

(Sub)Graph-(Iso)Morphie 2. Teil

Vorlesung Graphenalgorithmen WS 09/10
Karsten Klein

TU Dortmund Ls11



Zusammenfassung 1.VO

- (Subgraph)Isomorphie für Graphen: Umbenennung von Knoten und Erhalt der Adjazenz
- Komplexität GI in NP, Schwere unklar (wahrscheinlich nicht NP-schwer), SGI NP-vollst.
- Lösungsansätze schrittweise (BT mit Pruning) oder über kanonische Bezeichner, Invarianten
- Algorithmus von Ullmann – einfach, nicht State-of-the-Art
- Algorithmus VF2 – komplexer, schneller, weniger Platz
- In Praxis einige tausend Knoten bei 1:1 machbar

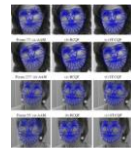
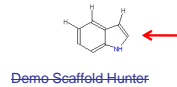
Übersicht

- Kurze Demo Scaffold Hunter Webinterface Molekülsuche
- Strategien für 1:n Isomorphietests
- Kanonische Bezeichner (Canonical Label)
- Algorithmus von McKay (Grundlagen)

TU Dortmund Ls11

Graph Isomorphie

- Schöne Theorie: Eines von nur zwei übriggebliebenen Problemen aus der Liste von Garey & Johnson (offene Komplexität)
- Praxisrelevant: Identifikation chemischer Verbindungen, Mustererkennung (Biometrie), Validierung von Schaltkreislayouts, Bioinformatik (Netzwerke), ...
Häufig 1:n Vergleiche statt 1:1 (2. VO) ←
- Stark untersucht in den letzten Jahrzehnten, sowohl in Theorie als auch in Praxis

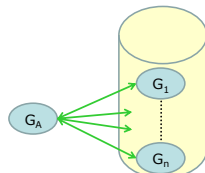


Skalierbarkeit

Bisher paarweiser Vergleich



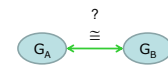
In Praxis meist Datenbanken mit sehr vielen Vergleichsgraphen



(Subgraph)Isomorphietest für jeden DB Graph schwierig:
DB Größe entscheidend

Filter-Verifikation

Möglichst Vergleiche auslassen



Preprocessing auf DB Graphen
Merkmale kompakt speichern



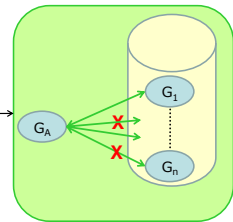
Welche Merkmale?

Wie speichern und vergleichen?

Eingabe

Filter

Verifikation



Graph Indexing

Einmaliges Preprocessing auf DB Graphen zur Erzeugung einer Index-Datenstruktur.

Datenstruktur:

- Speicherung der Merkmalmenge (IDs) für jeden Graph, sequentieller Durchlauf, häufig als *Feature vector / Fingerprint*

G ₁	G ₂	G ₃	G ₄
M ₁ , M ₃	M ₄ , M ₇	M ₁	M ₂

- Speicherung der Graphenmenge (IDs) pro Merkmal, *Invertierter Index*, Schnittmenge für Mustermerkmale bildet $cand(G)$. Entscheidend, dass G_A Merkmale enthält!

M ₁	M ₂	M ₃	M ₄
G ₁ , G ₃	G ₄	G ₁	G ₂

Merkmale: Graph Indexing

Menge von Merkmalen, die G_i gut charakterisieren mit

- schnellem Indexvergleich (Erzeugung von $cand(G_A)$)
- schnell extrahierbar (aus Muster)
- möglichst kleiner Kandidatenmenge $cand(G_A)$ (, Index)

Selektivitätsmaß für Qualität der Kandidatenmenge, z.B. $|cand(G_A)| / |Erg(G_A)|$
Eindeutiger Index: „Höchste“ Selektivität

Einfache Merkmale: Invarianten aus 1.VO, aber z.B. Knoten-/Kantenanzahl schlecht für Subgraphisomorphie geeignet
Besser: Strukturelle Merkmale – Teilgraphen

Merkmilverteilung in Praxis häufig:

- Einige Merkmale in fast allen Graphen (helfen wenig).
- Viele Merkmale in wenigen Graphen (potentiell große Merkmalmenge (schlecht)).

