VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

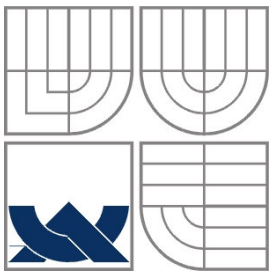
ZÍSKÁVÁNÍ ZNALOSTÍ Z XML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

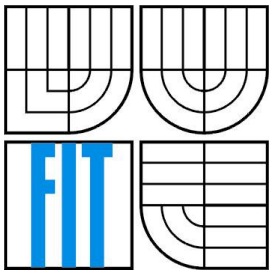
AUTOR PRÁCE
AUTHOR

Bc. LADISLAV MELICHAR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZÍSKÁVÁNÍ ZNALOSTÍ Z XML

DATAMINING FROM XML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LADISLAV MELICHAR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PAVEL JURKA

BRNO 2008

Abstrakt

Dolování v XML bylo až donedávna neprozkoumanou oblastí a dalo by se říci, že tomu tak stále ještě je. V projektu se zaměřuji nejprve obecně na problematiku získávání znalostí ze strukturovaných dat, speciálně na data ve formátu XML. Dále je zde prezentován stromový algoritmus HybridTreeMiner s cílem jeho aplikace pro získávání znalosti z XML dokumentů. Praktická část projektu se věnuje navržení koncepce pro začlenění algoritmu jako modulu do dolovacího systému vyvíjeného na FIT. Tento systém je implementován v programovacím jazyce Java, má modulární strukturu a jeho jednotlivé části spolu komunikují pomocí jazyka DMSL. Na závěr jsou prezentovány a diskutovány dosažené výsledky.

Klíčová slova

Získávání znalostí z databází, dolování dat, XML, DTD, HybridTreeMiner, Java

Abstract

The data mining is still little investigated area. This project is aimed firstly generally to the knowledge discovery from the structured data, especially from the datas in XML format. Furthermore the tree algorithm HybridTreeMiner is presented here with aim of its application for the knowledge discovery from XML documents. The practical part of this project is dedicated to the design of the conception for the algorithm integration to the mining system developed in FIT. This system is implemented in the programming language Java, it has modular structure and its parts communicate each other by means of the language DMSL. Reached results are presented and discussed in the end.

Keywords

Knowledge discovery in databases, data mining, XML, DTD, HybridTreeMiner, Java

Citace

Melichar Ladislav: Získávání znalostí z XML. Brno, 2008, diplomová práce, FIT VUT v Brně.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Pavla Jurky
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ladislav Melichar
10. května 2008

Poděkování

Děkuji svému vedoucímu Ing. Pavlu Jurkovi za odborné vedení, cenné rady a podněty, které mi při řešení tohoto projektu poskytl.

Obsah

1 Úvod

Díky neustálému rozvoji informačních technologií a zvětšování možností v oblasti sběru dat, které tyto technologie umožňují, dochází v poslední době k všeobecnému nárůstu uchovávaných dat. V důsledku toho nabývá na významu problém efektivního získávání znalostí založených na těchto datech. Jako součást řešení tohoto problému byl na naší fakultě vyvinut jazyk DMSL (Data Mining Specification Language), na jehož základě byl vytvořen systém pro získávání znalostí z databází.

Tato diplomová práce je zaměřena na problematiku získávání znalostí ze strukturovaných dat, především pak z XML. Cílem projektu je navrhnout koncepci modulu pro speciální dolovací algoritmus, poté tento modul realizovat a rozšířit jím stávající systém pro získávání znalostí vyvíjený na FIT VUT Brno. Tato práce částečně navazuje na ročníkový projekt, ve kterém jsem se zabýval zajištěním propojení jednotlivých modulů systému. Přitom jsem se seznámil s problematikou získávání znalostí z databází, dále se strukturou systému a také s jazykem DMSL.

Druhá kapitola, následující za úvodem, se zabývá problematikou získávání znalostí z databází v obecné formě. Popisuje proces získávání znalostí a kroky, které jsou nutné pro kvalitní výsledky dolovacího procesu.

Třetí kapitola se soustřeďuje na rozbor jazyka XML, v další kapitole, tj. v čtvrté, je pak stručně popsán jazyk DTD soužící jako popis syntaxe jazyka XML.

Pátá kapitola se zabývá dnešním stavem získávání znalostí z XML. Přehledně uvádí nové trendy a možnosti v tomto směru. Některé postupy jsou uvedeny jen výčtově s odkazem na příslušnou literaturu, jiné jsou stručně popsány.

Šestá kapitola popisuje teoreticky vybraný algoritmus HybridTreeMiner. Nejprve je pozornost zaměřena na definici důležitých pojmů, především z oblasti grafové teorie. V návaznosti na to, je prezentován vybraný algoritmus, je uveden jeho podrobný popis s vazbou na připravovanou implementaci.

Sedmá kapitola obsahuje popis dolovacího systému, jeho modulů a postup přidávání nového modulu. Dále je zde uváděn jazyk DMSL, který slouží jako komunikační prostředek mezi moduly a systémem. Pro nově vznikající modul je zde popsáno vytvoření nových prvků jazyka DMSL.

V osmé kapitole je popsána realizace dolovacího modulu implementujícího algoritmus HybridTreeMiner. Tato část popisuje konkrétně implementační proces v jazyce Java.

Devátá kapitola se zabývá testováním dolovacího modulu a vyhodnocením testů. Obsahuje i přehledové srovnání s jinými podobnými algoritmy.

Závěr obsahuje zhodnocení výsledků práce a návrh možných vylepšení.

2 Úvod do získávání znalostí

Získávání znalostí z databází si získalo značnou pozornost v posledních letech díky široké dostupnosti obrovského objemu dat a naléhavé potřebě přeměnit tato data na užitečnou informaci a znalost. Ta pak může být využita v oblastech od analýzy trhu, detekce podvodů, akcí na udržení si zákazníků po řízení výroby a vědecké bádání. V té době používané metody analýzy a zejména podpůrné nástroje dat založené na statistice nebyly schopny odhalit požadované znalosti v tak obrovských objemech dat. Bylo nutné hledat metody a algoritmy pro automatizovanou analýzu velkého množství dat a implementovat je. Výsledkem byl vznik získávání znalostí z databází nebo obecně z dat jako jednoho z rychle se rozvíjejících směrů počítačových věd a také vznik nové generace nástrojů pro analýzu dat. Získávání znalostí z databází lze chápat jako výsledek přirozeného vývoje databázové technologie. Podobně i prudký rozvoj webu, vznik jazyka XML a podpůrných technologií, které umožňují vytvářet globální informační systémy, staví před získávání znalostí z databází nové výzvy.

Lze říci, že *získávání znalostí z databází* je extrakce zajímavých (netriviálních, skrytých, dříve neznámých a potenciálně užitečných) modelů dat a vzorů z velkých objemů dat. Tyto modely a vzory reprezentují znalosti získané z dat. *Netriviálnost* znamená, že nejde o informaci, kterou lze získat například nějakým SQL dotazem nad databází, nýbrž je nutné použít nějaký sofistikovaný postup. Nejde tedy například o přehled počtu prodaných výrobků v jednotlivých prodejnách. Za získávání znalostí z databází se ani nepovažují deduktivní databázové či expertní systémy. Podobně *skrytost* definuje, že musí jít o modely a vzory, které nejsou v datech na první pohled vidět, musí se v datech nalézt a to netriviálním způsobem. Databáze nebyla navrhována s ohledem na to, abychom takový typ informace ukládali. *Potenciální užitečnost* získaných znalostí je měřena významem pro nějaké rozhodnutí. Může jít například o podklad pro rozhodnutí o půjčce klientovi banky nebo rozhodnutí o uspořádání zboží v supermarketu na základě znalosti druhů zboží.

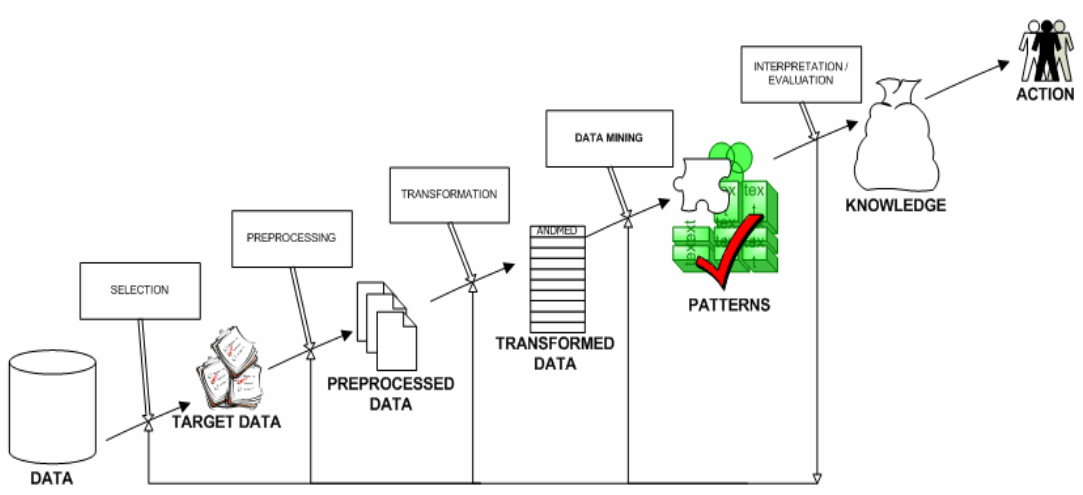
2.1.1 Proces získávání znalosti

Proces získávání znalostí z databází lze charakterizovat jako proces sestávající se z řady kroků, které se zpravidla v určitých iteracích opakují – viz Obrázek. 1.

1. Čištění dat – cílem je vypořádání se s chybějícími daty, odstranění šumu a vyřešení nekonzistence dat.
2. Integrace dat – cílem je integrace dat pocházejících z několika datových zdrojů. Často se integrace a čištění dat provádí společně. Jednak proto, že vyčištěná data

potřebujeme někam ukládat, jednak proto, že jedním ze zdrojů nekonzistence jsou typicky data pocházející z více zdrojů. V takovém případě jsou data jsou data ukládána do datového skladu, jak ukazuje obrázek.

3. Výběr dat – cílem je vybrat data, která jsou pro řešení dané analytické úlohy relevantní. Pokud získáváme znalosti z dat uložených v relační databázi, pracujeme typicky s jednou tabulkou. V tomto kroku tedy vybereme z tabulky relevantní sloupce. V případě datového skladu můžeme analogicky vybírat dimenze.
4. Transformace dat – cílem je transformovat data do konsolidované podoby vhodné pro dolování. Může jít například o sumarizaci nebo agregaci. V případě použití datového skladu pro dolování může transformace předcházet výběru dat, protože může být součástí tvorby datového skladu.
5. Dolování dat – jádro procesu získávání znalostí, jehož cílem je aplikací určité metody a konkrétního algoritmu extrahovat z dat vzory, resp. vytvořit model dat.
6. Hodnocení modelů a vzorů – cílem je identifikovat skutečně zajímavé vzory pomocí mír užitečnosti.
7. Prezentace znalostí – cílem je prezentovat výsledky dolování uživateli využitím technik vizualizace a reprezentace znalostí.



Obrázek 1: Proces získávání znalostí, převzato z [15]

2.1.2 Předzpracování dat

Reálné databáze, zpravidla nejsou v takovém stavu, aby se jejich data dala bezprostředně postoupit dolovacímu algoritmu k dolování. Obsahují velice často data, která jsou zašuměná, nekonzistentní a některé hodnoty mohou chybět. Nízká kvalita vstupních dat by mohla vést k nepřesným nebo

dokonce nesprávným a zavádějícím závěrům vyplývajícím z vydolovaných znalostí. Existuje řada kritérií hodnocení kvality dat. Někdy se hovoří o multidimenzionálním pohledu na kvalitu dat, kdy dimenzemi jsou jednotlivé aspekty kvality, které se hodnotí. Patří mezi ně:

- *přesnost* – data by měla co nejpřesněji modelovat realitu, kterou reprezentují
- *úplnost* – uvažuje se úplnost co do šířky (lze chápat jako dostupnost všech potřebných atributů) a do hloubky (dostatečný počet hodnot)
- *konzistence* – data musí být konzistentní
- *aktuálnost* – data musí být aktuální, pokud mají sloužit k podpoře rozhodnutí týkajícího se budoucnosti
- *důvěra* – věrohodnost dat by měla být co nejvyšší
- *přidaná hodnota* – míra prospěšnosti dat pro řešení dané úlohy
- *interpretovatelnost* – hodnoty by měly být snadno interpretovatelné, například hodnoty kódování intervalů (věku, ceny apod.), klasifikací (spokojenost se službami, schopnost splácet úvěr apod.)
- *dostupnost* – míra udávající, jak snadno jsou data dostupná.

