Der Subgraph Algorithmus

Stephan Epp

10. Januar 2026

Inhaltsverzeichnis

1 Einführung

Der Subgraph Algorithmus ist ein effizienter Algorithmus zum Vergleichen zweier Graphen G und G' mittels Adjazenzmatrizen und Signatur-Arrays. Der Algorithmus bestimmt durch zyklische Rotation, ob der Graph G als Subgraph in dem anderen Graphen G' enthalten ist.

1.1 Problemstellung

Gegeben sind zwei Graphen G und G' mit jeweils n Knoten, repräsentiert durch Adjazenzmatrizen A und B.

Ziel: Bestimme, ob Graph G in Graph G' enthalten ist, unter Berücksichtigung aller möglichen Knotenzuordnungen durch zyklische Rotation.

Ergebnis Der Algorithmus berechnet folgendes Ergebnis:

- Wenn $B \supseteq A$: Verwerfe A, behalte B (G' hat mehr Informationen)
- Wenn $A \supseteq B$: Behalte A, verwerfe B (G hat mehr Informationen)
- Wenn beide identisch sind: Beliebige behalten
- Wenn keiner den anderen enthält: Beide behalten

1.2 Motivation

Der Subgraph Algorithmus eignet sich besonders für die Verifikation von unterschiedlichen Programmen durch die Analyse der Abstract Syntax Trees (AST). Die Methode ermöglicht es, strukturelle Ähnlichkeiten zwischen Programmrepräsentationen effizient zu erkennen.

Es gibt Graphtransformationen zur Modellierung von Zustandsübergängen in Softwaresystemen. Dabei werden Systemzustände als Graphen dargestellt, und Zustandsübergänge durch farbcodierte Transformationsregeln beschrieben. Mit Hilfe des Subgraph Algorithmus lässt sich die Stabilität oder die Ruhelage eines Systems analysieren, welches in jedem globalen Zustand durch einen Graphen beschrieben wird. Was wurde spezifiziert, was modelliert und wie verhält sich das System in der Realität wirklich? Damit lassen sich Fehler im System finden und Aussagen treffen zur Performance und Sicherheit des Systems.

2 Grundlagen

2.1 Definitionen

Definition 1 (Graph)**.** Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten $V = \{v_1, v_2, \ldots, v_n\}$ und einer Menge von Kanten $E \subseteq V \times V$.

Definition 2 (Adjazenzmatrix)**.** Die Adjazenzmatrix A eines Graphen $G = (V, E)$ mit n Knoten ist eine $n \times n$ Matrix mit:

$$A_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$

Definition 3 (Subgraph)**.** Ein Graph $G = (V, E)$ ist ein Subgraph von $G' = (V', E')$, geschrieben $G \subseteq G'$, wenn $V \subseteq V'$ und $E \subseteq E'$.

Definition 4 (Zyklische Rotation)**.** Eine zyklische Rotation einer Sequenz $[a_0, a_1, \ldots, a_{n-1}]$ um k Positionen nach rechts ist definiert als:

$$\text{rotate}_k([a_0, a_1, \ldots, a_{n-1}]) = [a_k, a_{k+1}, \ldots, a_{n-1}, a_0, \ldots, a_{k-1}]$$

2.2 Grundidee

Der Algorithmus basiert auf zwei zentralen Konzepten:

2.2.1 Eindeutige Signaturen

Für jede Spalte j der Adjazenzmatrix wird eine eindeutige Signatur berechnet:

$$\sigma_j = \sum_{i=0}^{n-1} A_{ij} \cdot 2^i + j \cdot 2^n$$

Beispiel 1. Für $n = 4$ und Spalte $j = 0$ mit Vektor $[1, 0, 1, 0]^T$:

$$\sigma_0 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 = 1 + 4 = 5$$

Für Spalte $j = 1$ mit gleichem Vektor $[1, 0, 1, 0]^T$:

$$\sigma_1 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 1 + 4 + 16 = 21$$

Lemma 1 (Eindeutigkeit der Signaturen)**.** Die Signatur-Funktion $\sigma : \{0, 1\}^n \times \{0, \ldots, n-1\} \to \mathbb{N}$ ist injektiv.

Beweis. Die Zeilenkomponente $\sum_{i=0}^{n-1} A_{ij} \cdot 2^i$ kodiert jede Binärkombination eindeutig als Dezimalzahl. Die Spaltengewichtung $j \cdot 2^n$ unterscheidet gleiche Muster an verschiedenen Positionen, da $j \cdot 2^n$ für $j \in \{0, \ldots, n-1\}$ stets disjunkte Intervalle $[j \cdot 2^n, (j+1) \cdot 2^n)$ erzeugt. □

2.2.2 Zyklische Rotation

Statt alle $n!$ Permutationen zu prüfen, werden nur n zyklische Rotationen betrachtet. Dies erhält die sequentielle Ordnung und reduziert die Komplexität drastisch.

