Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Knowledge Discovery in Databases II
im Wintersemester 2012/2013

Kapitel 9: Graph Mining

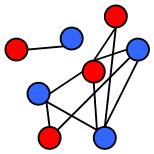
http://www.dbs.ifi.lmu.de/Lehre/KDD

Kapitelübersicht

1. Graphrepräsentationen
   Arten von Graphen, Grundlegende Definitionen, Anwendungsbeispiele
2. Isomorphiebasierter Graphvergleich und Edit-Distanz
   Max. Common Subgraph, Min. Common Supergraph, Edit Operationen auf Graphen
3. Topologische Deskriptoren und Graph-Kernel
   R-Convolution Kernel, All-Subgraphs Kernel, Random Walks Kernel , Shortest Path-Kernel
4. Frequent Subgraph Mining
   FSG, Gspan
5. Ranking von Knoten
   Centrality, PageRank, Hits
6. Link Prediciton
   Link-Klassifikation, Matrix Faktorisierung
7. Multi-Relationale Graphen
   Probabilistic Relational Models (PRM), Multiple Matrix Faktorisierung

Graphrepräsentation

- Graphen sind die allgemeinste Datenstruktur in der Vorlesung
- **Definition**: Ein Graph ist ein Tupel *G=(V,E)*, wobei *V* eine Menge von Knoten ist und $E \subseteq V \times V$ eine Menge von Kanten.



- Darstellung von Beziehungen zwischen Objekten durch Kanten
- Darstellung anderer Datenstrukturen durch Graphen meist möglich
  Beispiel: MI-Objekte sind Menge aus Knoten ohne Kanten.
- Problem mit Graphrepräsentationen:
  Vergleich 2er Graphen ist sehr teuer!
  $\Rightarrow$ Verwende einfachere Datenstrukturen wenn möglich !

Anwendungen

Beispiele:

- Molekulare Strukturen, RNA-Transkription
- Bio-Informatik: Protein-Interaktionsnetzwerke, Philogenetische Netzwerke, Metabolom-Netzwerke..
- Soziale Netzwerke: Vergleich ähnlicher sozialer Gruppen
  (Zusammenarbeit in Arbeitsgruppe, Zuspiel Fußballmannschaft..)
- WWW, Internet, Computernetzwerke: Web-Ringe, Netzwerk-Topologien..
- XML-Dokumente sind ebenfalls Graphen im allgemeinsten Fall.
  (Vorsicht viele XML-Datenquellen lassen sich ohne Informationsverlust durch Feature-Vektoren modellieren.)

Arten von Graphen

Erweiterungen der Definition:

GRAPH: Ein Graph ist ein Tupel G=(V,E), wobei V eine Menge von Knoten ist und E $\subseteq$ V×V eine Menge von Kanten.

GERICHTETER GRAPH: Gilt $(v_k,v_l) \neq (v_l,v_k)$ ist der Graph gerichtet.

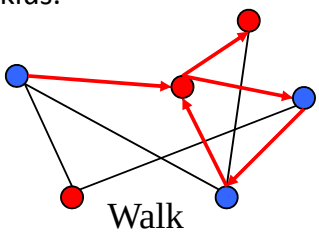
GELABELTE oder ATTRIBUTIERTE GRAPHEN: Seien F_V und F_E Feature-Räume.
Ein Graph G heißt gelabelt oder attributiert bzgl. der Knoten V, wenn es zu jedem Knoten $v \in V$ genau eine Feature-Beschreibung $l_v \in F_E$ gibt.
Ein Graph G heißt gelabelt oder attributiert bzgl. der Kanten E, wenn es zu jeder Kante $e \in E$ genau eine Feature-Beschreibung $l_e \in F_E$ gibt.

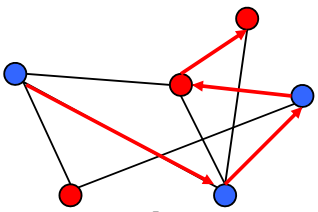
Bemerkung:

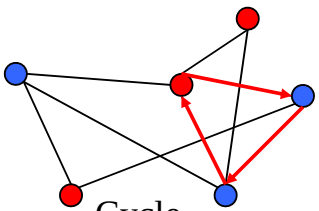
In der Regel sind Knoten und Kanten nur mit einzelnen diskreten Attributen gelabelt.

Darstellung von Graphen

- **Knotengrad:** Der Grad $d_G(v_i)$ eines Knotens v_i in G=(V,E) ist die Anzahl der anliegenden Kanten: $d_G(v_i) = \left|\{v_j \mid (v_i,v_j) \in E\}\right|$
- **Adjanzenz-Matrix**: Die Adjazenz-Matrix eines Graphen G=(V,E) ist gegeben durch: $[A]_{i,j} = \begin{cases} 1 & falls \quad (v_i,v_j) \in E \\ 0 & sonst \end{cases}$
- **Weg (Walk)**: Ein Weg in $G=(V,E)$ der Länge $k-1$ ist eine Sequenz von Knoten $w=(v_1,v_2,..,v_k)$ und $(v_{i-1}, v_i) \in E$ für alle $1 \leq i \leq k$.
- **Pfad (Path)**: w ist ein Pfad falls $v_i \neq v_j$ für alle $i \neq j$.
  (=> Kein Knoten darf in w zweimal auftauchen.)
- **Zyklus (Cycle):** Sei $w=(v_1,..,v_k)$ und $v_1 = v_k$ und für alle 1 < i,j < k gilt $v_i \neq v_j$ dann ist w ein Zyklus.


