

**Skript zur Vorlesung
Knowledge Discovery in Databases II
im Wintersemester 2013/2014**

Kapitel 6: Graph und Link Mining

Skript © 2013 Matthias Schubert und Karsten Borgwardt

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

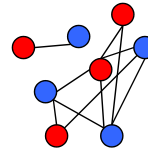
1

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 Prediction
Link-Klassifikation, Matrix Faktorisierung
7. Clustering und Graphen
Cliques und Spectral-Clustering

2

- 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!
⇒ Verwende einfachere Datenstrukturen wenn möglich !

Beispiele:

- Molekulare Strukturen, RNA-Transkription
- Bio-Informatik: Protein-Interaktionsnetzwerke, Phylogenetische Netzwerke, Metabolom-Netzwerke..
- Soziale Netzwerke: Vergleich ähnlicher sozialer Gruppen
(Zusammenarbeit in Arbeitsgruppe, Zuspil 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.)

Erweiterungen der Definition:

GRAPH: 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.

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 attribuiert bzgl. der Knoten V , wenn es zu jedem Knoten $v \in V$ genau eine Feature-Beschreibung $I_v \in F_V$ gibt.

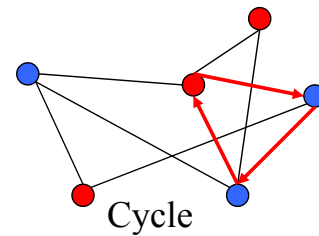
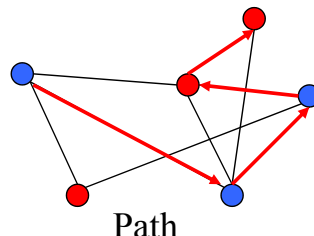
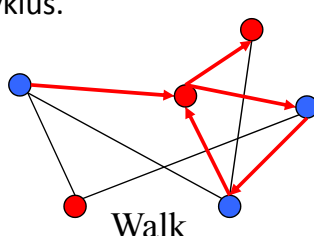
Ein Graph G heißt gelabelt oder attribuiert bzgl. der Kanten E , wenn es zu jeder Kante $e \in E$ genau eine Feature-Beschreibung $I_e \in F_E$ gibt.

Bemerkung:

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

- **Knotengrad:** Der Grad $d_G(v_i)$ eines Knotens v_i in $G=(V,E)$ ist die Anzahl der anliegenden Kanten: $d_G(v_i) = |\{v_j \mid (v_i, v_j) \in E\}|$
- **Adjanzenz-Matrix:** Die Adjanzenz-Matrix eines Graphen $G=(V,E)$ ist gegeben durch:

$$[A]_{i,j} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$$
- **Weg (Walk):** Ein Weg in $G=(V,E)$ der Länge $k-1$ ist eine Sequenz von Knoten $w=(v_1, v_2, \dots, v_k)$ 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 $i \neq j$.
($\Rightarrow$ Kein Knoten darf in w zweimal auftauchen.)
- **Zyklus (Cycle):** Sei $w=(v_1, \dots, v_k)$ und $v_1 = v_k$ und für alle $1 < i, j < k$ gilt $v_i \neq v_j$ dann ist w ein Zyklus.



Gegeben: 2 Graphen G und G' .

Gesucht: Abbildung $s: (V \times E) \times (V \times E) \rightarrow \mathbb{R}$, die die Ähnlichkeit von G und G' quantifiziert.

Ansätze:

Isomorphie: 2 Graphen sind gleich, wenn es eine eindeutige 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)

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 Bijektion $f: V \rightarrow V'$ 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 gemeinsamen 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.

- 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-hard).

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 Labels abhängen können.
- Eigenschaften wie Symmetrie, Definitheit und Dreiecksungleichung hängen von den Kosten der Edit-Operation ab.
- Die **Graph Matching Distanz** zwischen 2 Objekten entspricht:

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

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.