Graph Indexing

Ansätze:

- Data-Mining (DM) basiert:

Analyse der DB und entsprechende Auswahl der Merkmale (Hohe Selektivität)

Bei Änderung DB meist Änderung Index nötig

- Nicht Data-Mining basiert:

Keine Betrachtung der Gesamtheit der Graphen

Keine Analyse/Änderung nötig (!Online DBs!)

Evtl. erhöhte Extraktionskosten (viele Merkmale), geringere Selektivität

Merkmale: Graph Indexing

Viele Variationen implementiert: GraphGrep, gIndex, GraphFind, GrepVS, VFM, TreePi...

Üblicherweise: Knoten (und Kanten) als gelabelt betrachtet

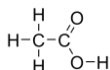
z.B. Atomtypen in Chemie, Proteine in Biologie, etc.

- Enthalten $\geq k$ Objekte mit Label l_i , Strukturen (Ringe bestimmter Größe, funktionale Gruppen,...) fix
- Pfade**: Berechne alle Pfade der Länge k
Praxis: Cheminformatik $0 \leq k \leq 8$
- Bäume**:... beschränkter Größe
- Ringe**:... beschränkter Größe
- Kombinationen dieser Strukturen

Pfade in Molekülen

Pfade (H wird vernachlässigt):

- Länge 1 C,O
- Länge 2 CO, CC, OC C=O, O=C
- Länge 3 CCO, CC=O, OC=O, ...



So ist das redundant.

Merkmale: Graph Indexing

Merkmale prüfen: Vergleich über *kanonische Bezeichner*

Kanonischer Bezeichner: Zeichenkette $B(G)$, die für isomorphe Graphen gleich ist, sich für nicht isomorphe Graphen jedoch unterscheidet.

Für einfache Teilgraphen schnell bestimmbar, für allgemeine Graphen nicht (sonst Isomorphieproblem gelöst)



ABC = CBA

⇒ Nach eindeutiger Regel, z.B. lexikographisch kleinsten Bezeichner speichern

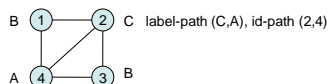
Generell Index für Gesamtgraph möglichst nahe am kanonischen Bezeichner.

(In Praxis auch Kantenlabel)

GraphGrep

[Shasha, Giugno 2002]

- Graphsuche mit Wildcards, eigene Anfragesprache
 - Knoten mit ID und Label vorausgesetzt
- ⇒ label-path, id-path:



h(CA)	1
...	...
h(ABCB)	2

Für label-path u.U. Menge von id-paths ⇒ Pfadrepräsentation

Indexaufbau (einfache Version): ...AB={{(4,1),(4,3)} AC={{(4,2)}...

Bestimme für jeden Graph/Knoten alle ausgehenden Pfade mit Länge $1 \leq k \leq l_p$, l_p kleine Konstante.

Label-Pfade w dienen als Schlüssel für Hash-Tabelle.

Eintrag #Vorkommen von w für jeden Graph (id-paths)

GraphGrep

Aufbau in $O(\sum_1^{|\text{DB}|} n_i \cdot \max \text{deg}_i^{l_p})$ Zeit und $O(\sum_1^{|\text{DB}|} |p_i| \cdot n_i \cdot \max \text{deg}_i^{l_p})$

Filterphase:

Fingerprint für Muster erstellen (Hash der Pfadmenge)

Wertevergleich: Falls Graphwert < Musterwert, verwerfen

Matching mit vf2

Mittlerweile mehrfach verbessert (GraphFind):

Indexverkleinerung durch Zusammenfassen gleich häufiger

Merkmale, Filterbeschleunigung, Suffix Tree statt

Hashtabelle

Graph Indexing

Datenstruktur:

Fingerprint: Bitvektor fester Länge, der Merkmale eines Graphs erfasst.