Walk


Path


Cycle

Das Graph-Vergleichsproblem

Gegeben: 2 Graphen G und G‘.

Gesucht: Abbildung $s:(V\times E)\times(V\times E) \rightarrow IR$, die die Ähnlichkeit von G und G‘ quantifiziert.

Ansätze:

Isomorphie: 2 Graphen sind gleich, wenn es eine eineindeutige Abbildung von Knoten und Kanten gibt.

=> Ähnlichkeit über die Größe der isomorphen Teilgraphen.

Edit-Distanz: Unähnlichkeit der Graphen wird durch Aufwand berechnet den einen Graphen in einen anderen zu verwandeln.

Topologische Deskriptoren: Zwei Graphen sind ähnlich, wenn ihre Topologie ähnliche Eigenschaften aufweist.

(z.B. ähnliche Anzahl von Knoten mit Grad *n*)

Isomorphiebasierte Graphvergleiche

Wann sind zwei Graphen gleich ?

Graph-Isomorphie:

Seien $G=(V,E)$ und $G'=(V',E')$ 2 Graphen. G und G' sind genau dann isomorph ($G \cong G'$), wenn es eine Bijection $f: V \rightarrow V'$ mit $(v,v') \in E \Leftrightarrow (f(v),f(v')) \in E'$ für alle $v,v' \in V$.

f heißt dann Isomorphismus.

Subgraph: Sei $G=(V,E)$ ein Graph, dann ist $G'=(V',E')$ ein Subgraph von G, wenn $V' \subseteq V$ und $E' \subseteq (V' \times V' \cap E)$.

Subgraph-Isomorphie: Seien $G=(V,E)$ und $G'=(V',E')$ 2 Graphen. G' ist subgraph-isomorph zu G falls es einen Subgraph G'' von G gibt mit $G'' \cong G'$.

Maximaler Gemeinsamer Subgraph : Seien $G=(V,E)$ und $G'=(V',E')$ 2 Graphen. Ein Graph S ist ein maximaler gemeinsamer Subgraph $mcs(G,G')$, wenn S sowohl Subgraph von G als auch von G' ist und es keinen anderen gemeinsame Subgraphen S' mit mehr Knoten gibt.

Minimaler Gemeinsamer Supergraph: Seien $G=(V,E)$ und $G'=(V',E')$ 2 Graphen. Ein Graph S ist ein minimaler gemeinsamer Supergraph $MCS(G,G')$, wenn sowohl G als auch von G' ein Subgraph von S ist und es keinen anderen gemeinsamen Supergraphen gibt der weniger Knoten hat.

Isomorphiebasierte Distanzen

- Sowohl die Größe des max. gemeinsamen Subgraphen als auch die Größe des minimalen gemeinsamen Supergraphen beschreiben Ähnlichkeit der Graphen.
- **Distanzmaß 1**: Relative Größe des maximalen gemeinsamen Subgraphen

$$d_1(G,G') = 1 - \frac{|mcs(G,G')|}{\max(|G|,|G'|)}$$

- **Distanzmaß 2:** Differenz der Größe zwischen MCS(G,G‘) und mcs(G,G‘)

$$d_2(G,G') = |MCS(G,G')| - |mcs(G,G')|$$

- Abhängig von Bestimmung der Größe eines Graphen:
  z.B. Anzahl der Knoten => auch unterschiedliche Graphen haben Abstand 0
- Distanzen sehr teuer in Berechnung, da Bestimmung von MCS und mcs das Subgraph-Isomorphie Problem enthält (NP-hart).

Edit Distanzen auf Graphen

Idee: Abstand zweier Graphen entspricht den minimalen Kosten um *G* so abzuändern, dass *G* zu *G‘* isomorph ist.

- integriert Fehlertoleranz, indem Unterschiede bewertet werden
- Operationen: Löschen, Einfügen, Umlabeln von Knoten und Kanten.
- Jede Operation hat Kosten, die von den Labeln abhängen können.
- Eigenschaften wie Symmetrie, Definitheit und Dreiecksungleichung hängen von den Kosten der Edit-Operation ab.
- Die ***Graph Matching Distanz*** zwischen 2 Objekten entspricht:

$$d(G,G') = \min_S \{c(S) | S \text{ ist Sequenz von Operationen , die G in G' ändert}\}$$

- wobei *c(S)* die Kosten der Edit-Operationen darstellt.

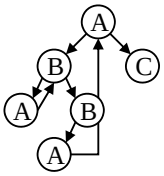
Problem:

- Graph- und Subgraph-Isomorphie können als Spezialfall der Edit-Distanz betrachtet werden => Berechnung sehr aufwendig
- Ergebnis stark von den Kosten der Edit-Operationen abhängig

Edit Distanzen auf Graphen

Performanz:

- allgemein kann die Komplexität nicht reduziert werden
- Einschränkung der Graphen
  z.B. auf Bäume.
  => Bäume können eindeutig als Strings dargestellt werden
  => Edit-Distanz auf Strings ist in $O(n^2)$
  => Problem: Einfügen eines Blatts verändert Baum anders als Einfügen eines inneren Knotens.



