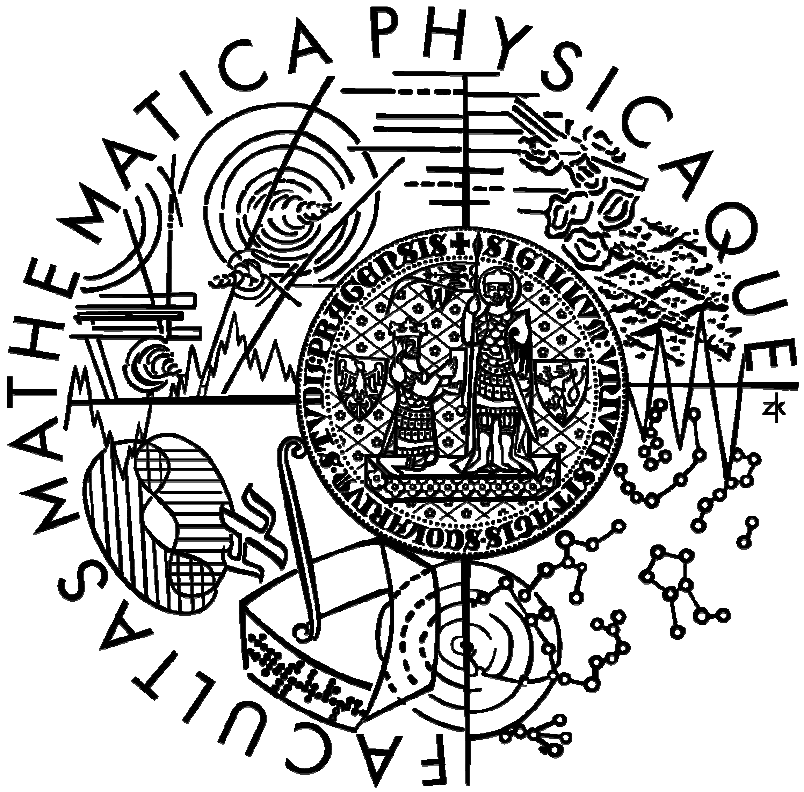
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Alan Eckhardt

Metody pro nalezení nejlepší odpovědi s různými uživatelskými preferencemi

Katedra softwarového inženýrství
Vedoucí diplomové práce: *Prof. RNDr. Peter Vojtáš, DrSc.*
Studijní program: *Informatika, I2 - Softwarové systémy*

Chtěl bych poděkovat vedoucímu Prof. RNDr. Peterovi Vojtášovi, DrSc. za vstřícné vedení a kontrolu průběhu vzniku diplomové práce. Dále děkuji rodičům, že mne podporovali po celou dobu studia, a všem přátelům, se kterými jsem tuto práci konzultoval.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7.srpna 2006

Alan Eckhardt

.....

Název práce: *Metody pro nalezení nejlepší odpovědi s různými uživatelskými preferencemi*
Autor: *Alan Eckhardt*
Katedra (ústav): *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Prof. RNDr. Peter Vojtáš, DrSc.*
e-mail vedoucího: *Peter.Vojtas@mff.cuni.cz*
Abstrakt: Uživatelské preference jsou jedním z nově zkoumaných aspektů v informatice, např. v doméně sémantického webu. Tato práce má za cíl pokrýt problematiku preferencí, jejich modelování, různé aspekty jejich generování a vyhodnocování. Jedna z částí se zabývá oblastí skupinového rozhodování a nalezení výsledku vyhovující celé skupině uživatelů. Další část navazuje na článek Ronalda Fagina, Amnona Lotema, Moni Naora o top-k algoritmu. Je zde prezentován nezávislý systém pro top-k algoritmus, testuje různé implementace tohoto algoritmu a probírá mnoho dalších aspektů, které ovlivňují jeho rychlost tak i počet prozkoumaných záznamů.
Klíčová slova: *sémantický web, uživatelské preference, skupinové rozhodování ,top-k algoritmus*

Title: Methods *for finding best answer with various user preferences*
Author: *Alan Eckhardt*
Department: *Department of software engineering*
Supervisor: *Prof. RNDr. Peter Vojtáš, DrSc.*
Supervisor's e-mail address: *Peter.Vojtas@mff.cuni.cz*
Abstract: User preferences are one of new aspects in informatics, e.g. in domain of semantic web. This work tries to cover the whole problematic of user preferences, their modeling, various aspects of generating user preferences and their evaluation. It also studies the area of group decision and finding answer that suits all members of group. Next part follows the paper of Ronald Fagin, Amnon Lotem, Moni Naor about the top-k algorithm. The new independent system for the top-k algorithm is presented, some implementations of this algorithm are tested and the influence of modules of algorithm on the speed and the number of rows are studied.
Keywords: *semantic web, user preferences, group decision, top-k algorithm*

1. Úvod

Sémantický web je nový směr výzkumu v oblasti informatiky. Jeho cílem je rozšířit funkčnost internetu, který je omezený http protokolem a normou HTML. Jednou z hlavních motivací je rozšíření vyhledávačů o smysl vyhledávaných slov, tedy jejich sémantiku. Tím by se významně zvýšila přesnost nalezených výsledků.

Uvažme nyní krátký přiklad - chceme jít dnes na oběd do restaurace a chtěli bychom si dát kuře. Zadejme tedy do vyhledávače Google slova "restaurace oběd kuře". Třetí odkaz ukazuje na deník ze zahraničního výletu Folklórního souboru Hořeňák v roce 2005 do Bulharska, což doopravdy neodpovídá představě dnešního oběda v restauraci.

Jeden z pojmů sémantického webu jsou ontologie. Ontologie zachycuje pojmy a vztahy mezi pojmy v nějaké konkrétní, omezené oblasti. Díky těmto ontologiím lze vidět pojem v kontextu dané problematiky. Důležitou vlastností ontologií je, že jsou srozumitelné pro stroje. Ontologie jsou nejčastěji zapisovány pomocí formátu RDF (Resource Description Framework) nebo některým z jeho rozšíření.

Tato práce se bude zabývat vnesením pojmu uživatel do světa internetu. Každý uživatel má jiné potřeby, zajímá se o jiné oblasti, líbí se mu různé věci. Budeme se snažit pokrýt celou problematiku uživatelských preferencí. Jedním z cílů bude zaznamenat tyto preference v RDF. Zakomponováním preferencí uživatelů do ontologií budeme moci provádět vyhledávání, které bude používat smysl (sémantiku) slov a navíc bude uživatelsky zohledněné, tedy využijeme znalosti typu - uživatel má rád kuřata a italské restaurace.

Uživatelské preference obvykle kombinují více kritérií, proto se budeme také zabývat vícekriteriálním rozhodováním. Vícekriteriální rozhodování je součást našeho každodenního přemýšlení, ale hraje roli i při klíčových rozhodnutích. Cílem bude naimplementovat různé algoritmy pro vícekriteriální rozhodování v jazyce Java, čímž se umožní využití těchto algoritmů v širokém spektru aplikací sémantického webu, který ve valné většině případů využívá právě jazyk Java.

Rozebereme také několik případů, jak zjistit uživatelovy preference na objektech, které explicitně nehodnotil. Navrhneme způsob, jak toto hodnocení zjistit na základě hodnocení ostatních objektů.

2. Vícehodnotová logika

V této části se budeme zabývat vícehodnotovou logikou. Popíšeme základní fakta o vícehodnotové logice a dokážeme některé věty. Nejdříve si přiblížíme výrokovou vícehodnotovou logiku, kterou v kapitole 2.4 rozšíříme na predikátovou vícehodnotovou logiku. Tato část byla převzata z [LP], až na rozšíření důkazu tvrzení 2.3.4 a 2.3.6 o body 5) a 6), které je naším vlastním příspěvkem.

2.1. Definice jazyka a značení

Ve značení budeme vycházet ze značení klasické dvouhodnotové logiky.

Výrokové proměnné budeme označovat písmeny p,q,r,... Množinu všech výrokových proměnných označíme VP. Výrokové proměnné budou označovány také jako fakty.

Množinu pravdivostních hodnot budeme označovat *T*. Tato množina v klasické dvouhodnotové logice je množina {0,1}. Ve vícehodnotové logice může *T* být libovolná uspořádaná množina. V dalším textu budeme uvažovat pro přehlednost *T*=[0,1]. Prvky množiny *T* budeme značit x,y,.. Speciálně hodnotu těla implikace budeme značit *b* jako "*body*", hodnotu hlavy *h* jako "*head*" a hodnotu vlastní implikace *r* jako "*rule*". Předpokládáme, že množina *T* je lineárně uspořádaná.

