

Skript zur Vorlesung
Knowledge Discovery in Databases II
im Sommersemester 2010

Kapitel 9: Link Mining

Skript © 2007 Matthias Schubert

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

461

Kapitelübersicht

8.1 Graphstrukturierte Datenmengen

Graphdatenbanken und ihre Anwendungen

8.2 Frequent Subgraph Mining

FSG, GSpan

8.3 Ranking von Knoten

Betweenness Centrality, Page-Rank, Hits

8.4 Weitere Graph Mining Aufgaben

Dense Subgraph Discovery, Graph Clustering, Link Prediction

462

8.1. Graphdatenbanken und Anwendungen

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

463

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, phylogenetische Netzwerke..
- Straßennetzwerke: Knotenpunkte, relevante Verbindungen,..
- Kriminalitätsnetzwerke
- Computer Netzwerke
- WWW: Linkstrukturen, Webringe, Ranking von Internetseiten..
- Relationale Datenbestände: Internet Movie Database,...

464

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 (Cliques) 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.

465

Lösungsansätze

- *Frequent Subgraph Mining ist ähnlich zum Itemset Mining*
 - Ausnutzen 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

466

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

467

Lösungen Frequent Subgraph Mining

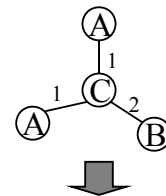
FSG [Kuramochi, Karypis 2001]

- 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

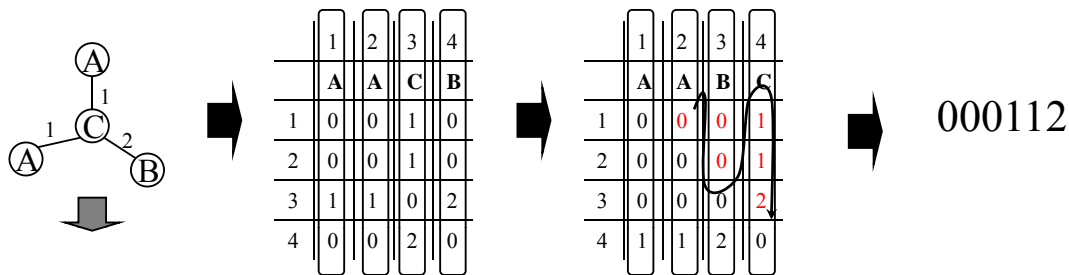
Finde eine eindeutige kanonische Form für jede Isomorphie-Klasse.



	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

468

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

469

FSG Algorithmus(1)

```
Vector<Graphmenge> fsg(Graphmenge D, double  $\delta$ )
    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;
```

470

FSG Algorithmus(2)

Kandidatengenerierung:

```
Graphmenge fsg-gen( $F^k$ )
  Graphmenge  $C_{k+1} = \{\}$ ;
  foreach Graph  $f_{1k} \in F^k$ 
    foreach Graph  $f_{2k} \in F^k$ 
      if( $f_{1k}.canonicalLabel \leq f_{2k}.canonicalLabel$ )
        foreach Edge  $e \in f_{1k}$ 
          Graph  $f_{1k-1} = f_{1k}.remove(e)$ ;
          if( $f_{1k-1}.isconnected \ \&\& \ f_{2k}.includes(f_{1k-1})$ )
            Graphmenge  $T_{k+1}$  = Alle durch Join von  $f_{1k}$  und  $f_{2k}$  entstehende Graphen
            foreach Graph  $t_{k+1} \in T_{k+1}$ 
              boolean allTkFrequent = true;
              foreach Edge  $ed \in t_{k+1}$ 
                Graph  $t_k = t_{k+1}.remove(ed)$ ;
                if( $t_k.isConnected \ \&\& \ t_k \notin FK$ )
                  allTkFrequent = false;
                  break;
              if(allTkFrequent)
                 $C_{k+1}.add(t_{k+1})$ ;
return  $C_{k+1}$ 
```

471

Komplexität von FSG

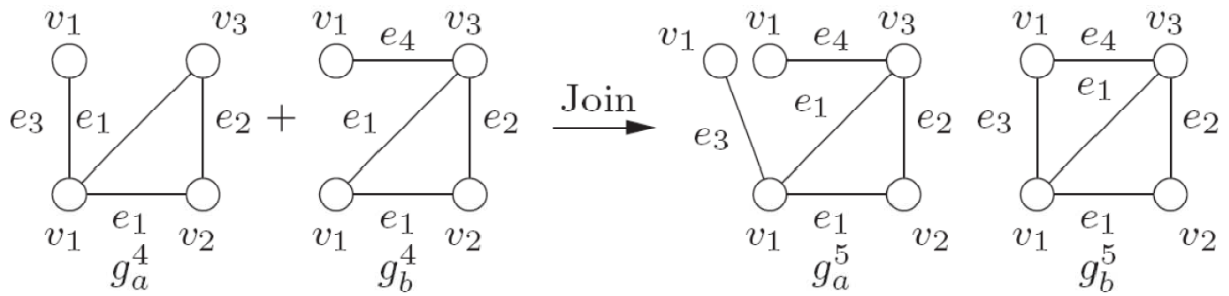
Im Code enthaltene komplexe Passagen:

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

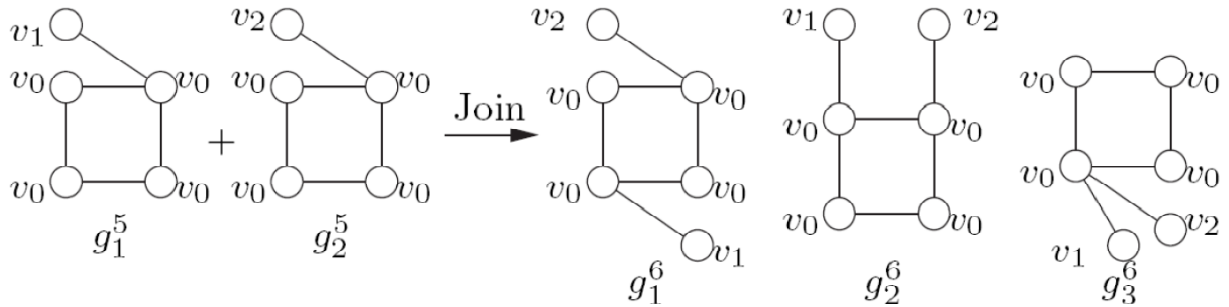
Fazit: Algorithmus ist sehr teuer !

472

Mögliche Kandidaten (1)

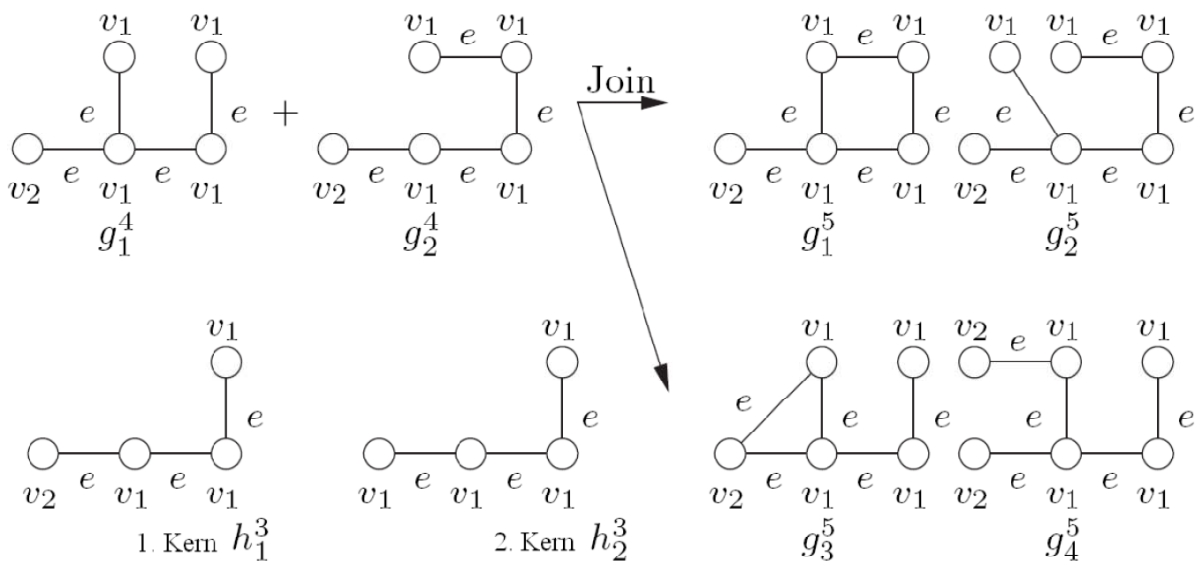


1. Gleiche Knoten-Markierung der den Kern erweiternden Knoten



473

Mögliche Kandidaten(2)



474

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)

475

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 $g \in S$ 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

$g' = \text{extend}(g,e)$

 GrowPatterns(g' ,DB,S)

end for

end function

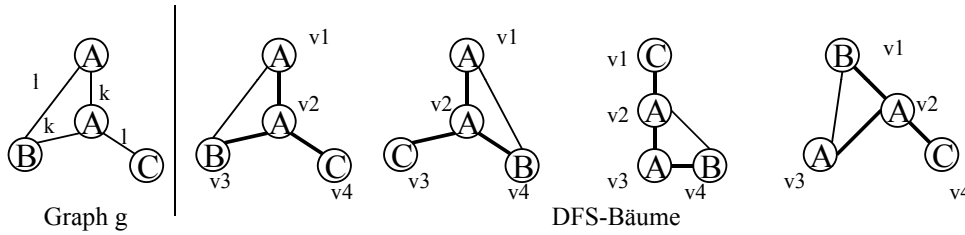
Bemerkung:

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

476

DFS Codes

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

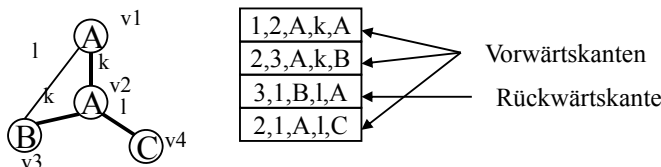
Ein DFS-Tree impliziert Reihenfolge der Kanten in Graph G (DFS-Code)

- Reihenfolge entspricht Tiefensuch-Reihenfolge (v_1, v_2, \dots)
- 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 $(i, j, l_i, l_{(i,j)}, l_j) \rightarrow (0, 1, A, u, B)$

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'

Right-Most-Only Extension

Idee: Vermeide mehrfache Untersuchung monotoner Kandidaten

=> Bilde Kandidaten nach speziellen Regeln

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

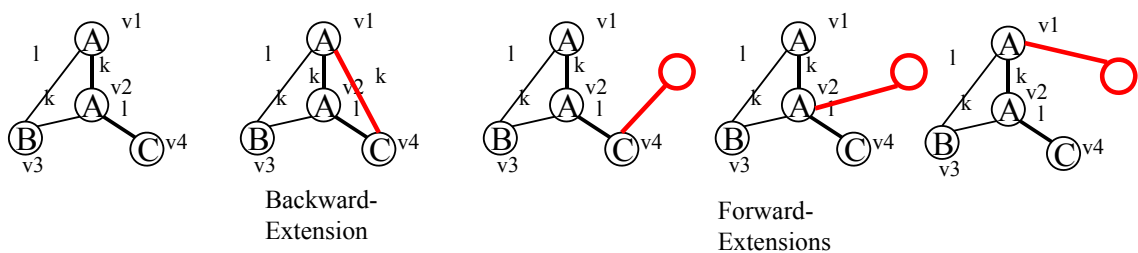
DFS-Tree:

- *Backward-Extension*

Verbinde Knoten auf dem rechten Pfad mit Rückwärtskante

- *Forward Extension*

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



479

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): 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

480

Frequent Subgraph Mining

- Finden häufiger Subgraphen ist ähnlich zum Itemset Mining
Aber:
 - Menge aller isomorphen Graphen ist größer als Menge aller Permutationen 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.

481

8.3. Ranking von Knoten

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

482

Zentralität eines 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 kürzester 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.

483

Zentralität eines 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-Warshall um alle kürzesten Pfade zu bestimmen.

=> Abzählen der Beteiligung 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

484

Anwendungen im Web Mining

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 .

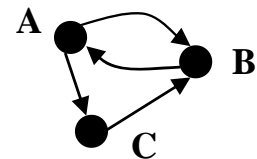
485

Anwendungen im Web Mining

Berechnung Pagerank

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

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



Adjazenzmatrix E

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“:
nach ca. 20-30 Iterationen stabil

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

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

486

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

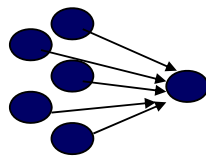
- *Zwei Typen von Webpages:*

Hubs: Verlinken auf viele relevante Webpages (Authorities)

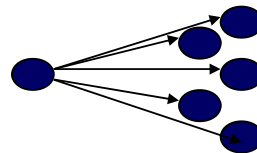
Authorities: Werden von vielen guten Hubs verlinkt.

=> 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.

487

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.

488

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

489

8.4. Weitere Aufgaben im Graph Mining

Weitere Aufgaben:

Link Prediction:

Annahme: Nicht alle Kanten im Graphen sind bekannt.

Aufgabe: Finde nicht beobachtete Verbindungen zwischen 2 Knoten. Vorhersage der Verbindungsstärke 2er Knoten.

Anwendungen:

- Finden von Kunden Produktbeziehungen
- Vorhersage von Beziehungen im sozialen Netzwerken

490

Link Prediction

Möglichkeiten zur Vorhersage:

- Link Prediction kann als Klassifikationstask gesehen werden.
 - hierbei sind Paare von Knoten die Objekte.
 - Klassen: „Haben Link“ und „Sind nicht verlinkt“
- Über SVD/PCA der Adjacence Matrix
 - Beispiel: Matrix M mit Kundenrezessionen für Produktmenge P der Kundengruppe K.
 - Zerlege M mit SVD: Singulärwerte repräsentieren jetzt die Kundenbedürfnisse. Eigenvektoren im Kundenraum repräsentieren die Präferenzen der Kunden. Eigenvektoren im Produktraum wie stark welches Produkt welches Bedürfnis abdeckt.

491

weitere Aufgaben

Dense Subgraph Discovery:

Finde „dichte“ Subgraphen in einem Netzwerk.

Definitionen für „dicht“:

- Cliques (Vollständige Teilgraphen)
- Quasi-Cliques (mindestens 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 => heuristische Lösungen sind weit verbreitet.

492

Graph Clustering

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.

493

Graph Clustering: Spektral Clustering

- 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.

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

494

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