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

Skript zur Vorlesung

Knowledge Discovery in Databases II

im Sommersemester 2007

Kapitel 8: Link Mining

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

Kapitelübersicht

8.1. 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
Objekt sind Knoten und Beziehungen sind Kanten

Anwendungen

Vergleich zu Kapitel 7:

Datenbank aus Graph-Objekten: Nicht zusammenhängender Graph, bei dem jede Komponente genau einem Objekt entspricht.

Anwendungen:

- Soziale Netzwerke: MySpace, Telefonnetzwerke, E-Mail Traffic, Co-Citation Graph,...
- Protein Interaktionsnetzwerke, philogenetische Netzwerke..
- Straßennetzwerke: Knotenpunkte, relevante Verbindungen,..
- Kriminalitätsnetzwerke
- Computer Netzwerke
- WWW: Linkstrukturen, Webringe, Ranking von Internetseiten..
- Relationale Datenbestände: Internet Movie Database,...

8.2 Frequent Subgraph Mining

Idee: Finde alle häufigen Subgraphen in Netzwerken.

Anwendungen:

- Auftreten von häufigen 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

Achtung: Viele Lösungen gehen vom Szenario von Kapitel 7 aus.
(Datenbank aus graphstrukturierten Objekten)

Anwendung auf Netzwerken ist aber direkt möglich.

Lösungsansätze

- *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 durch Kombination 2er existierender Graphen.
- *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.

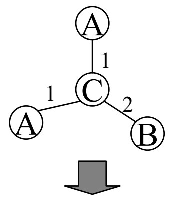
⇒ generiere nur 1 Subgraphen für jede Isomorphie-Klasse

Lösungen Frequent Subgraph Mining

FSG [Kuramochi, Karypis 2001]

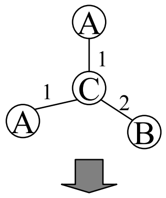
- Geht von Menge von gelabelten, ungerichteten Graphen aus.
- **Idee**: Erweiterung des Apriori-Algorithmus zum Itemset Mining auf Graphen.
- Darstellung der Graphen als Adjazenz-Listen
- Isomorphe Graphen entsprechen Permutationen der Adjazenz-Listen