Výrokové spojky konjunkce, disjunkce, ekvivalence, implikace a negace budeme označovat symboly $\&,\vee,\equiv,\rightarrow,\neg$. Pravdivostní funkce $T^2 \rightarrow T$ daných spojek budeme značit $\&^{\bullet},\vee^{\bullet},\equiv^{\bullet},\rightarrow^{\bullet},\neg^{\bullet}$.

Množinu výrokových formulí označíme jako *F*.

Ohodnocenou proměnnou *p* hodnotou *x* značíme $(p:x)$.

Ohodnocující funkci $VP \rightarrow T$ budeme značit v, její rozšíření na výrokové formule $F \rightarrow T$ potom $\bar{v}$. Hodnotu proměnné p označíme $v(p)=x$.

Symbol |= přebereme z klasické logiky, ale jeho obsah musíme nově definovat.

2.1.1. Definice

Pokud platí, že $v(p) \geq x$, pak řekneme, že $v \models (p:x)$. Dále pokud $\rightarrow^{\bullet}(v(B),v(H)) \geq r$, pak řekneme, že $v \models (H \leftarrow B:r)$.

Dokazování v Hilbertově stylu je ve vícehodnotové logice složitější. Pro různé spojky potřebujeme různé axiomatizace. V našem případě se omezíme na zobecnění datalogových dedukčních mechanismů.

2.2. Výrokové spojky

Nyní budeme chtít rozšířit význam výrokových spojek na vícehodnotovou logiku. Budeme vycházet z definice spojek pro dvouhodnotovou logiku. Tím zajistíme, že pro *T*={0,1} bude vícehodnotová logika ekvivalentní dvouhodnotové logice.

p	q	p & q	p v q	p=q	p->q	non p
1	1	1	1	1	1	0
1	0	0	1	0	0	0
0	1	0	1	0	1	1
0	0	0	0	1	1	1

2.2.1. Tabulka

Tato tabulka nám dává fixní hodnoty pro spojky v minimu a maximu *T*. Průběhem funkcí jednotlivých spojek mezi těmito hodnotami se budeme zabývat dále.

2.2.2. Příklad

Zmíníme zde několik standardních vícehodnotových spojek -

$\&_L^{\bullet}(x,y) = \max(0, x+y-1)$, Łukasiewiczova konjunkce

$\&_p^{\bullet}(x,y) = x * y$, produktová konjunkce.

$\&_G^{\bullet}(x,y) = \min(x,y)$, Gödelova konjunkce.

Takto definované konjunkce splňují okrajové podmínky definované tabulkou 1.1. Použitím de Morganových pravidel dostaneme z definic konjunkcí definice disjunkcí

$\vee_L^{\bullet}(x,y) = \min(1, x+y)$

$\vee_p^{\bullet}(x,y) = 1 - \&_p^{\cdot}(1-x, 1-y) = 1-(1-x)(1-y) = x+y+xy$

$\vee_G^{\bullet}(x,y) = 1 - \&_G^{\cdot}(1-x, 1-y) = 1 - \min(1-x, 1-y) = \max(x,y)$

Tyto disjunkce opět splňují okrajové podmínky.

2.2.3. Definice

Funkce $C: T^2 \to T$ se nazývá konjunktor, pokud splňuje následující podmínky

1) $C(1,1) = 1$
2) $C(0,1) = 0$
3) $C(1,0) = 0$
4) $C(0,0) = 0$
5) $\forall r > 0 : C(1,r) > 0$
6) C je spojitá zleva v druhé proměnné

Proměnné konjunktoru budeme mnemonicky značit $C(b,r)$.

2.2.4. Definice

Funkce $I: T^2 \to T$ se nazývá implikátor, pokud splňuje následující podmínky

1) $I(1,1) = 1$
2) $I(0,1) = 1$
3) $I(1,0) = 0$
4) $I(0,0) = 1$
5) $\forall h < 1 : I(1,h) < 1$
6) I je spojitá zprava v druhé proměnné

Proměnné implikátoru budeme mnemonicky značit $I(b,h)$.

2.3. Modus ponens

Z výrokové logiky zbývá převzít odvozovací pravidlo Modus Ponens (MP).

$$\frac{p,\ p \to q}{q}$$

2.3.1.1. Pravidlo modus ponens

Pro vícehodnotovou logiku bude mít tento tvar

$$\frac{(p:b),\ (p \to q : r)}{(q:h)}$$

2.3.1.2. Pravidlo modus ponens pro vícehodnotovou logiku

a dále budeme požadovat, aby $h = f_{\to}(b,r)$.

Jinak řečeno, pravidlo modus ponens říká, že pro každou ohodnocující funkci *v* takovou, že pro ni platí $v(p) \geq b$ a $v(p \to q) \geq r$, platí také $v(q) \geq h$.

Omezení, která klademe na $f_{\to}(b,r)$ se shodují s omezeními na konjunktor. Tento konjunktor bude závislý na použitém implikátoru.

2.3.2. Definice

Dvojice (C,I) je residuovaná, pokud $C(b,r) \leq h$ právě tehdy, když $I(b,h) \geq r$.

2.3.3. Značení

Resiudální konjunktor k implikátoru I budeme značit $C_I(b,r) = \inf\{h : I(b,h) \geq r\}$.
Resiudální implikátor ke konjunktoru C budeme značit $I_C(b,h) = \sup\{r : C(b,r) \leq h\}$.

Dále budeme používat zkratky

2.3.3.1. $H(b,r) = \{h : I(b,h) \geq r\}$

2.3.3.2. $R(b,h) = \{r : C(b,r) \leq h\}$

Tyto zkratky použijeme, pokud bude *I* nebo *C* fixované, jinak značíme H_I a R_C.

2.3.4. Tvrzení

Nechť I je implikátor. Pak $C_I(b,r)$ je konjunktor.

2.3.5. Důkaz

Ukážeme, že $C_I(b,r)$ splňuje všechny podmínky z definice konjunktoru.

1) $C_I(1,1) = 1$

Chceme, aby platilo $C_I(1,1) = 1$. Nahradíme $C_I(b,r)$ jeho definicí 2.3.3

$$\inf H(1,1) = 1$$

Rovnost platí, pokud $\forall h < 1 : I(1,h) < 1$. To je přesně podmínka 5) v definici implikátoru.

2) $C_I(0,1) = 0$

Opět přepíšeme na

$$\inf H(0,1) = 0$$

Z podmínky 2) v definici implikátoru platí, že pouze $I(0,1) = 1$. Pak $0 \in H(0,1)$ a infimum množiny $H(0,1)$ bude 0.

3) $C_I(1,0) = 0$

$$\inf H(1,0) = 0$$

Z podmínky 3) v definici implikátoru platí, že $I(1,0) = 0$. Pak $0 \in H(1,0)$ a infimum množiny $H(0,1)$ bude 0.

4) $C_I(0,0)=0$

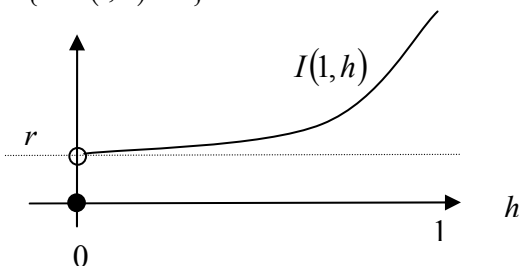
$$\inf H(0,0)=0$$

Z podmínky 4) v definici implikátoru platí, že $I(0,0)=1$. Pak $0\in H(0,0)$ a infimum množiny $H(0,0)$ bude 0.

5) $\forall r>0: C_I(1,r)>0$

$$\forall r>0: \inf H(1,r)>0$$

Důkaz sporem. Nechť $\exists r>0: \inf H(1,r)=0$. Přepíšeme si rovnost pomocí definice množiny *H* a dostaneme $\inf\{h: I(1,h)\geq r\}=0$.



2.3.5.1. Schéma

Uvažme podmínky 3) a 6) v definici implikátoru. Podle podmínky 3) platí, že $I(1,0)=0$ a podle podmínky 6) je I spojitý v druhé proměnné zprava. Tedy $\forall\varepsilon>0\exists\delta: I(1,(0+\delta))\leq\varepsilon$. Dále $\forall h>0$ platí, že $I(1,h)\geq r>0$. Tím jsme dostali spor se spojitostí *I*.

