist zu: es existiert ein k mit $A_{ik} = 1$ und $B_{kj} = 1$. \square

Satz 2 (Laufzeit). Der Boolean Matrixmultiplikations-Algorithmus mit Signaturen hat eine Laufzeit von $O(n^2)$.

Beweis. Der Algorithmus besteht aus zwei Phasen:

Phase 1: Signatur-Berechnung

- Berechnung aller Zeilen-Signaturen: n Zeilen $\times O(n)$ pro Signatur $= O(n^2)$
- Berechnung aller Spalten-Signaturen: n Spalten $\times O(n)$ pro Signatur $= O(n^2)$
- Gesamt Phase 1: $O(n^2)$

Phase 2: Multiplikation

- Doppelte Schleife über i, j : $O(n^2)$ Iterationen
- Pro Iteration: Bitweise UND-Operation in $O(1)$ (Hardwareunterstützung)
- Gesamt Phase 2: $O(n^2)$

$$\text{Gesamtkomplexität: } O(n^2) + O(n^2) = O(n^2) \quad \square$$

Korollar 1. Für dünnbesetzte Boolean Matrizen mit m Nicht-Null-Einträgen ($m \ll n^2$) kann die Laufzeit auf $O(m \cdot n)$ reduziert werden durch Verwendung von Adjazenzlisten statt vollständiger Matrizen.

3 Implementierung

In diesem Kapitel wird auf die Implementierung des Algorithmus 2.1 eingegangen. Dabei wird Python als Programmiersprache verwendet. Listing 1 zeigt die Kernfunktionen der Implementierung.

Listing 1: Signatur-Berechnung für Zeilen und Spalten

```

1 def _compute_row_signature(self, row: np.ndarray) -> int:
2     """Berechnet Signatur fuer eine Zeile."""
3     n = len(row)
4     signature = sum(2**i for i in range(n) if row[i] == 1)
5     return signature
6
7 def _compute_column_signature(self, col: np.ndarray) -> int:
8     """Berechnet Signatur fuer eine Spalte."""
9     n = len(col)
10    signature = sum(2**i for i in range(n) if col[i] == 1)
11    return signature
12
13 def _precompute_signatures(self, A: np.ndarray, B: np.ndarray):
14     """Vorberechnung aller Signaturen."""
15     n = A.shape[0]
16     m = B.shape[1]
17
18     # Zeilen-Signaturen von A
19     row_sigs_A = [self._compute_row_signature(A[i, :])
20                   for i in range(n)]
21
22     # Spalten-Signaturen von B
23     col_sigs_B = [self._compute_column_signature(B[:, j])
24                   for j in range(m)]
25
26     return row_sigs_A, col_sigs_B

```

Listing 2: Boolean Multiplikation via Signaturen

```

1 def multiply_optimized(self, A: np.ndarray, B: np.ndarray) -> np.
2     ndarray:
3         """Boolean Matrixmultiplikation in O(n^2)."""
4         n, k1 = A.shape
5         k2, m = B.shape
6
7         if k1 != k2:
8             raise ValueError("Dimensionen passen nicht")
9
10        # Phase 1: Signaturen vorberechnen
11        row_sigs_A, col_sigs_B = self._precompute_signatures(A, B)
12
13        # Phase 2: Multiplikation via Bitoperationen
14        C = np.zeros((n, m), dtype=int)
15
16        for i in range(n):
17            for j in range(m):
18                # Bitweise AND in O(1)
19                and_result = row_sigs_A[i] & col_sigs_B[j]
20
21                # Boolean OR Check
22                C[i, j] = 1 if and_result != 0 else 0

```

22
23 `return C`

3.1 Beispiel

In diesem Kapitel wird ein Beispiel zur Arbeitsweise des Algorithmus 2.1 vorgestellt.

Beispiel 2. Gegeben seien die Boolean Matrizen:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Schritt 1: Zeilen-Signaturen von A

$$\begin{aligned}\sigma(\text{row}_0) &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 5 \\ \sigma(\text{row}_1) &= 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 2 \\ \sigma(\text{row}_2) &= 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 3\end{aligned}$$

Schritt 2: Spalten-Signaturen von B

$$\begin{aligned}\sigma(\text{col}_0) &= 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = 6 \\ \sigma(\text{col}_1) &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 5\end{aligned}$$