Bestimmung der Edit-Kosten:

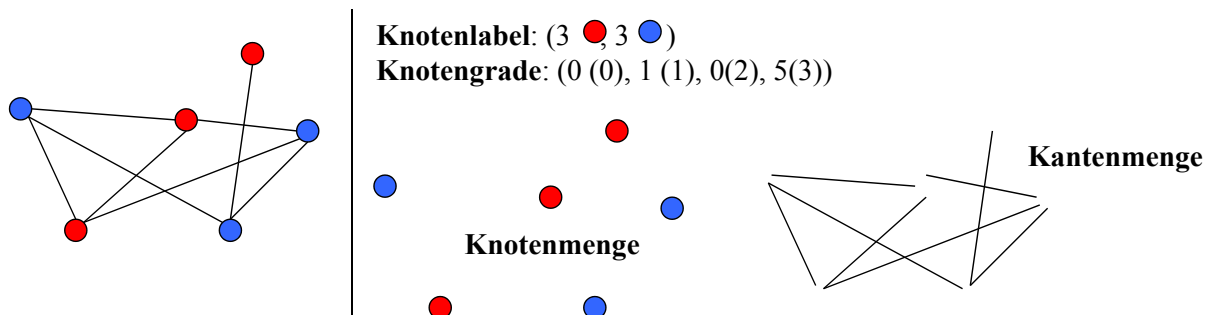
- Domain-Experten
- Mathematische Modelle
- Lernen der Kosten für Klassifikation

- 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

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



13

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 $G \cong G' \Rightarrow W(G) = W(G')$.

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

14

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.

- 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 $Z(o) = (x_1, \dots, x_n)$ (=Zerlegung von o), bei der jede Komponente x_i Teil des Raums X_i ist.
- $R: X_1 \times \dots \times X_n \rightarrow \{True, False\}$ gibt an, ob $(x_1, \dots, x_n)$ eine gültige Zerlegung von o ist.
- $R^{-1}(o) := \{x \mid R(o, (x_1, \dots, x_n)) = True\}$ die Menge aller gültigen Zerlegungen.
- Der R-convolution Kernel der Kernelfunktionen $K_1, \dots, K_n$ mit $K_i: X_i \times X_i \rightarrow \mathbb{R}$ ist

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

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

Beispiel: Spezialisierung auf Mengen und lineare Kernel

Gegeben: Graph $G=(V,E)$ mit $L: V \rightarrow \mathbb{R}^d$.

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 8 ist also ein R-Convolution Kernel.

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

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

wobei

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

Bemerkungen:

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

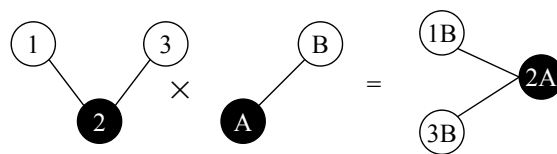
Idee: Um gemeinsame Wege von G und G' zu finden, kann man beide Graphen in einen Produktgraphen $G_{\times} = G \times G'$ zusammenfassen. Wege im Produktgraphen entsprechen dann den gemeinsamen Wegen von G und G' .

Produktgraph:

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

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

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



- **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:

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

- 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.)

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])

„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
- Verboten von Zyklen aus 2 Knoten.
 - ⇒ keine Verbesserung
 - ⇒ Tottering tritt auch über mehr als 2 Knoten auf

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.

Berechnung der kürzesten Pfade:

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

- Ergebnis definiert eine Matrix D:

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

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

- 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 Kernen auf Pfaden höher !)

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 8.

- 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.)

Bisher: Objekte sind iid (independent 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

Idee: Finde alle häufigen Subgraphen in Netzwerken.

Anwendungen:

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

- *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 => Kandidaten für $k+1$ elementige Graphen

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

31

FSG [Kuramochi, Karypis 2001]

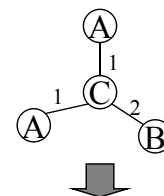
Geht von Menge von gelabelten, ungerichteten Graphen aus.

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

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

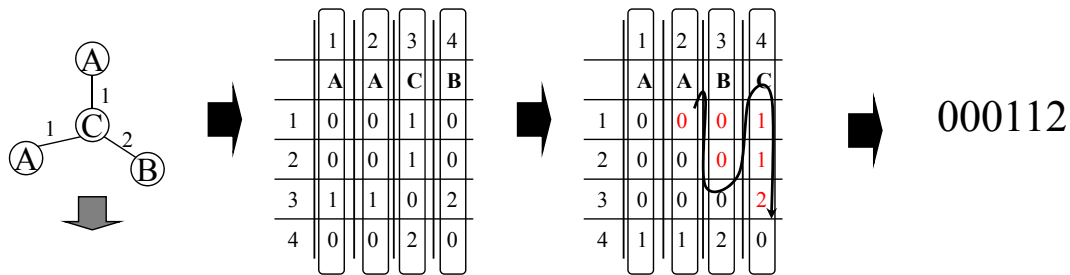
⇒ Canoncial Labelling

Finde eine eindeutige kanonische Form für jede Isomorphie-Klasse.