Berechne Fingerprint für alle Graphen der DB voraus, vergleiche mit Muster-Fingerprint (Bitvergleich einfach).

Structure-Key Fingerprint:

Vordefinierte Merkmalmenge (DM), ein Bit pro Merkmal.

In Praxis bekannte Fingerprint Standards z.B. PubChem für chemische Strukturen, 880bits. Häufig dünn besetzt.

Hash-Key Fingerprint:

Merkmalgenerierung aus Graphen, z.B. alle Pfade $\leq k$.

Mittels Hashfunktion Merkmal auf ≥ 1 Position abbilden.

Kollisionen möglich (verschlechtert Selektivität)

...weitere Varianten, auch für Ähnlichkeitssuche

Pubchem Molekül Fingerprint

Section 1: Hierarchic Element Counts - These bits test for the presence or count of individual chemical atoms represented by their atomic symbol.

Bit Position Bit Substructure

0 >= 4 H

1 >= 8 H

2 >= 16 H

3 >= 32 H

4 >= 1 Li

...

Section 2: Rings - These bits test for the presence or count of the described chemical ring system.

Bit Position Bit Substructure

115 >= 1 any ring size 3

116 >= 1 saturated or aromatic carbon-only ring size 3

117 >= 1 saturated or aromatic nitrogen-containing ring size 3

...

Section 3: Simple atom pairs - These bits test for the presence of patterns of bonded atom pairs, regardless of bond order or count.

Bit Position Bit Substructure

263 Li-H

264 Li-Li

265 Li-B

...

Graph Indexing

Erstellung der Fingerprints für die DB kann sehr aufwendig sein, aber die Suche erheblich beschleunigen. Structure Keys sind aber stark anwendungsorientiert, nicht beliebig übertragbar.

Tradeoff Fingerprint Länge vs. Anzahl Merkmale:

- Dünn besetzte Fingerprints verschwenden Platz
- Kollisionsreduktion durch langen FP braucht Platz (großer Index) und erhöht Laufzeit der Filterphase
- Dicht besetzte Fingerprints filtern nicht
- Kurze Pfade (wenige Merkmale) ergibt geringere Selektivität

Fingerprint



C 7 O 8

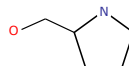
CC 5

COCC 3

...

CCCC 8

1	2	3	4	5	6	7	8	9	10
...	...	1	...	1	...	1	1



C 7

N 9

CC 5

...

CCCC 8

1	2	3	4	5	6	7	8	9	10
...	...	0	...	1	...	1	1	1	...

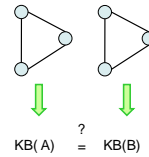
Bitweises AND ergibt String ungleich dem Musterstring,

also enthält Muster Pfad(e) die nicht im Molekül vorkommen.

Kanonische Bezeichner

Kanonische Bezeichner

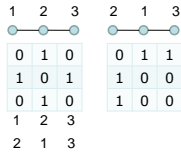
Letztes Mal: Sukzessiver Aufbau eines Matching.
 Jetzt: Bezeichner für G, der unabhängig erzeugt wird und zum Testen verwendet werden kann.



Kanonischer Bezeichner ist ein eindeutiger Repräsentant einer Klasse isomorpher Graphen.

Kanonische Bezeichner

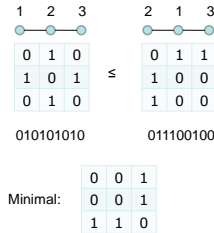
Wir wissen schon: Ändern der Reihenfolge in Adjazenzmatrix hilfreich für Isomphietest.
 „Umbenennung“ und tauschen:



Dies kann auch „eindeutig“ (kanonisch) geschehen:
 „Kleinste“ Adjazenzmatrix (lexikographisch).