[A[B[A][B[A]]][C]] → [A[B[A][B[A]]][C]] → [A[B[A][B]][C]]

Deletion A in Blatt

Bestimmung der Edit-Kosten:

- Domain-Experten
- Mathematische Modelle
- Lernen der Kosten für Klassifikation

Fazit Edit Distanz

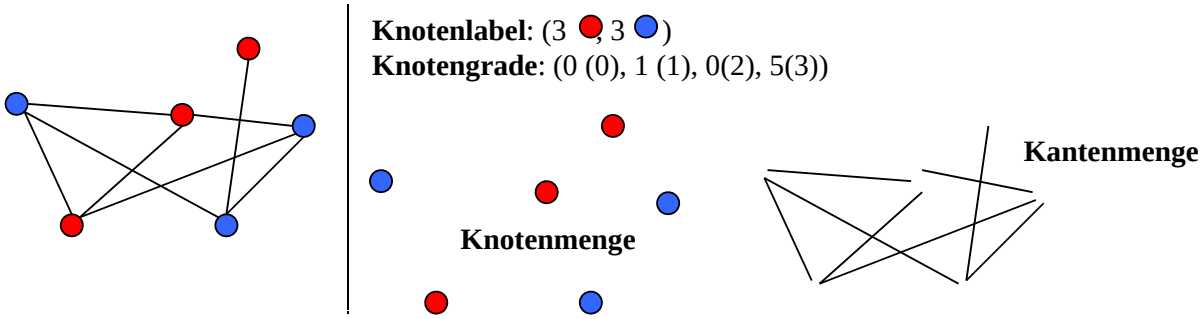
- Mathematische fundiertes Framework
- Graphen werden direkt und bzgl. all ihrer Eigenschaften verglichen
- Isomorphie-basierte Verfahren hängen davon ab wie $|G|$ definiert wird. (Kosten für Label, Kanten, Knoten..)
- Edit-Distanz verallgemeinert Isomorphie-basierte Verfahren
- Kosten und Art der Edit-Operationen bestimmen Distanz
- Kosten für den Vergleich von Graphen sehr hoch
  => nur auf wenige Graphen und kleine Graphen anwendbar
- Einschränkung auf bestimmte Topologien kann Problem entspannen. Aber: Verlust der Allgemeinheit von Graphen

Topologische Deskriptoren und Graph-Kernel

Idee: Da der direkte Vergleich der Graphen zu teuer ist, vergleiche nur die Eigenschaften der Graphen.

Mögliche Eigenschaften:

- Graph-Summarisierung: Bestimme Histogramme über Kantengrade, Kantenlänge, Label-Häufigkeiten..
- Betrachte Graphen als Menge von Kanten und Knoten
  => Graph besteht aus 2 Repräsentationen aus MI-Objekten



Topologische Deskriptoren

Aber: Bis jetzt keine Beschreibung der Graph-Topologie

⇒ Topologische Deskriptoren

z.B. Eigenschaften von Wegen, Pfaden, Subgraphen...

⇒ Topologische Deskriptoren zerlegen ein Graphen in eine Menge von einfacheren topologischen Objekten. Eventuell werden diese noch summarisiert.

Beispiel: Wiener Index

Sei G=(V,E) ein Graph. Dann ist der Wiener Index *W(G)* definiert durch:

$W(G) = \sum_{v_i \in G} \sum_{v_j \in G} d(v_i, v_j)$ wobei $d(v_i, v_j)$ die Länge des kürzesten Pfades von v_i nach v_j in *G* ist.

Bemerkung: Es gilt: Wenn $G \cong G'$ ⇒ **W(G) = W(G').**

Aber: W(G) = W(G') kann auch für unterschiedliche Graphen *G* und *G'* gelten.

Ähnlichkeitsmaße mit topologischen Deskriptoren

Idee: Benutze topologische Deskriptoren und Zerlegungen, um Ähnlichkeit zwischen Graphen zu beschreiben.

Ansätze:

- Ableiten von Feature-Räumen aus Topologischen Deskriptoren
- Integration von topologischen Zerlegungen in Distanzen und Kernel-Funktion
- Im folgenden werden überwiegend Kernel-Funktionen besprochen, da in diesem Gebiet mehr Forschungsergebnisse vorliegen.

R-Convolution Kernel

- Verallgemeinerung des Convolution-Kernels für Mengen auf zusammengesetzte Objekte
- Allgemeines Framework für fast alle Graph-Kernel.
- Korrektes Einsetzen in dieses Framework erleichtert den Beweis der positiven Definitheit.

- Sei $o \in O$ ein zusammengesetztes Objekt mit $Z(o) = (x_1, .. x_n)$ (=Zerlegung von o), bei der jede Komponente x_i Teil des Raums X_i ist.
- $R: X_1 \times .. \times R_n \rightarrow \{True, False\}$ gibt an, ob $(x_1, .. x_n)$ eine gültige Zerlegung von o ist.
- $R^{-1}(o) := \{x | R(o,(x1,..,xn)=True\}$ die Menge aller gültigen Zerlegungen.
- Der R-convolution Kernel der Kernelfunktionen $K_1,..K_D$ mit $K_i: X_i \times X_i \rightarrow IR$ ist

$$K(x, x') = K_1 * ... * K_n(x, x') = \sum_{x \in R^{-1}(x), x' \in R^{-1}(x')} \prod_{i=1}^{n} K_i(x_i, x_i')$$

Bemerkung:

- Alle Paare von gültigen Objektzerlegungen werden aufsummiert. Für jedes Paar aus Komponenten werden Kernel-Werte aufmultipliziert.
- Die Flexibilität steckt in der Menge der Zerlegungen und der Komponenten-Kernel

R-Convolution Kernel

Beispiel: Spezialisierung auf Mengen und lineare Kernel

Gegeben:Graph *G=(V,E)* mit $L: V \rightarrow IR^d$.