	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

32



- 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 Knotenlabels
- ⇒ verlangt nur noch Permutationen innerhalb jeder Gruppe.
- ⇒ weniger Permutationen müssen getestet werden

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 $\in$ Ck

int anzahl_c_in_D = 0;

foreach Graph d $\in$ D

if(d.includes(c))

anzahl_c_in_D ++;

if(anzahl_c_in_D < $\delta * |D|$)

ck.remove(c);

frequentSubgraphs.add(Ck);

return frequentSubgraphs;

Graphmenge fsg-gen(F^k)

```

Graphmenge Ck+1={};
foreach Graph f1k ∈ Fk
    foreach Graph f2k ∈ Fk
        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 entstehende Graphen
                    foreach Graph tk+1 ∈ Tk+1
                        boolean allTk_frequent = true;
                        foreach Edge ed ∈ tk+1
                            Graph tk = tk+1.remove(ed);
                            if(tk.isConnected && tk ∉ FK)
                                allTk_frequent = false;
                                break;
                        if(allTk_frequent)
                            Ck+1.add(tk+1);
return Ck+1

```

35

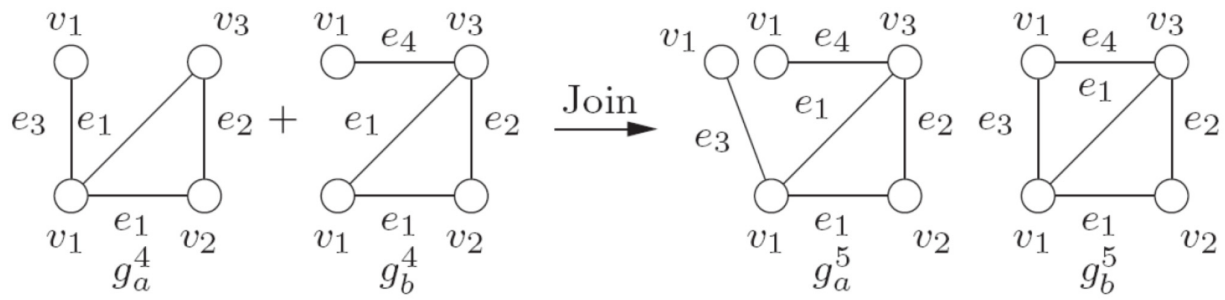
Im Code enthaltene komplexe Passagen:

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

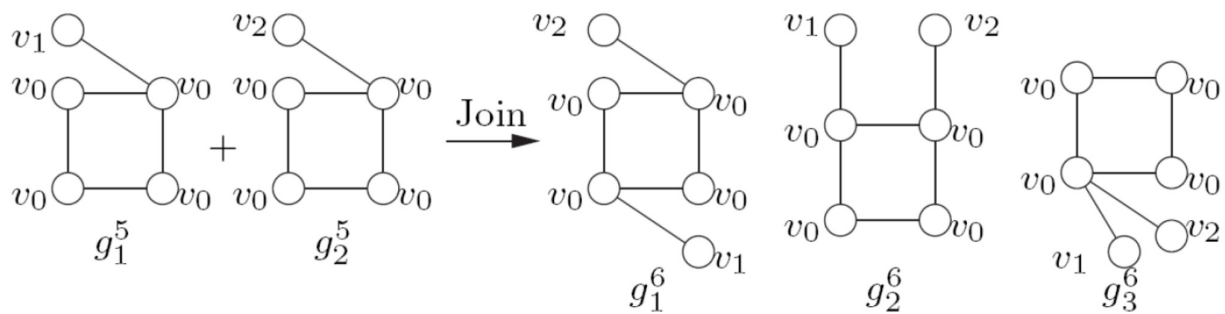
Fazit: Algorithmus ist sehr teuer !

