# PRÍLOHA A – Algoritmy dolovania frekventovaných množín v statických dátach

V tejto prílohe uvádzame podrobnejší opis niektorých známych algoritmov dolovania frekventovaných množín v statických datasetoch.

## Algoritmus Apriori a jeho variácie

Názov tohto algoritmu, ktorý bol navrhnutý ako jeden z prvých algoritmov riešiacich úlohu hľadania frekventovaných množín v práci (Agrawal & Srikant, 1994), je odvodený od *apriori* znalosti, ktorú tento algoritmus používa. Táto *apriori* znalosť hovorí, že všetky podmnožiny frekventovaných množín sú tiež frekventované množiny. Algoritmus objavuje frekventované množiny iteratívne.

1. Najskôr sa nájdu všetky frekventované *1-množiny*. To si vyžaduje jeden prechod celou databázou – aby sa zistili počty každej *1-množiny*. Vyberú sa iba tie *1-množiny*, ktoré spĺňajú podmienku minimálnej podpory. Označme výsledné nájdené *1-množiny* ako .
2. V ďalších iteráciách sa použije na nájdenie (čo je množina všetkých frekventovaných *2-množín*), sa použije na nájdenie a tak postupne až kým sa nenájdu žiadne ďalšie frekventované *k-množiny*.
3. Ak by sme nevyužili *a priori* znalosť tak by sme v každej iterácií z museli generovať príliš veľké množstvo kandidátov na frekventované *k-množiny* patriace .
4. Využitie *a priori* znalosti v algoritme *Apriori* spočíva v 2 krokoch:
   1. Generovanie kandidátov. Množina kandidátov sa vytvorí spojením s . Táto operácia spojenia predpokladá, že položky v transakciách sú zoradené. Spojenie prebehne pri dvojiciach transakcií, ktoré majú spoločných prvých k-2 položiek a rozdielne položky na pozícií k-1.
   2. Odstránenie kandidátov z , ktoré nie sú frekventovanými množinami. Na základe *a priori* znalosti platí, že akákoľvek *(k-1)-množina*, ktorá nie je frekventovaná nemôže byť podmnožinou frekventovanej *k-množiny*. Skontrolujeme všetky *(k-1) podmnožiny* kandidáta. Ak nie sú všetky frekventované (čo vieme zistiť jednoducho keďže všetky frekventované *(k-1)-množiny* boli nájdené v predchádzajúcej iterácií) tak vieme, že ani kandidát nebude frekventovanou množinou.
5. Po vygenerovaní kandidátov je potrebné ešte prejsť každú transakciu v databáze aby sa vypočítala podpora jednotlivých kandidátov. Nakoniec sa kandidáti s nedostatočnou podporou z odstránia a dostávame výslednú množinu frekventovaných *k-množín* , ktorá sa stáva vstupom do ďalšej iterácie.

Aj keď *Apriori* výrazne zmenšuje množinu kandidátov na frekventované množiny , tak si nepočína dobre najmä pri dlhých frekventovaných vzoroch a nízkych hodnotách minimálnej podpory. Nasledujúce tri uvedené problémy spôsobujú najvyššie straty na efektívnosti algoritmu (Han, Pei, & Yin, Mining Frequent Patterns Without Candidate Generation, 2000).

* V prvých iteráciách je problém s veľkým počtom generovaných kandidátov. Ak je počet frekventovaných *1-množín* tak počet generovaných kandidátov bude väčší ako .
* Ďalším problémom sú veľmi dlhé frekventované množiny. Napríklad na nájdenie frekventovanej množiny s dĺžkou 100 by sa muselo vygenerovať v priebehu algoritmu viac ako kandidátov.
* Výpočtovú náročnosť algoritmu tiež zvyšuje fakt, že sa vyžaduje v každej iterácií prechod celou databázou transakcií a hľadanie prítomnosti kandidátov v jednotlivých transakciách, kvôli výpočtu ich podpory. Na optimalizáciu tohto počítania bolo navrhnuté vylepšenie v práci (Agrawal & Srikant, Fast algorithms for mining association rules, 1994) kde sa používa dátová štruktúra *hash-tree* na ukladanie kandidátov *,* kde list stromu je zoznam kandidátnych k-množín a vnútorné uzly sú hashovacie tabuľky. Konkrétna k-množina je namapovaná na konkrétny listový uzol pomocou aplikácie hašovaciích funkcií v tabuľkách od koreňa stromu až po listový uzol. V i-tej úrovni je táto funkcia aplikovaná na i-tu položku množiny. Položky v k-množinách sú usporiadané. Samotné počítanie prebieha nasledovne. Pre každú transakciu *t* sa v koreňovom uzle pokračuje vo vetvách, ktoré sa vyberú aplikáciou hash-funkcie na každú položku množiny. Vo vnútornych uzloch ak bola *i*-ta položka z *t* poslednou zahašovanou tak sa pokračuje vo vetvách, ktoré sa vyberú aplikáciou hašovacej funkcie na všetky položky za ňou. Pri každom nájdenom listovom uzle sa inkrementuje podpora všetkých jeho kandidátnych k-množín (Aggarwal, Bhuiyan, & Hasan, 2014).