Mezi hlavní úlohy předzpracování dat patří:

- Čištění dat – cílem je odstranit chybějící hodnoty, identifikovat a odstranit odlehlé hodnoty a řešit nekonzistence. Pokud si uživatelé myslí, že data nejsou kvalitní, nebudou důvěřovat ani výsledkům dolování. Navíc nekvalitní data (např. chybějící hodnoty) mohou ovlivnit i samotný dolovací algoritmus, který potom může dávat nespolehlivé výsledky.
- Integrace dat – integrace dat pocházejících z různých datových zdrojů – databází datových kostek datového skladu nebo souborů. Čištění a integrace dat jsou dva kroky předzpracování dat pro datové sklady.
- Transformace dat – transformace dat do tvaru vhodného pro řešení dané dolovací úlohy. Patří sem normalizace a agregace. Agregace je typická pro předzpracování dat pro umístění v datovém skladu.
- Redukce dat – cílem je redukovat objem dat. Patří sem kromě již zmíněné agregace výběr podmnožiny atributů, redukce dimenzionality a redukce počtu hodnot (numerosity). Kromě agregace musí redukce zachovat charakter původního neredukovaného datového souboru.
- Diskretizace dat – patří také mezi metody redukce dat, ale má pro dolování zvláštní význam. Jde o redukci počtu hodnot atributů. Týká se především numerických (spojitých) dat.

2.1.3 Čištění dat

Jedním z údajů součástí souhrnné charakteristiky daného atributu počet chybějících hodnot. Ne každý dolovací algoritmus je schopen se s chybějícími hodnotami vyrovnat, proto se snažíme o odstranění chybějících hodnot. Při čištění dat pro klasifikaci se obvykle používá *Ignorování n-tice* – tento způsob se obvykle použije, pokud je cílový sloupec cílovou třídou. V ostatních případech tato metoda není příliš vhodná s výjimkou situace, kdy v daném řádku chybí hodnoty i v řadě jiných atributů. Způsob, který by mohl dávat dobré výsledky je *Manuální nahrazení*, protože je předpoklad, že uživatel má znalosti, které by mohl při nahrazování uplatnit. V praxi je ale tento přístup s ohledem na velkou časovou náročnost nepoužitelný. Proto se nejčastěji používá *Automatická náhrada.* Tu může představovat například *Globální konstanta.* Jedná se o obdobu NULL v databázích, ale zde je to nějaká skutečná hodnota, např. 0 pro numerický atribut. Pokud by to byla hodnota, která je mimo rozsah „platných" hodnot daného atributu, potom by mohla být ignorována jako odlehlá hodnota.

Pokud by chybějících hodnot bylo hodně, mohla by se takto uměle zavedená hodnota stát pro dolovací algoritmus významnou a zajímavou a mohla by tak negativně ovlivnit výsledek. Lepších výsledků se dosahuje použitím *Průměrné hodnoty atributu* nebo *Průměrem hodnot n-tic patřících do téže třídy* jako daná n-tice. Použitím sofistikovaných metod je možné určit *Nejpravděpodobnější hodnotou.* K jejímu nalezení se využije hodnot jiných atributů v n-tici, tj. řeší se vlastně úloha klasifikace nebo predikce s hledaným atributem jako cílem. Lze využít například regrese, Bayesovské klasifikace nebo rozhodovacího stromu.

2.1.4 Integrace a transformace

Při dolování dat je často třeba pracovat s daty pocházejícími z více zdrojů. V takovém případě je třeba je integrovat, tj. vytvořit z nich jeden koherentní zdroj. Mezi hlavní problémy integrace patří konflikt schématu, který znamená integraci metadat do jedněch metadat popisujících výsledný zdroj dat. K tomu by měla pomoci dostupná metadata. Může docházet také ke konfliktu hodnot. Nastává v případě, že hodnoty odpovídajících si atributů jsou různé. Příčin konfliktů hodnot může být celá řada, například různé formáty (datum, čas, hodnocení).

Cílem integrace je pokusit se detekovat redundanci atributů, kdy se pouze nejedná o výskyt stejných atributů (přesněji se stejným významem) v různých zdrojích dat, ale i o odvozené atributy, jejichž hodnotu lze odvodit z hodnot jiných atributů. Pojem redundance lze z pohledu dolování dat ještě poněkud rozšířit v tom smyslu, že nemusí jít o atribut, jehož hodnotu lze přesně odvodit z hodnot atributů jiných. Může jít o atributy, mezi kterými je silná korelace. Úkolem transformace dat je kromě potřeb plynoucích z integrace dat transformovat data do podoby vhodné pro dolování. Transformace dat typicky zahrnuje vyhlazení dat, tj. odstranění šumu z dat, dále pak agregaci (pro OLAP) a

generalizaci (zdrojová data jsou nahrazena koncepty z konceptuální hierarchie) atributů. Normalizace je proces transformace dat, jejímž cílem je mapování numerických hodnot na specifikovaný interval, typicky <-1.0, 1.0> nebo <0.0, 1.0>. Cílem transformace může být vytvořit nové atributy, které usnadní nebo zkvalitní dolování, jejichž hodnoty jsou odvozeny od atributů jiných. Ty se potom použijí pro dolování místo atributů původních.

2.1.5 Redukce dat

Podstata redukce dat spočívá ve zmenšení objemu dat, se kterým se bude při dolování pracovat. Redukovaná množina musí zachovávat integritu a charakter původního souboru, aby výsledky dolování byly stejné nebo alespoň přibližně stejné. Existuje řada strategií redukce dat.

- Agregace datové kostky – redukce dat agregováním údajů. Typický způsob pro datové sklady
- Výběr podmnožiny atributů – cílem je z množiny atributů, která se skutečně pro dolování použije, vyloučit nerelevantní nebo málo relevantní a redundantní atributy. V angličtině se označuje také jako „feature selection".
- Redukce dimenzionality – data se zakódují takovým způsobem, že dojde k redukci, ale bude možné provádět s daty potřebné operace. Příkladem je použití vlnkové transformace (wavelet transformation) a analýzy hlavních komponent (PCA – Primary Component Analysis). Redukce počtu hodnot (numerosity) – kompletní data jsou nahrazena nějakým modelem a reprezentována jeho parametry nebo jsou reprezentována v nějaké redukované podobě.
- Diskretizace a generace konceptuální hierarchie – hodnoty atributů jsou nahrazeny intervaly nebo pojmy z nějaké konceptuální hierarchie. Jde o speciální případ redukce počtu údajů, kdy se neredukuje počet n-tic, ale počet různých hodnot atributu.

Soubor zdrojových dat pro dolování může zahrnovat stovky atributů, z nichž řada není relevantní pro řešení dané dolovací úlohy. Některé z nich lze vyloučit na základě znalosti jejich sémantiky, případně odhalením redundantních atributů. I tak může zůstat jejich počet stále vysoký. Cílem výběru podmnožiny atributů je nalézt, pokud minimální množinu atributů, která zaručí stejné výsledky, které bychom dostali použitím úplné množiny atributů. Pro klasifikaci, což je typický příklad použití této strategie, bychom mohli cíl zpřesnit jako nalezení minimální množiny atributů takových, že rozdělení pravděpodobnosti různých tříd při použití těchto atributů je co možná nejbližší původnímu rozdělení při použití všech atributů. Má-li původní množina N atributů, zahrnuje prostor řešení 2N možností. Prohledávání celého tohoto prostoru by bylo neúnosně časově náročné, proto se používají heuristické metody, které nezaručují obecně nalezení globálního optima. Základní heuristické metody zahrnují následující techniky:

- Postupný dopředný výběr – procedura začíná s prázdnou množinou výsledku. V každém kroku iterativního postupu je vyhodnocena relevance zbývajících atributů a nejlepší z nich je zařazen do množiny výsledku. Procedura je ukončena na základě nějakého kritéria ukončení.
- Postupná zpětná eliminace – v tomto případě procedura začíná s množinou výsledku obsahující všechny atributy a v každé iteraci naopak vyřazuje nejhorší atribut.
- Kombinace dopředného výběru a zpětné eliminace – kombinace obou předchozích metod. V každém kroku se vybere nejlepší atribut a současně vyřadí atribut nejhorší.
- Indukce rozhodovacího stromu – algoritmy konstrukce rozhodovacího stromu, jako je ID3 a C4.5 používají kritéria pro výběr atributů do uzlů stromu, které odpovídají i kritériím pro výběr atributů. Je-li takový algoritmus použit pro dopředný výběr atributů, zkonstruuje se rozhodovací strom a výslednou množinu atributů budou tvořit pouze ty, které se vyskytují ve stromu.

Metody redukce počtu hodnot můžeme rozdělit do dvou skupin:

- Parametrické – reprezentují data nějakým modelem dat a data jsou pak reprezentována parametry tohoto modelu.
- Neparametrické modely – reprezentují data nějakou redukovanou reprezentací.

Příkladem parametrických metod může být použití regrese, kdy například v případě lineární regrese jsou data reprezentována parametry regresní přímky. Mezi neparametrické metody patří použití histogramů, shlukování a vzorkování. V případě shlukování jsou vytvořeny shluky dat a data potom mohou být reprezentována reprezentanty těchto shluků, například těžištěm shluku, průměrem shluku a počtem hodnot ve shluku. Podstata vzorkování spočívá v redukci dat výběrem nějakého reprezentativního vzorku. Jestliže originální soubor dat obsahuje N n-tic, potom soubor vzniklý vzorkováním bude obsahovat s n-tic, kde $s < N$.

3 Jazyk XML

Jazyk XML (Extensible Markup Language) byl standardizován konsorciem W3C a jeho základem se stala nejpoužívanější podmnožina jazyka SGML, která byla dále omezena pevně určenými parametry a přísnou syntaxí. Je určen především pro ukládání, předávání a publikování semi-strukturovaných dat. Jedná se ve své podstatě o metajazyk, což znamená, že je určen pro popis struktury dalších jazyků.

3.1 Elementy

Základními prvky jazyka XML jsou elementy, které jsou do sebe vzájemně vnořovány. Elementy jsou určeny pomocí značek, přičemž většině elementů odpovídají dvě značky - počáteční a koncová. Výjimkou je prázdný element, který je určen pouze jednou značkou. Značky jsou uzavřeny mezi znaky < a >, koncová značka má navíc na začátku znak /, značka pro prázdný element má znak / na konci.

Každý neprázdný element musí být uzavřen mezi obě značky a toto uzavření musí být správně uzávorkované (tj. značky se nesmějí křížit). Dále musí být celý XML dokument uzavřen v právě jednom kořenovém elementu. Splňuje-li XML dokument tato základní syntaktická pravidla, říkáme, že je správně strukturovaný (well-formed). Názvy, přípustný obsah a vzájemné vztahy elementů je možné definovat XML schématem popsaný v jazycích jako je DTD, XML Schema apod.

3.2 Atributy

Dalšími prvky XML dokumentu jsou atributy. Jsou součástí počáteční značky elementu a obsahují dodatečné informace vztahující se k danému elementu. Každý atribut má dvě části - název a hodnotu uzavřenou do uvozovek, která se odděluje znakem =. Názvy a datové typy atributů je také možné definovat prostřednictvím XML schématu.

