Mining Free Trees

Seminar aus maschinellem Lernen
Referent: Markus Biesinger

Motivation

Graph

- Repräsentation von Daten mit komplexen Beziehungen
- Datenanalyse um „höhere“ Informationen zu gewinnen

Probleme des Graph Mining

- Enthält ein Graph einen anderen Graphen?
- Sind zwei Graphen isomorph?

Übersicht

- Grundlagen
  - Free Trees
  - Kanonische Form
- Free Tree Miner
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang

Free Trees

- Frequent tree mining
  - TreeMiner
  - FreqT
  - ...
  - Free Tree

Fokus liegt auf Wurzelbäumen

Free Tree

- Ungerichteter, zusammenhängender, azyklischer Graph ohne Wurzelknoten und Ordnung
- Werden beim Tree Mining in Wurzelbäume umgewandelt

- **Anwendung von TreeMiner, FreqT, ... auf Free Trees möglich?**

Free Trees

Anwendung von TreeMiner, FreqT, ... auf Free Trees möglich?

- Transaktionen/Teilbäume haben keine Vorgänger-Nachfolger-Beziehung
- Umwandlung in Wurzelbaum erfolgt auf jeder Stufe des Mininig-Prozess
- Beziehung kann sich auf jeder Stufe ändern

→ Anwendung nicht möglich

Kanonische Form

TECHNISCHE UNIVERSITÄT DARMSTADT

Motivation

- Viele Repräsentationsmöglichkeiten
- Eindeutigkeit erforderlich

Kanonische Form

- Vereinfachten Speicherung, Indizierung und Manipulation
- Vorgehen
  - Normalisierung
  - Bestimmung einer kanonischen String-Repräsentation

Kanonische Form

Normalisierung

- Bestimmung der Center und Selektion als Wurzelknoten
- Reorganisation der Kinder nach vordefinierter Ordnung
  → Free Tree → Ungeordneter Wurzelbaum → Geordneter Wurzelbaum

Kanonische String-Repräsentation

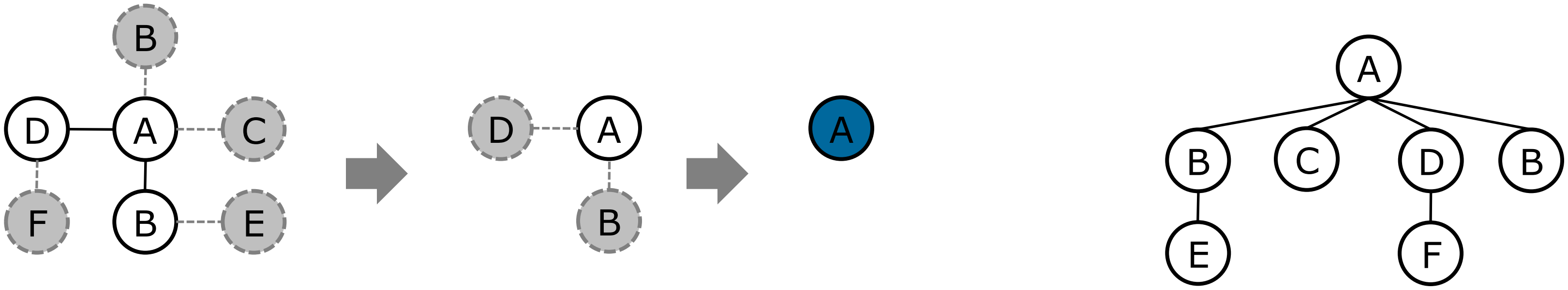
- Äquivalente, aber „einfachere" Repräsentation
- Ableitung durch DFS- oder BFS-Traversierung

Normalisierung

Center

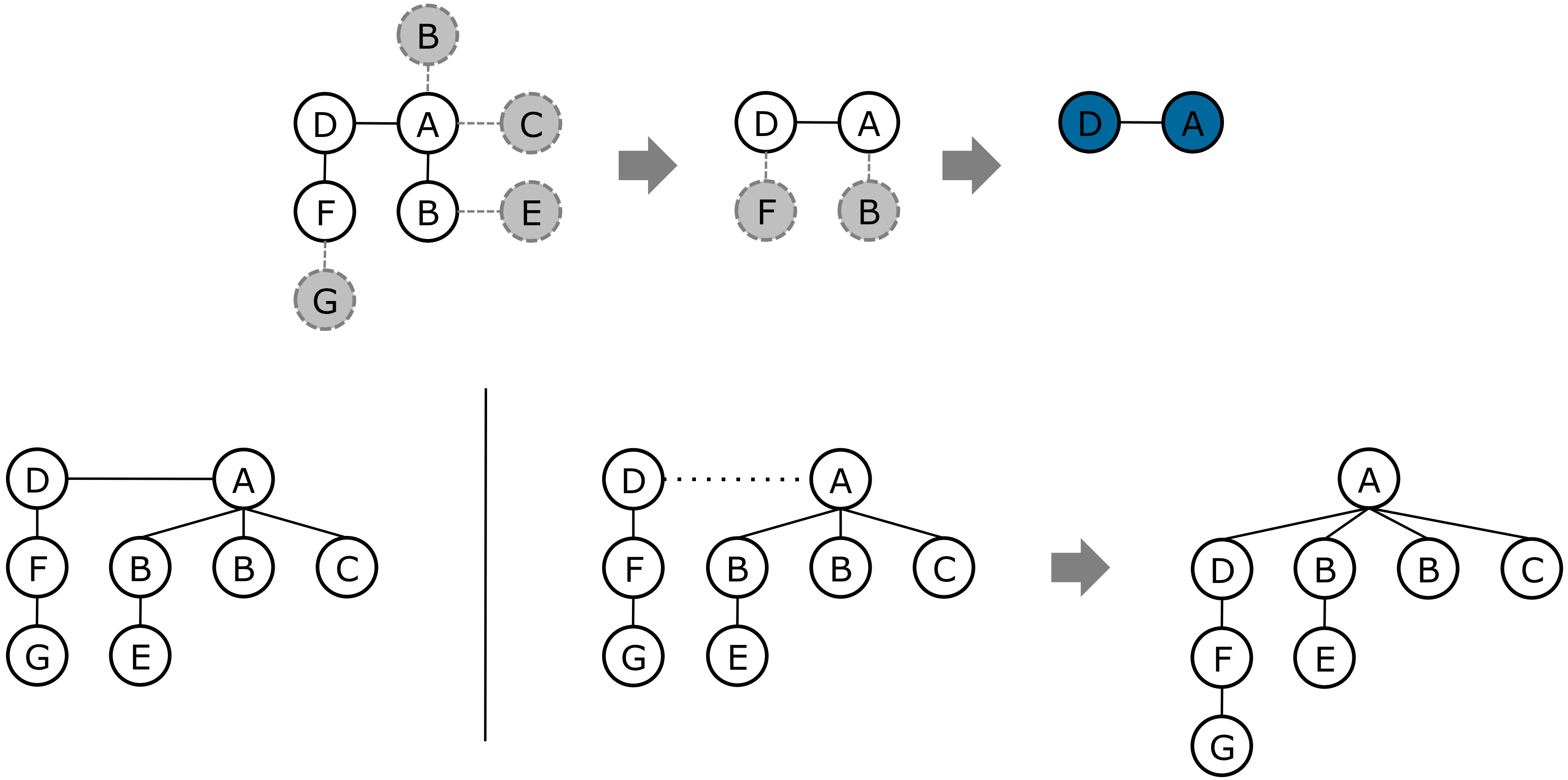
- Knoten, die maximale Distanz zu allen anderen Baumknoten minimieren
- Es gibt höchstens zwei Center → *centered* bzw. *bicentered free tree*