Bolo navrhnutých viacero optimalizácií algoritmu Apriori. V pôvodnej práci (Agrawal & Srikant, 1994) boli navrhnuté optimalizácie *AprioriTID*, *AprioriHybrid.* Nech R(T, C\_k) označuje množinu identifikátorov kandidátnych k-množín C\_k, ktoré sa nachádzajú v transakcií T. V algoritme *AprioriTID* sa všetky takéto množiny uložia do novej databázy transakcií , ktorá sa použije namiesto . Nová databáza je úspornejšia, transakcie sú v nej kratšie. V niektorých prípadoch nebude mať transakcia za podmnožinu žiadnu kandidátnu *k-množinu* a to hlavne v prípadoch keď je *k* väčšie. V tom prípade je transakcia vymazaná z čo má za následok zrýchlenie výpočtu. V niektorých prípadoch však bude mať transakcia za podmnožinu viac kandidátnych podmnožín čo je nežiadúce. Variácia *AprioriHybrid* používa toto vylepšenie až v neskorších iteráciach kedy je k-väčšie a teda prirodzene viacero transakcií z je odstraňovaných (Aggarwal, Bhuiyan, & Hasan, 2014).

## FP-growth

*FP-Growth* (skrátené od ang. *frequent pattern growth*) je algoritmus navrhnutý v práci (Han, Pei, & Yin, 2000), ktorý sa pokúša prekonať popísané problémy algoritmu *Apriori*, ktoré spôsobujú jeho výpočtovú náročnosť spôsobenú najmä veľkým počtom generovaných kandidátov.

Autori v článku (Han, Pei, & Yin, 2000) navrhli dátovú štruktúru na ukladanie frekventovaných množín s názvom *FP-tree*. Pri konštrukcií *FP-tree* štruktúry vychádzajú z týchto poznatkov :

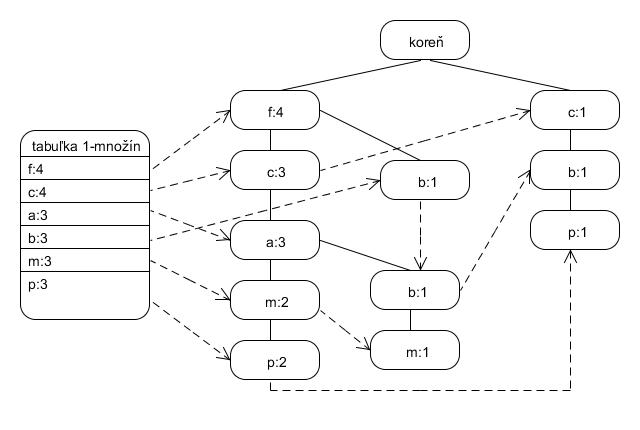
1. Je nutné vykonať minimálne jeden prechod databázou aby sa našli všetky frekventované 1-množiny.
2. Ak dokážu ukladať frekventované množiny nachádzajúce sa v jednotlivých transakciách do nejakej kompaktnej dátovej štruktúry tak nebude potrebné prechádzať celou databázou kvôli hľadaniu frekventovaných *k-množín* pre každé *k*.
3. Ak sa vo viacerých transakciách nachádza identická frekventovaná množina, môžu byť zlúčené a uložené ako jedna položka v dátovej štruktúre.
4. Predpokladajme, že položky v transakcií sú usporiadané zostupne podľa podpory jednotlivých položiek vzhľadom na celú databázu. Potom bude celkom veľká šanca, že viacero transakcií bude mať spoločný prefix. Ak majú dve transakcie spoločný prefix, tak túto spoločnú časť môžeme zlúčiť.