Beispiel 2. Für $n = 4$ Spalten:

$$\begin{aligned} \text{Original:} &\quad [\sigma_0, \sigma_1, \sigma_2, \sigma_3] \\ \text{Rotation 1:} &\quad [\sigma_1, \sigma_2, \sigma_3, \sigma_0] \\ \text{Rotation 2:} &\quad [\sigma_2, \sigma_3, \sigma_0, \sigma_1] \\ \text{Rotation 3:} &\quad [\sigma_3, \sigma_0, \sigma_1, \sigma_2] \end{aligned}$$

Dies sind nur 4 Rotationen und nicht $4! = 24$ Permutationen.

3 Algorithmus

In diesem Kapitel wird der Subgraph Algorithmus formal beschrieben.

3.1 Arbeitsweise

Der Algorithmus arbeitet in drei Hauptschritten:

Schritt 1: Signatur-Berechnung für Graph G
Berechne für jede Spalte j der Adjazenzmatrix A die Signatur:

$$\sigma_j^A = \sum_{i=0}^{n_A-1} A_{ij} \cdot 2^i + j \cdot 2^{n_A}$$

Schritt 2: Signatur-Berechnung für Graph G'
Berechne analog für Matrix B:

$$\sigma_j^B = \sum_{i=0}^{n_B-1} B_{ij} \cdot 2^i + j \cdot 2^{n_B}$$

Schritt 3: Rotation-Match
Für jede der n_B zyklischen Rotationen von B:

1. Rotiere Spalten zyklisch nach rechts
2. Extrahiere Zeilenkomponenten (ohne Spaltengewichtung)
3. Vergleiche Signatur-Sequenzen mit Longest Common Subsequence (LCS)
4. Falls LCS ≥ 2: Subgraph-Beziehung gefunden

3.2 Vergleich der Signatur-Sequenzen

Der Vergleich zweier Signatur-Sequenzen erfolgt mittels dynamischer Programmierung zur Berechnung der längsten gemeinsamen Teilsequenz:

Satz 1 (Subgraph-Kriterium)**.** Eine Subgraph-Beziehung existiert genau dann, wenn die längste gemeinsame Teilsequenz der Signatur-Arrays mindestens die Länge 2 hat.

Beweis. Eine Subgraph-Beziehung kann nur dann existieren, wenn mindestens zwei Knoten durch eine Kante miteinander verbunden sind. Dies entspricht einer gemeinsamen Teilsequenz der Länge mindestens 2 in den Signatur-Arrays. □

3.3 Implementierung

Der Subgraph Algorithmus wurde in Python implementiert. Listing 1 zeigt die Berechnung der Signaturen für jede Spalte.

Listing 1: Signatur-Berechnung

```
1 def _compute_column_signature(self, matrix: np.ndarray) -> List[
    int]:
2         n = matrix.shape[0]
3         signatures = []
4
5         for col in range(n):
6                 column_vector = matrix[:, col]
7
```

```
                # Polynomiale Hash-Funktion
                row_signature = sum(2**i for i in range(n) if
                    column_vector[i] == 1)

                # Spaltenindex mit Gewichtung 2^n einbeziehen
                col_weight = col * (2**n)

                signature = row_signature + col_weight
                signatures.append(signature)

        return signatures
```

Listing 2 zeigt das Prüfen auf ein Matching für alle Rotationen.

Listing 2: Rotation-Match

```
def _find_signature_rotation_match(self, sig_A: List[int], sig_B:
    List[int]) -> bool:
    n_A = len(sig_A)
        n_B = len(sig_B)

        if n_A > n_B:
                return False

        if n_A == 0:
                return True

        # Extrahiere Zeilenkomponenten (ohne Spaltengewichtung)
        # Fuer Matrix n x n ist col_weight = 2^n
        col_weight_A = 2 ** n_A
        col_weight_B = 2 ** n_B

        row_sigs_A = [sig % col_weight_A for sig in sig_A]
        row_sigs_B = [sig % col_weight_B for sig in sig_B]

        # Pruefe alle n_B zyklischen Rotationen von B
        for rotation in range(n_B):
                # Rotiere row_sigs_B um 'rotation' Positionen
                    nach rechts
                rotated_B = row_sigs_B[rotation:] + row_sigs_B[:
                    rotation]

                # Pruefe ob A in diesem Fenster der Rotation
                    enthalten ist
                if n_A == n_B:
                        # Gleich gross: direkter Vergleich
                        if self._compare_signature_sequences(
                            row_sigs_A, rotated_B):
                                return True
                else:
                        # A ist kleiner: Pruefe alle
                            Startpositionen in rotated_B
                        for start in range(n_B - n_A + 1):
```

```
32                                  window = rotated_B[start:start +
                                        n_A]
33                                  if self.
                                        _compare_signature_sequences(
                                        row_sigs_A, window):
34                                          return True
35
36          return False
```

4 Analyse

In diesem Kapitel wird die Laufzeit und die Korrektheit des Subgraph Algorithmus analysiert. Außerdem wird ein Beispiel gegeben, das die Arbeitsweise des Subgraph Algorithmus verdeutlicht.