6) Spojitost v druhé proměnné zleva

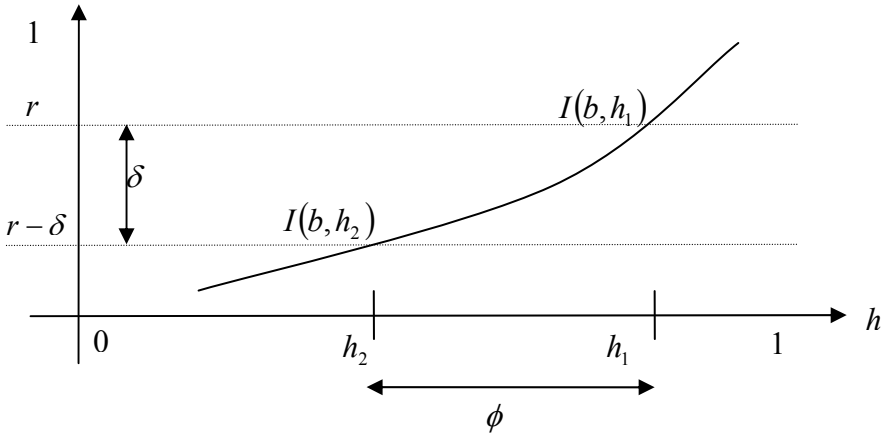
Chceme dokázat, že $\forall b\forall r\forall\varepsilon>0\exists\delta: |C(b,(r))-C(b,r-\delta)|\leq\varepsilon$.

Opět podmínku přepíšeme pomocí definice residovaného konjunkturu.

$|\inf H(b,r)-\inf H(b,r-\delta)|\leq\varepsilon$

$|\inf\{h_1: I(b,h_1)\geq r\}-\inf\{h_2: I(b,h_2)\geq r-\delta\}|\leq\varepsilon$

Víme, že *I* je spojitý v druhé proměnné zprava, tedy že $\forall b\forall h\forall\delta>0\exists\phi: |I(b,(h))-I(b,h-\phi)|\leq\delta$.



2.3.5.2. Schéma

Pro dostatečně malé δ tedy bude i platit, že $|h_1 - h_2| \le \varepsilon = |\phi| \le \varepsilon$.

□

2.3.6. Tvrzení

Nechť C je konjunktor. Pak $I_C(b,h)$ je implikátor.

2.3.7. Důkaz

Ukážeme, že $I_C(b,h)$ splňuje všechny podmínky z definice implikátoru..

1) $I_C(1,1)=1$

$$\sup R(1,1)=1$$

Z podmínky 1) v definici konjunktoru platí, že $C(1,1)=1$. Pak $1 \in R(1,1)$ a supremum množiny $R(1,1)$ bude 1.

2) $I_C(0,1)=1$

$$\sup R(0,1)=1$$

Z podmínky 1) v definici konjunktoru platí, že $C(0,1)=0$. Pak $1 \in R(0,1)$ a supremum množiny $R(0,1)$ bude 1.

3) $I_C(1,0)=0$

$$\sup R(1,0)=0$$

Rovnost platí, pokud $\forall r>0 : C(1,r)>0$. To je přesně podmínka 5) v definici konjunktoru.

4) $I_C(0,0)=1$

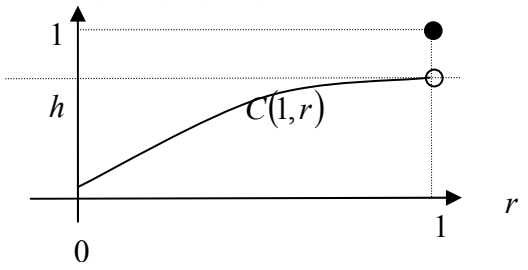
$$\sup R(0,0)=1$$

Z podmínky 1) v definici konjunktoru platí, že $C(0,1)=0$. Pak $1 \in R(0,0)$ a supremum množiny $R(0,0)$ bude 1.

5) $\forall h<1 : I_C(1,h)<1$

$$\forall h<1 : \sup R(1,h)<1$$

Důkaz sporem. Nechť $\exists h<1 : \sup R(1,h)=1$. Přepíšeme si rovnost pomocí definice množiny *R* a dostaneme $\sup\{r : C(1,r) \le h\}=1$.



2.3.7.1. Schéma

Uvažme podmínky 3) a 6). Podle podmínky 3) platí, že $C(1,1)=1$ a podle podmínky 6) je C spojitý v druhé proměnné zleva. Tedy $\forall\varepsilon>0\exists\delta: C(1,(1-\delta))\geq 1-\varepsilon$. Dále $\forall r<1$ platí, že $C(1,r)\leq h<1$. Tím jsme dostali spor se spojitostí *C*.

6) Spojitost v druhé proměnné zprava

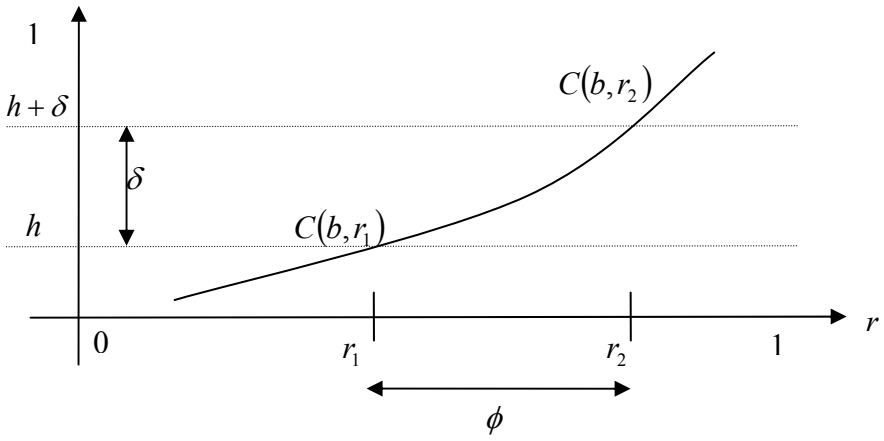
Chceme dokázat, že $\forall b\forall r\forall\varepsilon>0\exists\delta:\left|I(b,h)-I(b,h+\delta)\right|\leq\varepsilon$.

Opět podmínku přepíšeme pomocí definice residovaného konjunkturu.

$\left|\sup R(b,h)-\sup R(b,h+\delta)\right|\leq\varepsilon$

$\left|\sup\{r_1: C(b,r_1)\leq h\}-\sup\{h_2: C(b,r_2)\leq h+\delta\}\right|\leq\varepsilon$

Víme, že *C* je spojitý v druhé proměnné zleva, tedy že $\forall b\forall h\forall\delta>0\exists\phi:\left|C(b,r)-C(b,r+\phi)\right|\leq\delta$.



2.3.7.2. Schéma

Pro dostatečně malé δ tedy bude i platit, že $\left|h_1-h_2\right|\leq\varepsilon=\left|\phi\right|\leq\varepsilon$.

□

2.3.8. Věta

Nechť I je implikátor a C je konjunktor. Pak dvojice (I_C,C) a (C_I,I) jsou residuované.

2.3.9. Důkaz

Nejdříve dokážeme větu pro (C_I,I).

Ekvivalenci rozdělíme na dvě implikace. Jako první dokážeme

2.3.9.1. $I(b,h)\geq r\Rightarrow C_I(b,r)\leq h$.

Nahradíme $C_I(b,r)$ jeho definicí.

2.3.9.2. $\inf(H(b,r))\leq h$.

Z předpokladu $I(b,h)\geq r$ víme, že $h\in H(b,r)$. Pak v množině $H(b,r)$ máme alespoň jeden prvek. Pak infimum $H(b,r)$ je nejvýše ten prvek *h*. Tím jsme dokázali 1.3.9.2 a celou implikaci.

Druhá implikace je

$$C_I(b,r) \le h \Rightarrow I(b,h) \ge r$$

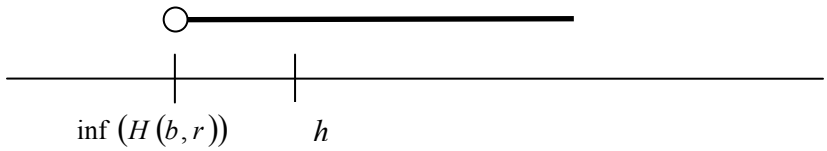
Upravíme předpoklad stejným způsobem jako v první implikaci pomocí definice

2.3.9.3. $\inf(H(b,r)) \le h$

Nyní předpokládáme 2.3.9.3. Z něj chceme odvodit, že $I(b,h) \ge r$.

Protože $\inf(H(b,r)) \le h$, pak je množina $H(b,r)$ neprázdná. Chceme dokázat, že $h \in H(b,r)$.

Situace, kdy platí, že $\inf(H(b,r)) < h$, je znázorněna na schématu 1.3.9.4



2.3.9.4. Schéma

Ze schématu je zřejmé, že $h \in H(b,r)$. Tím je implikace dokázána pro případ $\inf(H(b,r)) < h$.

Pokud $\inf(H(b,r)) = h$, pak existuje klesající posloupnost $h_n,...,h_2,h_1,h_0 = h$. $I(b,h)$ je spojitá zprava v h-proměnné, takže platí, že $I(b,h) \ge r$.