36

Mögliche Kandidaten (1)

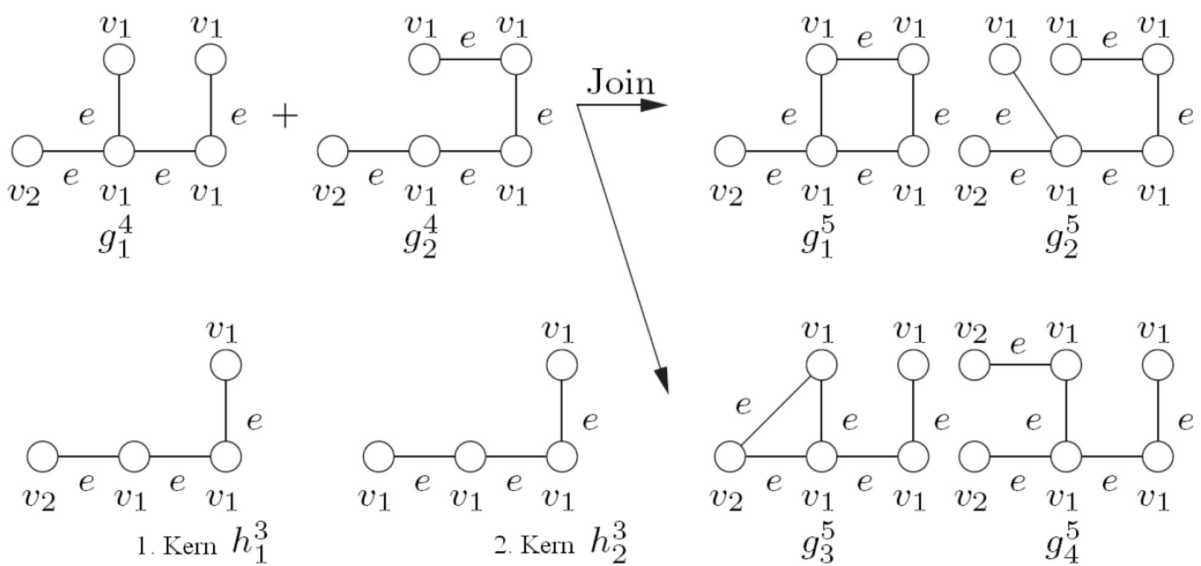


1. Gleiche Knoten-Markierung der den Kern erweiternden Knoten



37

Mögliche Kandidaten(2)



38

Ideen:

- Erweitere häufige Sugraphen mit k Knoten direkt um 1 Kante.
- Annotation von Graphen durch Tiefensuchbaum (DFS-Code)
- Einschränkung der Kandidatengenerierung durch „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)

39

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 $g \in S$ then return;

else $S.insert(g)$

EdgeSet $E = findAdjacentEdges(DB, g, MinSup)$; // finden aller Kanten in DB die g erweitern können

for each frequent $e \in E$ DO //es werden nur Kanten betrachtet die öfter als MinSup auftreten

$g' = extend(g, e)$

GrowPatterns(g', DB, S)

end for

end function

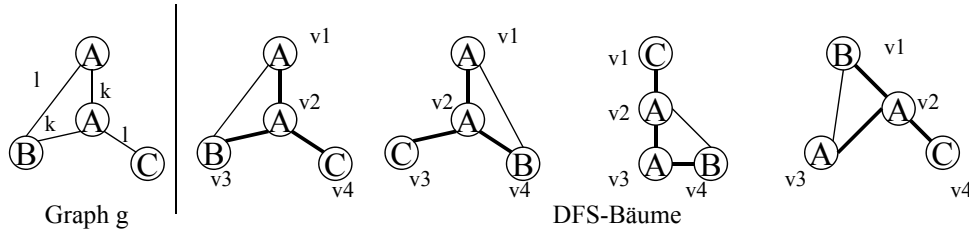
Bemerkung:

Finden der Erweiterung sehr teuer und Isomorphie-Test für $g \in S$

Klassen von isomorphen Graphen sollten in findAdjacentEdges nur 1 mal untersucht werden

40

- Einführen eines kanonischen Labelings zur Erkennung isomorpher Graphen
- Beschreibe Graph durch Kantenreihenfolge in einem Tiefensuchbaum (Depth First Search Baum)

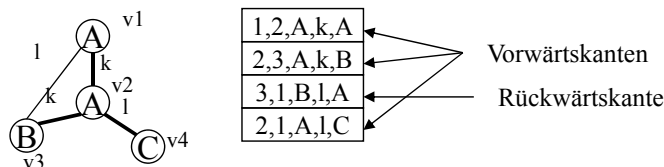


- Unterscheide **Vorwärtskanten**: Erweitern Graph um neuen Knoten
- **Rückwärtskanten**: Verbinden bereits vorkommende Knoten
- Ein DFS-Tree impliziert Reihenfolge der Kanten in Graph G (DFS-Code)
- Reihenfolge entspricht Tiefensuch-Reihenfolge (**v1,v2,..**)
- Rückwärtskanten werden eingefügt, nachdem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