Spôsob konštrukcie *FP-tree* uvedieme na upravenom príklade z (Han, Pei, & Yin, Mining Frequent Patterns Without Candidate Generation, 2000). V Tabuľka 1 uvádzame príklad databázy transakcií s 5 transakciami, z ktorej zostrojíme *FP-tree.* Nech hodnota minimálnej podpory v tomto príklade je 3.

|  |  |  |
| --- | --- | --- |
| TID | Položky transakcie | Frekventované položky transakcie |
| 1 | f, a, c, d, g, i, m, p | f, c, a, m, p |
| 2 | a, b, c, f, l , m, o | f, c, a, b, m |
| 3 | b, f, h, j, o | f, b |
| 4 | b, c, k, s, p | c, b, p |
| 5 | a, f, c, e, l, p, m, n | f, c, a, m, p |

Tabuľka 1 Príklad databázy transakcií

1. Najskôr prechádzame databázu a hľadáme frekvencie výskytov jednotlivých *1-množín*. Dostaneme tabuľku *1-množín* ako je uvedené na Obrázok 1, kde sú jednotlivé položky zoradené podľa počtu výskytov v databáze (počet výskytov je číslo uvedené pri položke).
2. Vytvoríme prázdny koreň stromu.
3. Znova prechádzame celú databázu transakcií.
   1. Pri prechode transakciou 1 zostrojíme prvú vetvu stromu (zľava): *<f:1,c:1,a:1, m:1, p:1>* s frekventovanými *1-množinami* z transakcie. Dôležité je, že zachovávame pri konštrukcií danej vetvy usporiadanie z tabuľky *1-množín*.
   2. Pri prechode druhou transakciou by sme zostrojili vetvu *<f:1,c:1,a:1, b:1, m:1>*. Vidíme, že má spoločný prefix *<f,c,a>* s prvou vetvou. Zvýšime početnosť položiek so spoločného prefixu v prvej vetve a vytvorím podvetvu *<b:1,m:1>*, ktorú pripojíme k rodičovskému uzlu *<a:2>*
   3. Podobne pri prechode treťou transakciou je zoznam frekventovaných položiek *<f:1,b:1>*. Spoločný prefix je iba *<f>*. Inkrementujeme početnosť uzlu *<f:2>* na *<f:3>* a vytvoríme nový uzol *<b:1>*, ktorý pripojíme k rodičovskému uzlu *<f:3>*.
   4. Pri prechode štvrtou transakciou vytvoríme novú vetvu *<c:1, b:1,p:1>*. Keďže nezdieľa prefix s žiadnou inou transakciou, tak ju pripojíme priamo na koreň.
   5. V poslednej transakcií máme vetvu *<f:1,c:1,a:1, m:1, p:1>*, ktorá je celá identická s prvou. Stačí teda inkrementovať počty v tejto vetve.
4. Aby bolo možné uskutočniť prechod stromom, z tabuľky *1-množín* zostrojíme ukazovatele na spájané zoznamy, ktoré spájajú uzly s rovnakým názvom *1-množiny* nachádzajúcej sa v strome.
5. Zostrojenú štruktúru vidíme na Obrázok 1.



Obrázok 1 Príklad zostrojenej štruktúry FP-tree.

V ďalšej časti práce (Han, Pei, & Yin, 2000) autori navrhli algoritmus, ktorý z takto vytvorenej štruktúry nájde všetky frekventované vzory, ktorý nazvali *FP-growth* (ang. *frequent pattern growth*).

Tento algoritmus využíva vlastnosti dátovej štruktúry *FP-tree*:

* Pre frekventovanú *1-množinu* je možné nájsť všetky frekventované množiny v ktorých sa vyskytuje nasledovaním liniek spájaného zoznamu, ktorý začína v tejto *1-množine*. Teda v našom príklade vieme napr. pre *1-množinu* <*p>* nájsť vetvy v ktorých sa nachádza a to sú : *<f:4, c:3, a:3, m:2, p:2>* a druhá *<c:1, b:1, p:1>*. Z prvej uvedenej vetvy vieme, že početnosť množiny *<f,c,a,m,p>* v databáze je 2, keďže prefix *<f,c,a,m>* sa nachádza spoločne s položkou *p* v databáze presne 2 krát.
* Na výpočet frekventovaných množín, ktoré obsahujú položku *i*je potrebné vo vetve stromu v ktorej sa nachádza uvažovať len časť vetvy, ktorej uzly sú predkami tejto položky – teda prefixová časť vetvy. Tiež platí, že všetky uzly tejto prefixovej časti budú mať rovnakú početnosť ako položka *i.* Množinu všetkých prefixových časti pre všetky vetvy v ktorých sa položka *i*nachádza nazveme *podmienený základ vzoru* (ang. *conditional pattern base*). Označíme ju ako “*základ vzoru*|*i “.* Z tohto podmieneného základu vzoru vieme vypočítať všetky frekventované množiny obsahujúce položku *i* a to pomocou zostrojenia ďalšieho menšieho *FP-tree* a ďalším rekurzívnym hľadaním podmienených základov a konštruovaním ďalších *FP-tree.*
* Nech *a*je frekventovaná množina v databáze *D.* Nech *B* je *podmienený základ vzoru* a β je množina . Potom platí že je frekventovaná množina ak a iba ak je β frekventovaná množina.
* Pre *1-množiny* teda zostrojíme *podmienený základ vzoru*, čo je napríklad pre *1-množinu* *p* množina jej tzv. prefixových ciest s upravenou početnosťou. Teda : |*p*.

Pre *1-množinu* *m* je to zas: .

* Ďalej pokračujeme rekurzívnym zostrojovaním podmienených *FP-tree* z nájdených *podmienených základov vzorov.* Pričom pre každú položku z tabuľky frekventovaných *1-množín* nového *FP-tree* vytvárame rastúci vzor , kde je vzor, ktorý bol doposiaľ vytvorený (teda bol vstupom do rekurzívneho volania) pričom podpora je rovná podpore položky .
* Ďalej platí, že ak nový *FP-tree* má len jednu vetvu, tak nájdeme z nej všetky frekventované množiny tak, že nájdeme všetky kombinácie položiek tejto vetvy, ktoré spojíme s doposiaľ vytvoreným vzorom. Podpora každej kombinácie bude rovná podpore položky s minimálnou podporou v nej.
* Napríklad z podmieneného základu vzoru zostrojíme *FP-tree*, ktorý bude mať len jednu vetvu *<c:3>.* Tak vieme povedať, že frekventovaná množina je *<c, p>* s podporou 3.

Pre prípad rekurzívnej vetvy, ktorá začínala s *1-množinou* *m b*ude skonštruovaný *FP-tree* s jednou vetvou *<f:3,c:3,a:3>*. Z nej vieme nájsť všetky frekventované množiny obsahujúce *m.* Jednoducho vytvoríme z tejto vetvy všetky kombinácie, ktoré spojíme s doteraz nájdeným vzorom, ktorým je v tomto prípade zatiaľ len položka *m.* Budú to tieto frekventované množiny:

*<f,m>, <c,m>, <a,m>, <f,c,m>, <f,a,m>, <c,a,m>,<f,c,a,m>.* Všetky budú mať podporu rovnakú ako je podpora najmenšej položky v kombinácií a to je vo všetkých prípadoch podpora položky *m,* ktorá je 2.

V uvedenej práci autori ukazujú, že algoritmus *FP-growth* je efektívnejší ako *Apriori* a jeho iné varianty založené na generovaní kandidátov podľa apriori vlastnosti aj v prípadoch krátkych aj dlhých frekventovaných množín.

## Eclat

Eclat (z ang. Equivalence Class Transformation) je jeden zo 6 navrhovaných algoritmov dolovania frekventovaných množín v práci (Zaki M. J., 2000).

Všetky navrhované algoritmy sú postavené na podobnom princípe. Kľúčové prvky tohto princípu sú tieto :

1. Použitie vertikálneho formátu dát, teda indexu kde je pre každú množinu položiek uvedený zoznam transakcií v ktorých sa nachádza.
2. Rozdeľovanie priestoru prehľadávania do nezávislych častí.
3. Oddelenie úlohy rozdeľovania problému na menšie časti a úlohy hľadania frekventovaných množín v týcho častiach.
4. Minimalizovanie I/O operácií vďaka menšiemu počtu prechodov databázy. V najlepšom prípade len jeden.

Ako horizontálny formát dát je označovaný klasický formát, kde jeden záznam predstavuje jednu transakciu a pozostáva z identifikátora transakcie (skr. TID) ku ktorej je uvedený zoznam položiek transakcie. Ako vertikálny formát dát je označovaný formát, kde jeden záznam predstavuje množinu položiek ku ktorej je uvedený zoznam TID v ktorých sa nachádza. Transformácia horizontálneho formátu do vertikálneho formátu dát si vyžaduje jeden kompletný prechod databázou transakcií.