Tím jsme větu dokázali pro (C_I, I).

Nyní větu dokážeme pro (I_C, C).

Ekvivalenci rozdělíme na dvě implikace. Jako první dokážeme

2.3.9.5. $C(b,r) \le h \Rightarrow I_C(b,h) \ge r$.

V našem případě $I(b,h) \ge r$ je $I_C(b,h) \ge r$. Nahradíme $I_C(b,h)$ jeho definicí.

2.3.9.6. $I(b,h) \ge r \equiv I_C(b,h) \ge r$.

Z předpokladu $C(b,r) \le h$ víme, že $r \in R(b,h)$. Pak v množině $R(b,h)$ máme alespoň jeden prvek. Pak supremum $R(b,h)$ je nejméně ten prvek *r*. Tím jsme dokázali 1.3.9.6 a celou implikaci.

Druhá implikace je

$$I_C(b,h) \ge r \Rightarrow C(b,r) \le h$$

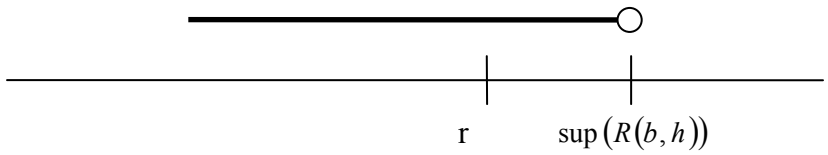
Upravíme předpoklad stejným způsobem jako v první implikaci

2.3.9.7. $\sup(r_0 : C(b,r_0) \le h) \ge r$

Nyní předpokládáme 1.3.9.7. Z něj chceme odvodit, že $I(b,h) \ge r$.

Protože $\sup(R(b,h)) \geq r$, pak je množina $R(b,h)$ neprázdná. Chceme dokázat, že $r \in R(b,h)$.

Situace, kdy platí, že $\sup(R(b,h)) > r$, je znázorněna na schématu 1.3.9.8



2.3.9.8. Schéma

Ze schématu je zřejmé, že $r \in R(b,h)$. Tím je implikace dokázána pro případ $\sup(R(b,h)) > r$.

Pokud $\sup(R(b,h)) = r$, pak existuje klesající posloupnost $r_n,...,r_2,r_1,r_0 = r$. $C(b,r)$ je spojitá zleva v r-proměnné, takže platí, že $C(b,r) \leq h$.

Tím jsme větu dokázali pro (I_C, C).

□

Nakonec dokážeme korektnost pravidla Modus Ponens ve vícehodnotové logice ve tvaru

$\frac{(B:b),(I(B,H):r)}{H:C_I(b,r)}$. Volně interpretováno nám toto pravidlo říká, že pokud víme, že *B* platí alespoň na *b* a *I(B,H)* alespoň na *r*, pak *H* platí nejméně na $C_I(b,r)$.

2.3.10. Důkaz

Dokážeme korektnost sporem. Předpokládejme, že existuje lepší výsledek h_0 než je $C_I(b,r)$. Nechť tedy existuje h_0 takové, že platí $h_0 > C_I(b,r)$ a zároveň *H:* h_0. Víme, že $C_I(b,r) = \inf(h : I(b,h) \geq r)$. Tedy existuje $h_1 : h_0 > h_1 \geq C_I(b,r) \& I(b,h_1) \geq r$ a tedy *H:* h_1. Což je spor s tím, že chceme, aby hodnocení *H* bylo minimální.

□

Tímto jsme dokázali platnost pravidla Modus Ponens ve vícehodnotové logice. Toto pravidlo budeme dále využívat v logických programech.

2.4. Rozšíření vícehodnotové logiky na predikátovou logiku

Ukázali jsme si, jak rozšířit dvouhodnotovou výrokovou logiku na vícehodnotovou výrokovou logiku. Nyní rozšíříme vícehodnotovou výrokovou logiku na vícehodnotovou predikátovou logiku, které bude rozšířením klasické dvouhodnotové predikátové logiky na vícehodnotovou.

2.4.1. Definice jazyka a značení

Proměnné budeme nyní označovat pomocí písmen x,y,z,.. Množinu všech proměnných označíme *PP*.

Predikátové symboly budeme psát velkými písmeny P,Q,R,... Pokud budeme chtít explicitně uvést aritu predikátu, budeme psát P^k.

Funkční symboly budu značeny písmeny *f,g,…* Speciálně nula-ární funkce budeme značit *a,b,c,...* a chápat je jako individuové konstanty. Množinu všech individuových konstant označíme *C*. V této práci si vystačíme pouze s individuovými konstantami.

Množinu atomických formulí budeme značit *AF*, množinu formulí potom *F*.

Pojmem term budeme rozumět všechny individuové konstanty a proměnné.

Realizaci jazyka predikátové logiky budeme značit $\Re$.

$\Re$ obsahuje množinu individuí, kterou budeme značit U, prvky této množiny budou $\alpha, \beta, \gamma, ...$ a budeme je nazývat individua.

Pro každý funkční symbol f arity k obsahuje $\Re$ funkci $f_{\Re}^{k} : U^{K} \to U$. Speciálně individuové konstanty budou funkce $f_{\Re} : 0 \to U$.

Pro definici splňování potřebujeme uvažovat také ohodnocení proměnných $PP \to U$, které budeme značit e, ohodnocení termu budeme psát $t[e, \Re]$.

Nakonec $\Re$ realizuje každý predikátový symbol $P_{\Re}^{k}$. Realizaci tohoto predikátu budeme značit $P_{\Re}^{k} : U^{K} \to T$. Pouze v tomto případě se uplatní "vícehodnotovost".

2.4.2. Interpretace

Interpretace $\Im$ je funkce $\Im : F \to T$. Interpretace je závislá na realizaci jazyka $\Re$, proto ji budeme značit $\Im_{\Re}$.

Interpretace výrokových spojek zůstává stejná, zbývá tedy definovat interpretaci kvantifikátorů $\forall, \exists$, predikátových symbolů, termů a atomických formulí. Uvažujme tedy interpretaci formule *A*.

1) *A* je term.

a) *A* je proměnná *x*, pak $\Im_{\Re}(x) = e(x)$

b) *A* je individuová konstanta a, pak $\Im_{\Re}(a) = a_{\Re}$

2) *A* je ve tvaru $\forall xB$, pak $\Im_{\Re}(\forall xB)$ bude infimum výstupů všech interpretací s ohodnocující funkcí *e'* formule *A* $\Im_{\Re \setminus e \cup e'}(B)$.

3) *A* je ve tvaru $\exists xB$, pak $\Im_{\Re}(\exists xB)$ bude supremum výstupů všech interpretací s ohodnocující funkcí *e'* formule *A* $\Im_{\Re \setminus e \cup e'}(B)$.

4) *A* je predikátový symbol P^k, pak $\Im_{\Re}(P^k) = P_{\Re}^{k}$.

5) *A* je atomická formule $P^k(x_1,...,x_k)$, pak $\Im_{\Re}(P^k(x_1,...,x_k)) = \Im_{\Re}(P^k)(\Im_{\Re}(x_1),...,\Im_{\Re}(x_k))$

Proměnné i predikátový symbol jsme interpretovali pomocí interpretace $\Im_{\Re}$.

Ve dvouhodnotové logice atomická formule $P(x)$ určovala, zda proměnná *x* má vlastnost *P*. Nyní $P(x)$ vyjadřuje, v jakém stupni má proměnná *x* vlastnost *P*. Sémantická funkce přiřazuje tedy jednotlivým individuím α stupeň vlastnosti $P(\alpha)$.

2.5. Závěr kapitoly

V této kapitole jsme položili základy vícehodnotové logiky do takové míry, jakou budeme dále potřebovat. Primární motivace bylo využití vícehodnotové logiky jako prostředku k vyjádření uživatelských preferencí. Spojky ve dvouhodnotové logice definované tabulkou jsou ve vícehodnotové logice funkce, které musí splňovat určité požadavky.

Vícehodnotová logika nám bude sloužit jako základ pro další kapitolu - logické programování.

3. Logické programování

Nyní použijeme teorii vícehodnotové logiky a vnoříme ji do logického programování. Dostaneme tím vícehodnotový (fuzzy) logický program. Tato část je opět převzata z [LP].

3.1. Struktura logického programu ve výrokové logice

Logický program je konečná množina ohodnocených proměnných a ohodnocených pravidel.

Fakty vyjadřují přímo ohodnocení výrokových proměnných. Pravidla jsou použita pro odvození dalších znalostí, jež nejsou přímo uvedeny ve faktech.

Konjunktory budeme značit $\&_1,\&_2,...$, implikátory pak $\leftarrow_1,\leftarrow_2,...$. Ve vícehodnotovém programování se tradičně používá šipka implikace v obráceném směru.