3.3 XML deklarace

Na začátku XML dokumentu by měla být umístěna tzv. XML deklarace. Jedná se o element obsahující informaci o použité verzi XML (version), znakové sadě (encoding) a informaci o tom, zda je pro správnou interpretaci obsahu nutné použít externí definované deklarace značek (standalone). Např:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

4 Jazyk DTD

DTD (definice typu dokumentu) je nepovinnou součástí XML dokumentů, ale jeho používání přináší dost podstatné výhody a ulehčuje práci. DTD je šablona, podle které se tvoří a kontrolují XML dokumenty. Existuje mnoho standardních DTD požívaných v určitých oborech např. ve zdravotnictví, finančnictví atd. Asi nejznámějším DTD je DocBook, který definuje elementy a atributy vhodné pro značkování technické dokumentace. Další výhodu přináší DTD programátorům aplikací, které zpracovávají XML dokumenty. Programátor se nemusí obávat, že program narazí na neočekávaný vstup, se kterým si nedokáže poradit. Dokument nevyhovující požadovanému DTD bude jednoduše vyřazen ze zpracování. Tvůrcům dokumentů přináší použití DTD možnost kontroly správné struktury dokumentu pomocí parseru.

Použití DTD je podmíněno jeho deklarací. Deklarace typu dokumentu (DOCTYPE) se umísťuje na začátek dokumentu hned za XML deklaraci. Deklarace může mít několik tvarů a to podle toho, jak je DTD s dokumentem sloučeno.

1. TD je přímo součástí XML dokumentu. V tom případě bude deklarace vypadat takto:

```
<!DOCTYPE "root element" [
<!-- DTD -->
<! ... >
<! ... >
]>
```

V prvním řádku je uvedeno, že se jedná o DTD dokumentu (DOCTYPE). Za ním je uveden název kořenového elementu, což je element, ve kterém je obsažen celý dokument XML. Název elementu se píše bez uvedených uvozovek. Po této deklaraci následuje samotné DTD.

2. Umístění DTD do externího souboru s příponou .dtd. Takto použité DTD je oproti předchozímu použití daleko výhodnější. Pokud provádíme v DTD změny, stačí změnit jeden soubor a nemusíme změny provádět ve všech ostatních dokumentech. Deklarace při použití tohoto způsobu vypadá následovně:

```
<!DOCTYPE "root element" SYSTEM "URL file with DTD" [
<!—Here can be specific DTD for this document-->
<! ... >
<! ... >
]>
```

Umístění DTD je dáno systémovým identifikátorem, který je určen klíčovým slovem SYSTEM. Za slovem SYSTEM je v uvozovkách uvedena URL adresa dokumentu obsahujícího DTD.

Př. : XML dokument s definovaným interním a externím DTD.

```
<?xml version="1.0" encoding="windows-1250"?>
<!DOCTYPE root_element SYSTEM "..dtd/dtd_document.dtd" [
<!—DTD specific for this document -->
<! ... >
<! ... >
]>

<root_element>
 others elements
</root_element>
```

4.1.1 Elementy

Elementy jsou základem XML dokumentů a DTD určuje jaké elementy mohou být v daném dokumentu použity a co mohou obsahovat. Obecná deklarace elementu v DTD vypadá takto

```
<!ELEMENT element-name (element-content)>
```

Obsah elementu může být definován následovně:

<!ELEMENT element-name (EMPTY)>

<!ELEMENT element-name (#CDATA)>

<!ELEMENT element-name (#PCDATA)>

<!ELEMENT element-name (ANY)>

Klíčové slovo EMPTY určuje, že se jedná o prázdný element a tedy, že nemůže obsahovat text ani žádné jiné elementy. Na rozdíl od prázdných elementů můžeme však mít elementy, které mohou obsahovat úplně všechno. Takový element se deklaruje pomocí klíčového slova ANY. Deklarace pomocí klíčového slova ANY se však moc nepoužívá, protože příliš uvolňuje strukturu dokumentu a to je proti samotné myšlence XML. Dále může element obsahovat samotný text, to se v DTD vyjádří pomocí klíčového slova #PCDATA. Podobným klíčovým slovem je #CDATA. Odlišnost je však v tom, #CDATA nejsou na rozdíl od #PCDATA zpracovávána parserem.

Nejčastější případ je ten, že element obsahuje další elementy. V tom případě se použije pro deklaraci tzv. modelová skupina (model group). Modelová skupina je vždy uzavřena do kulatých závorek a obsahuje alespoň jedno slovo (tím je nejčastěji jméno elementu, který může být obsažen v právě deklarovaném elementu). Vnořené elementy lze kombinovat pomocí znaků "," a "|". Pokud jsou elementy odděleny čárkou, musí následovat v pořadí, v jakém jsou zapsány.

Deklarace modelové skupiny:

<!ELEMENT element-name (child-element-name)>

<!ELEMENT element-name (child-element-name,child-element-name,.....)>

Kromě pořadí elementů musíme určit také jejich počet, zda jsou povinné nebo zda se mohou opakovat. Jestliže v modelové skupině uvedeme pouze jméno elementu, musí být přítomen právě jednou. Za jménem elementu je možné uvést násobnost výskytu pomocí následujících znaků. Znak ? vyjadřuje nepovinnost elementu(0 až 1), + nejméně jeden výsky elementu, * žádný nebo několik výskytů elementu

4.1.2 Atributy

Elementy v XML dokumentu mohou mít libovolné množství atributů, které se používají především pro připojení různých metainformací k elementům nebo k tvorbě odkazů atd. Obecná deklarace atributu elementu vypadá následovně:

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```

Klíčové slovo ATTLIST určuje, že se jedná o deklaraci atributu. Za jméno elementu se dosadí název elementu pro který atributy definujeme. Samotná deklarace atributu se skládá ze tří částí. První částí je jméno atributu. Pro vytváření jmen atributů platí stejná omezení jako pro elementy. Za jménem následuje typ atributu, který může nabývat těchto hodnot:

- CDATA – množinou hodnot jsou jakákoliv data. Pokud by náhodou obsahovala značky, nebudou rozpoznávány.
- IDREF, IDREFS – jde o odkaz, odkazy (reference) na jiný prvek s atributem ID
- NMTOKEN, NMTOKENS – jde o jméno, jména, tedy víceméně o řetězec alfanumerických znaků
- ENTITY, ENTITIES – entity, seznam entit
- NOTATION – jméno notace
- xml: - hodnota je předdefinovaná

Poslední částí deklarace atributu je specifikace, jak se má analyzátor chovat, pokud příslušná hodnota atributu není zadána. Jsou definovány následující možnosti

- #REQUIRED – hodnota je povinná
- #IMPLIED – hodnota nemusí být zadána
- #CURRENT – vezme se poslední zadaná hodnota tohoto atributu
- *hodnota* – je zadána konkrétní implicitní hodnota

4.1.3 Entity

XML umožňuje informace obsažené v dokumentu rozdělit na menší části, kterým se říká entity. Každá entita má své jméno, pomocí kterého může být jednoznačně identifikována. XML podporuje několik druhů entit, které se liší svými vlastnostmi. Entity mohou obsahovat buď data ve formátu XML, nebo v jiném formátu. Podle toho je dělíme na entity textové (formát XML) a entity binární (ostatní datové formáty). Dále můžeme entity rozlišovat podle toho, zda jsou uloženy přímo v hlavím dokumentu nebo v externím souboru. Podle tohoto rozlišení máme entity interní textové, externí textové a externí binární. Entita je definována v DTD a do dokumentu se vkládá pomocí odkazů na požadovanou entitu. V textu může být použito několik odkazů na stejnou entitu a její obsah nahradí každý výskyt odkazu. U deklarací entit je jedno, zda jsou umístěny v lokálním či externím DTD dokumentu. Obecná deklarace vypadá následovně:

<!ENTITY entity-name "entity-value">

Takto definovanou entitu pak můžeme v textu použít pomocí odkazu & "entity_name".

Externí entitu definujeme následovně:

```
<!ENTITY entity-name SYSTEM "URI/URL">
```

5 Dolování v XML

5.1 Dnešní stav

Dolování v XML bylo až donedávna neprozkoumanou oblastí a dalo by se říci, že tomu tak stále ještě je. XML se často zpracovává tak, že se provede extrakce vlastních dat na základě potřeb a znalosti konkrétního dialektu. Získá se tak jediná tabulka, která se předloží propozičnímu učícímu se systému. Hovoříme pak o tzv. propozicionalizaci. Převod dat se však může do jisté míry provést automaticky, např. pomocí systému pro multirelační dolování znalostí. Vzhledem ke stále rostoucímu počtu XML dokumentu však roste potřeba metod, které pracují přímo s XML dokumenty a využívají všech informací, které jsou v nich obsaženy – tedy i těch o struktuře.

5.2 Nové metody

Pro dolování ve struktuře XML navrhl Termier a kol. [7] algoritmus TreeFinder, který hledá pomocí upraveného algoritmu Apriori [1] nejčastěji se vyskytující nespecifičtější podstromy. Ve výsledném podstromu jsou zachovány tranzitivní vztahy předek–potomek ale ne nutně rodič–dítě. Algoritmus nejprve z každého vstupního XML dokumentu vytvoří transakci, která obsahuje všechny možné kombinace (element–element). Elementy jsou přitom bud ve vztahu rodič–dítě a nebo předek–potomek. V této množině se pomocí AprioriTree naleznou nejčastější relace a z nich se znovu sestaví nejméně obecný strom. Zaki a kol. [9] navrhl metodu XRules pro klasifikaci XML dokumentu. Vychází z algoritmu TreeFinder, na jehož základě je vytvořen algoritmus Xminer, pro hledání tzv. klasifikačních asociačních pravidel (asociační pravidlo s identifikátorem třídy v závěru) v XML dokumentech. Z pravidel, která jsou splněna pro klasifikovanou instanci, se vybírá podmnožina se shodnou třídou a nejvyšším kombinovaným efektem (pokrytí, spolehlivost a korelace). Nevýhodou tohoto přístupu pak muže být fakt, že celá klasifikace je prováděna pouze na základě struktury XML Proto Theobald a kol. vyvinul metodu převodu XML na propoziční data, která využívá jak části strukturní informace, tak dat uložených v dokumentu. Nové rysy jsou vytvářeny jako kombinace vlastních dat a základní strukturní informace: atribut–hodnota, element–atribut, element–termín, nebo využívají pouze strukturu: element–element (rodič–dítě), element–element–element (levé dítě–rodič–pravé dítě). Problém různého pojmenování elementu v dokumentech z různých zdrojů byl řešen mapováním ontologií. Metodu úspěšně použili na klasifikaci dat z IMDb.

5.2.1 Klasifikace

Klasifikace je jednou z nejčastěji používaných úloh dolování znalostí. Uplatňuje se totiž v mnoha doménách, např. při filtrování spamu (klasifikace dokumentu). Většinu stávajících systému strojového učení však nelze použít přímo na data v XML. Mohl by se sice použít některý systém pro relační dolování, ty ale vyžadují definici doménové znalosti a jejich použití je tak dost náročné. Nejjednodušší by bylo informaci o struktuře úplně pominout a klasifikovat data jako prostý text. Tím se však připravíme o informace, které mohou pomoci při analýze dat. Vhodnější je proto transformovat data na jedinou tabulku takovou metodou, která zachová alespoň část strukturní informace. Dokumenty jsou tak nejprve převedeny do jediné tabulky, ze které jsou poté vybrány pouze významné rysy.