Zerlegung eines Graphen und Kernel: *Z(G)=V*

Kernel K: $\langle x,y \rangle$ linearer Kernel

$$K(G,G') = \sum_{\substack{v \in V \\ v' \in V'}} \prod_{i=1}^{1} \langle L(v), L(v') \rangle = \sum_{\substack{v \in V \\ v' \in V'}} \langle L(v), L(v') \rangle$$

Bemerkung:

- Der Convolution Kernel aus Kapitel 6 ist also ein R-Convolution Kernel.

R-Convolution auf topologischen Deskriptoren

- Sei *S(G)* die Menge aller Subgraphen von *G*.
- **All-Subgraph-Kernel** für *G* und *G'*:

$$K_{Subgraph}(G,G') = \sum_{g \in S(G)} \sum_{g' \in S(G)} K_{isomorphism}(g,g')$$

wobei

$$K_{isomorphism}(g,g') = \begin{cases} 1 & falls \quad g \cong g' \\ 0 & sonst \end{cases}$$

Bemerkungen:

- Vergleich aller Teilgraphen auf Isomorphie
- NP-harter Kernel, da das Subgraph-Isomorphie Problem Teil der Berechnung ist.

Der Produktgraph

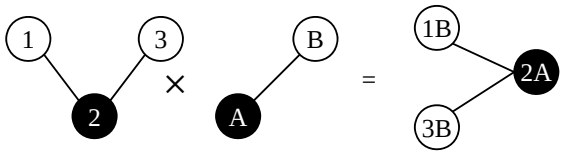
Idee: Um gemeinsame Wege von *G* und *G'* zu finden, kann man beide Graphen in einen Produktgraphen $G_\times = G \times G'$ zusammenfassen. Wege im Produktgraphen entsprechen dann den gemeinsamen Wegen von *G* und *G'*.

Produktgraph:

$G_\times = G \times G'$ für $G=(V,E,L)$ und $G=(V',E',L')$ ist folgendermaßen definiert:

$$V_\times = \{(v_i, v'_j) : v_i \in V \wedge v'_j \in V' \wedge L(v_i) = L(v'_j)\}$$

$$E_\times = \{((v_i, v'_j),(v_k, v'_l)) \in V \times V' : (v_i, v_k) \in E \wedge (v'_j, v'_l) \in E' \wedge L(v_i, v_k) = L(v'_j, v')\}$$



Random Walk Kernel

- **Idee:** Vergleiche 2 Graphen bzgl. der Anzahl aller Wege, die in beiden Graphen gelaufen werden können. Wege sind gleich, wenn man die gleichen Label und die gleiche Anzahl von Knoten in beiden Wegen beobachten kann.

- **Berechnung**:
  Zähle alle Wege in beiden Knoten auf und vergleiche diese.
- **Aber**: Wege können unendlich lang sein und Matching ist teuer.

- **Lösung:** Berechnung mit dem Produktgraphen:

$$K_\times(G,G') = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda^n A_\times^n \right]_{ij} = \sum_{i,j=1}^{|V_\times|} \left[(I - \lambda A_\times)^{-1} \right]_{i,j}$$

  - Bemerkung: Faktor $0 < \lambda < 1$ sorgt für Konvergenz.
  - Unter Konvergenz ist der Random Walk Kernel positiv definit.
  - (*I* ist eine Diagonalmatrix mit 1 in der Hauptdiagonale.)

Random Walk Kernel

Zeitkomplexität:

- für 2 Graphen G und G‘ sei n = max(|V|,|V‘|)
- Berechnung des Produktgraphen:
  - Vergleich aller Paare von Kanten: n^2 potentielle Kanten
  - Komplexität: $O(n^4)$
- Invertieren der Adjazenzmatrix:
  - Inversion einer $n^2 \times n^2$ Matrix : $O(n^6)$ (Invertieren ist kubisch)
- Laufzeitkomplexität für Berechnung eines Graphvergleichs: $O(n^6)$
- Fazit: Sehr teures Ähnlichkeitsmaß !!
  (Verfahren kann allerdings auf $O(n^3)$ beschleunigt werden [Vishwanathan et al. 2006])

Weitere Probleme mit Random Walks

„Tottering“

- Walk-Kernel erlauben, dass Wege die gleichen Knoten und Kanten
- mehrmals enthalten => durch hin und her Wandern zwischen den
- gleichen 2 Knoten wird die Ähnlichkeit künstlich erhöht.

Lösungsansätze:

- Einführen zusätzlicher Knoten-Labels
  ⇒ Matchende Knoten nehmen ab
  ⇒ Klassifikationsgenauigkeit steigt
- Verbieten von Zyklen aus 2 Knoten.
  ⇒ keine Verbesserung
  ⇒ Tottering tritt auch über mehr als 2 Knoten auf

Shortest Path Kernel

Idee: Einschränkung der Random Walks auf kürzeste Pfade (Shortest Path). Hierdurch weniger Teilkomponenten, die zu vergleichen sind und kein Tottering mehr.

- Berechne die Menge der kürzesten Pfade für G und G‘ separat
- Vergleich der Graphen als Menge aus kürzesten Pfaden
- Über Aufsummieren aller Kernel-Werte über alle Paare von kürzesten Pfaden kann ein Kernel auf Graphen definiert werden.