41

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

- Beispiel: DFS-Code



- DFS-Lexicographical-Order: Vergleich mehrerer DFS-Codes
- Lexikografischer Vergleich der Kantensequenz
- Vergleich der Kanten: Startindex, Zielindex, Startlabel, Kantenlabel, 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.
- $\Rightarrow$ Haben 2 Graphen G, G' gleiche Min-DFS-Codes $\Leftrightarrow G$ ist isomorph zu G'

42

Idee: Vermeide mehrfache Untersuchung monotoner Kandidaten

⇒ Bilde Kandidaten nach speziellen Regeln

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

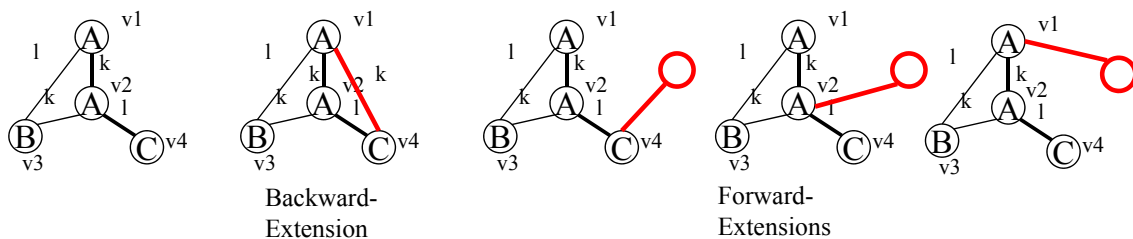
- **DFS-Tree:**

- *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.



43

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): Minimale 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

44

Finden häufiger Subgraphen ist ähnlich zum Itemset Mining

Aber:

- Menge aller isomorphen Graphen ist größer als Menge aller Permutationen über Strings $\Rightarrow$ 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 $\Rightarrow$ Kandidatengenerierung sehr teuer
- **FSG:** Apriori-basierter Ansatz mit paarweiser Kandidatengenerierung
 - **GSpan:** Pattern-Growth Ansatz, der momentan als am schnellsten berichtet wird.

45

Ziel: Selektieren und Ranken besonders relevanter und interessanter Knoten/Subgraphen in großen Netzwerken.

Aufgaben:

- Ranking der Knoten nach Einfluss
- Suche nach Schlüsselknoten

Anwendungen:

- Ranking von Webpages
- Bestimmung von zentralen Personen in sozialen und Zitationnetzwerken
- Ranking von Kreuzungen in Straßennetzwerken

46

Idee: Zentralität hängt von der Position des Knotens bzgl. der Netzwerkdistanz (= Kosten des kostenminimalen Pfades) ab.

Sei $d(v,t)$ die Länge des kürzesten Pfades von v zu t ($v,t \in V$) in $G(V,E)$:

- **Closeness Centrality:** $C_c(v) = \frac{1}{\sum_{t \in V'} d(v,t)}$
- **Graph Centrality:** $C_G(v) = \frac{1}{\max_{t \in V'} (d(v,t))}$

Sei σ_{st} Anzahl der kürzesten Pfade von s nach t und sei $\sigma_{st}(v)$ Anzahl kürzester Pfade von s nach t , die v enthalten.

- **Stress Centrality:** $C_S(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)$
- **Betweenness Centrality:** $C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$

47

Beispiel: Betrachte Knoten als Router

Bei Ausfall des Routers mit der höchsten Betweenness Centrality, fallen am meisten direkte Kommunikationsverbindungen aus.

Berechnung: Menge aller kürzesten Pfade lässt sich mit dem Algorithmus von Floyd-Warshall in $O(n^3)$ bzgl. und in $O(n^2)$ bzgl. Speicher bestimmen.

Lemma: v liegt genau dann auf einem kürzesten Pfad von s nach t wenn gilt

$$d(s,t) = d(s,v) + d(v,t)$$

$$\Rightarrow \sigma_{st}(v) = \begin{cases} 0 & \text{wenn } d(s,t) < d(s,v) + d(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{sonst} \end{cases}$$

$\Rightarrow$ zur Berechnung der Betweenness sind nicht alle Pfade notwendig

$\Rightarrow$ Es existieren schnellere Lösungen:

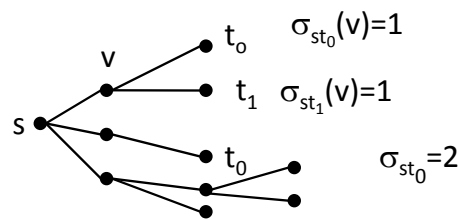
- $O(nm)$ ohne Kantengewichte
- $O(nm + n^2 \log n)$ mit Kantengewichten

mit $n = |V|$ und $m = |E|$ für den Graph $G(V,E)$

48

Grundidee:

- Starte für jeden Knoten s einen best-first Suche zu allen anderen Knoten. Ergebnis ist ein Baum der alle kürzesten Pfade in s zu allen anderen Knoten t in G enthält. (wird auch als Dijkstra-Baum bezeichnet)
- Laufe von jedem Blatt zurück in Richtung s und zähle die Knoten, die im Baum hinter jedem Knoten v liegen ($\sigma_{st}(v)$) und die Menge der kürzesten Pfade (σ_{st})



Erläuterungen:

- S: Stack der die Knoten nach Distanz zu Startknoten s aufammelt
- Q: Priority Queue die Knoten nach Distanz zu s verwaltet (best-first search)
- P[v]: Liste mit Vorgängern von v auf kürzesten Pfaden
- d[v]: aktuell kürzester Pfad von s zu v
- $\sigma[v]$: Anzahl der kürzesten Pfade von s zu v
- $\delta[v]$: Sei $\delta_{st}[v] = \frac{\sigma_{st}[v]}{\sigma_{st}}$ dann ist $\delta[v] = \delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}[v] = \sum_{w: v \in P[w]} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$

Ablauf:

1. Phase: Algorithmus berechnet Dijkstra-Baum bzgl. aktuellem s
2. Phase: Stack S wird abgebaut und dabei wird die Anzahl der hinter dem Knoten Ziele nach vorne weitergegeben.

<pre> CB[v] := 0 $\forall v \in V$ for s $\in V$ S := empty Stack; P[w] := empty List $\forall w \in V$; $\sigma[t] := 0 \quad \forall t \in V$; $\sigma[s] := 1$; d[t] := -1 $\forall t \in V$; d[s] := 0; Q := empty Queue; Q.push(0, s); while Q not empty do v := Q.pop(); S.push(v); foreach neighbor w of v do if d[w] < 0 then d[w] := d[v] + 1; Q.push(d[w], w); end if end foreach end while </pre>	<pre> if d[w] = d[v] + 1 then $\sigma(w) := \sigma(w) + \sigma(v)$ P[w].add(v) end if end foreach end while $\delta[v] := 0; \forall v \in V$; while S not empty do w := S.pop(); for v $\in$ P[w] do $\delta[v] := \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$ end for if w $\neq$ s then CB[w] := CB[w] + $\delta[w]$; end if end while end for </pre>
---	---

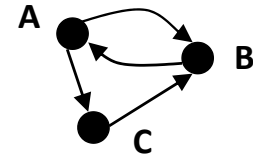
PageRank: (S.Brin/B. Page 1996)

- wichtige Komponente im Ranking moderner Suchmaschinen.
(in Kombination mit Ankertexten, Abstand, Worthäufigkeit)
- Web als stark verbundener, gerichteter Graph $G(V, E)$.
(Betrachtet gesamten Graph aus bekannten Webpages
z.B. alle in Suchmaschine gespeicherten Pages.)
- Zufallssurfer macht unendlichen Random-Walk.
Suche: Aufenthaltswahrscheinlichkeit für Page v
= „Prestige“ der Webpage v.

Berechnung Pagerank

Startverteilung: $p_0(u) = 1 / |V|$

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



Adjazenzmatrix E

$$\text{Übergangsw'keit: } L[u, v] = \frac{E[u, v]}{\sum_{\beta} E[u, \beta]}$$

$$L = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

W'keit für Zeitpunkt i und Page v: $p_i[v] = \sum_{u \in V} L[u, v] p_{i-1}(u)$

Als Vektor aller Knoten: $\vec{p}_i = L^T \vec{p}_{i-1}$

Berechnung über „Power Iterations“: $\vec{p}_i \leftarrow L^T \vec{p}_{i-1}$

nach ca. 20-30 Iterationen stabil

Lösung für nicht stark verbundene Graphen: 1. Weglassen von Knoten ohne Links
2. Zufallsprünge auf beliebige Seite

HITS (Kleinberg 1998): Hyperlink Induced Topic Search

- Betrachte alle Seiten, die für Anfrage q relevant sind oder die mit einer relevanten Seite verlinkt sind (In- und Outlinks).