6 Hybrid Tree Miner

6.1 Vymezení základních pojmů

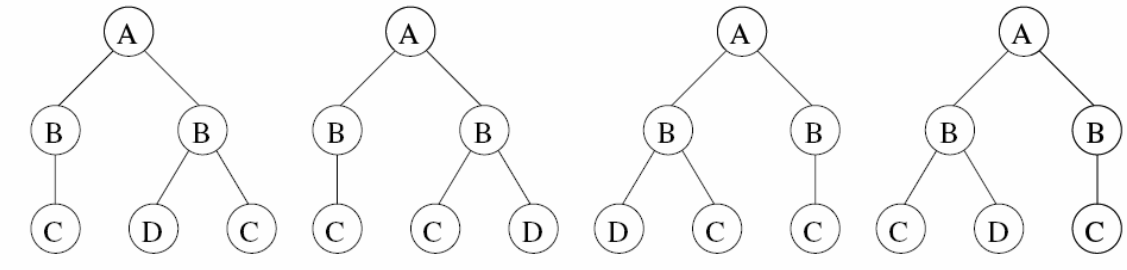
V této sekci se zaměřím na definování pojmů, které jsou pak dále použity. Označený graf $G = [V, E, \Sigma, L]$ sestává z množiny vrcholů V, množiny hran E, abecedy $\sum$ pro vrcholy a hrany, funkce L takové, že $V \cup L \rightarrow \Sigma$. Funkce L přiřazuje označení hranám a vrcholům. *Volný strom* je neorientovaný, propojený a acyklický graf. *Kořenový strom* je volný strom, který obsahuje vybraný vrchol, který se nazývá kořen. V kořenovém stromu, jestliže vrchol *v* je na cestě z kořene do vrcholu *w*, potom vrchol *v* je předek *w* a *v* je následník vrcholu *w* .Dále pak, pokud vrcholy *v* a *w* jsou sousedící, potom *v* je otcem *w* a *w* je synem *v*. *Kořenový uspořádaný strom* je kořenový strom, který má definované pořadí levého a pravého syna každého vrcholu. Označený volný strom *t* je podstrom jiného volného stromu *s*, pokud může být t získán s opakovaným odstraňováním vrcholů stupně. Podstrom kořenového stromu je definován podobně. Dva *volné stromy t* a *s* jsou vzájemně isomorfní, pokud existuje jednoznačné mapování t na mapování vrcholů s, které zachovává označení vrcholů, hran a sousednosti. Isomorfismus pro kořenové stromy je definován podobně, jen mapování zachovává kořen. Automorfismus je isomorfismus, který mapuje strom sám na sebe.

Nechť D označuje databázi, kde každá transakce s∈ D je *kořenový uspořádaný strom* (nebo D je databáze *volných stromů*). Říkáme, že *t*(strom) je vzor, pokud se vyskytuje v transakci *s*, tj. v transakci *s* existuje nejméně jeden isomorfní strom k *t*. *Podpora* vzoru t je část v databázi D, která podporuje *t*. Vzor t je nazván frekventovaný, pokud jeho hodnota je rovna nebo větší, než hodnota definovaná uživatelem. Dolování frekventovaných stromů je tak problém hledání všech frekventovaných stromů obsažených v databázi.

6.2 Kanonická forma

Z kořenového neorientovaného stromu lze odvodit mnoho orientovaných kořenových stromů, viz. Obr. 2. Nejprve definujeme breadth-first string (BFS) kódování pro kořenový uspořádaný strom. Předpokládejme, že existují dva speciální symboly $ a #, které nejsou v abecedě vrcholů a hran. BFS kódování kořenového uspořádaného stromu je dáno průchodem stromu do šířky, úroveň po úrovni. Průchod zaznamenává jména vrcholů, symbol $ je použit jako oddělovač mezi rodinami sourozenců a symbol # označuje konec posloupnosti. Dále předpokládáme, že existuje úplné uspořádání na

množině hran a vrcholů, dále předpokládáme, že # > $ a zároveň jsou oba tyto symboly větší než jakýkoli symbol z množiny hran nebo vrcholů. Pro neuspořádaný kořenový strom můžeme vytvořit několik různých uspořádaných stromů a k nim korespondující BFS. *Breadth-first canonical string* (BFCS) kořenového neuspořádaného stromu je definován jako minimální BFS, na základě lexikálního uspořádání.



Obrázek 2 4 kořenové uspořádané stromy odvozené z neuspořádaného stromu, převzato z [6]

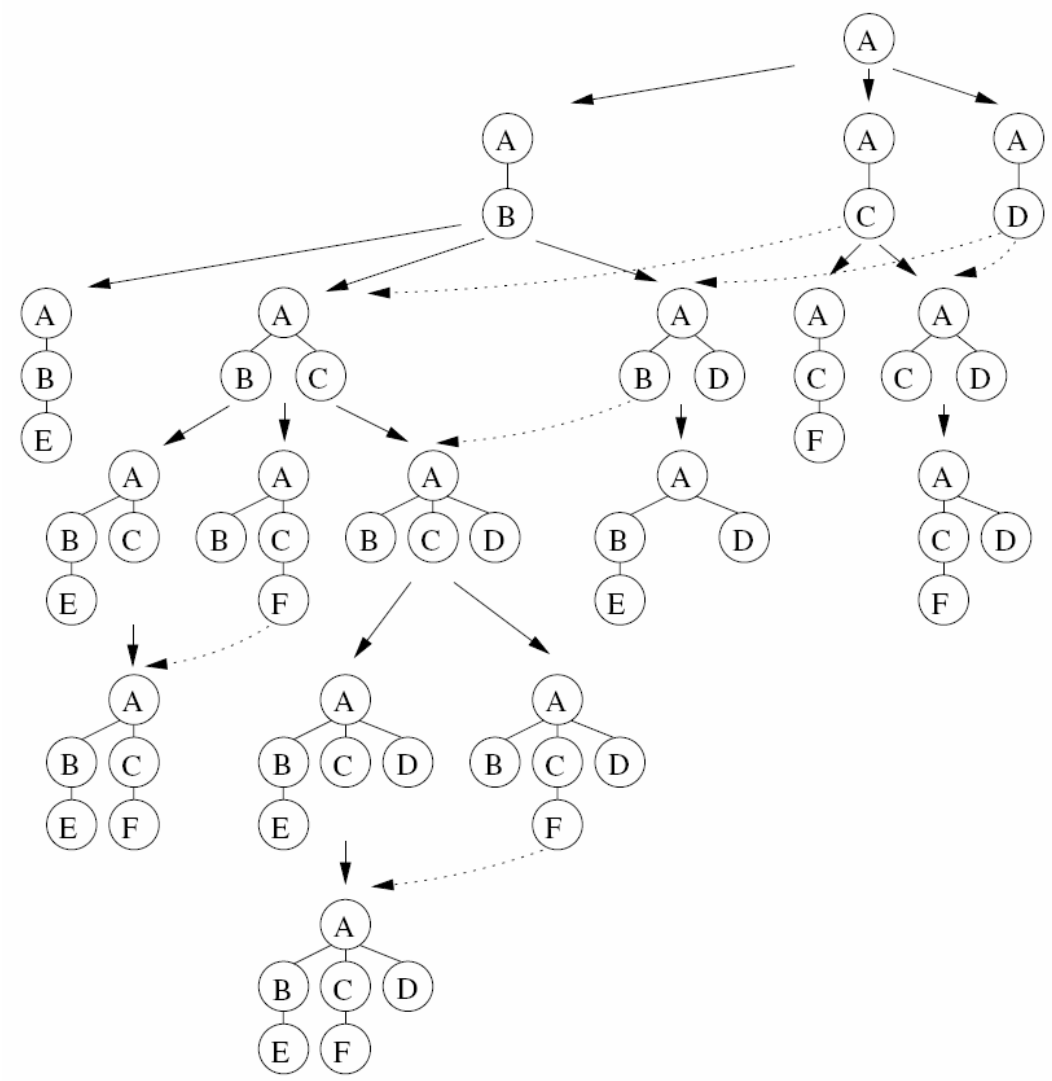
Pro stromy na obrázku lze psát následující BFS. Pro a) ABBC$DC#, b) A$BBCCD#
c) ABBDC$C# d) A$BBCDC#. Dle lexikálního uspořádání je jako BFCS vybrán řetězec d) tj. ABBCD$C#. Existují i jiné metody definování kanonické formy neupořádaného kořenového stromu, jejichž bližší popis lze nalézt v [8] nebo v [9].

6.3 Enumeration Tree

Nyní definujme pojem Enumeration Tree (dále jen ET). Zaveďme nejprve konvenci, že list stromu (společně s jeho předkem), který je na nejnižší úrovni BFCF formy stromu, budeme nazývat *rameno*. Rameno, které je na nejnižší úrovni stromu a je nejpravější, nazveme *poslední rameno*. Následující lema definuje základní myšlenku ET.

Lema: Odstraněním posledního ramena (tj. nejpravějšího ramena) z BFCF kořenového neuspořádaného stromu výšky (k+1), obdržíme BFCF jiného kořenového neuspořádaného stromu, který bude výšky k.

Na základě tohoto lema lze budovat ET, ve kterém uzly představují kořenové neuspořádané stromy v jejich BFCF. Otec každého tohoto stromu je vytvořen odstraněním posledního ramene z příslušného BFCF.



Obrázek 3: Příklad vytváření Enumeration Tree, převzato z [6]

6.4 Operace nad Enumeration Tree

Cílem algoritmu je efektivní vybudování ET. Pro návrh algoritmu na vybudování ET stromu je převzata metoda představená prostřednictvím Huan[6] a kol, která dovoluje speciální logické spojení stromů. Pokud se podíváme na obrázek 3, můžeme vidět, že potomci vrcholu *v* mohou být vytvořeni dvěma metodami. A to takzvaným **rozšířením**, tj. přidáním ramene do nejnižší úrovně stromu. Druhou metodou je takzvané **spojení,** které probíhá mezi sourozenci. Tím vznikne strom (BFCF) opět výšky h. Jsou tedy definovány dva způsoby pro odvozování potomků vrcholu *v*. Při budování ET, se však operace rozšířením provede jen tehdy, pokud vzroste výška stromu.

Definice rozšíření: Pro uzel *v* ET stromu nazýváme jeho BFCT t_v a přiřazujeme mu výšku h. Operaci rozšíření aplikujeme pro vznik nového stromu $t_{v'}$ který má výšku h+1 a má nové rameno.

Definice spojení: Předpokládejme dva sourozenecké uzly v_1 a v_2, které mají společného předka v a t_{v1} reprezentuje BFCF v_1 a t_{v2} reprezentuje v_2, stromy mají výšku h. Dále předpokládejme, že $t_{v1} <= t_{v2}$ (dle lexikálního uspořádání). Operace spojení aplikovaná na v_1 a na v_2 vytvoří nový uzel (strom) v'_1 (BFCF $t_{v'_1}$) je potomkem v_1 a má výšku h. $t_{v'_1}$ vznikne z t_{v1} přidání posledního (nejpravějšího) ramene z t_{v2}.

6.4.1 Výpočet podpory

Pro výpočet podpory listů v neuspořádaném stromu se zavadí seznam výskytů L_{tv}. Tento seznam obsahuje záznamy o každém výskytu uzlu t_v v databázi transakcí. Každý element $l \in L_{tv}$ je struktura tvaru $l = (tid, i_1 ... i_k)$, kde *tid* je id transakce která obsahuje příslušný t_v. $i_1 ... i_k$ reprezentuje mapování vrcholů transakce na vrcholy stromu t_v. Ze seznamu výskytů lze určit, zda je t_v frekventovaný nebo zda není. Podpora je součet různých id transakcí v příslušném t_v .Operace spojení kombinuje dva seznamy výskytů L_{t1} a L_{t2}, z nich se vytvoří nový seznam výskytů L_{t12}. Kombinace probíhá mezi elementy $l_1 = (tid_1, i_1 .. i_k), l_2 = (tid_2, j_1 .. j_k)$ seznamů $l_1 \in L_{t1}, l_2 \in L_{t2}$, pokud $tid_1 = tid_2$ a $i_m = j_m$, pro $m = 1.., k-1$. Výsledkem kombinace l_1, l_2 je $l_{12} = (tid_1, i_1 ..., i_k, j_k) \in L_{t12}$. Pro operaci rozšíření předpokládejme, že l je prvkem L_{tv} a má strom t_v výšky k. l lze rozšířit na l' a vznikne tak nový prvek seznamu výskytů pro $L_{tv'}$, kde $t_{v'}$ je strom výšky $k+1$ a $t_{v'}$ je potomkem t_v. Nechť $l = (tid, i_1 ... i_k)$, potom l může být rozšířen na $l' = (tid, i_1 ... i_k, i_{k+1})$, právě tehdy, pokud platí, že $i_1 ... i_k, i_{k+1}$ je validní mapování mezi vrcholy náležející k $t_{v'}$ a transakci tid a zároveň platí, že $i_{k+1} \neq i_m \; pro \; m = 1..., k$.