4.1 Laufzeit

Satz 2 (Laufzeit)**.** Der Subgraph Algorithmus arbeitet mit einer Laufzeit von $O(n^3)$.

Beweis. Der Algorithmus arbeitet in drei Schritten:

Schritt 1: Signatur von G berechnen

- Iteration über $n \times n$ Matrix-Einträge
- Komplexität: $O(n^2)$

Schritt 2: Signatur von G' berechnen

- Iteration über $n \times n$ Matrix-Einträge
- Komplexität: $O(n^2)$

Schritt 3: Alle zyklischen Rotationen prüfen

- Für jede der n Rotationen:
  - Signatur neu berechnen: $O(n^2)$
  - Sequenzen vergleichen (LCS): $O(n^2)$
- Komplexität: $O(n \cdot (n^2 + n^2)) = O(n^3)$

Gesamtkomplexität: $O(n^2) + O(n^2) + O(n^3) = O(n^3)$ □

Zur Prüfung beider Richtungen $A \subseteq B$ und $B \subseteq A$ benötigt der Subgraph Algorithmus ebenfalls eine Laufzeit von $O(n^3)$.

4.2 Adjazenzlisten

Für weniger dichte Graphen kann die Verwendung von Adjazenzlisten effizienter sein:

- Laufzeit für einfache Vergleiche: $O(n + m)$
- Speicherbedarf: $O(n + m)$ für m Kanten
- Vorteil bei $m \ll n^2$

4.3 Korrektheit

Satz 3 (Korrektheit)**.** Der Subgraph Algorithmus bestimmt korrekt, ob eine Subgraph-Beziehung zwischen zwei Graphen existiert.

Beweis. Die Rotation der Spalten entspricht dem Drehen des Graphen. Dabei bleibt die Struktur des Graphen, gegeben durch die Verbundenheit der Knoten und Kanten, immer erhalten. Deshalb sind nur n Rotationen zu betrachten und nicht $n!$ viele Permutationen.

Der Graph wird so lange gedreht, bis eine Subgraph-Beziehung existiert oder das Drehen vollständig durchgeführt wurde, ohne dass eine Subgraph-Beziehung existiert.

Zur Überprüfung der Subgraph-Beziehung für jede Drehung werden die Elemente der Signatur-Arrays in $O(n^2)$ Laufzeit verglichen. Dabei wird die längste gemeinsame Teilsequenz beider Signatur-Arrays ermittelt.

Die Injektivität der Signatur-Funktion (siehe Lemma 1) garantiert, dass verschiedene Spaltenvektoren verschiedene Signaturen erhalten und somit keine Fehlerkennungen auftreten. □

4.4 Beispiel

Nachfolgend wird ein kleines Beispiel zur Anwendung des Subgraph Algorithmus gegeben.

Beispiel 3 (Grundlegendes Beispiel)**.** Gegeben seien zwei Graphen:

Graph G mit 4 Knoten:

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

Graph G' mit zusätzlicher Kante:

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

Der Algorithmus berechnet:

- Signaturen von G: $[1, 18, 36, 48]$
- Signaturen von G': $[5, 18, 36, 48]$
- Entscheidung: `keep_B` (G' hat mehr Informationen)

5 Zusammenfassung

Der Subgraph Algorithmus bietet eine effiziente Methode zum Vergleich von Graphen mit einer Laufzeit von $O(n^3)$. Durch die Verwendung von:

- eindeutigen Signaturen basierend auf polynomialer Hash-Funktion,
- zyklischen Rotationen statt vollständiger Permutationen,
- der Longest Common Subsequence zur Subgraph-Erkennung,

erreicht der Algorithmus eine deutlich bessere Laufzeit als naive Ansätze.

5.1 Anwendungen

Der Algorithmus eignet sich besonders für die Verifikation von Programmen durch AST-Analyse oder zur Erkennung struktureller Ähnlichkeiten in Code, Deduplizierung von Graph-Datenbanken oder Pattern Matching in strukturierten Daten. Außerdem lässt sich mit Hilfe des Subgraph Algorithmus die Stabilität und die Ruhelage eines Systems analysieren, welches in jedem globalen Zustand durch einen Graphen beschrieben wird. Damit sind Fehler im System identifizierbar und es lassen sich Aussagen zur Performance und Sicherheit des Systems treffen.

5.2 Ausblick

Mögliche Erweiterungen des Subgraph Algorithmus umfassen die Parallelisierung der Rotation-Prüfungen, die Optimierung für sehr große Graphen ($n > 10000$), die Approximative Algorithmen für Echtzeit-Anwendungen, die Erweiterung auf gewichtete und gerichtete Graphen und die Integration von Heuristiken zur Reduktion der zu prüfenden Rotationen.

5.3 Implementierung

Der Python Code für die Implementierung des Subgraph Algorithmus und die Tests wurden mit Claude AI generiert. Der Code ist verfügbar unter: `https://github.com/hjstephan/subgraph`. In diesem Repository befindet sich auch der generierte Code Coverage Report im HTML Format.