$\Rightarrow G_q(V_q, E_q)$ für Anfrage q

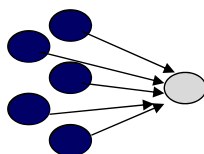
- Zwei Typen von Webpages:

Hubs: Verlinken auf viele relevante Webpages (Authorities)

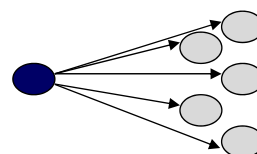
Authorities: Werden von vielen guten Hubs verlinkt.

=> Jede Webpage ist zu gewissen Grad sowohl Hub als auch Authority.

Für Webpage u sei $h[u]$ der Hub-Score und $a[u]$ der Authority-Score.



Eine gute Authority wird von vielen guten Hubs referenziert.



Ein guter Hub referenziert viele gute Authorities.

Berechnung von HITS:

- $\vec{a}$ Vektor aus Authority-Scores aller $v \in V_q$
- $\vec{h}$ Vektor aus Hub-Scores aller $v \in V_q$
- Berechnung über wechselseitige Iterationen: $\vec{a} = E^T \vec{h}$ (Authorityscore)
 $\vec{h} = E \vec{a}$ (Hubscore)

Vorgehen:

1. Bestimme alle relevanten Pages (Rootset).
2. Bestimme alle Pages, die Rootset verlinken.(Extendedset)
3. Iteriere über Hub- und Authority-Scores (Extended Set)
4. Sortiere Ergebnisse nach Hub-Score, Authority-Score oder Kombination.

Gegeben: Ein Graph $G(V,E)$ und 2 Knoten $v,u \in V$ mit $(v,u) \notin E$.

Gesucht: Berechne ob der Link (v,u) in Zukunft entstehen wird oder ob er existiert und nur noch nicht beobachtet wurde.

Beispiele:

- Freundschaftsanfragen in sozialen Netzwerken
- unbekannte Proteininteraktionen
- Vorhersage von Kundenbewertung oder Käufen (Collaborative Filtering)

Grundidee: Paare von verlinkten Knoten haben meistens Eigenschaften, die für diesen Link verantwortlich sind.

Beispiele:

- Freunde haben häufig ähnliche Interessen.
- Co-Autoren forschen meist in ähnlichen Gebieten.
- Proteine haben komplementäre reaktive Zentren

⇒ Links entstehen nicht zufällig sondern durch die Kontakt von Objekten die komplementäre Eigenschaften haben.

⇒ Link Prediction lernet eine Funktion, die aus Paaren von Objektbeschreibungen die Existenz oder die Stärke eines Links vorhersagen

Formal: Sei $u, v \in V$ und $F(v), F(u)$ ihre Featurebeschreibung, dann versucht Link-Prediction, die Funktion: $(F(v), F(u)) \rightarrow L$ zu lernen.

(L ist dabei entweder {link, no link} oder $[0, \dots, \text{max_Strength}]$)

Problem: Bei der Feature-Basierten Vorhersage wird nicht berücksichtigt, ob die Knoten Kontakt haben.

Beispiel:

- Personen mit gleichen Interessen lernen sich nicht kennen und verlinken sich daher nicht.
- Proteine können docken, kommen aber in der Natur nicht in den gleichen Zellen vor.

Lösungsansatz: Berücksichtige die Umgebung der Knoten v und u in G .

⇒ Haben beide Knoten ähnliche Nachbarn wächst die Wahrscheinlichkeit eines Kontakts und damit einer Verlinkung.

⇒ Beschreibe einen Knoten durch seine Adjanzenzliste im Graphen.

⇒ Wahrscheinlichkeit/Stärke eines Links kann durch die Ähnlichkeit der benachbarten Knoten vorhergesagt werden.

Gegeben: Graph $G(V,E)$ mit Adjazenzmatrix A , des Graphen $G(V,E)$ mit der Annahme das $a_{i,j} \in E$ unbekannt sind.

Annahmen:

- die Stärke der *Links* $U \subseteq A$ ist unbekannt
- Es gibt einen latenten k -dimensionalen Vektor-Raum mit $k < |V|$, der die Eigenschaften der Knoten beschreibt. $\Rightarrow f(v)$ ist ein k -dimensionaler Vektor
- Die Stärke eines Links zwischen v und u besteht aus dem linearen Kernel $K(u,v)$

Vorgehen:

- Suche eine Zerlegung von in eine $n \times k$ Matrix B : $A' = BB^T$

so dass $L(B) = \sum_{a_{i,j} \in A \setminus U} |a_{i,j} - a'_{i,j}|^2 = \sum_{a_{i,j} \in A \setminus U} |a_{i,j} - \langle b_{i,*}, b_{*,j} \rangle|^2$ minimiert wird.