DATABASE SYSTEMS GROUP

Shortest Path Kernel

LMU

Berechnung der kürzesten Pfade:

- Berechnung über All-Pair Shortest Path Algorithmus
  (Floyd-Warshal Algorithmus: $O(n^3)$)

- Ergebnis definiert eine Matrix D:

$$M_{ShortestPath}(G)_{ij} = \begin{cases} d_{i,j} & falls \quad v_i \text{ erreichbar von } v_j \\ \infty & sonst \end{cases}$$

- Menge der kürzesten Pfade SD(G) beschreibt Graph G
  (unendliche Wege nicht in SD(G) enthalten)

Shortest Path Kernel

- Vergleich 2er Graphen über Convolution Kernel:

$$K_{shortestPath}(G,G') = \sum_{s_1 \in SD(G)} \sum_{s_2 \in SD(G')} k(s_1, s_2)$$

Bemerkung:

$k(s_1,s_2)$ ist eine Kernelfunktion auf Pfaden/Wegen

Möglichkeiten: -

- Vergleich der Distanzen
- Berücksichtigen der Start- und End-Label
- Berücksichtigen aller Label

Komplexität: $O(n^4)$

da n^2 mögliche kürzeste Pfade, die paarweise verglichen werden müssen. (bei komplexen Kerneln auf Pfaden höher !)

Bezug zu Distanzen auf Graphen

Häufig lassen sich Kernel direkt in Distanzen überführen, wenn das von der Anwendung her erforderlich ist.

1. Jeder Kernel impliziert auch ein Distanzmaß:

$$D(G,G') = \sqrt{K(G,G) + K(G,G') - 2 \cdot K(G,G')}$$

2. Auch konzeptionell können viele Ideen direkt übertragen werden:
   1. All-Subgraph: Zähle Anzahl nicht-isomorphe Subgraphen
   2. Shortest Path: Vergleich Mengen der kürzesten Pfade mit einem der Abstandsmaße aus Kapitel 6.

Fazit Graphvergleiche

- Modellierung von Objekten als Graphen erlaubt die Beschreibung von Beziehungen zwischen Teilobjekten.
- Graphen können gerichtet sein und gelabelte Knoten und Kanten haben. Als Label können beliebige andere Objektrepräsentationen dienen. (Meistens: 1 diskretes Attribut)
- Komplexität von Graphen schränkt ihre Anwendbarkeit ein.
- Vergleich über topologische Eigenschaften erlaubt die Verwendung von Beziehungswissen mit vertretbarem Aufwand.
- Topologischer Vergleich arbeitet oft auf Transformation eines Graphen in weniger komplexe Darstellungen
  (z.B. MI-Objekte oder Feature-Vektoren).
- Dabei geht ein Teil der Information verloren.
  (z.B. Ungleiche Objekte können Distanz 0 haben.)

Graphdatenbanken und Anwendungen

Bisher: Objekte sind iid (idependent and identical distributed)

⇒ Bedeutung der Objekte hängt nur von ihren Objektwerten ab.

⇒ Es gibt keine gegenseitige Beeinflussung und keine Beziehungen.

Jetzt: Link-Mining

Daten sind durch Beziehungen untereinander verbunden.

Bsp: Wichtigkeit eines Papers wird an Referenzierungen gemessen.

Ein Paper wird als wichtig betrachtet, wenn es in vielen anderen erwähnt wird. Der Inhalt wird dabei nicht betrachtet.

⇒ Objekte beeinflussen sich gegenseitig

⇒ Modellierung der Daten als großer Graph
Objekte sind Knoten und Beziehungen sind Kanten

Frequent Subgraph Mining

Idee: Finde alle häufigen Subgraphen in Netzwerken.

Anwendungen:

- Auftreten von häufiger Subgraphen kann als topologischer Deskriptor dienen.
- Finden typischer Subnetze (Cliquen) in sozialen Netzwerken
- Graph-Kompression: Substituieren häufiger Subgraphen durch neue Knoten => Graph wird kleiner.
- Ableiten von Regeln über Verbindungen von Personen

Ansätze im Frequent Subgraph Mining

- *Frequent Subgraph Mining ist ähnlich zum Itemset Mining*
  - Ausnützen von Monotonie für die Kandidatenbildung
    => *k* Itemset *I* nur frequent, wenn alle *k-1* Itemsets in *I* auch frequent
    analog: Subgraph *G* mit k Knoten kann nur frequent sein, wenn alle Subgraphen von *G* mit *k-1* Knoten auch frequent sind
  - Generierung von Kandidaten der Größe *k* durch Kombination von 2 häufigen Graphen der Größe *k-1*.
- *Direktes Vergrößern der Graphen um jeweils einen Knoten*
  - Finde alle Graphen mit *k* Knoten und erweitere diese um einen weiteren Knoten => Kandiaten für *k+1* elementige Graphen

Grundproblem für Frequent Subgraph Mining

Test auf Gleichheit 2er Subgraphen (Subgraph-Isomorphie !!):

- Entscheidung, ob Kandidat in einem Datenbankgraphen enthalten ist, ist bereits Subgraph-Isomorphie-Test.

  ⇒ Es ist wichtig möglichst viele DB-Subgraphen auszuschließen bevor wirklich auf Subgraph-Isomorphie getestet wird.

- Bestimmen des korrekten Supports

  Mehrere Gruppen von isomorphen Subgraphen werden getrennt betrachtet

  ⇒ Support jeder Gruppe für sich zu niedrig, aber alle zusammen können frequent sein.

- Vermeidung von doppelten Kandidaten-Subgraphen

  Es gibt exponentiell viele isomorphe Subgraphen.

  ⇒ generiere alle Graphen und streiche alle, die zu einem bereits vorhandenen Subgraphen isomorph sind.

  ⇒ generiere nur 1 Subgraphen für jede Isomorphie-Klasse

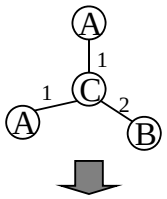
Lösungen Frequent Subgraph Mining

FSG [Kuramochi, Karypis 2001]

Geht von Menge von gelabelten, ungerichteten Graphen aus.