⇒ 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

Canonical Labeling



	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

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

000112

- 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(Fk)
    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
```

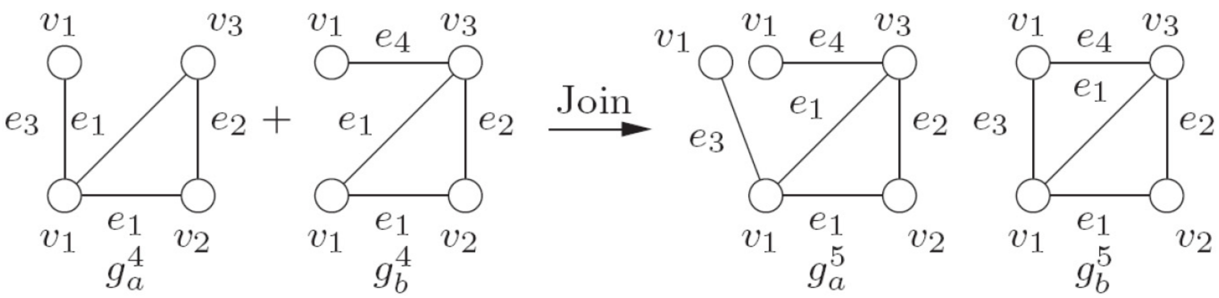

Komplexität von FSG

Im Code enthaltene komplexe Passagen:

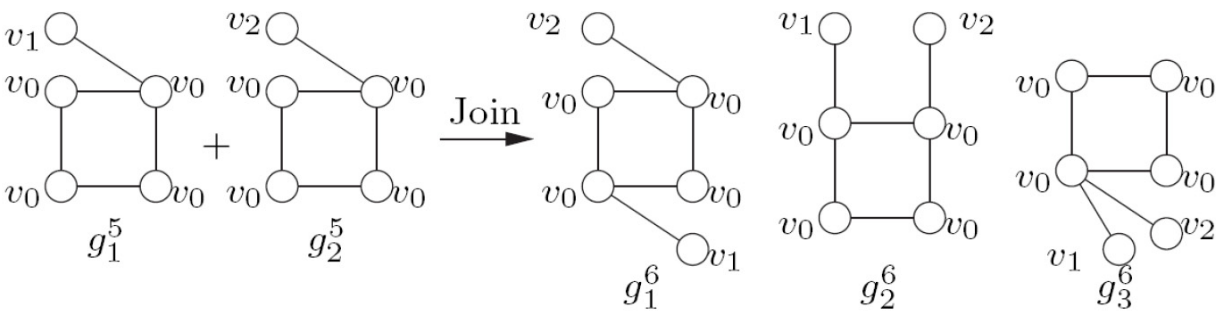
1. Subgraph-Isomorphie-Test (g.includes(s))
   - notwendig beim Datenbank-Scan im Hauptalgorithmus
   - notwendig bei der Kandidatengenerierung:
     Test auf gemeinsame *k-1* Subgraph
2. Join zweier Graphen auf Basis eines **k-1** Subgraphen
   - ⇒ es gibt mehrere mögliche Ergebnisse
   - ⇒ alle müssen als Kandidaten in Betracht gezogen werden

Fazit: Algorithmus ist sehr teuer !

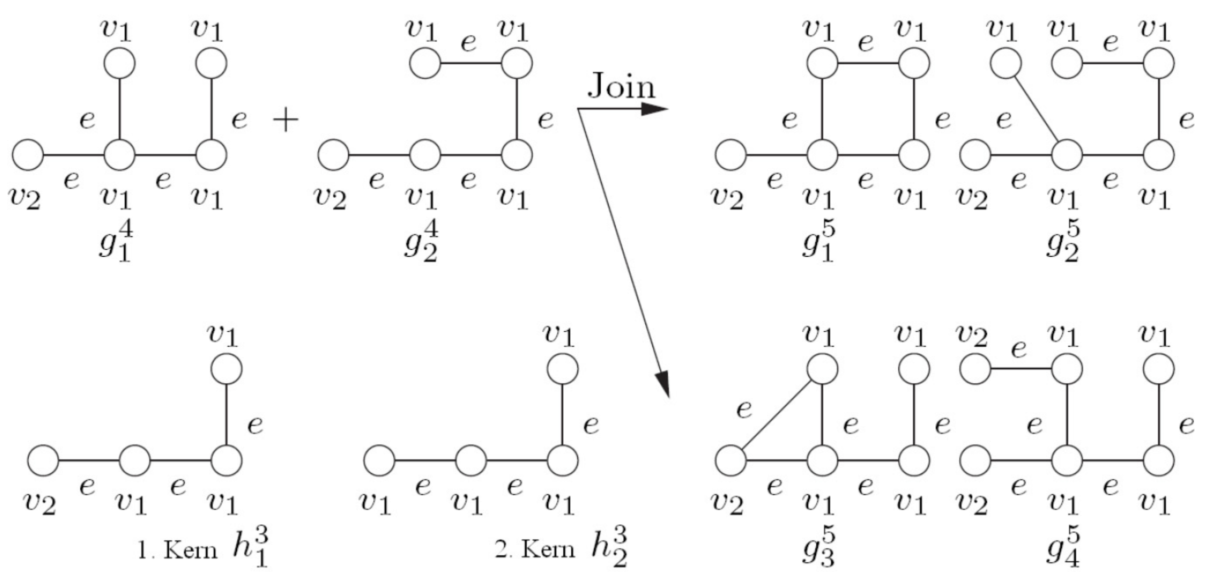
Mögliche Kandidaten (1)



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

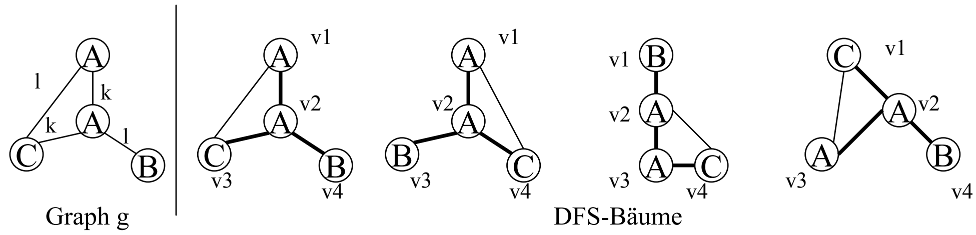
```
S:={}
GrowPatterns(g,DB, S)

Function GrowPatterns(g,DB,S)
    if g ∈ S then return;
    else S.insert(g)
    EdgeSet E = findAdjacentEdges(D,g MinSup); // finden aller Kanten in DB in g erweitern können
    for each frequent e ∈ E DO
      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 Graphe durch Kantenreihenfolge in einem Tiefensuchbaum
(***D***epth ***F***irst ***S***earch Baum)



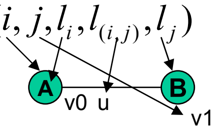
Unterscheide *Vorwärtskanten*: Erweitern Graph um neuen Knoten
Rückwärtskanten: Verbinden bereits vorkommende Kanten

Ein DFS-Tree impliziert Reihenfolge der Kanten in Graph *G* (DFS-Code)

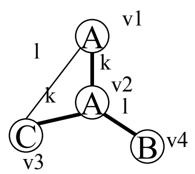
- Reihenfolge entspricht Tiefensuch-Reihenfolge **(v1,v2,..)**
- Rückwärtskanten werden eingefügt, nachdem Startknoten erreicht wurde
- Reihenfolge der Rückwärtskanten nach Besuchsreihenfolge der Zielknoten

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

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

=> Bilde Kandidaten nach speziellen Regeln

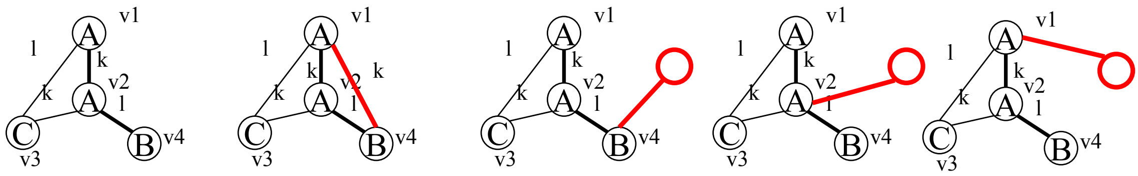
Right-Most-Only Extension erlaub nur Erweiterungen entlang des rechtesten Pfades im DFS-Baum der Min-DFS-Code impliziert.