Beispiel *centered free tree*



Normalisierung

Beispiel *bicentered free tree*



DFCF & DFCS

Depth-first canonical string (DFCS)

- Minimaler DFS-String hinsichtlich lexikografischer Ordnung

Depth-first canonical form (DFCF)

- Zu DFCS korrespondierender Baum

Vorgehen

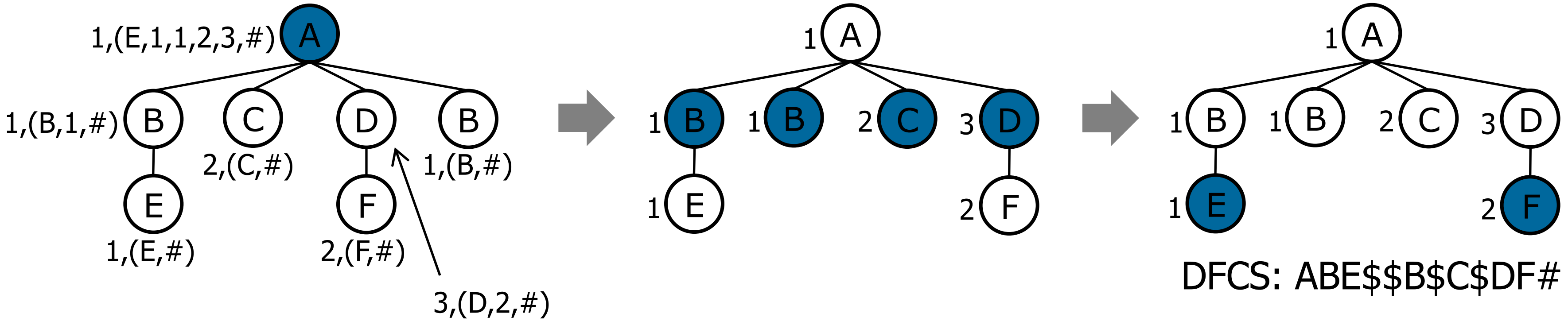
- Bestimme durch Sortierung den Rang (bottom-up)
- Reorganisiere die Kinder (top-down)

DFCF & DFCS – Beispiel

Notation

- Rang, (Label, Rang $Kind_1$, Rang $Kind_2$, ...,#)
- Symbole: $ (Backtrack), # (Ende des Strings)

Beispiel



DFCS: ABE$$BCDF#

BFCF & BFCS

- BFCF und DFCF können identisch
- ...

- Siehe Anhang

Übersicht

- Grundlagen
- Free Tree Miner
  - FTM von Chi et al.
  - FTM von Rückert et al.
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang

Free Tree Miner (Chi et al.)

- Findet häufige Free Trees in Baum-Datenbanken
- Basiert auf *breadth-first pattern-join* Apriori Ansatz

Apriori Ansatz

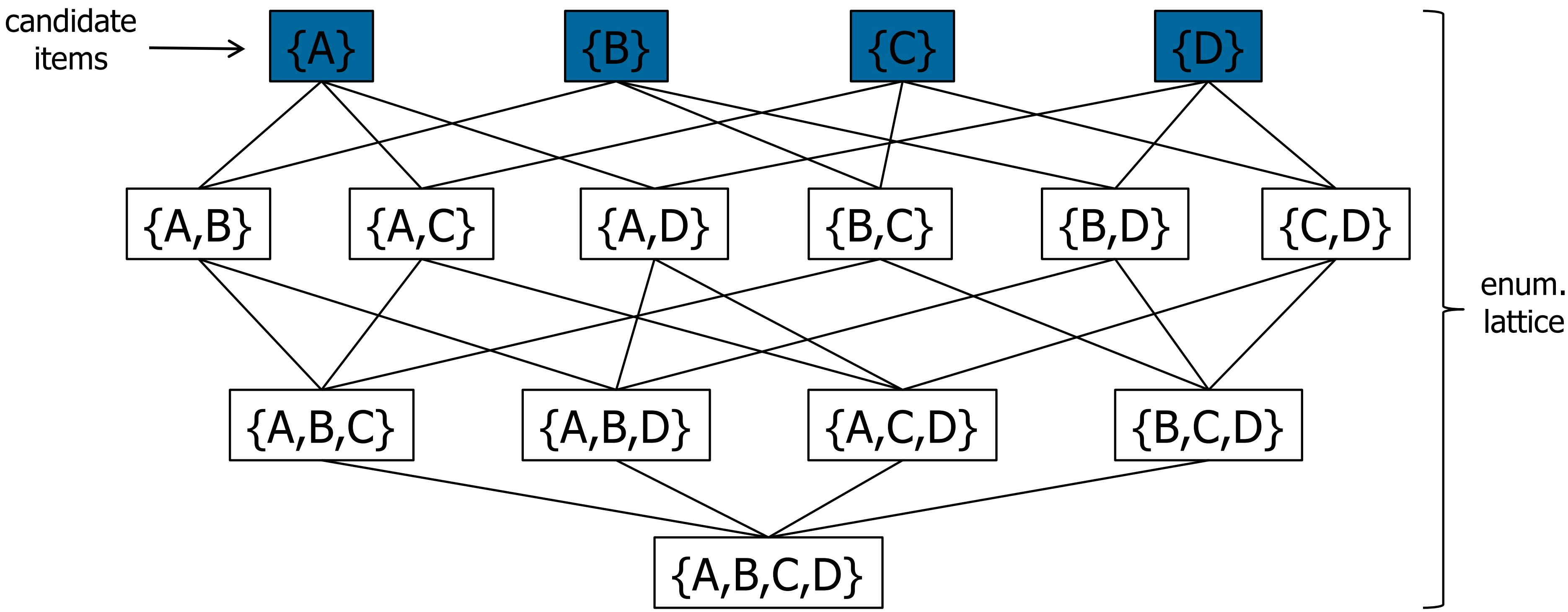
- Ursprung liegt im *frequent itemset mining*
- *enumeration lattice* wird generiert und mit BFS traversiert
- (k+1)-Itemsets werden aus häufigen k-Itemsets erzeugt