Ohodnocené pravidlo bude implikace ve tvaru $(H \leftarrow B, r)$, kde $H \in AF, r \in T$

Tělo pravidla *B* bude ve tvaru $(p \,\&_1\, q)\&_2\, r...$ Později zavedeme alternativu těla pravidla v podobě agregační funkce.

Dotaz ?-*Q* je výroková proměnná.

Zobecněný dotaz *GQ* je výraz tvořený z výrokových proměnných, konjunktorů a pravdivostních hodnot.

$t \in T$ je správná odpověď na dotaz ?-*Q* vzhledem k programu *P*, pokud pro každý model v programu *P* platí, že $v(Q) \geq t$ a neexistuje žádné $t' \in T$ takové, že $t' > t$ a $v(Q) \geq t'$.

3.1.1. Definice

Odvozovací krok je funkce $GQ \rightarrow GQ$. Je definován čtyřmi kroky

V1) Vybereme (nedeterministicky) nějakou výrokovou proměnnou *p* z *GQ*. *GQ* má tedy tvar L*p*R. Pak provedeme krok V2) nebo V3), pokud už nejsou v *GQ* žádné proměnné, provedeme krok V4).

V2) Pokud existuje ohodnocený fakt $(p : x)$, nahradíme *x* za *p*. Výstup funkce bude L*x*R.

V3) Pokud existuje ohodnocené pravidlo $(p \leftarrow B : r)$, nahradíme *p* výrazem $C_{\rightarrow}(B, r)$. Výstup funkce bude $LC_{\rightarrow}(B, r)R$.

V4) Pokud *GQ* neobsahuje žádnou výrokovou proměnnou, výsledkem je vyhodnocení výrazu *GQ*. Bez výrokových proměnných jde o aritmetickou úpravu.

3.1.2. Definice

$t \in T$ je vypočítaná odpověď na dotaz *Q* vzhledem k programu *P*, pokud existuje posloupnost zobecněných dotazů $Q = Q_0,...,Q_n = t$. Mezi Q_i a Q_{i+1} je použit odvozovací krok pro všechna $i < n$.

3.2. Agregační operátory

Agregační operátory jsou možnost, jak rozšířit vícehodnotovou logiku tam, kde nám nedostačují konjunktory.

3.2.1. Definice

$@$ je k-ární spojka, $k \geq 1$. Její pravdivostní funkce $@^{\bullet}$ je agregační operátor $T^k \to T$ a platí

1) $@^{\bullet}(0,..,0) = 0$
2) $@^{\bullet}(1,..,1) = 1$
3) $\forall x_1,...,x_k, y_1,...,y_k \mid (x_1 \leq y_1 \,\&\, ... \,\&\, x_k \leq y_k) \to @^{\bullet}(x_1,..,x_k) \leq @^{\bullet}(y_1,..,y_k)$

Agregační operátory budeme umisťovat na místo těla *B* pravidla $(H \leftarrow B, r)$. To bude potom mít tvar $(H \leftarrow @(p_1,..., p_k), r)$

Tyto agregační operátory mohou již mít libovolný tvar. Typickým příkladem agregačního operátoru je vážený průměr.

Motivace pro zavedení agregačních operátorů je přesnější a jasnější vyjádření uživatelských preferencí. V případě váženého průměru je zřetelně vidět, jaký atribut uživatel preferuje.

3.2.2. Příklad agregačních funkcí

$$@_1(x_1, x_2, x_3) = \frac{3x_1 + x_2 + 5x_3}{9}$$
$$@_2(x_1, x_2, x_3) = \max(x_1, x_2, x_3)$$
$$@_3(x_1, x_2, x_3) = \min(x_1, x_2, x_3)$$

3.2.3. Příklad programu

Mějme následující logický program

a) $(x_1 \leftarrow_G @_1(x_2, x_3), 0.8)$
b) $(x_2 \leftarrow_p x_3 \,\&_p\, x_4, 0.6)$
c) $(x_3, 0.9)$
d) $(x_4, 0.5)$

Kde $@_1(x, y) = \frac{x + y}{2}$.

Budeme hledat vypočítanou odpověď na dotaz $?- x_1$

Krok 1)

Máme k dispozici pouze jednu výrokovou proměnnou a to x_1. Jelikož neexistuje ohodnocený fakt s proměnnou x_1, musíme použít pravidlo a). Dotaz se tedy změní na $C_G(@_1(x_2, x_3), 0.8)$. (Použili jsme pravidlo V3.)

Krok 2)

Nyní vybereme proměnnou x_3. K ní máme ohodnocený fakt, nahradíme tedy tuto proměnnou její hodnotou. Dostaneme $C_G(@_1(x_2, 0.9), 0.8)$. (Použili jsme pravidlo V2.)

Krok 3)

Dále máme proměnnou x_2, ke které nemáme ohodnocený fakt. Použijeme tedy pravidlo b) a proměnnou nahradíme residovaným konjunktorem k implikaci. $C_G(@_1(C_p(x_3 \&_p x_4, 0.6), 0.9), 0.8)$. (Použili jsme pravidlo V3.)

Kroky 4,5)

Nakonec nahradíme proměnné x_3 a x_4 jejich hodnotou v ohodnocených faktech c) a d). Nakonec dostáváme $C_G(@_1(C_p(0.9 \&_p 0.5, 0.6), 0.9), 0.8)$. (Použili jsme pravidlo V2.)

Krok 6)

Poslední krok algoritmu je aritmetická úprava výsledku, což odpovídá pravidlu V4.

$C_G(@_1(C_p(0.9 \&_p 0.5, 0.6), 0.9), 0.8) = C_G(@_1(C_p(0.45, 0.6), 0.9), 0.8) = C_G(@_1(0.27, 0.9), 0.8) =$
$C_G(@_1(0.27, 0.9), 0.8) = C_G(0{,}585, 0.8) = 0.585$

Odpověď na dotaz $?- x_1$ je tedy 0.585.

3.3. Logické programování v predikátové logice

Hlavní rozdíl oproti logickému programu ve výrokové logice je zavedení universa *U* a individuí. Ve logickém programu výrokové logiky byl výsledek dotazu na výrokovou proměnnou *x* její pravdivostní hodnota.

Dotaz na logický program tedy bude mít tvar formule ?-*F*, obvykle atomické formule nebo ve tvaru $@_1(B_1, .., B_k)$. Pokud formule neobsahuje žádné volné proměnné, výsledek bude pravdivostní hodnota formule. Pokud ovšem *F* obsahuje volné proměnné, výsledkem budou možná nahrazení volných proměnných individui spolu s pravdivostní hodnotou *F* po nahrazení.

3.4. Struktura logického programu v predikátové logice

Ohodnocené pravidlo bude implikace ve tvaru $(H \leftarrow B, r)$, kde $H \in AF, r \in T$

Tělo pravidla *B* bude formule ve tvaru $(A \&_1 B) \&_2 C..., A, B, C \in AF$.

Ohodnocený fakt bude $P(a_1, ..., a_k) : r$. Vyjadřuje stupeň vlastnosti *P* pro k-tici $a_1, ..., a_k$.

Dotaz ?- *F* je atomární formule.

Substituce je funkce $\Theta : PP \rightarrow C$.

Nechť F má n volných proměnných $x_1, ..., x_n$. Pak $\Theta = \{x_1 / a_1, ..., x_n / a_n\}$ a $t \in T$ je správná odpověď na dotaz ?-*F* vzhledem k programu *P,* pokud po nahrazení volných proměnných v *F* je $\Im_{\mathfrak{R}}(F(\overline{x} / \overline{a})[e]) \geq t$ pro každé *e*.

3.4.1. Poznámka

Zřejmě může být správných odpovědí více. Ve výrokovém počtu byly správné všechny odpovědi menší než libovolná správná odpověď. Zde navíc každá hodnota *t* je závislá na n-tici *A*.

3.4.2. Příklad

Mějme logický program *P*:

$(P(x,y) \leftarrow_P Q(x,y),0.5)$

$Q(a,.b): 0.2$

$Q(a,c): 0.8$

a dotaz $?- P(a,x)$

Potom vypočítané odpovědi jsou $(\Theta = (x/b),0.1)$ i $(\Theta = (x/c),0.4)$. Získanou odpověď můžeme také zapsat ohodnocenými fakty $P(a,.b): 0.1$ a $P(a,.c): 0.4$.

Odpověď jsme získali analogickým postupem jako ve výrokovém programu. Nahradili jsme pravidlo residuovaným konjunktorem a poté použili ohodnocené fakty. Důležité zde je, že individuové proměnné v použitém faktu a ve zobecněném dotazu se musí shodovat, jinak fakt nemůžeme použít.