6.4.2 Algoritmus HybridTreeMiner

Hlavní část algoritmu je funkce Enum-Grow, která vybuduje celý ET strom. V algoritmu jsou obsaženy dvě operace a to spojení a rozšíření, které jsou vykonávány odděleně. Dá se ukázat, že složitost algoritmu je $O|F|.(hk^2 + |D|kc)$, kde F je množina všech frekventovaných podstromů, *D* je velikost databáze, *h* je maximální výška, *k* je maximální velikost frekventovaného podstromu a *c* je maximální velikost transakce v databázi.

```
HybridTreeMiner(D,minsup)
      F1, F2 ← {frequent 1, and 2-trees};
      F ← F1 ∪ F2;
      C ← sort(F2);
      Enum-Grow(C,F,minsup);
      return F;
end;

Enum-Grow(C,F,minsup)
      for i ← 1, . . . , |C| do
            J ← ∅;
            for j ← i, . . . , |C| do
                  p ← join(ci, cj);
                  if supp(p) ≥ minsup then J ← J ∪ p;
            F ← F ∪ J;
      Enum-Grow(J,F,minsup);
      E ← ∅;
      for each leg lm of ci do
            for each possible new leg ln do
                  q ← ci plus leg ln at position lm;
                  if supp(q) ≥ minsup then E ← E ∪ q;
            F ← F ∪ E;
      Enum-Grow(E,F,minsup);
end;
```

6.5 Isomorfismus grafů

Izomorfizmus mezi grafy může být neformálně chápán jako shoda jejich struktury, respektive jejich identita až na označení. Obecný izomorfizmus grafů spadá do tzv. NP problémů, u kterých není známo řešení v polynomiálním čase. Proto je nutné nalézt algoritmy, které dokáží částečně obejít jeho velkou časovou a paměťovou náročnost. Mnoho grafů se liší pouze způsobem kreslení a zejména označením svých vrcholů a hran, což vystihuje pojem izomorfizmus Grafy $G=(V,E)$ a $G^{'}=(V^{'},E^{'})$ jsou **izomorfní**, zapisujeme $G \cong G^{'}$, pokud existuje bijektivní zobrazení, tj. isomorfismus:

$\phi(V \to V^{'})$ takové, že platí $(v_i,v_j) \in E \Leftrightarrow (\varphi(v_i),\phi(v_j)) \in E^{'}$

zobrazení tedy zachovává vrcholy i hrany dané incidenční relací z definice.

Podgraf je graf, který je izomorfní s nějakou částí nadgrafu, tedy pro φ je postačující injektivní zobrazení, respektive:

Říkáme, že graf $G' = (V', E')$ je podgrafem grafu $G = (V, E)$, pokud platí:

$V' \subseteq V \quad a \quad E' \subseteq E$

Izomorfizmus grafů je označován jako GI problém. Jeho definice je velmi jednoduchá, ale jeho řešení spadá do třídy NP problému. Nejjednodušším způsobem zjištění izomorfizmu je užitím hrubé síly, řešením je permutace všech vrcholů V a V‘, složitost tohoto přístupu roste exponenciálně s počtem vrcholů, což není v současné době akceptovatelné. Naštěstí ale existuje několik technik, které dokáží tuto složitost značně zlepšit, jak je popsáno níže. Na druhou ale stranu ale nebylo prokázáno, že lze GI problém zařadit do nějaké praktické třídy, jako P, RP nebo BPP. Proto bylo vyvinuto několik technik pro řešení tohoto problému, například Ullmanův algoritmus nebo využití kanonické formy. Podrobnější informace lze nalézt například v [10].

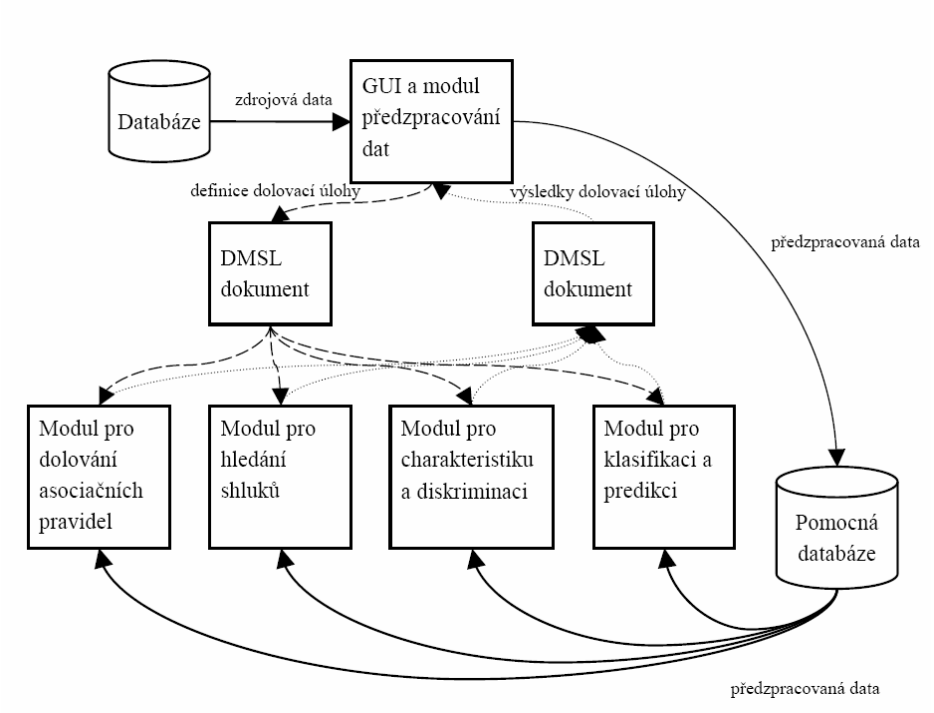
7 Systém pro získávání znalostí

Systém pro získávání znalostí vyvíjený na Fakultě informačních technologií Vysokého učení technického je navržen tak, aby byl použitelný na různých platformách a měl modulární architekturu. Je psán v jazyce Java, což zaručuje například tyto výhody: Přenositelnost mezi platformami, jednoduchost a snadné programování. Komunikace s databází je řešena pomocí technologie JDBC (Java Database Connectivity). To umožňuje programátorovi využít jednotné rozhraní pro přístup do libovolné databáze, která poskytuje JDBC ovladač. V dnešní době jsou to prakticky všechny hlavní databázové systémy. Ovladače jsou vyvíjené a optimalizované samotnými výrobci databázových strojů. Pro uložení dat byla vybrána databáze MySQL, což je jednoduchá relační databáze, volně dostupná pro nekomerční využití podle licence GPL (GNU General Public License).

Komunikace mezi moduly systému je zajištěna pomocí jazyka DMSL, který umožňuje výměnu strukturovaných informací.

7.1 Struktura systému

Systém se skládá ze základního modulu, ve kterém je implementováno grafické uživatelské rozhraní (GUI) spolu s předzpracováním dat, a dalších modulů, které jsou zaměřeny na konkrétní dolovací úlohy. Systém pracuje tak, že uživatel pomocí GUI definuje, nad jakými daty a jak má proces získávání znalostí probíhat. Poté základní modul získá data ze zdrojové databáze, provede na nich předzpracování a uloží je do pomocné databáze. Potom předá definici dolovací úlohy ve formátu DMSL jednomu z dolovacích modulů. Ten na základě informací předaných v DMSL dokumentu provede nad daty v pomocné databázi dolovací úlohu. Po jejím úspěšném dokončení uloží dolovací modul získané informace opět do DMSL dokumentu a předá ho zpět základnímu modulu. Ten pomocí GUI provede vizuální prezentaci výsledků, tedy vlastně získaných znalostí. Tento děj je graficky znázorněn na obrázku.



Obrázek 4: Koncepce systému, převzato z [14]

7.2 Jazyk DMSL

Pro popis procesu získávání znalostí z databází již bylo vyvinuto několik jazyků, ovšem tyto jazyky se většinou zaměřovaly jen na určité kroky tohoto procesu (hlavně na samotné dolování dat), některé byly přímo závislé na podpoře jazyka SQL (Structured Query Language) (především při předzpracování dat), či byly implementovány pouze pro určitou platformu.

Jazyk DMSL (Data Mining Specification Language), byl tedy navržen z důvodu potřeby standardizace jazyka pro popis celého procesu získávání dat z databází, který by byl nezávislý na platformě. Jazyk DMSL je založen na standardu XML.

DMSL se tedy snaží pokrýt celý proces získávání dat z databází, přičemž se zaměřuje hlavně na předzpracování dat. Proto také nemají elementy jazyka popisující dolování dat a následující kroky přesně určenou syntaxi. Jejich definice se provádí pomocí jiných jazyků, například jazykem PMML (Predictive Model Markup Language), popsaným v [14], nebo pomocí postupně definovaných rozšíření jazyka DMSL

7.3 Struktura jazyka DMSL

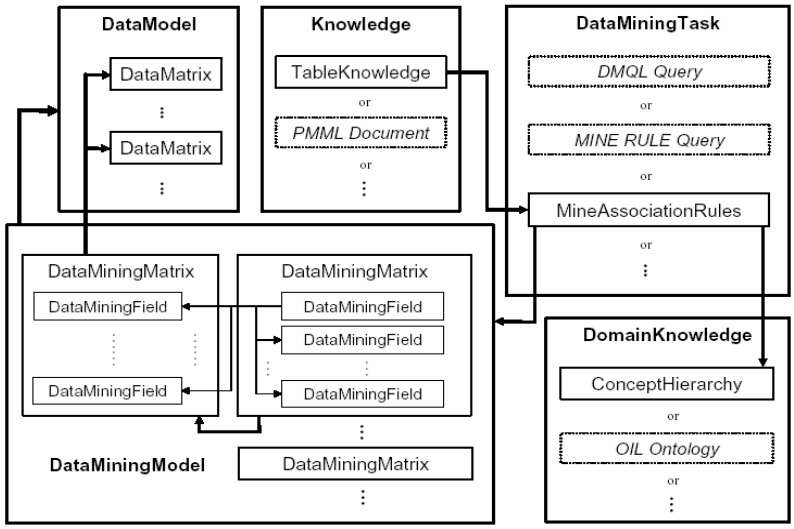
Pomocí DTD lze strukturu jazyka DMSL popsat následujícím způsobem:

```
<!ELEMENT DMSL (Header?, (FunctionPool | DataModel | DataMiningModel
|DomainKnowledge |  DataMiningTask | Knowledge)+) >
```

Nejvyšším elementem jazyka DMSL je element DMSL, sloužící jako rozcestník k dalším elementům. Těmito elementy jsou:

- *Header* – slouží pro popis obecných vlastností DMSL dokumentu, například název aplikace, která daný dokument vyprodukovala, verzi aplikace, datum vytvoření dokumentu, jméno autora, operační systém, ve kterém vznikl daný dokument atd.
- *FunctionPool* – slouží pro popis funkcí použitých v projektu.
- *DataModel* – reprezentuje schéma vstupních dat.
- *DataMiningModel* – reprezentuje schéma dat, transformovaných do podoby potřebné pro dolování dat.
- *DomainKnowledge* – reprezentuje znalosti o datech, ty mohou být využity dolovací úlohou.
- *DataMiningTask* – popisuje úlohu dolování dat.
- *Knowledge* – reprezentuje znalosti získané dolovací úlohou.

Vztahy mezi hlavními primitivy (DataModel, DataMiningModel, DomainKnowledge, DataMiningTask , Knowledge) jsou uvedeny na obrázku.



Obrázek 4.1 Vztah základních elementů v DMSL

Syntaktická a sémantická omezení:

- Každý DMSL element může obsahovat jeden nebo žádný element *Header*.
- Každý DMSL element může obsahovat libovolný nenulový počet libovolně zkombinovaných elementů: *FunctionPool*, *DataModel*, *DataMiningModel*, *DomainKnowledge*, *DataMiningTask* a *Knowledge*.