Einschub
Frequent Item Mining

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel

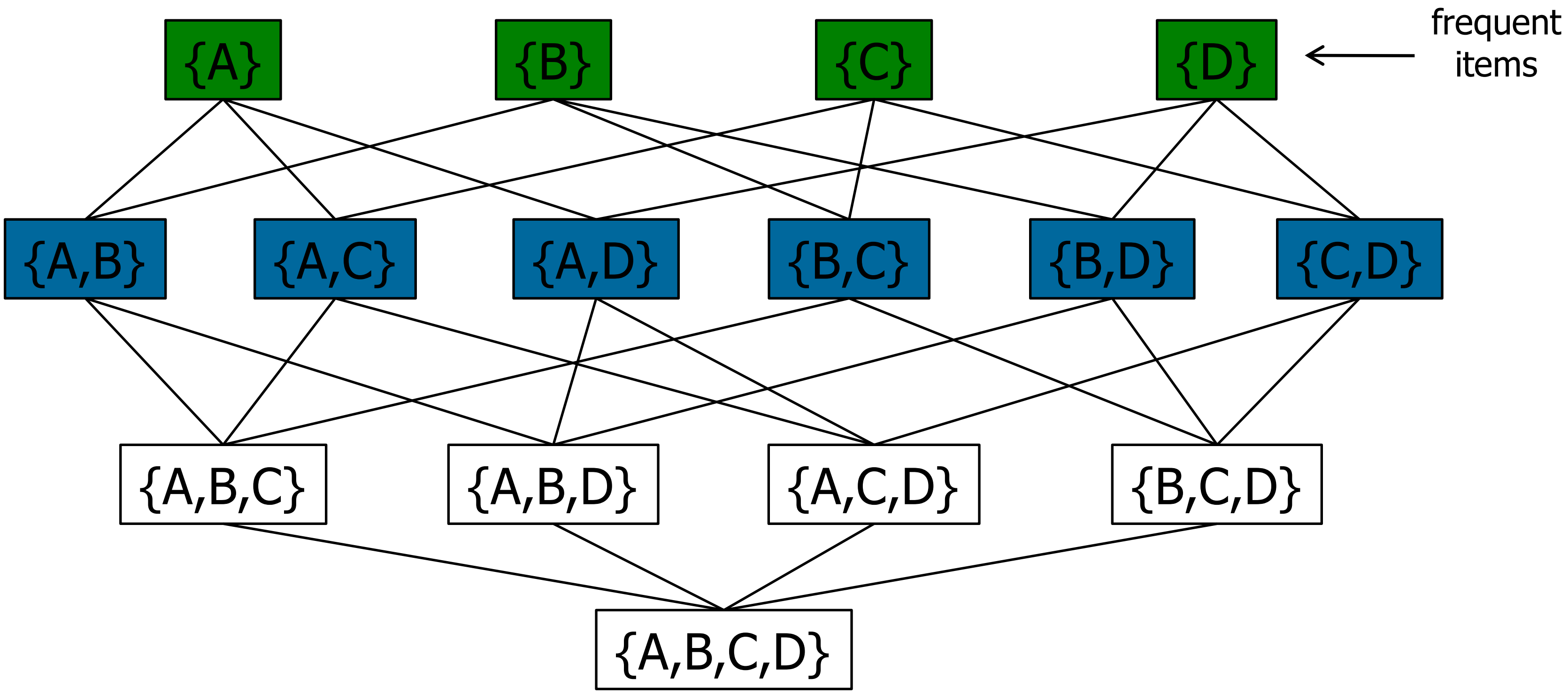
- m = 2, Datenbank: {A,B}, {A,C}, {A,B,D}, {A,B,C,D}



Einschub
Frequent Item Mining

Beispiel (Fortsetzung)

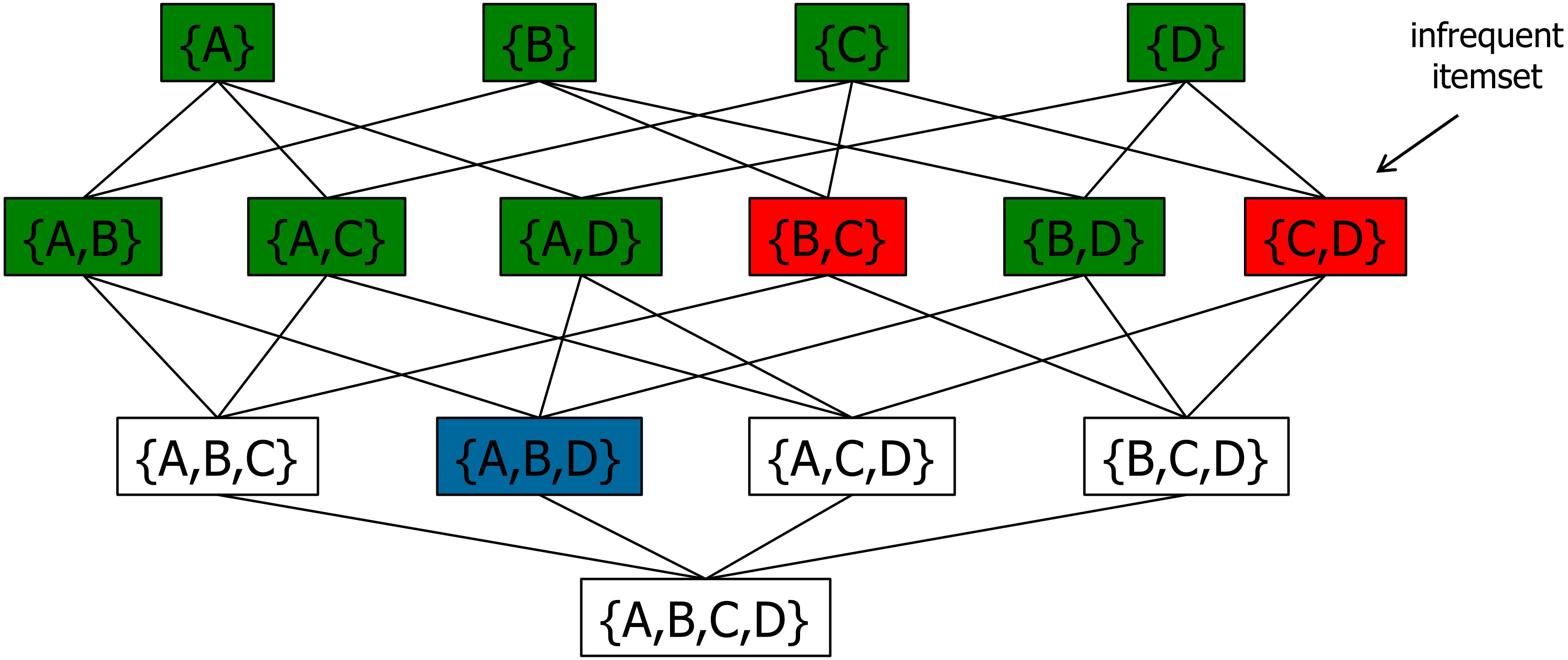
- m = 2, Datenbank: {A,B}, {A,C}, {A,B,D}, {A,B,C,D}



Einschub
Frequent Item Mining

Beispiel (Fortsetzung)