Idee: Erweiterung des Apriori-Algorithmus zum Itemset Mining auf Graphen.

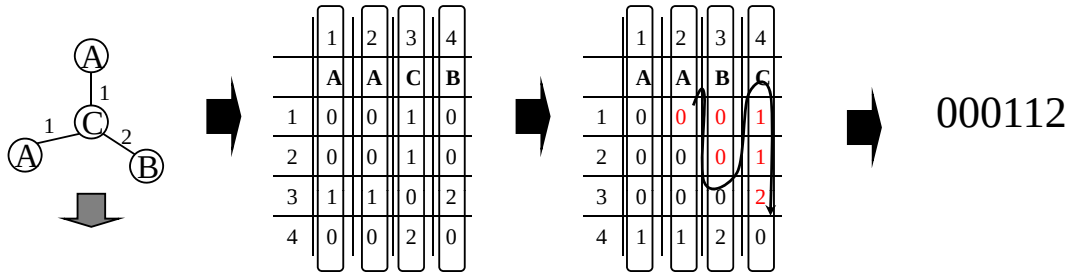
- Darstellung der Graphen als Adjazenz-Listen
- Isomorphe Graphen entsprechen den Permutationen der Adjazenz-Listen



	1	2	3	4
	A	**A**	**C**	**B**
1	0	0	1	0
2	0	0	1	0
3	1	1	0	2
4	0	0	2	0

⇒ Canoncial Labelling
Finde eine eindeutige kanonische Form für jede Isomorphie-Klasse.

Canonical Labeling



- Sortieren der Spalten nach Grad der Knoten
- bilden aller Permutationen
- Auslesen der oberen Dreiecksmatrix
- Auswahl der lexikografisch kleinsten Zeichenkette

⇒ eindeutige Zeichenkette repräsentiert jetzt Menge von isomorphen Graphen
Verbesserung durch Gruppieren der Listen nach Knotenlabeln

⇒ verlangt nur noch Permutationen innerhalb jeder Gruppe.

⇒ weniger Permutationen müssen getestet werden

FSG Algorithmus(1)

Vector<Graphmenge> fsg(Graphmenge D, double δ)

```
Graphmenge F1 = Menge aller häufigen Subgraphen mit einer Kante
Graphmenge F2 = Menge aller häufigen Subgraphen mit zwei Kanten
int k=3
Vector<Graphmenge> frequentSubgraphs;
frequentSubgraphs.add(F1)
frequentSubgraphs.add(F2)
while(frequentSubgraphs.getLastElement()!= {})
        Graphmenge Ck= fsg-gen(frequentSubgraphs.getLastElement());
        foreach Graph c ∈ Ck
                int anzahl_c_in_D =0;
                foreach Graph d ∈ D
                        if(d.includes(c))
                           anzahl_c_in_D ++;
                if(anzahl_c_in_D < δ*|D|)
                  ck.remove(c);
        frequentSubgraphs.add(Ck);
return frequentSubgraphs;
```

FSG Algorithmus(2) (Kandidatengenerierung)

Graphmenge fsg-gen(F^k)

```
Graphmenge Ck+1={};
foreach Graph f1k ∈ F^k
        foreach Graph f2k ∈ F^k
          if(f1k.canonicalLabel <= f2k.canonicalLabel)
            foreach Edge e ∈ f1k
                Graph f1k-1=f1k.remove(e);
                if(f1k-1.isconnected && f2k.includes(f1k-1))
                  Graphmenge Tk+1 = Alle durch Join von f1k und f2k enstehende Graphen
                forech Graph tk+1 ∈ Tk+1
                   boolean all_tk_frequent = true;
                  foreach Edge ed ∈tk+1
                     Graph tk = tk+1.remove(ed);
                      if(tk.isConnected && tk ∉ FK)
                            all_tk_frequent = false;
                            break;
                     if(all_tk_frequent)
                            Ck+1.add(tk+1);
return Ck+1
```


Komplexität von FSG

Im Code enthaltene komplexe Passagen:

1. Subgraph-Isomorphie-Test (g.includes(s))
   - notwendig beim Datenbank-Scan im Hauptalgorithmus
   - notwendig bei der Kandidatengenerierung:

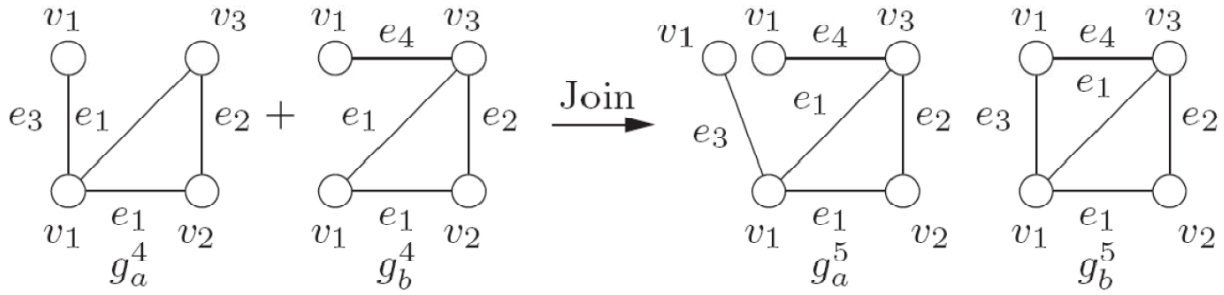
     Test auf gemeinsame *k-1* Subgraph
2. Join zweier Graphen auf Basis eines **k-1** Subgraphen