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.



GSpan

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

GSpan

S :Menge der frequent Graphs;
c : a DFS Code
DB: Graphdatenbank
MinSup: minimaler Support des SubGraphen

```
S:={}
GSpan(c,DB, S)

Function GrowPatterns(g,DB,S)
    if s ≠ dfs(S) then return;
    else S.insert(g)
    C := {}
    EdgeSet E = findRightMostOnlyExtensions(D,s, MinSup); // finden aller Kanten in DB in g erweitern können
    C = extend(s,E);
    C.sortInLexDFSOrder;
    for each frequent g ∈ C DO
        G Span(g,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.

8.3. Graph Analysis

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

Aufgaben:
- Ranking der Knoten nach Einfluss
- Suche nach Schlüsselknoten
- Suche nach Verbindungsgraph zwischen 2 oder mehreren Knoten
- Finden von stark verbundenen Subgraphen
- Link Prediction: Wird in naher Zukunft Kontakt zwischen 2 noch nicht verbundenen Knoten entstehen.

Zentralität eine Knotens im Netzwerk

Betweenness Centrality

Definition: Die Anzahl der kürzesten Pfade, die einen Knoten v im Graphen G enthalten, bezeichnet man als „Betweenness Centrality".

Formal: σ_{st} Anzahl der kürzesten Pfade von s nach t.

$\sigma_{st}(v)$ Anzahl kürzeste Pfade von s nach t, die v enthalten.

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Interpretation: Je höher die Betweenness Centrality desto wichtiger der Knoten innerhalb des Graphen.

Zentralität eine Knotens im Netzwerk

Beispiel: Betrachte Knoten als Router

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

Berechnung: Verwende Algorithmus von Floyd-Warshal um alle kürzesten Pfade zu bestimmen.

=> Abzählen der Beteilung jedes Knotens an den kürzesten Pfaden.

Komplexität: $O(n^3)$

Schnellere Berechnung möglich durch Aggregation von Teilergebnissen: Für *m* Kanten und *n* Knoten

$O(nm)$ ohne Kantengewichte

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

Anwendungen im Web Mining

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

- wichtige Kompente im Ranking moderner Suchmaschinen.
  (in Kombination mit Ankertexten, Abstand, Worthäufigkeit)

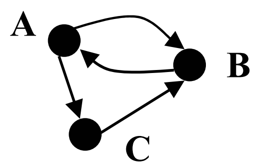
- Web als stark verbundener, gerichteter Graph G(V,E).
  (Betrachtet gesamten Graph aus bekannten Webpages
  z.B. alle in Suchmaschine gespeicherten Pages.)

- Zufallssurfer macht unendlichen Random-Walk.
  Suche: Aufenthaltswahrscheinlichkeit für Page v
  = „Prestige" der Webpage v.

Anwendungen im Web Mining

Berechnung Pagerank
Startverteilung: $p0(u) = 1 / |V|$

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



Adjazenzmatrix E

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

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

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

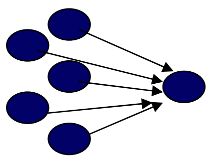
Berechnung über „Power Iterations": $\vec{p}_i \leftarrow L^T \vec{p}_{i-1}$
nach ca. 20-30 Iterationen stabil
Lösung für nicht stark verbundene Graphen: 1. Weglassen von Knoten ohne Links
2. Zufallsprünge auf beliebige Seite

Anwendungen im Web Mining

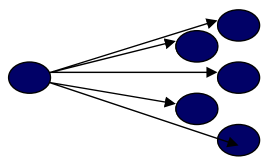
HITS (Kleinberg 1998): Hyperlink Induced Topic Search

- Betrachte alle Seiten, die für Anfrage *q* relevant sind oder die mit einer relevanten Seite verlinkt sind (In- und Outlinks).
  => $G_q(V_q, E_q)$ *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.

Anwendungen im Web Mining

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.

Zusammenfassung

- Link Mining untersucht, wichtige Substrukturen und Knoten in großen Netzwerken verlinkter Objekte
- Häufig Auftretende Subgraphen sind interessant, weil sie Muster in der Struktur des Netzwerkes darstellen
- Einzelne Knoten werden untersucht um die Bedeutung oder den Einfluss einzelner Objekte auf andere zu untersuchen
- Forschungsgebiet noch sehr jung

  => es ist noch nicht klar welche Strukturen besonders interessant aus Anwendungssicht sind.
- Da Objektnetzwerke sehr komplex sind, kann man interessante Muster in Vielerlei Hinsicht definieren

Literatur

- 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