3.4.3. Definice

Odvozovací krok je funkce $GF \rightarrow GF$. Je definován čtyřmi kroky

P1) Vybereme nějakou atomární formuli P^k z *GF*. *GF* má tvar $LP(a_1,...,a_l,x_{l+1},...,x_k)R$. V atomární formuli je použito *l* individuových konstant $a_1,...,a_l$ a *k-l* proměnných $x_{l+1},...,x_k$.

P2) Pokud existuje ohodnocený fakt $(P(a_1,...,a_l,a_{l+1},...,a_k): x)$, výstup funkce bude $(LxR)\Theta$.

P3) Pokud existuje ohodnocené pravidlo $(P(x_1,...,x_k) \leftarrow B: r)$, nahradíme za *A* $C_{\rightarrow}(B,r)$. Výstup funkce bude $(LC_{\rightarrow}(B,r)R)\Theta$.

P4) Pokud *GF* neobsahuje žádnou atomární formuli, výsledek bude $(\Theta_1 \circ ... \circ \Theta_n, t)$, kde *t* je výsledná pravdivostní hodnota.

3.4.4. Poznámka

Pokud v krocích 2) a 3) části dotazu *L* nebo *R* obsahují některé z volných proměnných těla pravidla *B*, musíme tyto volné proměnné přejmenovat. Použijeme jména proměnných, která se v částech *L* a *R* nevyskytují. To nám zajišťuje substituce Θ.

3.4.5. Definice

$\Theta = \{x_1/a_1,...,x_n/a_n\}$, $t \in T$ je vypočítaná odpověď na dotaz *F* vzhledem k programu *P*, pokud existuje posloupnost zobecněných dotazů $Q = Q_0,...,Q_n$ a $\Theta_1 \circ ... \circ \Theta_n = \Theta$. Mezi Q_i a Q_{i+1} je použit odvozovací krok pro všechna $i < n$.

3.4.6. Příklad

Ukážeme si příklad při hledání výletu, kde svítilo sluníčko. Tento příklad nás bude provázet celou prací. V praktické části budeme pracovat s ontologií (pojem ontologie bude

vysvětlen dále), která se zabývá výletními lokalitami. Proto jsme zvolili příklad, který se týká této oblasti.

1) $SunnyTrip(x) \leftarrow_p @_1(hasWeather(x,y), SunnyWeather(y)) : 0.9$
2) $SunnyWeather(x) \leftarrow_G hasAtmosphericConditions(x, Sunny) : 0.9$

3) $hasWeather(Brdy050501, OvercastsMildTemperature) : 1$
4) $hasAtmosphericConditions(OvercastsMildTemperature, Overcasts) : 0.9$
5) $hasAtmosphericConditions(OvercastsMildTemperature, Sunny) : 0.1$
6) $hasWeather(Polsko030810, SunnyHighTemperature) : 1$
7) $hasAtmosphericConditions(SunnyHighTemperature, Sunny) : 0.9$

Agregační operátor $@_1$ bude funkce $@_1(x,y) = (x + 2 * y)/3$

Dotaz bude mít tvar ?-SunnyTrip(x).

Jelikož neexistuje žádný ohodnocený fakt s predikátem SunnyTrip, použijeme pravidlo 1). Tím vykonáme krok P3) logického programu.

Dostaneme dotaz ve tvaru $C_{\rightarrow}(@_1(hasWeather(x,y), SunnyWeather(y)), 0.9)$.

Nyní použijeme fakt 3).

$C_{\rightarrow_p}(@_1(hasWeather(Brdy050501, OvercastsMildTemperature),$
$SunnyWeather(OvercastsMildTemperature)), 0.9)$

Dále nahradíme atomickou formuli $hasWeather(Brdy050501, OvercastsMildTemperature)$ za její pravdivostní hodnotu.

$C_{\rightarrow_p}(@_1(1, SunnyWeather(OvercastsMildTemperature)), 0.9)$

Vykonali jsme krok P2) logického programu. Pro názornost jsme krok rozdělili do dvou částí.

Nahradíme predikát *SunnyWeather* podle pravidla 2).

$C_{\rightarrow_p}(@_1(1, C_{\rightarrow_G}(hasAtmosphericConditions(OvercastsMildTemperature, Sunny), 0.9)), 0.9)$

a použijeme fakt 5)

$C_{\rightarrow_p}(@_1(1, C_{\rightarrow_G}(0.1, 0.9)), 0.9)$

Nakonec vypočítáme výsledek

$C_{\rightarrow_p}(@_1(1, C_{\rightarrow_G}(0.1, 0.9)), 0.9) = C_{\rightarrow_p}(@_1(1, 0.1), 0.9) = C_{\rightarrow_p}(0.4, 0.9) = 0.36$.

Vypočítaná odpověď na dotaz ?-SunnyTrip(x). tedy je (x=Brdy050501, 0.36).

Další vypočítaná odpověď je (x=Polsko030810, 0.84).

3.5. Závěr kapitoly

Přiblížili jsme logické programování ve výrokové i predikátové logice, zavedli jsme pojem faktu a pravidla. Dále jsme ukázali dva možné tvary těla pravidla. Tyto struktury budeme v dalších kapitolách dále využívat.

Jedním z cílů této práce je vytvořit možný zápis vícehodnotového logického programu v RDF. V této zjednodušené formě se jej pokusíme vyzkoušet v experimentální implementaci v jazyku RDF.

Vztah mezi vypočítanou a správnou odpovědí v logickém programování a další hlubší aspekty této problematiky jsou mimo rámec této práce. Podrobnosti můžete nalézt např. v [LP].

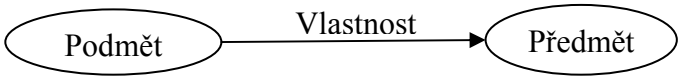
4. Prostředí pro popis zdrojů (RDF)

V této kapitole se budeme zabývat jazykem Resource Description Framework (RDF). V české literatuře se zatím nevžil žádný český výraz, proto budeme používat dobře známou zkratku RDF. RDF je jazyk určený pro reprezentování informací v prostředí internetu. Je navržen na jednoduchém principu grafového modelu dat. Rozšíření RDF, RDF Schema (dále RDFS), má jasně definovanou sémantiku, která umožňuje získávat nová fakta z již existujících pomocí logického důsledku.

Přehled RDF čerpal hlavně ze zdrojů skupiny W3C, tyto dokumenty lze nalézt na [W3C].

4.1. Základy RDF

Základní stavební kámen RDF jsou trojice.



4.1.1. Schéma

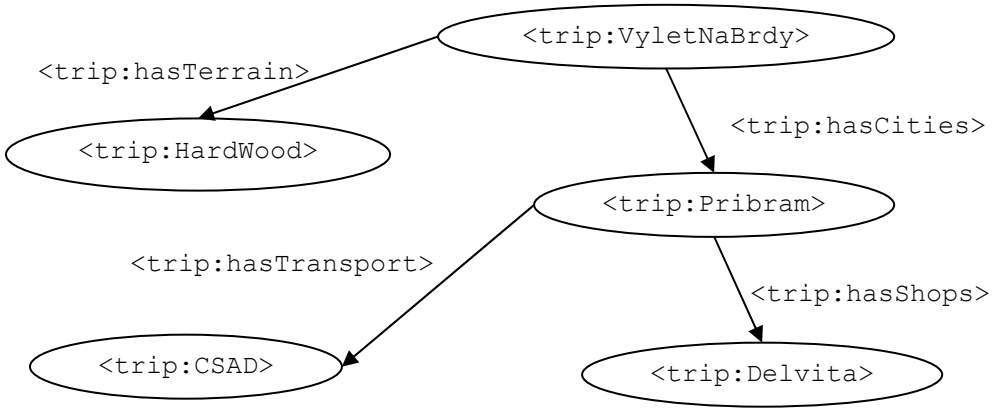
Trojice se skládá z podmětu, vlastnosti a předmětu. Budeme používat tento lingvistický překlad anglických slov "subject", "predicate" a "object". RDF dokument se skládá z trojic, jež poté tvoří orientovaný graf. Uzly grafu jsou tvořeny podměty a předměty, vlastnosti pak tvoří orientované hrany grafu.

Podměty v grafu musejí být platné URI identifikátory. Požadovaný tvar URI je definován v [URI]. Objekt, který je identifikovaný URI, budeme označovat jako zdroj. URI identifikátory budeme psát mezi úhlové závorky.