Minimale Adjazenzmatrix

Minimal in Bezug auf lexikographische Sortierung der konkatenierten Zeilen.



Kanonische Bezeichner

Minimale Adjazenzmatrix repräsentiert eine isomorphe „Normalform“ eines Graphen und der dazugehörige String ist kanonischer Bezeichner, also für alle isomorphen Graphen gleich.

Welche für Isomphietest einsetzbaren Bezeichner gibt es und wie können wir solche Bezeichner berechnen?

Partitionierung

Schon gesehen: Invarianten (Grad etc.) teilen Knoten in Klassen.

Isomorphe /automorphe Abbildung nur zwischen äquivalenten Klassen/Knoten.

⇒ Beschleunigung des Tests durch Bestimmung möglichst kleiner Klassen.

Optimal: In jeder Klasse genau ein Knoten ⇒ eindeutig.

Partitionierung: Unterteilung in disjunkte Teilmengen, die *Zellen*.

Iterativer Ansatz:

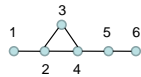
- Starte mit grober Partitionierung der Knoten in Zellen z.B. nach Knotengrad.
- Verfeinere schrittweise.

Idee: Verfeinere Zelle durch Klassifizierung der Knoten nach ihrer Nachbarschaft zu *Pivotvertex* oder *Pivotmenge*.

Knotenpartitionierung

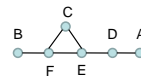
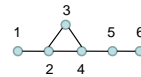
- Teile auf nach Knotengrad, kleineren Grad, Index zuerst
 $1\ 6\ | \ 3\ 5\ | \ 2\ 4$
- Offensichtlich nicht eindeutig, wir können verfeinern
 Benutze Grad bzgl. der anderen Zellen:
 Teile jede Partition mit mehr als einem Knoten bzgl. des Grades zur ersten Zelle auf, wiederhole dies mit Zelle 2 usw. (1-dim Weisfeiler-Lehman Refinement).
 $1\ 6\ | \ 3\ 5\ | \ 2\ 4 \rightarrow 1\ 6\ | \ 3\ 5\ | \ 2\ 4 \rightarrow 1\ 6\ | \ 3\ 5\ | \ 4\ 2 \rightarrow$
 $1\ 6\ | \ 3\ 5\ | \ 4\ 2 \rightarrow 1\ 6\ | \ 3\ 5\ | \ 4\ 2$

Wir sind damit fertig!
 Und nun?



Knotenpartitionierung

- Diskrete Partitionierung
 $1\ | \ 6\ | \ 3\ 5\ | \ 4\ | \ 2$
- Stellen wir uns vor, wir haben einen zweiten, ganz anderen Graph.
- Wir führen dieselbe Partitionierung durch:
 $AB\ | \ CD\ | \ EF \rightarrow B\ | \ A\ | \ C\ | \ D\ | \ E\ | \ F$
 - Diskrete Partitionierung ergibt kanonische Umbenennung:
 $1 \rightarrow 1\ 6 \rightarrow 2\ 3 \rightarrow 3\ 5 \rightarrow 4\ 4 \rightarrow 5\ 2 \rightarrow 6$
 $B \rightarrow 1\ A \rightarrow 2\ C \rightarrow 3\ D \rightarrow 4\ E \rightarrow 5\ F \rightarrow 6$
 - Das wiederum erlaubt ein Matching



Partitionierung

Verfeinerung:
Eingabe Partition $\Pi=(V_1, \dots, V_r)$
Ausgabe Finale verfeinerte Partition $f(\Pi)=(V_1, \dots, V_{r'})$ mit $\deg(v_i, V_j) = \deg(w_i, V_j)$ für beliebige $v_i, w_i \in V_{i'}$ und $V_{i'}, V_j$ aus $f(\Pi)$.