7.4 Rozšíření jazyka DMSL

Jak bylo uvedeno výše, jazyk DMSL nemá ve své specifikaci přesně definovány elementy pro popis dolovací úlohy (*DataMiningTask*) a pro reprezentaci získaných znalostí (*Knowledge*). Protože základní modul systému pro získávání informací komunikuje s dolovacím modulem pomocí DMSL, je třeba tyto elementy jazyka DMSL dodefinovat. Jako předlohu pro rozšíření jsem použil jazyk PMML [14]. Stávající systém neobsahuje podporu pro data v podobě XML, proto jsem se zaměřil na tuto problematiku a dodefinoval podporu jak pro XML data, tak pro specifikaci parametrů pro algoritmu.

7.4.1 Element DataMiningTask

V DMSL je element pro popis dolovací úlohy definován takto:

```
<!ELEMENT DataMiningTask ANY >
<!ATTLIST DataMiningTask
     name                 CDATA                          #REQUIRED
     type                 CDATA                          #IMPLIED
     dataMiningModelRef   CDATA                          #IMPLIED
>
```

Význam atributů:

- **name** – název dolovací úlohy,
- **type** – typ dolovací úlohy, v tomto případě *hybridTreeMiner,*
- **dataMiningModelRef** – jméno dolovacího modelu dataMiningModel, použitého dolovací úlohou pro získávání znalostí.

Eelement HybridTree je základním elementem pro algoritmus hybridTreeMiner. Obsahuje základní údaje, které jsou pro něj potřebné.

```
<!ELEMENT HybridTree EMPTY>
<!ATTLIST HybridTree
     support              %INT-NUMBER                    #REQUIRED
     input                %INT-NUMBER                    #REQUIRED
     output               %INT-NUMBER                    #REQUIRED
     rootElement          CDATA                          # IMPLIED
     property             CDATA                          # IMPLIED
>
```

Význam atributů:

- **support** – nastavená minimální podpora pro hledané vzory
- **input** – vstupní soubor s daty
- **output** – výstupní soubor s frekventovanými vzory
- **rootElement -** lze specifikovat kořenový element vstupního souboru
- **propery –** zle specifikovat na kterou vlastnost se dolovací algoritmus zaměří

7.5 Dolovací moduly

Pro systém již bylo vytvořeno několik modulů, které implementují tyto dolovací úlohy:

- Popis pojmů (*concept description*) – skládá se ze dvou úloh, charakterizace (*characterization*) a porovnání (*discrimination*). Obě úlohy jsou téměř totožné, liší se jen výběrem dat z databáze. Základem tohoto modulu je třída *MineCD*, která obsahuje pouze vytvoření instance třídy *Concept*. Ta aktivuje vlastní dolování popisu pojmů. Třída *Concept* obsahuje metody pro načtení DMSL dokumentu, jeho rozlišení a provedení dolování charakteristiky nebo porovnání a nakonec vytvoření výstupního DMSL dokumentu.
- Klasifikace (*classification*) – základem tohoto modulu je třída *MineClassification*, která obsahuje jen vytvoření instance třídy *Classification*. Ta aktivuje vlastní dolování klasifikace. V této třídě jsou opět metody pro načtení DMSL dokumentu, generování rozhodovacího stromu pomocí ID3 algoritmu a vytvoření výstupního DMSL dokumentu.
- Asociační analýza (*associations*) – základem tohoto modulu je třída *Miner*, která spouští jednotlivé kroky procesu získávání asociačních pravidel.

7.5.1 Přidání dolovacího modulu do systému pro získávání znalostí

Popis začlenění dolovacího modulu do systému pro získávání znalostí je již součástí dokumentace k diplomovému projektu [13]. Přidání dolovacího modulu do stávajícího systému je celkem snadné a je ho možné provést bez zásahu do původních zdrojových kódů základního modulu. Za prvé je nutné do balíku *preprocessing.task* přidat třídu implementující rozhraní *MiningTaskGui*. Rozhraní *MiningTaskGui* definuje metody potřebné pro zobrazení prvků GUI, pomocí kterých lze zadat parametry a zobrazit výsledky dolovací úlohy, a dále definuje metody pro načtení/uložení těchto informaci z/do DMSL dokumentu:

```
public interface MiningTaskGui {
      public String toString();
```

```
    public MiningTask getMiningTask();
    public void init(MainWindow wnd, MiningTask task);
    public String getDmslType();
    public Component getDefinitionCard();
    public Component getResultCard();
    public boolean checkDefinition();
    public void SaveDefinitionToDmsl(Node dmsl);
    public void LoadDefinitionFromDmsl(Node dataMiningTask,
    Map fieldList) throws SAXException;
    public void SaveResultToDmsl(org.w3c.dom.Node dmsl);
    public void LoadResultFromDmsl(Node knowledge, Map fieldList)
    throws SAXException;
}
```

Popis jednotlivých metod rozhraní *MiningTaskGui*:

- *init()* – inicializace grafických komponent potřebných pro zadání parametrů dolovací úlohy a pro zobrazení jejích výsledků,
- *getDmslType()* – vrátí typ dolovací úlohy, který se potom používá v atributu type v DMSL elementech *DataMiningTask* a *Knowledge*. Systém pomocí tohoto atributu pozná, která ze tříd implementujících rozhraní *MiningTaskGui* má tyto elementy zpracovat,
- *getDefinitionCard()* – vrací grafickou komponentu, která obsahuje vizuální rozhraní pro zadání parametrů dolovací úlohy,
- *getResultCard()* – vrací grafickou komponentu, která obsahuje vizuální rozhraní pro zobrazení výsledků dolovací úlohy,
- *CheckDefinition()* – provede kontrolu správnosti zadání parametrů dolovací úlohy. Pokud tato kontrola proběhne v pořádku, vrací boolovskou hodnotu „pravda“, v opačném případě vrací „nepravda“,
- *SaveDefinitionToDmsl()* – uloží definici dolovací úlohy, zadanou pomocí vizuálního rozhraní, do vnitřního DMSL dokumentu základního modulu systému,
- *LoadDefinitionFromDmsl()* – načte definici dolovací úlohy, abychom ji pak mohli zobrazit pomocí vizuálního rozhraní pro zadání parametrů dolovací úlohy,
- *SaveResultToDmsl()* – uloží výsledek dolovací úlohy do vnitřního DMSL dokumentu základního modulu systému,
- *LoadResultFromDmsl()* – načte výsledek dolovací úlohy, abychom ji pak mohli zobrazit pomocí vizuálního rozhraní pro zobrazení výsledků dolovací úlohy.

Druhým krokem je vytvoření třídy implementující rozhraní MiningTask, která musí být součástí dolovacího modulu. Rozhraní *MiningTask* je definováno takto:

```
public interface MiningTask {
      public Reader RunMiningTask( InputStream taskSpecification,
ProgressListener progress);
      public String getTaskName();
}
```

Popis jednotlivých metod rozhraní *MiningTask*:

- *RunMiningTask()* – tato metoda funguje jako prostředník mezi základním a dolovacím modulem. Má na starosti samotné spuštění dolování dat pomocí dolovacího modulu. Jako vstupní parametry získává odkaz na DMSL dokument s definicí dolovací úlohy, dále referenci na okno vizuálního rozhraní, umožňující modulu zobrazit postup dolovací úlohy. Výstupem této metody je pak odkaz na DMSL dokument se znalostmi získanými dolovací úlohou. Odkaz musí obsahovat původní DMSL dokument s přidaným elementem *Knowledge*, který bude obsahovat získané znalosti.
- *GetTaskName()* – vrátí typ dolovací úlohy. Poté je třeba zajistit, aby adresář *classes* systému pro získávání informací obsahoval přeložené zdrojové soubory (soubory s koncovkou *.class*) dolovacího modulu a třídy implementující rozhraní *MiningTask*.

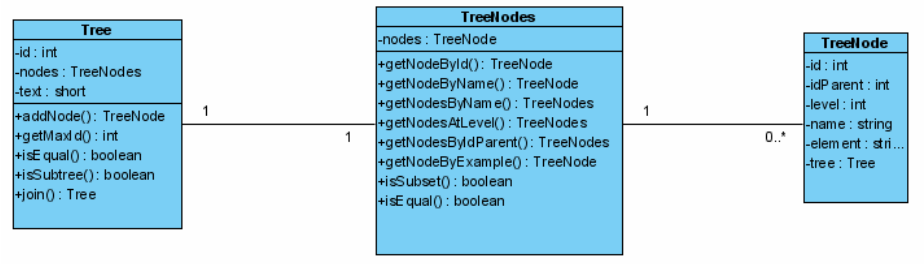
Třetím krokem je zadání informací o třídách implementujících rozhraní *MiningTaskGui* a *MiningTask* do souboru *MinerOptions.xml* nacházejícím se v adresáři systému pro získávání znalostí. Tento soubor obsahuje XML dokument zahrnující mimo jiné informace, pomocí nichž základní modul systému připojuje dolovací moduly. Do kořenového elementu *MinerOptions* je třeba přidat element *TaskModule*, jehož atribut *taskClass* bude obsahovat jméno třídy implementující rozhraní *MiningTask* a atribut *taskGuiClass* bude obsahovat jméno třídy implementující rozhraní *MiningTaskGui*. Například v mém případě vypadá element TaskModule takto:

```
<TaskModule
taskClass="MineClustering"
taskGuiClass="preprocessing.tasks.ClusteringGui"
/>
```

8 Realizace dolovacího modulu

8.1 Implementace algoritmu

Implementace algoritmu a jeho integrace do dolovacího systému je založena na jádru algoritmu popsaném v kapitole 6.4.2. Pro implementaci algoritmu bylo nutné vytvořit speciální datové struktury, především se jedná o vhodný návrh stromových struktur a seznamů. Základním prvkem algoritmu je n-ární stromu s jedním specifickým vrcholem, tj. kořen stromu. Možností, jak implementovat strom je několik, nejjednodušší a nejefektivnější metoda je implementace pomocí jednoduchého lineárního seznamu. Prvkem seznamu je uzel grafu (stromu). Vazba mezi uzly je tvořena odkazem na svého rodiče. Tímto způsobem je zajištěna hierarchie stromu a vytvořen tak vztah rodič- potomek. Dalším požadavkem na realizaci stromů bylo jejich opětovné seskupení do stromové struktury. Vhodný způsob, jak realizovat tyto abstraktní datové typy (ADT) nabízí paradigma objektového programování. Výstavba datových struktur je zajištěna třídami a metody tříd implementují potřebné operace nad těmito daty. Pro zajištění implementace je vhodné nejprve vytvořit diagram tříd. Následující obrázek uvádí zjednodušený diagram tříd.



Obrázek 5: Zjednodušený diagram tříd stromu

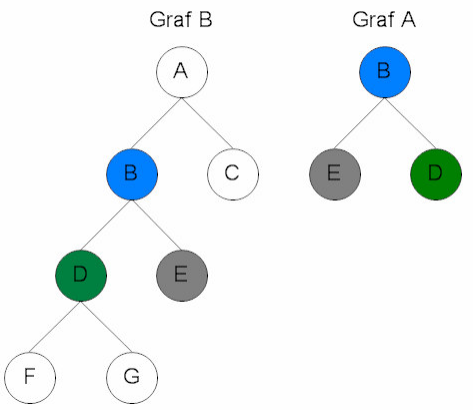
Diagram tříd uvádí významné vlastnosti a metody, které jsou důležité pro realizaci stromového abstraktního datového typu.

Základním prvkem ADT stromu je třída implementující uzel toho stromu. Třída *TreeNode* obsahuje jednoznačný identifikátor, který musí být unikátní v rámci stromu, ke kterému tento uzel patří. Proto je generování identifikátoru v režii patřičného stromu. Jak bylo uvedeno výše, je nutné vyjádřit vztah rodič-potomek. Tento vazba je modelována prostřednictvím vlastnosti *idParent*, která obsahuje identifikátor rodiče. Pro efektivní prácí s uzly stromu třída dále obsahuje úroveň uzlu v rámci stromu (0 je virtuální kořen). Datové informace v uzlu jsou tvořeny vlastnostmi *name* a *element*. Vlastnost

name obsahuje jméno uzlu a vlastnost *element* se vztahuje k příslušnému XML elementu. Pro některé operace nad třídou TreeNode je nutná informace o příslušnosti uzlu k jistému stromu, proto třída obsahuje odkaz *tree* na instanci rodičovského stromu.