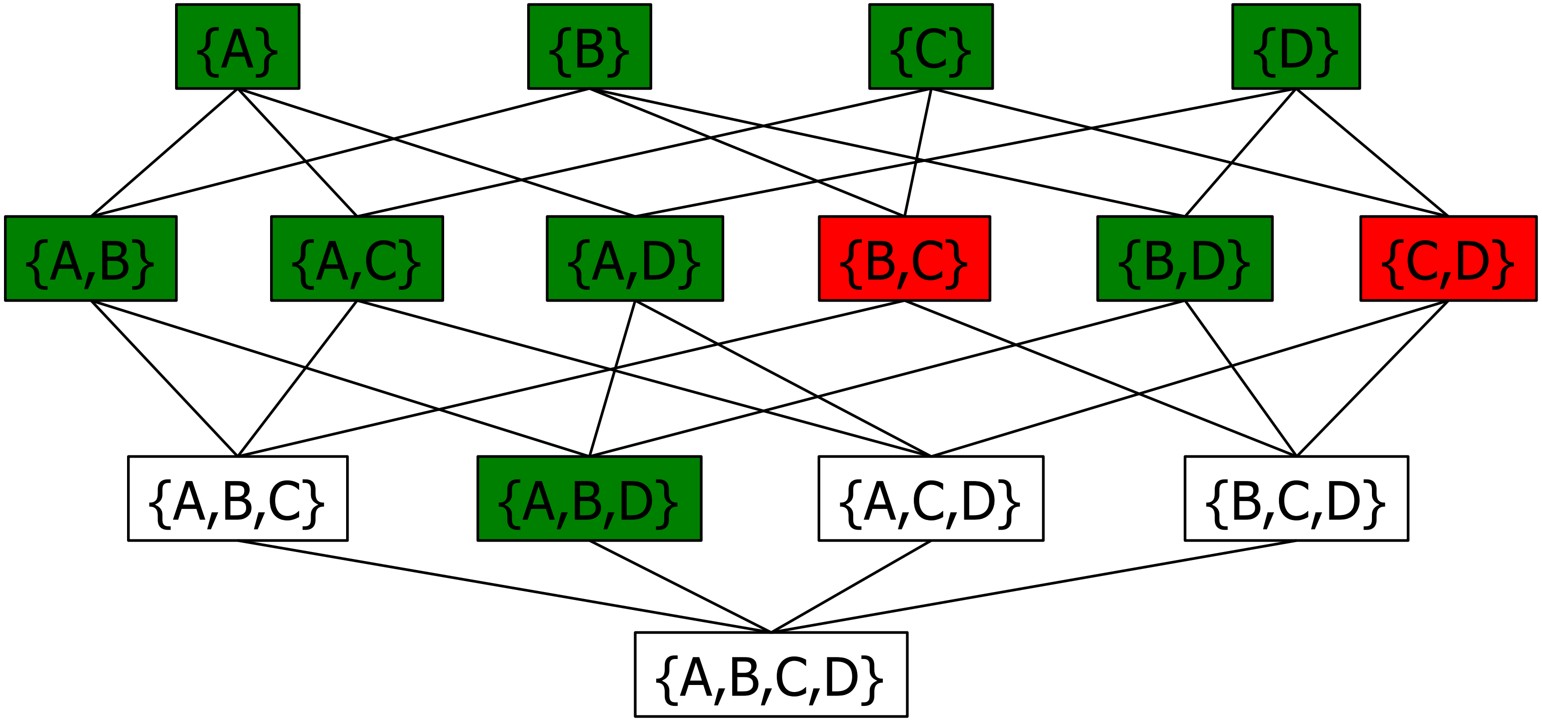
- m = 2, Datenbank: {A,B}, {A,C}, {A,B,D}, {A,B,C,D}



Einschub
Frequent Item Mining

Beispiel (Fortsetzung)

- m = 2, Datenbank: {A,B}, {A,C}, {A,B,D}, {A,B,C,D}



Free Tree Miner (Chi et al.)

Übertragung der Idee auf Free Tree Miner

- Häufige 1 bzw. 2-Bäume werden mit *brute-force* Methode bestimmt
- (k+1)-Kandidaten ergeben sich durch Verknüpfung häufiger k-Bäume
- Nach Kandidaten-Generierung *downward closure checking*

downward closure checking

- Filtert nicht häufige (k+1)-Kandidaten heraus
- Vorgehen
  - Lösche jeweils ein Blatt des (k+1)-Kandidaten
  - Überprüfte resultierende k-Bäume auf Häufigkeit
  - Nicht häufiger k-Baum → Kandidat wird verworfen

Free Tree Miner (Chi et al.)

```
procedure freeTreeMiner(d, m) {
    F1, F2 = { frequent 1 and 2-trees };
    for ( k = 3; Fk-1 ≠ ∅ ; k++) {
        Ck = generateCandidate(Fk-1);

        for each transaction t ε d do
            for each candidate c ε Ck do
                if ( t supports c )
                    c.count++;

        Fk = { c ε Ck | c.count ≥ m };
    }
    output ⋃k Fk
}
```

Free Tree Miner (Chi et al.)

candidate generation (Pseudocode siehe Anhang)

- Zwei k-Bäume werden verknüpft, wenn
  - beide den gleichen *core* haben
  - und die korrespondierenden *limbs* die *top two leafs* im (k+1)-Baum sind

core* und *limb

- Ein (k-1)-Baum heißt *core* des k-Baums und der entfernte Knoten *limb*

top two leafs

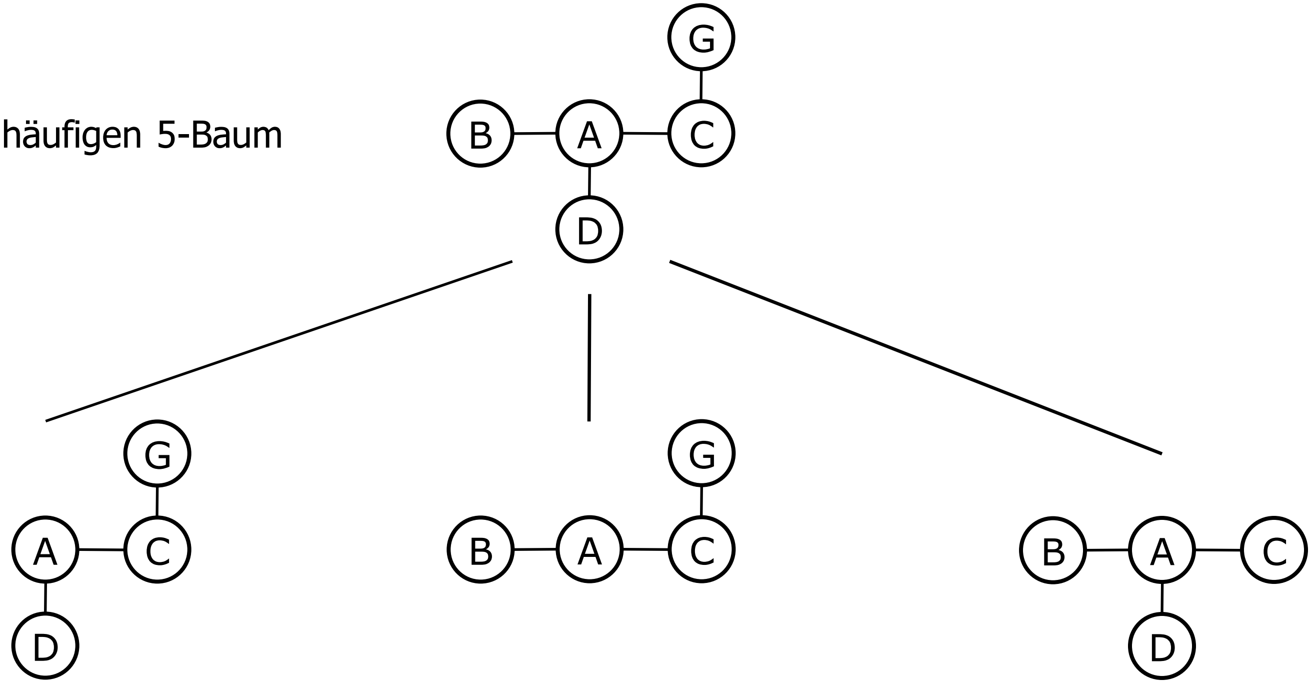
- Die beiden größten Label aller (sortierten) Blattknoten des (k+1)-Baums

Free Tree Miner (Chi et al.)