Platí, že TID zoznamy pre frekventované (k+1)-množiny je možné získať s využitím apriori znalosti prienikom všetkých párov TID zoznamov frekventovaných k-množín. Pričom platí, že kardinalita TID zoznamov bude so zvyšujúcim sa k klesať, čo prakticky znamená stale rýchlejší výpočet väčších frekventovaných množín. Kritický je teda najmä začiatok algoritmu, kedy z frekventovaných 1-množín generujeme frekventované 2-množiny.

Autori navrhli tiež spôsob ako výpočet rozdeliť do nezávislých častí a tak znížiť pamäťové zaťaženie (v pamäti nemusia byť ukladané všetky medzivýsledky) a umožniť aj prípadné paralelizovanie výpočtu. Nezávisle je možné hľadať všetky frekventované k+1-množiny pre jednotlivé triedy ekvivalentnosti na úrovni *k*. Dve *(k+1)*-množiny sú v rovnakej triede ekvivalentnosti definovanou reláciou ak zdieľajú rovnaký prefix dĺžky *k*. (predpokladáme, že TID zoznamy sú vždy udržiavané usporiadané).

Algoritmus môže byť potom založený na rekurzívnej dekompozícií každej triedy ekvivalentnosti definovanej reláciou do menších tried ekvivalentnosti. Samotné prehľadávanie priestoru riešení potom môže byť hľadaním do hĺbky alebo hľadaním do šírky. Pri hľadaní do šírky sa najskôr vygeneruje všetky k+1-frekventované množiny pre všetky triedy ekvivalentnosti na úrovni k.

## CHARM

CHARM (Zaki & Hsiao, 2002) je algoritmus na dolovanie uzavretých frekventovaných množín, ktorý podobne ako algoritmus *Eclat* (Zaki M. J., 2000) využíva vertikálny formát transakčných dát. Nech P a Q sú množiny položiek. Nech je zoznam identifikátorov transakcií v ktorých sa nachádza množina P. CHARM využíva tieto tri vlastnosti pre zmenšovanie prehľadávaného priestoru:

* Ak t (P) = t(Q), potom platí aj

To znamená, že každý výskyt P môžeme nahradiť a odstrániť Q aj celú vetvu pod ním v prehľadávanom priestore, keďže obidve množiny majú rovnaký uzáver.

* Ak , potom platí aj:. Takže môžeme nahradiť každý výskyt P množinou pretože ak sa P vyskytuje v nejakej transakcií tak sa tam nachádza aj Q. Keďže však platí, že Q sa nachádza aj v iných transakciách tak nemožno odstrániť z prehľadávaného priestoru Q.
* Ak , potom platí aj . V tomto prípade nemôže byť odstránená ani jedna množina, keďže P aj Q vedú k odlišným uzavretým frekventovaným množinám.

Vstupnými parametrami metódy je zoznam *I* dvojíc frekventovaných 1-množín s ich zoznamami identifikátorov transakcií a minimálna podpora.

1. Množiny v *I*sa zoradia podľa stúpajucej podpory (vypočítaná ako kardinalita množiny transakcií).
2. Pre každú množinu sa algoritmus pokúsi rozšíriť ju o inú množinu a aplikovať uvedené 3 vlastnosti pre zmenšovanie prehľadávaného priestoru. Ak je novovytvorená množina z P a Q frekventovaná tak overuje či je uzavretá.
3. Algoritmus CHARM je volaný rekurzívne pre upravenú množinu I až pokiaľ táto nie je prázdna.

## Citované práce

Aggarwal, C. C., Bhuiyan, M. A., & Hasan, M. A. (2014). Frequent Pattern Mining Algorithms:A Survey. In V. E. Lee, R. Jin, & G. Agrawal, *Frequent Pattern Mining* (s. 19-61).

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. *Proceeding VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases*, 487-499.

Han, J., Pei, J., & Yin, Y. (2000). Mining Frequent Patterns Without Candidate Generation. *SIGMOD Rec.*, 1-12.

Zaki, M. J. (2000). Scalable Algorithms for Association Mining. *IEEE Trans. on Knowl. and Data Eng.*, 372-390.

Zaki, M. J., & Hsiao, C.-J. (2002). CHARM: An efficient algorithm for closed itemset mining. *Proceedings of the 2002 SIAM International Conference on Data Mining*, (s. 457--473).