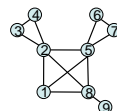
```

repeat
   $\Pi_{\text{old}} = \Pi_{\text{new}}$  (ist  $(V_1, \dots, V_{r_{\text{alt}}})$ )
  for  $i = 1$  to  $r_{\text{alt}}$ 
    Nutze  $V_i$  als Pivotmenge zur Partitionierung der  $V_k : |V_k| > 1, 1 \leq k \leq r_{\text{alt}}$ 
    Ersetze  $V_k$  durch Partitionierung
  until  $\Pi_{\text{new}} = \Pi_{\text{old}}$ 
    
```

Knoten in derselben Zelle heißen auch *strukturell äquivalent*.

Partitionierung

- Das geht auch mit Automorphismen:
Orbit von Knoten v : Menge der Knoten auf die ihn Automorphismen abbilden.
- Definition: Orbitpartitionierung**
 Partitionierung P mit: Knoten v, w gehören zu derselben Zelle von $P \Leftrightarrow \exists$ Automorphismus φ mit $\varphi(v)=w$



$1\ | \ 2,5\ | \ 3,4,6,7\ | \ 8\ | \ 9$
 Besser geht's nicht.

Zueinander isomorphe Graphen haben äquivalente Automorphismengruppe, die kennen wir aber nicht.

McKays Algorithmus

McKay's Algorithmus

Algorithmus zur Berechnung kanonischer Bezeichner und der Automorphismengruppe [McKay '81].
 Implementiert im *nauty* Paket – lange Zeit der schnellste Isomorphietest-Algorithmus.
 Nutzt das gerade vorgestellte Schema mit Knotenklassifizierung und Suchbaum, Automorphismen werden zum Pruning benutzt.
 Hohe Zahl an Automorphismen finden problematisch (Speicher, Zeit) aber Auto. zum Beschleunigen – es ex. Graphklassen mit exp. Laufzeit.

Verbesserung durch [Lopez-Presa, Fernandez 04]: *Conauto*
 Benutzt ebenfalls Automorphismen, aber nicht alle.

McKay's Algorithmus

Idee: Bezeichner ist minimale Adjazenzmatrix.

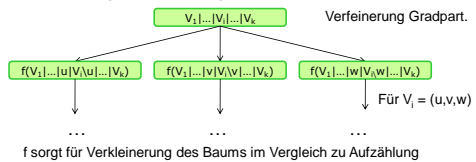
- Diese kann man durch Aufzählung bestimmen (ineffizient).
- Knotenpartitionierung schränkt die Fälle ein.
- Automorphismen ergeben dieselbe Matrix.
- Starte mit bester Partitionierung einen Suchbaum, der die noch möglichen Fälle aufzählt (willkürlich Zellen aufteilen).
- Versuche Verfeinerung der Partitionierung in jedem Schritt.
- Vergleiche gefundene Matrizen, um Automorphismen zu finden.
- Nutze Automorphismen, um Berechnungen zu sparen.
- Weitere Tricks, u.A. BB ähnlich (nicht hier, siehe Literatur).

Partitionierung 2

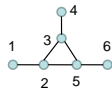
Für reguläre Graphen zum Beispiel haben wir leider auch nur eine initiale Zelle, Partition (V_1).

Nutze bekanntes Schema (B&B etc.): Aufteilung in Suchbaum

- Knoten enthalten geordnete Partitionen
 - Blattpartitionen sind diskret
 - Knotenkinder: $V_1 | \dots | V_i | \dots | V_k \rightarrow f(|V_1 | \dots | v | V_i \setminus v | \dots | V_k)$ $\forall v \in V_i$
- Also aus zu großen Zellen jeden Knoten in Einzelhaft nehmen.



Partitionierung 2



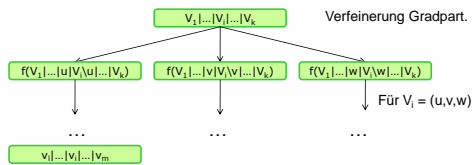
Initiale Verfeinerung 1 4 6 | 2 3 5

1 | 4 6 | 3 5 | 2

Ein Blatt! 1 | 4 | 6 | 5 | 3 | 2

McKays Algorithmus

Blattpartitionen geben Anordnung der Adjazenzmatrix vor.