Schritt 3: Berechnung von C

$$\begin{aligned}C_{00} &= (5 \& 6 \neq 0) = (0100_2 \neq 0) = 1 \\ C_{01} &= (5 \& 5 \neq 0) = (0101_2 \neq 0) = 1 \\ C_{10} &= (2 \& 6 \neq 0) = (0010_2 \neq 0) = 1 \\ C_{11} &= (2 \& 5 \neq 0) = (0000_2 = 0) = 0 \\ C_{20} &= (3 \& 6 \neq 0) = (0010_2 \neq 0) = 1 \\ C_{21} &= (3 \& 5 \neq 0) = (0001_2 \neq 0) = 1\end{aligned}$$

Ergebnis:

$$C = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

Zur Überprüfung berechnen wir C nach klassischer Definition:

$$\begin{aligned}C_{00} &= (A_{00} \wedge B_{00}) \vee (A_{01} \wedge B_{10}) \vee (A_{02} \wedge B_{20}) \\ &= (1 \wedge 0) \vee (0 \wedge 1) \vee (1 \wedge 1) = 0 \vee 0 \vee 1 = 1 \quad \checkmark\end{aligned}$$

Alle weiteren Einträge können analog überprüft werden.

4 Vergleich herkömmlicher Methoden

Die nachfolgende Tabelle zeigt die verschiedenen Algorithmen und ihre Laufzeiten für die Berechnung der Boolean Matrixmultiplikation.

Algorithmus	Laufzeit	Speicher
Naive Multiplikation	$O(n^3)$	$O(n^2)$
Strassen	$O(n^{2.807})$	$O(n^2)$
Coppersmith-Winograd	$O(n^{2.376})$	$O(n^2)$
Signatur-Methode (Boolean)	$O(n^2)$	$O(n^2)$

Die Signatur-Methode hat folgende Einschränkungen:

1. **Nur für Boolean Matrizen:** Funktioniert nicht für allgemeine Zahlen
2. **Maximale Dimension:** Begrenzt durch Wortgröße (64-Bit $\Rightarrow n \leq 64$)
3. **Hardwareabhängig:** Bitoperationen müssen effizient unterstützt werden

Für $n > 64$ können erweiterte Techniken verwendet werden:

- Aufteilung in Blöcke der Größe 64
- Verwendung von Bit-Arrays oder speziellen Bibliotheken
- Hybride Ansätze für sehr große Matrizen

5 Zusammenfassung

In diesem Kapitel wird auf Anwendungen hingewiesen und die Arbeit als solche zusammengefasst.

5.1 Anwendungen

Anwendungen in der Graphen-Theorie: Boolean Matrixmultiplikation ist fundamental in der Graphentheorie:

- **Transitive Hülle:** Berechnung aller erreichbaren Knoten
- **All-Pairs Shortest Paths:** Mit wiederholter Boolean Multiplikation
- **Pfadexistenz:** Prüfung ob Pfad zwischen zwei Knoten existiert

Anwendungen in der formalen Verifikation: In Kombination mit dem ursprünglichen Subgraph Algorithmus:

- Effiziente Analyse von Abstract Syntax Trees
- Strukturelle Ähnlichkeit von Programmen
- Zustandsübergänge in Graphtransformationssystemen

Anwendungen in Datenbanken:

- Relationale Joins als Boolean Matrixoperationen
- Transitive Abfragen in Graphdatenbanken
- Zugriffsrechte-Propagation

Wir haben gezeigt, dass die Signatur-Technik aus dem Subgraph Algorithmus erfolgreich auf Boolean Matrixmultiplikation angewendet werden kann:

- Theoretische Laufzeit: $O(n^2)$ (optimal für Boolean Fall)
- Praktische Implementierung mit modernen Bitoperationen
- Korrektheitsbeweis durch algebraische Eigenschaften
- Anwendungen in Graph-Theorie und Verifikation

Die Verbindung zwischen Boolean Matrixmultiplikation und dem Subgraph Algorithmus zeigt die Vielseitigkeit der Signatur-Technik:

Subgraph Algorithmus	Boolean MatMul
Spalten-Signaturen	Zeilen/Spalten-Signaturen
Zyklische Rotation	Bitweise AND-Operation
LCS-Vergleich	OR-Check auf Bitebene
Subgraph-Erkennung	Pfadexistenz

Beide Algorithmen nutzen die polynomiale Hash-Funktion zur effizienten Strukturkodierung und erreichen damit deutliche Laufzeitverbesserungen gegenüber naiven Ansätzen.

5.2 Implementierung

Der Python Code für die Implementierung des Algorithmus zur Boolean Matrixmultiplikation und die Tests wurden mit Claude AI generiert. Der Code ist verfügbar unter: <https://github.com/hjstephan/bool-mm>. In diesem Repository befindet sich auch der generierte Code Coverage Report im HTML Format.