4.1.2. Příklad URI

```
<http://www.muaddib.wz.cz/trips#TripLocation>
```

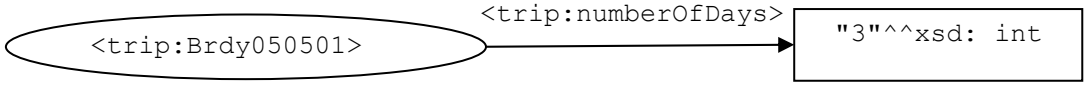
URI je možné zkracovat použitím jmenného prostoru URI. Pro náš případ definujme předponu trip: jako jmenný prostor `<http://www.muaddib.wz.cz/trips#>`. Nyní můžeme psát jasněji `<trip:TripLocation>`. Dále budeme používat standardní předpony rdf:, rdfs:, owl:, xml: a xsd:. Pokud v příkladu nebude záležet na daném URI, použijeme předponu foo:, která se typicky v dokumentech W3C i jinde používá pro nedůležité předpony.



4.1.3. Příklad RDF grafu

Pokud nějaký zdroj je zapsaný ve tvaru <_:Nazev>, jde o tzv. blank node neboli prázdný uzel. Tyto uzly nemají přiřazený jmenný prostor a používají se pro případy, kdy nechceme definovat plné jméno zdroje.

Kromě URI identifikátorů lze v RDF použít také datové typy. Datový typ je identifikován URI identifikátorem, typicky ve jmenném prostoru xsd:. Hodnota předmětu je uchována v textovém řetězci. Uzlu, který je datového typu, se říká literál.



4.1.4. Příklad datového typu

4.2. Jazyk RDF

RDF má pevně definované některé vlastnosti a podměty.

Budeme používat jmenný prostor rdf:, což je zkratka za http://www.w3.org/1999/02/22-rdf-syntax-ns#.

4.2.1. Obecný jazyk RDF

<rdf:type> je vlastnost. Trojice <trip:Brdy050501> <rdf:type> <trip:Trip> říká, že zdroj <trip:Brdy050501> je typu <trip:Trip>. Tato vlastnost definuje přímé náležení do nějaké třídy.

<rdf:Property> je třída. Všechny zdroje náležející do této třídy jsou vlastnosti.

```
<trip:numberOfDays> <rdf:type> <rdf:Property>
```

<rdf:XMLLiteral> je třída literálů, jejichž obsah je nějaký XML dokument.

<rdf:value> je vlastnost, která označuje hodnotu podmětu. Očekávané využití je tam, kde je předmět trojice nějaký složený typ, např. cena. Potom se předmět rozdělí na dvě trojice, kde první bude označovat vlastní cenu pomocí **<rdf:value>** a druhá měnu.

```
<foo:X> <foo:price> <foo:Price1>
<foo:Price1> <rdf:value> "27"^^xsd:int
<foo:Price1> <foo:currency> "CZK"
```

4.2.2. Jazyk reifikace

Pomocí tzv. reifikace lze vytvářet trojice, jejichž podmět je trojice. Tím je umožněno vyjádřit, že např. Ondra v úterý 16.5.2006 v 17:20 řekl, že na výletu na Brdy pršelo. První trojice má podmět fakt, že *na výletu na Brdy pršelo*, vlastnost je *řekl* a předmět je *Ondra*. Další trojice má stejný podmět, vlastnost *vČase* a předmět *16.5.2006 v 17:20*.

<**rdf:Statement**> je třída, která obsahuje trojice. Každý zdroj v této třídě má definované tři vlastnosti definované níže. Instance této třídy mohou být předměty dalších trojic.

<**rdf:subject**> určuje podmět trojice.

<**rdf:predicate**> určuje vlastnost trojice.

<**rdf:object**> určuje předmět trojice.

4.2.2.1. Příklad

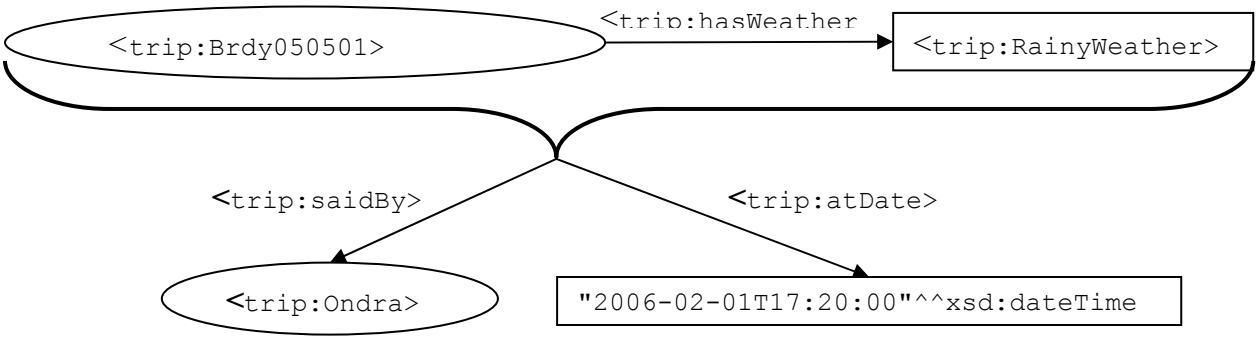
Nejdříve vytvoříme trojici tvrdící, že na výletu na Brdy pršelo.

```
<_:ReifiedTriple1> <rdf:type> <rdf:Statement>
<_:ReifiedTriple1> <rdf:subject> <trip:Brdy050501>
<_:ReifiedTriple1> <rdf:predicate> <trip:hasWeather>
<_:ReifiedTriple1> <rdf:object> <trip:RainyWeather>
```

Nyní řekneme, že výše uvedený fakt řekl Ondra.
```
<_:ReifiedTriple1> <trip:saidBy> <trip:Ondra>
```

A nakonec přidáme datum, kdy jej řekl.
```
<_:ReifiedTriple1> <trip:atDate> "2006-02-01T17:20:00"^^xsd:dateTime
```

Celý výrok je znázorněn na schématu 4.2.2.2.



4.2.2.2. Schéma

4.2.3. Jazyk seznamů

Seznamy umožňují vytvářet seznamy zdrojů a datových typů. V RDF jsou definované tři typy seznamů.

<**rdf:Seq**> je posloupnost zdrojů. Tento typ je uspořádaný.

<**rdf:Bag**> je neuspořádaný seznam, který povoluje duplicitní záznamy.

<**rdf:Alt**> vyjadřuje seznam alternativ. Může být uspořádaný podle preferovaného pořadí.

<**rdf:_1**> <**rdf:_2**> ... jsou vlastnosti, které jsou použity pro vyjmenování obsahu seznamů.

4.2.4. Jazyk kolekcí

<**rdf:List**> reprezentuje třídu kolekcí. Kolekce se skládá z prvního prvku (hlavy) a zbytku kolekce.

<**rdf:first**> reprezentuje hlavu kolekce.

<**rdf:rest**> reprezentuje zbytek kolekce, typicky další kolekci.

<**rdf:nil**> označuje konec kolekce nebo prázdnou hodnotu (tak, jak ji známe z funkcionálních programovacích jazyků).

4.2.5. Axiomy

Axiomy RDF definují význam jednotlivých definovaných zdrojů. Tyto axiomy mohou být použity pro odvození dalších trojic.

Reifikační vlastnosti
```
<rdf:subject> <rdf:type> <rdf:Property>
<rdf:predicate> <rdf:type> <rdf:Property>
<rdf:object> <rdf:type> <rdf:Property>
```
Vlastnosti pro seznamy
```
<rdf:first> <rdf:type> <rdf:Property>
<rdf:rest> <rdf:type> <rdf:Property>
<rdf:_1> <rdf:type> <rdf:Property>
<rdf:_2> <rdf:type> <rdf:Property>
```

...
<rdf:nil> <rdf:type> <rdf:List>
Další vlastnosti
<rdf:value> <rdf:type> <rdf:Property>
<rdf:type> <rdf:type> <rdf:Property>

Ve [VANEKOVA] je podrobně dokázáno, že tyto axiomy stačí, aby RDF získalo výrazovou sílu RDFS. K tomu účelu je třeba definovat struktury a interpretace jazyka RDF, v naší práci se vydáme jiným směrem.

4.3. Jazyk RDFS

Budeme používat jmenný prostor rdfs:, což je zkratka za http://www.w3.org/2000/01/rdf-schema#.

4.3.1. Jazyk tříd

<**rdfs:Resource**> je nadtřída všech ostatních tříd.
<**rdfs:Class**> označuje třídu. Tento termín je v RDF používán neurčitě. Každý zdroj, který je předmětem trojice s vlastností <rdf:type>, považováno za třídu.
Např. z trojice
<trip:Brdy050501> <rdf:type> <trip:Trip>
vyplývá, že zdroj <trip:Trip> je třída.
<**rdfs:Literal**> je instance třídy <rdfs:Class>. Jde o třídu literálů.
<**rdfs:Datatype**> je podtřídou <rdfs:Class>. Je to třída datových typů.