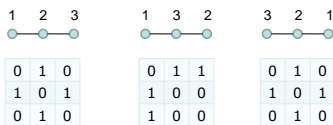


Bezeichner ist die minimale Matrix aller Blätter. Wir merken uns also den bisherigen Spitzenreiter.

McKays Algorithmus

Pruning: Nutze Automorphismen

Automorphismen finden: Zwei Blätter haben dieselbe Matrix



Tradeoff: Mehr Matrizen merken \Rightarrow mehr Automorphismen findbar, aber Speicher- und Rechenaufwand.

McKays Algorithmus

Pruning: Nutze Automorphismen

Automorphismen finden: Zwei Blätter haben dieselbe Matrix

Äquivalente Knoten $N_1 \sim N_2$ in Suchbaum:



\exists Automorphismus ϕ , der Zellen ineinander überführt:

$$\phi(V_i) = W_{\phi(i)} \quad i=1, \dots, m$$

„Farberhaltender Automorphismus“ für zugehörige Zellen $V_i, W_{\phi(i)}$.

McKays Algorithmus

Pruning: Nutze Automorphismen

Automorphismen finden: Zwei Blätter haben dieselbe Matrix

Äquivalente Knoten $N_1 \sim N_2$ in Suchbaum:

$V_1 | \dots | V_i | \dots | V_k$ $W_1 | \dots | W_j | \dots | W_k$

Falls $N_1 \sim N_2$, so gibt es für jeden Knoten N_1^* im Unterbaum von N_1 einen Knoten N_2^* mit $N_1^* \sim N_2^*$ (ohne Beweis).

Diese Unterbäume können wir abschneiden.

McKays Algorithmus

Wie merken wir uns Automorphismen?

Wir nutzen die Orbitpartition bzgl. der schon gefundenen Automorphismen.

Finden eines Automorphismus während Suche \Rightarrow
Anpassen der Orbitpartition (Vergrößerung).

Falls $V_1 | \dots | V_j | \dots | V_k$ Partitionierung in N und $v_i, v_j \in V_i$ liegen im selben Orbit, dann gilt $f(V_1 | \dots | v_i | v_j | \dots | V_k) \sim f(V_1 | \dots | v_j | v_i | \dots | V_k)$.

Es reicht daher, einen Nachfolger (min. Index v) zu erzeugen. Dies wird evtl. erst im Unterbaum für v erkannt.

Kanonisierung

Vorteil:

Kanonisierung kann unabhängig vom Muster geschehen

\Rightarrow Preprocessing, extreme Zeitersparnis bei Vergleich

Zusammenfassung

- (Subgraph)Isomorphie für Graphen
- Verfahren für 1:n Suche – Graph Indexing
- Fingerprints
- Kanonische Bezeichner und ihre Berechnung
- Grundlagen des McKay Algorithmus – schnell, aber überholt
- Das war jetzt alles noch exakt, in der Praxis aber approximative Ansätze nötig

TU Dortmund Ls11

Literatur

Graph Indexing:

- D. Shasha et al.: *Algorithmics and Applications of Tree and Graph Searching*, Proc. PODS'02, June 3 -- 5 2002
- H. Jiang et al.: *GString: A Novel Approach for Efficient Search in Graph Databases*, Proc. ICDE 2002, pp. 566-575

Labeling:

- B. McKay: *Practical Graph Isomorphism*, Congressus Numerantium, 30 (1981), pp. 45-87
- L. Babai, E. M. Luks: *Canonical Labeling of Graphs*, Proc. 15th ACM Symp. on Theory of Computing, 1983, 171-183
- T. Miyazaki: *The complexity of McKay's canonical labeling algorithm*, Groups and Computation. II, DIMACS Ser. Discrete Math. Theoret. Comput. Sci., vol. 28, Amer. Math. Soc., Providence, R.I., 1997, pp. 239-256