Jednotlivé uzly stromu jsou sdruženy do třídy *TreeNodes*. Základní vlastností třídy je tedy lineární seznam objektů *TreeNode*. Třída *TreeNodes* definuje na kolekcí uzlů množství metod především pro vyhledávání a získávání uzlů podle jeho všech vlastností. Speciálnější jsou metody pro porovnávání dvou seznamů uzlů a zjišťování, zda je jeden seznam podseznamem jiného. Seznam A je podseznam B, pokud existuje injektivní zobrazení tj. $f : A \rightarrow B$, že platí $\forall(x_1, x_2 \in A)(x_1 \neq x_2)(f(x_1) \neq f(x_2))$. Seznamy A, B jsou ekvivalentní, pokud A je podseznam B a zároveň se rovnají počty jejich prvků.

Celkovou abstrakci stromu vytváří třída *Tree*. Zapouzdřuje seznam uzlů TreeNode a definuje vlastnosti stromu, především jeho identifikátor a textový popis. Třída dále implementuje důležité operace pro zjišťování ekvivalence s jiným stromem, a zda je podstromem jiného stromu. Náročná je zejména metoda pro zjišťování podstromů *isSubtree*. Jedná se tedy o problém zjišťování isomorfismu grafů a hledání podgrafu v grafu. Z hlediska organizace a návrhu struktur je implementován následující algoritmus. Uvažujme situaci dvou grafů A, B. Cílem je zjistit, zda A je podstromem B. Množina vrcholů A je podmnožinou vrcholů B. Jedná se tedy o zjištění izomorfizmu mezi grafy G_1, G_2. Důležitá je předpoklad, že grafy mají stejné množiny označení vrcholů.



Obrázek 6: Hledání podgrafu

Algoritmu projde všechny uzly cílového grafu. V každém uzlu tak vznikne stromu o 1 menší než původní a porovná se zdrojovým stromem. Situaci ilustruje následující příklad. Cílový graf označme G_1, zdrojový G_2. Algoritmus prochází cílovým grafem do postupně hloubky. Navštíví uzly v tomto pořadí A, B, D, F, G, E, C. V uzlu B dojde k detekci zdrojového podgrafu G_1, neboť child(B_{G1}) = {D, E} a child(B_{G2}) = {D, E}. Protože $child(B_{G1}) \subseteq child(B_{G2})$ a zároveň $B_{G1} \cong B_{G2}$ lze konstatovat, že $G_1 \subseteq G_2$.

8.2 Realizace generátoru

Jako podpůrný nástroj pro ověření funkčnosti algoritmu jsem vytvořil generátor XML dokumentu. Jeho implementace je součástí balíku *xmlGenerator*. Hlavní třída *Generator* spouští vytváření XML dokumentu dle zadaných parametrů, především se jedná o jméno výstupního souboru, počet vytvářených transakcí, počet transakcí v prvním stupni stromu, hloubku stromu a množinu atributů XML elementu. Hlavní třída, která generuje jednu transakci je třída *transactionGenerator*. Realizace generování transakce je založena na pseudo-náhodném budování XML stromu. V první fázi je vytvořen kořen stromu. Následně podle zadaného parametru na počet elementů v první fázi stromu (*first level*) se náhodně vybírají z množiny elementů prvky, které se připojují ke kořenu. V další fázi dochází k vyhledávání uzlů v již vytvořeném stromu a k nim se připojí zadaný uzel. Dochází tak k rozšiřování stromu jak do šířky, tak do hloubky. Generování končí po dosažení nastavené hloubky stromu.

Ukázka výstupu generátoru:

```
<?xml version="1.0" encoding="UTF-8"?>
<root value="root">
    <transaction value="Transaction">
        <element value="radio">
            <element value="hifi">
            </element>
            <element value="television">
            </element>
        </element>
    </transaction>
</root>
```

8.3 Začlenění do systému

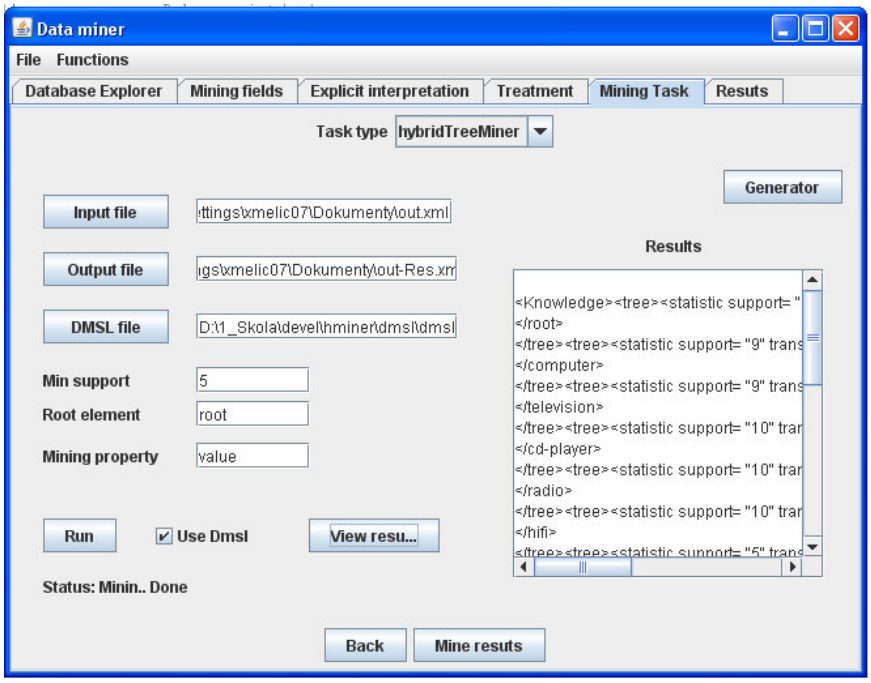
Za účelem začlenění dolovacího modulu do systému pro získávání znalostí bylo dále třeba provést několik kroků:

1. Vytvořit třídu *ClusteringGui*, která implementuje rozhraní *MiningTaskGui*, jehož popis lze najít v kapitole 7.5.1. Tato třída byla přídána do balíku *preprocessing.task* systému pro získávání znalostí. V třídě *ClusteringGui* jsem naprogramoval metody potřebné pro zobrazení prvků GUI, pomocí kterých lze zadat parametry dolovací úlohy a načíst či uložit tyto informace z nebo do DMSL dokumentu.
2. Vytvořit třídu *MinHybridTreeMiner*, která implementuje rozhraní *MiningTask*, jehož popis lze také najít v kapitole 7.5.1. V této třídě jsem naprogramoval metodu pro spuštění

dolovací úlohy a pro vrácení typu dolovací úlohy. Metoda pro spuštění dolovací úlohy nejdříve pomocí instance třídy *DmslParser* analyzuje vstupní DMSL dokument, poté vytvoří instanci odpovídající třídy (*Hminer*) a spustí dolování.

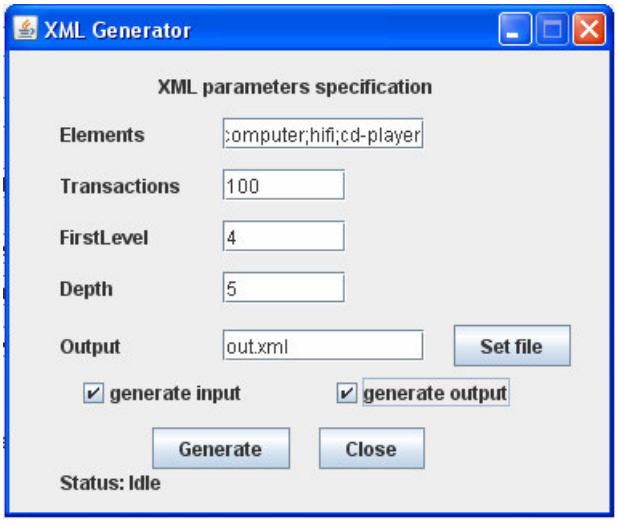
3. Podle postupu z kapitoly 7 jsem do souboru *MinerOptions.xml* přidal informaci o třídě *HminerGui*. Tímto krokem jsem prakticky připojil dolovací modul do systému pro získávání informací.
4. V třídě *HminerGui* jsem implementoval metody pro načtení a uložení znalostí získaných dolovací úlohou z/do DMSL dokumentu. Dále jsem dopsal třídy a metody pro zobrazení získaných znalostí pomocí GUI.

Nejprve byl realizován dolovací modul v samostatném balíku *hminer*. Podle výše popsaného postupu jsem jej začlenil do dolovacího systému. Tato práce spočívala především v realizaci GUI pro tento nově vznikající modul. Záložka Mining Task systému je doplněna o výběr mého dolovacího algoritmu GIU obsahuje několik voleb, především se jedná o specifikaci vstupů systému. Je možné nastavit vstupní soubor obsahující transakce a výstupní soubor, který bude obsahovat výstup dolování. Tyto údaje není nutné nastavovat v případě zatržení checkboxu *Use dmul*, což je XML soubor se specifikací dolovací úlohy. Dále je nutné nastavit požadovanou minimální podporu a pak volitelně kořenový element popř. vlastnost, na kterou se algoritmus ve vstupu zaměří. Výsledek dolování lze zobrazit do okna v pravé části. Výsledek pak obsahuje XML dokument s frekventovanými podstromy. Ukázka okna modulu je následujícím obrázku.



Obrázek 7: GUI Modul hybridTreeMiner

Jak bylo popsáno výše, pro účel testování jsem vytvořil generátor XML dokumentu. Jeho GUI jsem začlenil do dolovacího modulu. Okno generátoru je přístupné z modulu stiskem tlačítka *Generator*. Umožňuje nastavit parametry podle požadavku na výstupní XML dokument. Je nutné specifikovat množinu elementů, počet transakcí ve výstupu, počet elementů na první úrovni stromu a hloubku stromu. Dále okno obsahuje dva checkboxy, které nastavují jako vstup dolovacímu algoritmu výstup z generátoru a checkbox, který nastavuje dolovacímu algoritmu výstupní soubor. Ukázka okna je uvedena níže.

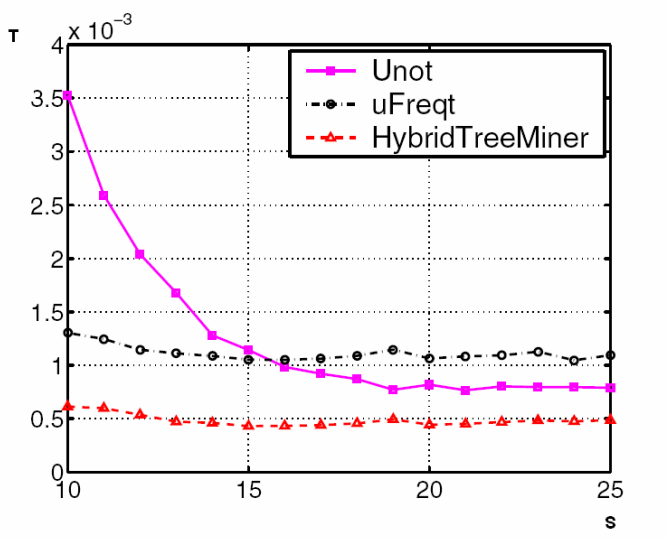


Obrázek 8: GUI Generátor

9 Testování

Tato kapitola je zaměřena na praktické ověření a vyhodnocení výsledů implementovaného algoritmu. Pro testování algoritmu byl vytvořen generátor vstupního XML dokumentu. Účelem generátoru je vytvořit syntetický XML dokument splňující některé zadané metriky. Generátor vytvoří dle zadaných parametrů vstupní dokument, který poté slouží jako vstup dolovacího modulu. Požadavkem na generátor bylo variabilní a náhodné vytváření dokumentu dle zadaných metrik. Především se jedná o počet transakcí, velikost množiny elementů a výšku stromu transakce. Detailnější popis generátoru je součástí kapitoly 8.2.