Beispiel *candidate generation*

Kandidat für einen häufigen 5-Baum

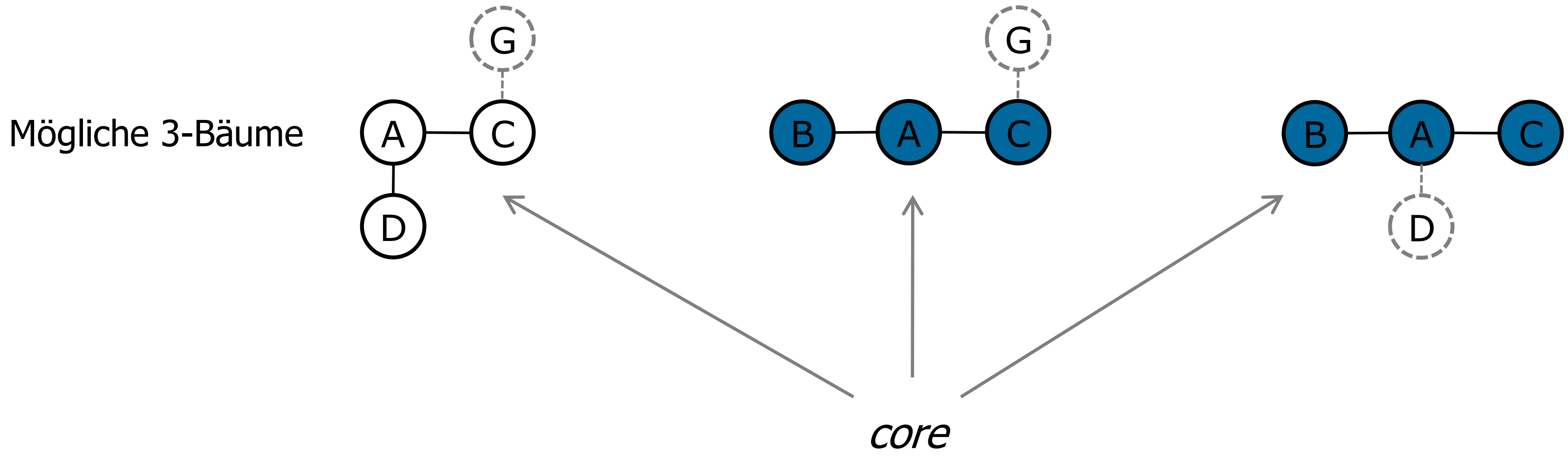
Häufige 4-Bäume



Free Tree Miner (Chi et al.)

Beispiel *candidate generation* (Fortsetzung)

- Haben die 4-Bäume den gleichen *core*?

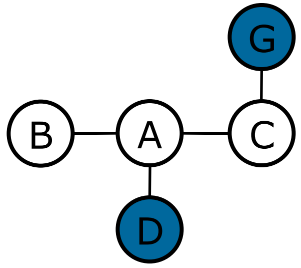


Free Tree Miner (Chi et al.)

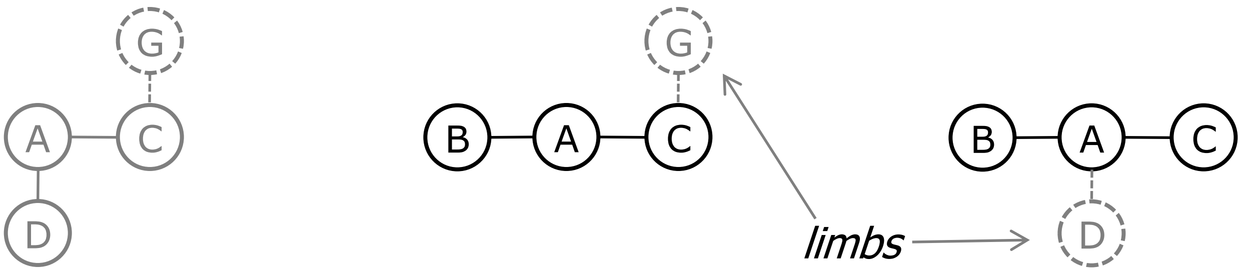
Beispiel *candidate generation* (Fortsetzung)

- Entsprechen die korrespondierenden *limbs* den *top two leafs*?

Kandidat für einen häufigen 5-Baum



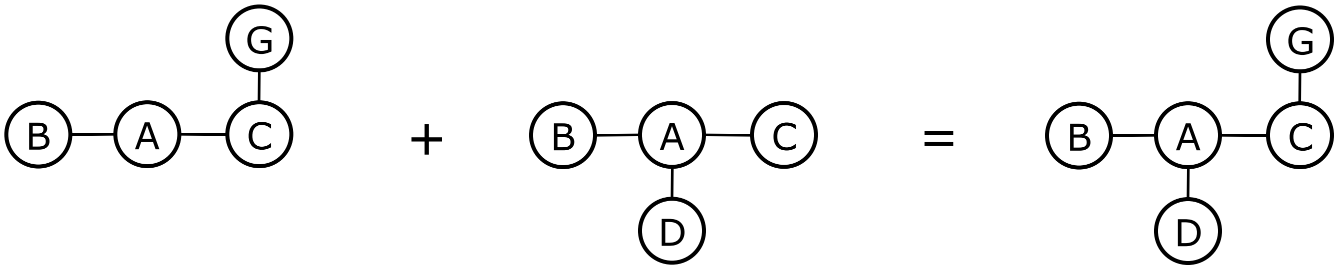
Mögliche 3-Bäume



Free Tree Miner (Chi et al.)