   ⇒ es gibt mehrere mögliche Ergebnisse

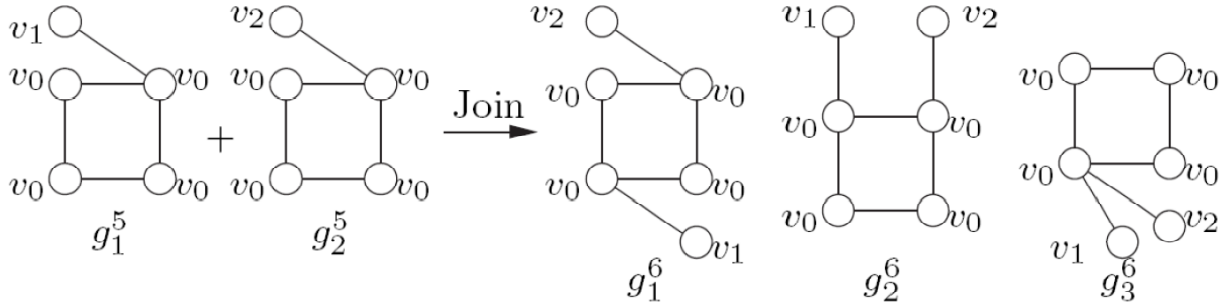
   ⇒ alle müssen als Kandidaten in Betracht gezogen werden

Fazit: Algorithmus ist sehr teuer !

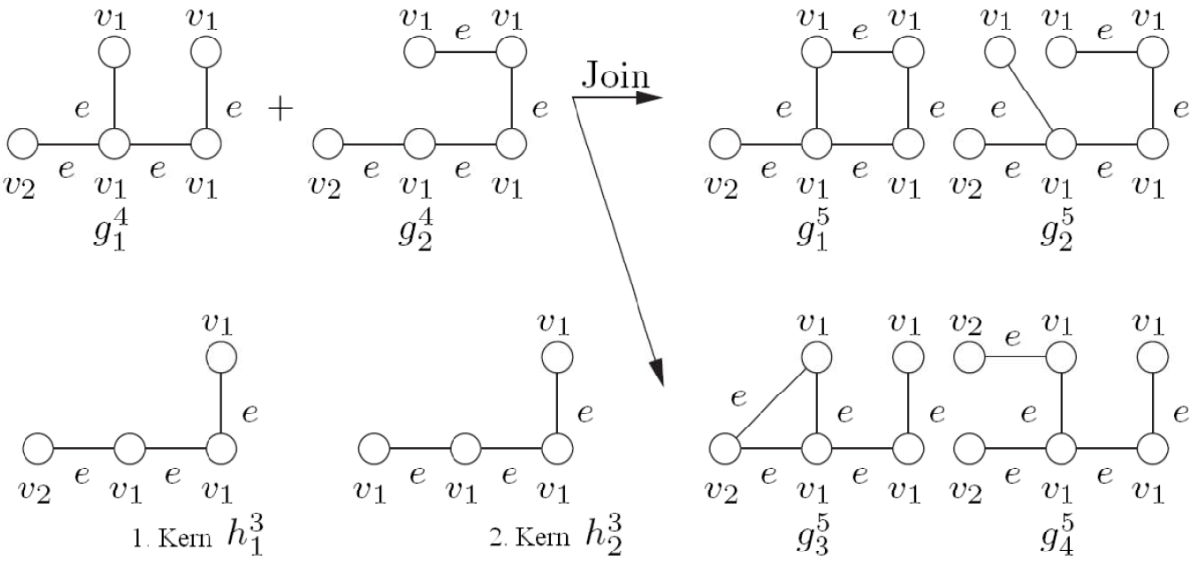
Mögliche Kandidaten (1)



1. Gleiche Knoten-Markierung der den Kern erweiternden Knoten



Mögliche Kandidaten(2)



GSpan

Ideen:

- Erweitere häufige Sugraphen mit *k* Knoten direkt um 1 Kante.
- Annotation von Graphen durch Tiefensuchbaum (DFS-Code)
- Einschränkung der Kandidatengenerierungedurch „right-most-only growth“

Ziel:

- Vermeide das Generieren von isomorphen Kandidaten
- Vermeide möglichst viele Isomorphismie-Tests

Verwendete Konzepte:

- DFS-Lexikographische Ordnung
- minimaler DFS-Code (kanonische Annotation eines Graphen)

Pattern Growth

Naiver Pattern Growth Ansatz:

S :Menge der frequent Graphs;
g :frequent Subgraph,
DB: Graphdatenbank
MinSup: minimaler Support des SubGraphen

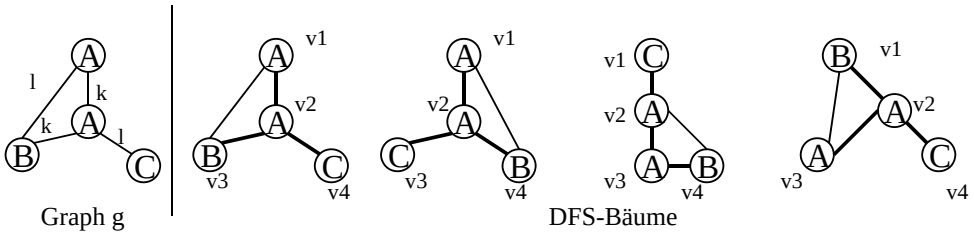
```
S:={}
GrowPatterns(g,DB, S)

Function GrowPatterns(g,DB,S)
    if g ∈ S then return;
    else S.insert(g)
    EdgeSet E = findAdjacentEdges(DB,g MinSup); // finden aller Kanten in DB die g erweitern können
    for each frequent e ∈ E DO   //es werden nur Kanten betrachtet die öfter als MinSup auftreten
      g‘ = extend(g,e)
      GrowPatterns(g‘,DB,S)
    end for
end function
```