Lösung: Gradientenverfahren über der Ableitung von $L(B)$.

Anmerkung: Funktioniert auch für Two-Node Graphen (Kunde/Produkt)

- Finde „dichte“ Subgraphen in einem Netzwerk.
- Definitionen für „dicht“:
- Cliques (Vollständige Teilgraphen)
- Quasi-Cliques (mindstens x % der möglichen Kanten sind vorhanden)
- relative Dichte zur Umgebung: Jeder Knoten in Teilgraphen G' hat mehr Links zu Knoten in G' als zu Knoten in $G \setminus G'$.
- ...
- Problem: Fast alle Formulierungen führen zu NP-harten Suchproblemen $\Rightarrow$ heuristische Lösungen sind weit verbreitet.

- Klasse von Clustering Algorithmen die Datamenge als Graph betrachtet:
- Objekte = Knoten; Links sind mit Distanzen gelabelt.
- Häufig werde nicht alle Links verwendet: beschränken der Links auf die k nächsten Nachbarn oder ε -Umgebung
=> gerichtete und ungerichtete Graphen möglich
- Clustering über weighted k-Mincut: Zerlege Graphen in k möglichst gleich große, disjunkte Teilgraphen, wobei die Menge der durchtrennten Kanten minimiert und die Kosten der getrennten Kanten maximiert werden sollen.
- Problem ist ebenfalls NP-hart.

- bilde symmetrische Ähnlichkeitsmatrix : $S_{i,j} = sim(x_i, x_j)$
- Transformiere S in eine Graph Laplacian Matrix L:

$$L = I - D^{-\frac{1}{2}} S D^{-\frac{1}{2}} \quad D_{i,j} = \begin{cases} \sum_k sim(x_i, x_k) & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

- nach Eigenwertzerlegung von L:
 - Eigenvektoren mit Eigenwert 0 repräsentieren Zusammenhangskomponenten des Graphen
 - Eigenvektoren enthalten lineare Gewichte für das Bilden eines Clusterrepräsentanten.

$$r_k = \sum_{i=1}^{|DB|} EV_i \cdot o_i$$

- Graph-Mining bezeichnet viele neue Data Mining Tasks
 - Ranking wichtiger Knoten
 - Link-Prediction
 - Dichte-Teilgraphen und Community Detection
 - Frequent Subgraph Mining
- Clustering kann auch als Graphproblem formuliert werden
 - Dichtebasiertes Clustering: Finden von Zusammenhangskomponenten im Graphen wobei Links einem Ähnlichkeitsprädikat entsprechen.
 - Spectral Clustering
 - weighted k-Mincut Problem: Aufteilen einen Graphen in k Teilgraphen, so dass alle Teilgraphen möglichst gleich groß sind und die Summe der Gewichte aller „durchtrennten“ Kanten Minimal wird

- Borgwardt K., Kriegel H.-P.: „Shortest-path kernels on graphs“. In Proc. Intl. Conf. Data Mining (ICDM 2005), 2005
- Borgwardt K.: „Graph Kernels“, Dissertation im Fach Informatik, Ludwig-Maximilians-Universität München, 2007
- Bunke, H. : „Recent developments in graph matching“. In ICPR, pages 2117–2124. 2000
- Gärtner, T., Flach, P., and Wrobel: „On graph kernels: Hardness results and efficient alternatives.“ Proc. Annual Conf. Computational Learning Theory, pages 129–143, 2003
- Wiener, H.: „Structural determination of paraffin boiling points“. J. Am. Chem. Soc., 69(1):17–20, 1947

- Yan X., Han J.:“gSpan: Graph-based substructure pattern mining“, In ICDM, 2002.
- Kuramochi M., Karypis G.:“Frequent Subgraph Discovery“, In ICDM, 2001
- Brin S., Page L.:“The anatomy of a large-scale hypertextual Web search engine“, Computer Networks and ISDN Systems, Vol 30, Nr.1-7, S.107-117,1998
- Kleinberg J. M.:“Authoritative sources in a hyperlinked environment“, Journal of the ACM,Vol.46, Nr. 5, S. 604-632,1999
- Brandes U.:“A faster Algorithm for Betweenness Centrality“, Journal of Mathematical Sociology, 25(2):163-177, 2001