Beispiel *candidate generation* (Fortsetzung)

- Ergebnis



Free Tree Miner (Chi et al.)

```
procedure freeTreeMiner(d, m) {
    F1, F2 = { frequent 1 and 2-trees };
    for ( k = 3; Fk-1 ≠ ∅ ; k++) {

        Ck = generateCandidate(Fk-1);
        for each transaction t ε d do
            for each candidate c ε Ck do
                if ( t supports c )
                    c.count++;
        Fk = { c ε Ck | c.count ≥ m };
    }
    Output ∪_k Fk
}
```

$$\text{procedure } \textbf{\textit{freeTreeMiner}}(d, m)\ \{$$
$$F_1, F_2 = \{\text{ frequent 1 and 2-trees }\};$$
$$\textbf{for}\ (k = 3;\ F_{k-1} \neq \emptyset;\ k{+}{+})\ \{$$
$$C_k = \textit{generateCandidate}(F_{k-1});$$
$$\left.\begin{array}{l}\textbf{for each } \text{transaction } t \in d \textbf{ do}\\ \quad\textbf{for each } \text{candidate } c \in C_k \textbf{ do}\\ \quad\quad\textbf{if } (t \text{ supports } c)\\ \quad\quad\quad c.count{+}{+};\end{array}\right\}\ \textit{frequency counting}$$
$$F_k = \{ c \in C_k \mid c.count \geq m \};$$
$$\}$$
$$\text{Output } \bigcup_k F_k$$
$$\}$$

Übersicht

- Grundlagen
- Free Tree Miner
  - FTM von Chi et al.
  - FTM von Rückert et al.
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang

Free Tree Miner (Rückert et al.)

- Findet häufige Free Trees in Graph-Datenbanken
- Basiert auf *depth-first pattern-growth* Ansatz

pattern-growth Ansatz

- Häufige k-Bäume werden an extension points erweitert

Datenbank-Scan

- FTM sammelt Informationen über Pattern → Häufigkeit, *extension points*
- *extension table* speichert Erweiterungen und *support sets*

Free Tree Miner (Rückert et al.)

procedure ***mineFreeTrees****(d, m)* {

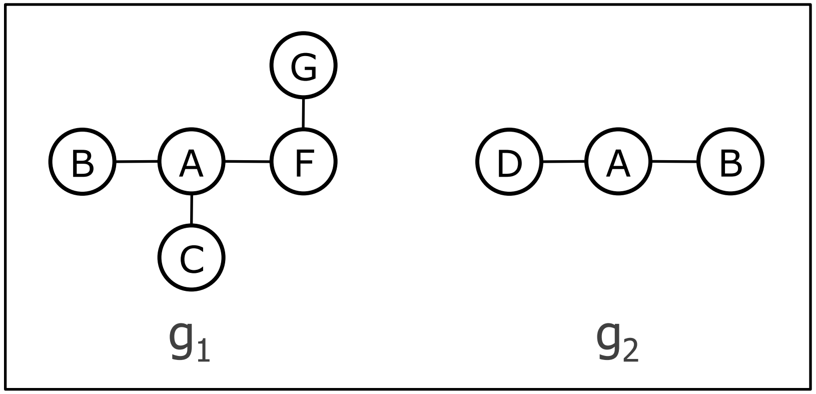
$\boxed{F_1 = \{\text{ set of all frequent 1-trees }\};}$ m = 2 ➡ $F_1 = \{$ A , B $\};$

for all t $\in$ F_1 **do** {

depthSearch(t, d, m);

}

}



Free Tree Miner (Rückert et al.)

procedure ***mineFreeTrees****(d, m)* {

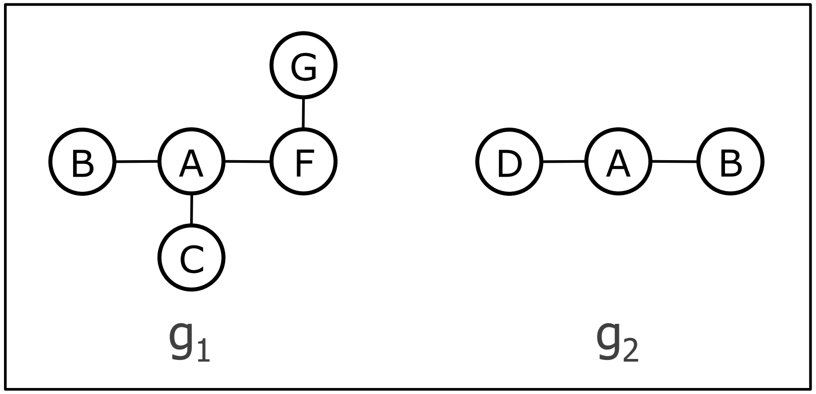
F_1 = { set of all frequent 1-trees };

m = 2 ➡ F_1 = { A , B };

```
for all t ∈ F1 do {
    depthSearch(t, d, m);
}
```

}

Beispiel: Aufruf von *depthSearch(...)* mit Pattern t = A



Free Tree Miner (Rückert et al.)

```
procedure depthSearch(t, d, m) {
    ext = databaseScan(t, d);
    ...
}
```

Free Tree Miner (Rückert et al.)

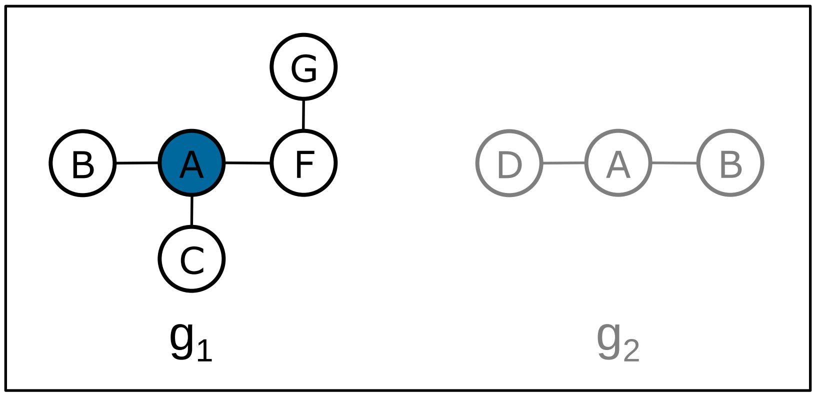
TECHNISCHE UNIVERSITÄT DARMSTADT

```
procedure databaseScan(t, d) {
  ext = empty table;
  for graph g in d do
    for all occurrences of t in g do
      for all extension points p of t do
        for all extensions e to p do {
          if ( (p : e) ∉ ext )
            add (p : e) to ext with empty support set;
          add g to support set in row (p : e);
        }
  return ext;
}
```

Bestimmung der *extension table*

Free Tree Miner (Rückert et al.)

Beispiel *extension table*



extension

support set

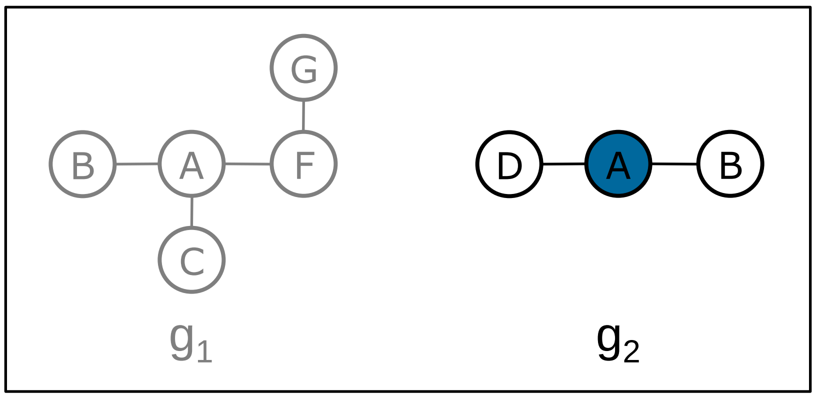
$(e_1 : B)$	$\{g_1\}$
$(e_1 : C)$	$\{g_1\}$
$(e_1 : F)$	$\{g_1\}$

extension table

Pattern $t = A_{e_1}$

Free Tree Miner (Rückert et al.)