Bemerkung:

Finden der Erweiterung sehr teuer und Isomorphie-Test für g ∈ S
Klassen von isomorphen Graphen sollten in findAdjacentEdges nur 1 mal untersucht werden

DFS Codes

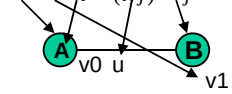
- Einführen eines kanonisches Labelings zur Erkennung isomorphen Graphen
- Beschreibe Graph durch Kantenreihenfolge in einem Tiefensuchbaum (***D***epth ***F***irst ***S***earch Baum)



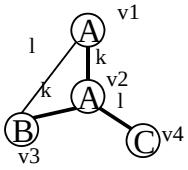
- Unterscheide ***Vorwärtskanten***: Erweitern Graph um neuen Knoten
  Rückwärtskanten: Verbinden bereits vorkommende Kanten
- Ein DFS-Tree impliziert Reihenfolge der Kanten in Graph *G* (DFS-Code)
- Reihenfolge entspricht Tiefensuch-Reihenfolge **(v1,v2,..)**
- Rückwärtskanten werden eingefügt, nachdem Startknoten erreicht wurde
- Reihenfolge der Rückwärtskanten nach Besuchsreihenfolge der Zielknoten

DFS-Lexikographical Order

- Graph kann durch Menge seiner DFS-Trees beschrieben werden
- DFS-Tree ist eindeutig durch Kantensequenz DFS-Code gegeben
- Darstellung einer Kante: durch $(i, j, l_i, l_{(i,j)}, l_j)$ ⇒ **(0, 1, A, u, B)**



- Beispiel: DFS-Code



1,2,A,k,A
2,3,A,k,B
3,1,B,l,A
2,1,A,l,C

Vorwärtskanten
Rückwärtskante

- DFS-Lexicographical-Order: Vergleich meherer DFS-Codes
- Lexikografischer Vergleich der Kantensequenz
- Vergleich der Kanten: Startindex, Zielindex, Startlabel, Katenlabel, Ziellabel.
- Durch Ordnung kann man jetzt kanonischen DFS-Code bestimmen:
- Kleinster DFS-Code (Min DFS-Code) von G bezüglich DFS-Lexicographical Order beschreibt Graph G eindeutig.
- => Haben 2 Graphen *G,G'* gleiche Min-DFS-Codes ⇔ *G* ist isomorph zu *G'*

Right-Most-Only Extension

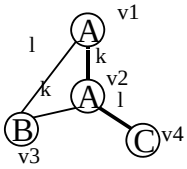
Idee: Vermeide mehrfache Untersuchung monotoner Kandidaten

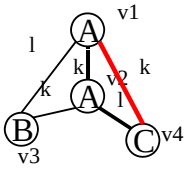
$\Rightarrow$ Bilde Kandidaten nach speziellen Regeln

- ***Right-Most-Only Extension*** erlaub nur Erweiterungen entlang des rechtesten Pfades im DFS-Baum der Min-DFS-Code impliziert.
- **DFS-Tree**:
  - *Backward-Extension*

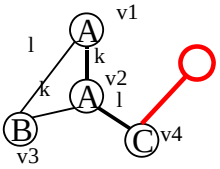
    Verbinde Knoten auf dem rechtesten Pfad mit Rückwärtskante
  - *Forward Extension*

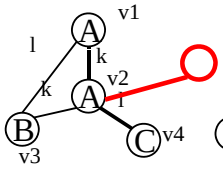
    Erweitere Graph um einen neuen Knoten. Verbindende Kante beginnt auf dem rechtesten Pfad.

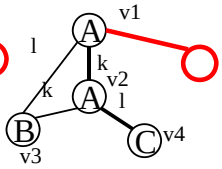




Backward-Extension







Forward-Extensions

GSpan

Pattern Growth Algorithmus, der nur right-most-only Extensions, des minimalen DFS-Tree erlaubt.

GSpan

S :Menge der frequent Graphs;
s : a DFS Code
min_dfs(s): Mininmale DFS-Code von S.
DB: Graphdatenbank
MinSup: minimaler Support des SubGraphen

S:={}
GSpan(s,DB, S)

```
Function GrowPatterns(g,DB,S)
    if s ≠ min_dfs(s) then return;
    else S.insert(s)
    C := {}
    EdgeSet E = findRightMostExtensions(DB,s, MinSup); // finden aller gültigen Erweiterungen minimaler DFS-Trees
    C = extend(s,E);
    C.sortInLexDFSOrder;
    for each frequent s ∈ C DO
        GSpan(s,DB,S)
    end for
end function
```

Frequent Subgraph Mining

Finden häufiger Subgraphen ist ähnlich zum Itemset Mining

Aber:

- Menge aller isomorphen Graphen ist größer als Menge aller Permuationen über Strings ⇒ Isomorphie schwieriger als Itemset- Vergleich.
- Finden kanonischer Labels ist deutlich aufwendiger
- Menge der möglichen Erweiterung eines Graphen ist auch deutlicher größer als Erweiterung von Itemsets ⇒ Kandidatengenerierung sehr teuer

- **FSG**: Apriori-basierter Ansatz mit paarweiser Kandidatengenerierung

- **GSpan**: Pattern-Growth Ansatz, der momentan als am schnellsten berichtet wird.