Cílem bylo analyzovat chování algoritmu v závislosti na různých parametrech vstupního dokumentu. Jedná se především o časovou složitost získávání frekventovaných vzorů. Dle dostupných informací například v [5] je chování algoritmu velice příznivé a dokazuje dobré výsledky při srovnání například s algoritmem *Unot* nebo s *uFreeTree*. Následující obrázek uvádí srovnání těchto dvou algoritmů.



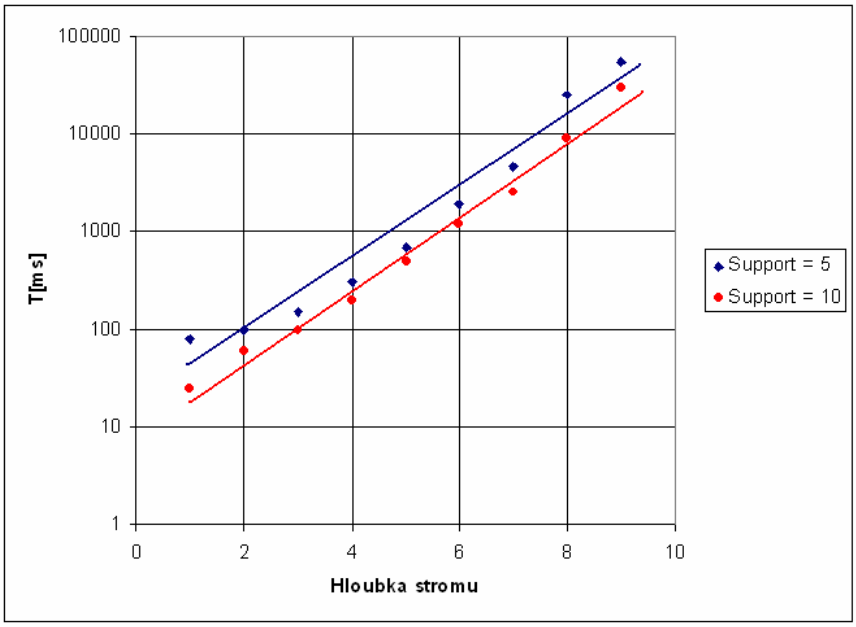
Obrázek 9: Srovnání algoritmů, převzato z [5]

T = čas na získání vzorku

S = velikost vzorku

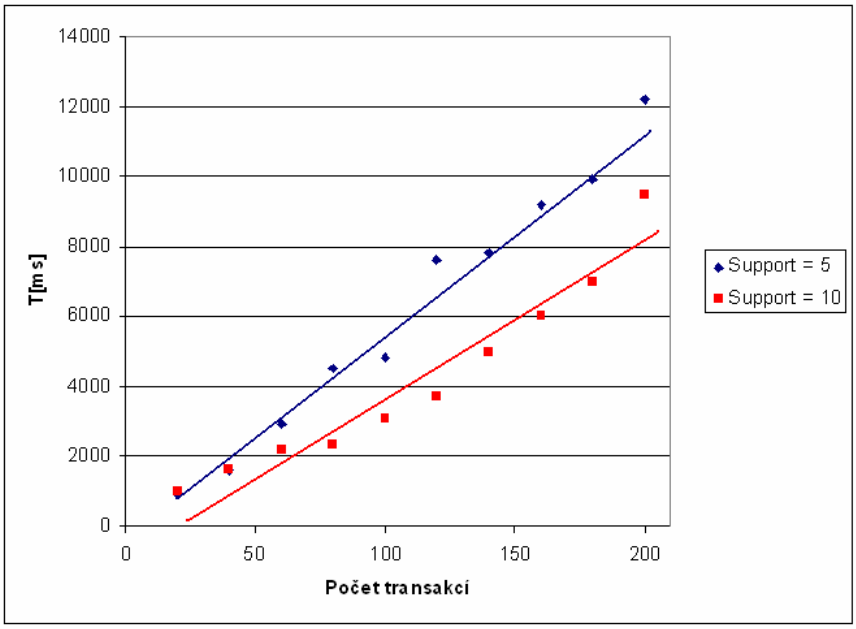
Ve své práci jsem se zaměřil na několik různých sad testů, které statisticky ukazují vlastnosti algoritmu. Následující grafy shrnují statistické parametry algoritmu. Graf 10 ukazuje trend exponenciální (časová osa je logaritmická) závislosti času dolování frekventovaných podstromů v závislosti na hloubce stromu. Graf zobrazuje dvě rovnoběžné charakteristiky rozdílné v parametru

nastavené minimální podpory pro hledané vzory. Ostatní parametry jako je množina elementů jsou konstantní.



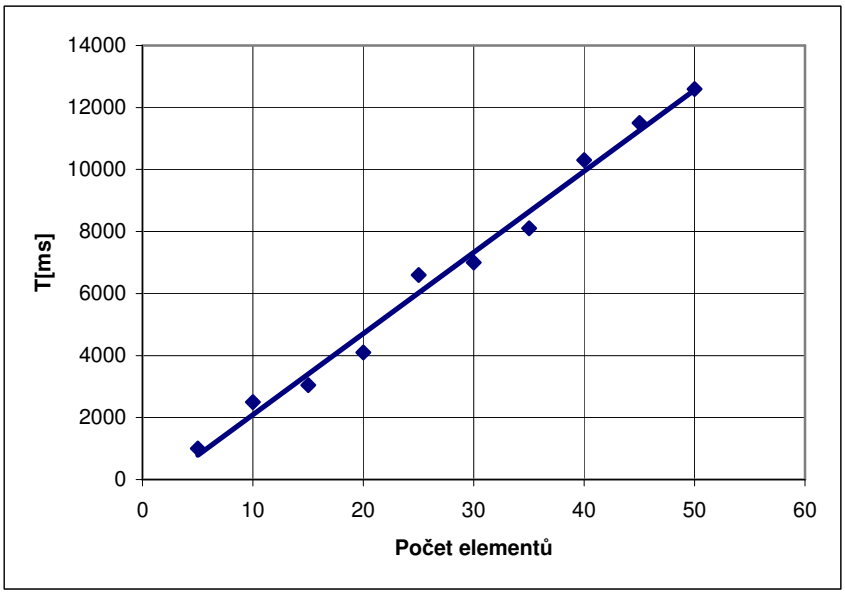
Obrázek 10: Test 1

Další test je zaměřen na zjištění charakteristiky doby trvání na počtu transakcí ve vstupním souboru. Charakteristika má podle očekávání lineární průběh. Při nastavené vyšší podpoře je generováno méně kandidátů a tedy i výsledných frekventovaných množin. Při nastavené vyšší podpoře dochází ke sklápění charakteristik.



Obrázek 11: Test 2

Následující test měl prověřit časovou složitost hledání frekventovaných vzorů na základě velikosti množiny. Graf opět ukazuje lineární průběh tohoto testu.



Obrázek 12: Test 3

10 Závěr

Má diplomová práce se zabývá problematikou získávání frekventovaných vzorů ze strukturovaných dat, zejména pak v rámci dat v podobě XML. Práce navazuje na semestrální projekt. Cílem práce bylo vytvořit modul HybridTreeMiner a začlenit ho do systému pro získávání znalostí, který je vyvíjen na naší fakultě.

Začal jsem nastudováním problematiky získávání znalostí z databází, především jsem se věnoval tématu strukturovaných dat.

Poté jsem se zaměřil na problematiku jazyků popisujících proces získávání znalostí z databází. Prostudoval jsem jazyky DMSL a PMML. Poté jsem Jazyk DMSL rozšířil o nové elementy potřebné pro popis vybraného algoritmu.

Dále jsem prozkoumal a prakticky se věnoval připojení modulu do systému. Praktická část projektu se týkala implementace algoritmu a jeho přizpůsobení pro získávání znalostí z XML. Následovně jsem tento modul začlenil do systému, vytvořil třídy pro zadání parametrů dolovací úlohy a pro zobrazení znalostí získaných pomocí tohoto modulu.

Možná vylepšení vidím především v efektivnějším vyhledávání v XML datech, což by přispělo k lepším parametrům systému. Doporučoval bych implementovat některou sofistikovanější metodu pro zjišťování isomorfismu grafů. Přínos práce je především v praktické implementaci algoritmu, začlenění jej do dolovacího systému a posouzení jeho parametrů. Zajímavé by bylo vyzkoušet algoritmus na praktickém vzorku dat nebo jako zdroj dat použít např. generátor BRITE[16].

Literatura

[1] Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, 2001.

[2] Zendulka, J. a kol., Získávání znalostí z databází, Studijní opora, FIT VUT, Brno, 2006

[3] Dunham, M., H.: Data mining introductory and advanced topics. Pretice Hall, 2003

[4] Berka, P.: Dobývání znalostí z databází. ACADEMIA, Praha, 2003.

[5] Yun, C., Yirong, Y., Richard R. Muntz: HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Tree and Free Trees Using Canonical Forms, USA 2006

[6] Huan, J., Wang W.,Prins J.: Efficient mining of frequent subgraph in the presence of isomorphism, USA, 2003.

[7] Theobald, M., Schenkel, R., Weikum, G.:Exploiting structure, annotation and ontological knowledge for automatic classification of XML data, USA, 2003

[8] Nijssen S., Kok N.: Efficient discovery of frequent unordered trees., USA, 2003.

[9] Asai, T., Arimura, H., Uno T., Nakano S.: Discovering frequent substructures in large unordered trees, Kyushu University, 2003.

[10] Chmelař, P.: Isomorfismus grafů, http://www.fit.vutbr.cz/~chmelarp/public/06grafizo.pdf

[11] Worl Wide Web Consortium. Extensible Markup Language (XML) 1.0, W3C ecommendation edition, 2000.

[12] Zaki, M. J., Aggarwal C.: Xrules: Effective structural classifier for XML data, USA, 2001.

[13] Hromčík, P.: Systém pro získávání znalostí z databází. [Diplomová práce], Vysoké učení technické, Fakulta informačních technologií, Brno, 2003.

[14] Kotásek, P.: DMSL: Data mining specification language. [Disertační práce], Vysoké učení technické, Fakulta informačních technologií, Brno, 2003.

[15 Business Intelligence, Knowledge discovery in databases, http://www.kmining.com/info_definitions.html (10.května 2008)

[16] Medina, A., Lakhina, I., Matta, J. Byers. Brite: Universal topology generation from a user's perspective [Odborý článek], Boston University, 2001.

Příloha 1

Ukázka vstupního XML souboru

```
<?xml version="1.0" encoding="UTF-8"?>
<root value="root">
    <transaction id="0">
        <hifi value="hifi">
            <cd-player value="cd-player">
            </cd-player>
        </hifi>
        <cd-player value="cd-player">
        </cd-player>
    </transaction>
    <transaction id="1">
        <radio value="radio">
            <hifi value="hifi">
            </hifi>
        </radio>
        <cd-player value="cd-player">
        </cd-player>
    </transaction>
    <transaction id="2">
        <hifi value="hifi">
            <cd-player value="cd-player">
            </cd-player>
        </hifi>
        <cumputer value="cumputer">
        </cumputer>
    </transaction>
    <transaction id="3">
        <radio value="radio">
            <cd-player value="cd-player">
            </cd-player>
        </radio>
        <cd-player value="cd-player">
        </cd-player>
    </transaction>
</root>
```

Ukázka výstupu

```
<Knowledge>
    <tree>
        <statistic support= "4" transactions="0,1,2,3" />
        <cd-player value="cd-player">
        </cd-player>
    </tree>
    <tree>
        <statistic support= "2" transactions="1,3" />
        <radio value="radio">
        </radio>
    </tree>
    <tree>
        <statistic support= "1" transactions="2" />
        <cumputer value="cumputer">
        </cumputer>
    </tree>
    <tree>
        <statistic support= "3" transactions="0,1,2" />
        <hifi value="hifi">
        </hifi>
    </tree>
    <tree><statistic support= "2" transactions="0,2" />
        <hifi value="hifi">
            <cd-player value="cd-player">
            </cd-player>
        </hifi>
    </tree>
</Knowledge>
```