Beispiel *extension table* (Fortsetzung)



$(e_1 : B)$	$\{g_1, g_2\}$
$(e_1 : C)$	$\{g_1\}$
$(e_1 : D)$	$\{g_2\}$
$(e_1 : F)$	$\{g_1\}$

Pattern $t = A_{e_1}$

Free Tree Miner (Rückert et al.)

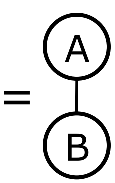
```
procedure depthSearch(t, d, m) {
    ext = databaseScan(t, d);
    if ( support of t ≥ m ) {
        cand = { set of all extension candidates };
        while ( cand ≠ ∅ ) {
            newCand = ∅ ;
            for all candidates c ∈ cand do {
                if ( support of c ≥ m ) {
                    build t‘ from t and c;
                    if ( t‘ is in canonical form )
                        depthSearch(t‘, d, m);
                    if ( t‘ is well ordered )
                        newCand = newCand U { c U (p : e) };
            }}
            cand = newCand; }}
}
```

Kandidaten-Generierung

Free Tree Miner (Rückert et al.)

Beispiel Kandidaten-Generierung

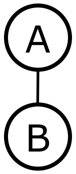
$(e_1 : B)$	$\{g_1, g_2\}$
$(e_1 : C)$	$\{g_1\}$
$(e_1 : D)$	$\{g_2\}$
$(e_1 : F)$	$\{g_1\}$

- Support extension $(e_1 : B) \geq 2$? ✓
- Pattern t = A + extension $(e_1 : B)$ = B ➡ extension candidate t' = 

Free Tree Miner (Rückert et al.)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel Kandidaten-Generierung (Fortsetzung)

- Ist t′ = 

in kanonischer Form? ✓

- Rekursiver Aufruf von *depthSearch(...)*
→ Test auf Häufigkeit von t‛, Generierung weiterer Kandidaten

Überblick

- Grundlagen
- Free Tree Miner
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang

Experimentelle Ergebnisse

TECHNISCHE UNIVERSITÄT DARMSTADT

Datengrundlage

- AIDS-Dataset bestehend aus 42390 chemische Verbindungen
- Kontenlabel: Atomtypen, Kantenlabel: Verbindungen zwischen Atomen

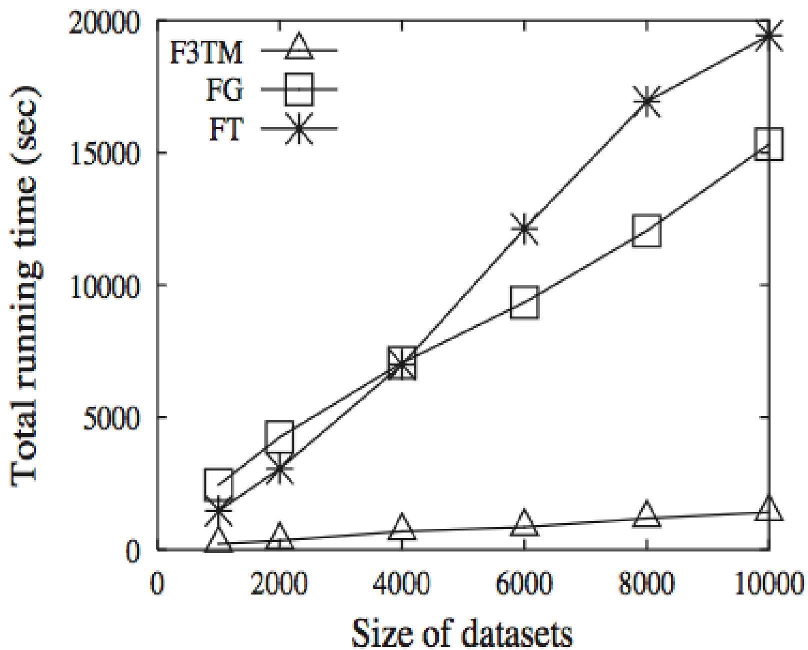
Experiment

- Anzahl der chemischen Verbindungen wird von 1.000 auf 10.000 erhöht

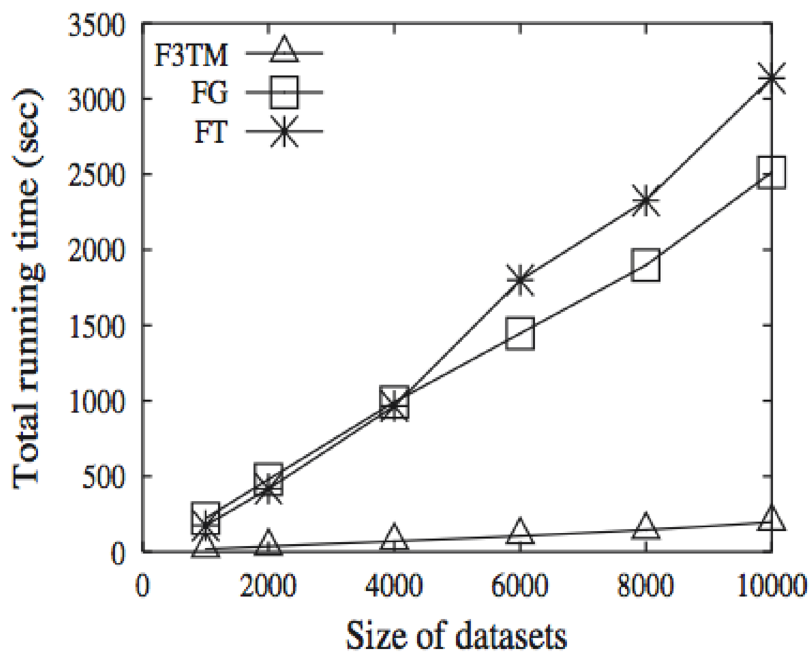
Experimentelle Ergebnisse

Mining-Performance

- Vergleich des Free Tree Miner von Chi (FT) und Rückert (FG) mit dem Fast Frequent Free Tree Miner (F3TM) von Zhao et al.



σ= 0.1



σ= 0.5

Überblick

- Grundlagen
- Free Tree Miner
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang

Fazit & Ausblick

TECHNISCHE
UNIVERSITÄT
DARMSTADT

pattern-join Ansatz (Chi et al.)

- Exponentielles Wachstum von potenziellen (k+1)-Kandidaten

→ Hoher Speicherbedarf → Schlechte Performance

pattern-growth Ansatz (Rückert et al.)

- Geringer Speicherbedarf
- Aber: Kandidaten-Generierung kann Redundanzen erzeugen

Literatur

TECHNISCHE UNIVERSITÄT DARMSTADT

Hauptquelle

- Yun Chi, Yirong Yang, Richard R. Muntz: **Canonical Forms for Labeled Trees and Their Applications in Frequent Subtree Mining**, Knowledge and Information Systems, vol. 8, no. 2, pp. 203-234, 2005.
- Rückert, U., Kramer, S.: **Frequent Free Tree Discovery in Graph Data**, Special Track on Data Mining, ACM Symposium on Applied Computing (SAC'04), 2004.

Vertiefung

- Yun Chi, Yirong Yang, Richard R. Muntz: **Indexing and Mining Free Trees**, Proceedings of the Third IEEE International Conference on Data Mining, p. 509, 2003
- Peixiang Zhao, Jeffrey Xu Yu: **Fast Frequent Free Tree Mining in Graph Databases**, Proceedings of the Sixth IEEE International Conference on Data Mining, pp. 315-319, 2006
- Yun Chi, Richard R. Muntz, Siegfried Nijssen, Joost N. Kok: **Frequent Subtree Mining - An Overview**, Fundamenta Informaticae, vol. 66, issues 1-2, pp. 161-198, 2004
- Rakesh Agrawal, Ramakrishnan Srikant: **Fast Algorithms for Mining Association Rules**, Morgan Kaufmann Series In Data Management Systems - Readings in database systems (3rd ed.), pp. 580-592, 1998

Fragen?

Überblick

- Grundlagen
- Free Tree Miner
- Experimentelle Ergebnisse
- Fazit & Ausblick
- Anhang
  - *breadth-first canonical form*
  - *candidate generation* des Free Tree Miner (Chi et al.)

BFCF & BFCS

TECHNISCHE UNIVERSITÄT DARMSTADT

Breadth-first canonical string (BFCS)

- Minimaler DFS-String unter allen ungeordneten Wurzelbäumen

Breadth-first canonical form (BFCF)

- Zu DFCS korrespondierender Baum

Vorgehen

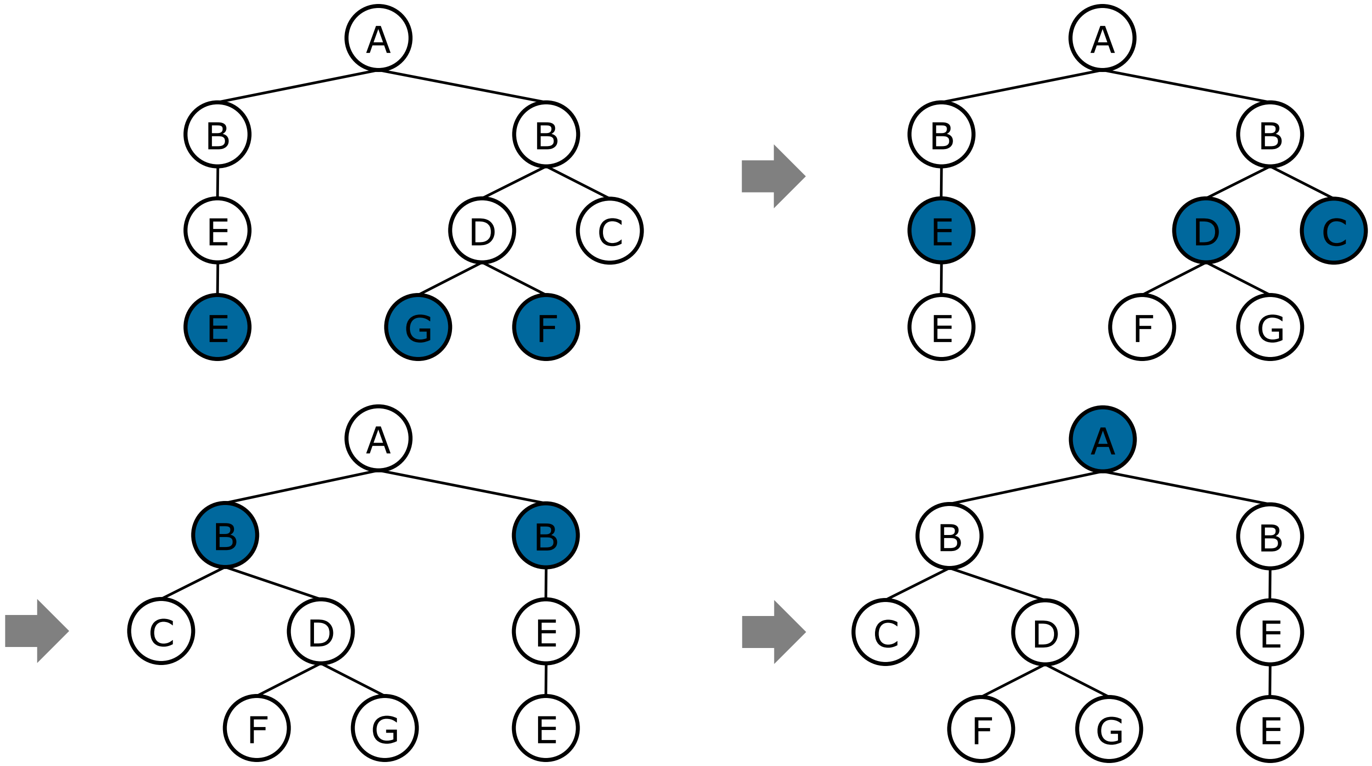
- Sortiere bottom-up auf jeder Stufe von links nach rechts die Knoten von klein nach groß bis alle Kinder der Wurzel reorganisiert sind

Notation

- Symbole: $ (Partition der Kinder), # (Ende des Strings)

BFCF & BFCS

Beispiel



BFCS: ABBCDEFG$E#

Free Tree Miner (Chi et al.)

```
procedure generateCandidate(F_k) {
    C_{k+1} = ∅ ; CL = ∅ ;
    for each tree f in F_k do
        for each leaf l among top 2 leaves of f do
            cl = remove l from f
            if ( cl ∉ CL )
                CL = cl U CL;
            register l to cl in CL;
    for each core cl ∈ CL do
        for each limb pair (l_1, l_2) of cl do
            for each automorphism of cl related to l_1, l_2 do
                c = attach l_1 and l_2 to cl;
                if ( downwardCheck(c, F_k) )
                    C_{k+1} = c U C_{k+1};
    return C_{k+1};
}
```

Free Tree Miner (Chi et al.)

Automorphismus

- Isomorphismus eines Baums auf sich selbst
- Automorphismus des core erschweren Verknüpfung von k-Bäumen
- Schema notwendig um alle möglichen Automorphismen zu erfassen

→ Äquivalenzklasse

Äquivalenzklasse (Automorphismus)

- Knoten u und v eines Baums gehören zur selben Äquivalenzklasse, wenn
  - u und v sich auf der gleichen Stufe befinden,
  - Teilbäume von u und v isomorph sind und
  - u und v den gleichen Elternknoten besitzen

Free Tree Miner (Chi et al.)

Verknüpfung

- Ann.: Ein *limb* gehört zu einer Äquivalenzklasses mit mehreren Elementen
- Betrachte dann alle Kombinationen die entstehen, wenn ein *limb* an jedes Element einer Äquivalenzklasse angehängt wird

Beispiel Automorphismus