4.3.2. Jazyk vlastností

<**rdfs:range**> je vlastnost, která určuje obor hodnot vlastnosti, jež je předmětem trojice.
<**rdfs:domain**> je vlastnost, která určuje definiční obor vlastnosti, jež je předmětem trojice.
<**rdfs:subClassOf**> říká, že podmět je podtřídou předmětu.

Ze dvou trojic
<foo:B> <rdf:subClassOf> <foo:A> a
<foo:c> <rdf:type> <foo:B>
plyne trojice
<foo:c> <rdf:type> <foo:A>.

<**rdfs:subPropertyOf**> říká, že podmět je podvlastnost předmětu.

Ze dvou trojic
<foo:p> <rdf:subPropertyOf> <foo:q> a
<foo:c> <foo:p> <foo:d>
plyne trojice
<foo:c> <foo:q> <foo:d>.

Tedy všechny trojice s vlastností <foo:p> platí i se všemi nadvlastnostmi dané vlastnosti <foo:p>.

<**rdfs:label**> pojmenovává daný zdroj. Jde o název, který je určený pro lidského čtenáře.
<**rdfs:comment**> popisuje zdroj. Tento popis je také určený pro lidského čtenáře, ovšem je rozsáhlejší než <**rdfs:label**>.

Tento seznam jazyku RDF a RDFS byl úvodem pro jazyk OWL, který umožňuje přesnější a bohatší definici hierarchie tříd a vlastností než RDFS.

4.4. Databáze pro RDF

Pro formát RDF existuje několik databází. Tyto databáze uskladňují RDF trojice v nějakém úložišti, ke kterému se poté dá přistupovat pomocí dotazovacích jazyků. Tyto jazyky jsou přiblíženy v následující kapitole.

4.4.1. Inference

Další funkce těchto databází oproti tradičním relačním je v tzv. inferenci. Inference se také označuje jako odvozování, jde o proces vytváření nových trojic na základě vztahů mezi třídami. Tato inference může být různě silná - základní úroveň poskytuje RDFS, kde je definován vztah podtřída. Potom z trojic

```
<trip:HardWoods> <rdfs:subClassOf> <trip:Woods>
<trip:HardWood> <rdf:type> <trip:HardWoods>
```

lze odvodit novou trojici

```
<trip:HardWood> <trip:type> <trip:Woods>.
```

Totéž platí i pro vlastnosti a podvlastnosti. Jmenný prostor foaf: odkazuje na `<http://xmlns.com/foaf/0.1/>`. Foaf je jedna z nejznámějších ontologií, modeluje vztahy mezi lidmi a informace o zálibách, publikacích apod. Používá pouze syntax RDFS.

Z následujících trojic

```
<foaf:homepage> <rdfs:subPropertyOf> <foaf:page>
<trip:AlanEckhardt> <foaf:homepage> <foaf:MojeStranka>
```

lze odvodit novou trojici

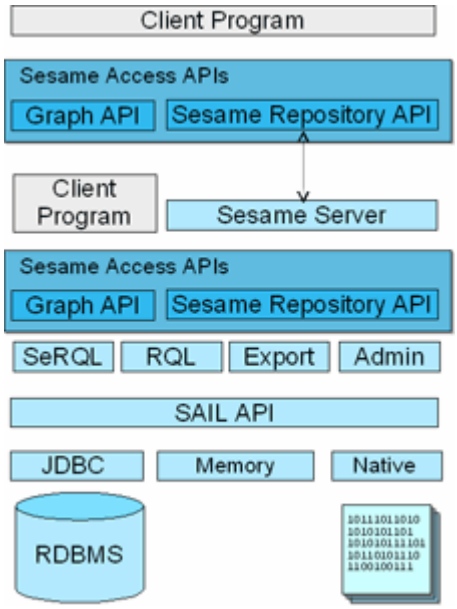
```
<trip:AlanEckhardt> <foaf:page> <foaf:MojeStranka>.
```

OWL nabízí mnohem bohatší slovník, který definuje další odvozovací pravidla. Pomocí OWL lze zapsat podstatně přesnější vztahy.

Odvozování těchto nových trojic je ponecháno na databázi. Existují také programy, které umožňují provádět odvozování na RDF souborech. Těmto programům se říká reasonery neboli odvozovací stroje. Ty mohou ověřit konzistenci RDF souboru a vygenerovat nové odvozené trojice. Nekonzistence se dá docílit až v OWL, kde lze zadat omezení na individua třídy. Pokud je nějaký zdroj individuum této třídy a zároveň nesplňuje zadaná omezení, nejsou data konzistentní.

4.4.2. Sesame

Jednou z nejznámějších RDF databází je Sesame. Sesame je rozdělen do několika logických vrstev, čímž umožňuje velmi dobrou flexibilitu. Vrstvy mezi sebou komunikují pomocí pevně daných rozhraní, každou z vrstev lze tedy doprogramovat.



4.4.2.1. Obrázek [SESAME]

Schéma Sesamu je na obrázku 4.4.2.1. K Sesamu se dá přistupovat z libovolného Java programu, pak se využívá Sesame Access APIs. Přes toto rozhraní lze zadávat dotazy do RDF dat, případně provádět další operace jako je mazání trojic nebo jejich přidávání.

Zajímavou částí obrázku je SAIL API. SAIL je zkratka za Storage And Inference Layer. Tato vrstva provádí odvozování a samotné získávání dat. SAIL využívají moduly pro dotazovací jazyky a také administrátorský modul, který zabezpečuje přidávání a mazání trojic. Tuto vrstvu budeme označovat obecně jako úložiště.

Výhoda této vrstvy je ta, že je nezávislá na použitém úložišti RDF dat. To umožňuje lehkou přenositelnost programu mezi databázovými platformami nebo např. testování programu na datech uložených v paměti a poté přechod na plně databázový přístup. Zajímavé je, že odvozování provádí tato vrstva. Pokud náš program potřebuje odvozování, musíme tedy vybrat takové úložiště, které ho podporuje.

Sesame dále umožňuje spravovat jednotlivá úložiště přes rozhraní internetového prohlížeče pomocí Apache Tomcat. Správa přes toto rozhraní je jednoduchá a uživatelsky příjemná, lze zde mimo jiné vyprázdnit úložiště, nahrát do něj data z nějakého souboru, pokládat do něj dotazy apod. Mezi další funkce patří i export dat do RDF souboru. K úložištím se přistupuje přes Apache Tomcat, čímž je programátor plně odstíněn od nutnosti konfigurace připojení k databázi apod.

4.4.3. Jena

Jena je oproti Sesamu již dlouhodobější projekt, verze 2.0 byla vydána 27.8.2003. Jena poskytuje programátorovi více funkcí než Sesame, chybí jí ovšem internetové rozhraní pro správu úložišť. Programátor musí sám přímo v programu uvést, odkud chce získávat data.

Jena umožňuje získávat data z úložiště pomocí funkcí tohoto úložiště. Např. funkce listSubjectsWithProperty vrátí všechny podměty s danou vlastností. Takto lze získávat další data bez nutnosti používat dotazovací jazyk.

4.5. Dotazovací jazyky pro RDF

V této kapitole si přiblížíme dva dotazovací jazyky pro RDF, SeRQL a SPARQL. SeRQL je vyvinut přímo pro Sesame, je to proprietární jazyk. SPARQL je standard konsorcia W3C, čímž je zajištěna jeho použitelnost na více RDF databázích.

Oba dva jazyky podporují dva přístupy - dotazovací a vytvářející. Dotazovací část se podobá tradičnímu SQL z relačních databází, vytvářející potom vytváří RDF graf z dat získaných v dotazu. V přiblížení jazyků se budeme věnovat pouze jejich dotazovací části.

4.5.1. SeRQL

SeRQL vychází z notace SQL, ze které si bere klauzule "SELECT", "FROM" a "WHERE". Dále zavádí klauzuli "USING NAMEPSPACE", která umožňuje využití jmenných prostorů.

Typický SeRQL dotaz může vypadat takto

```
select distinct Y
from {X} rdf:type {trip:Trip},
        {X} p {Y}
where p like "*City*"
using namespace
        rdf  =<http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
        rdfs  =<http://www.w3.org/2000/01/rdf-schema#>,
        trip =<http://www.muaddib.wz.cz/>
```

Tento příklad vrací takové zdroje, jež jsou předmětem vlastnosti, která obsahuje řetězec "City" a podmětem této vlastnosti je nějaký zdroj typu `<trip:Trip>`.

Z příkladu je vidět použití příkazu distinct, který eliminuje duplicitní záznamy. Dále je zde použit výraz like, který umožňuje omezit hodnotu nějaké proměnné pomocí regulárního řetězce.

Dále v klauzuli from je výraz pro jednotlivé trojice. Tyto trojice jsou odděleny čárkou. Sesame umožňuje i zkracování zápisu v případě shodného podmětu takto

```
from {X} rdf:type {trip:Trip};
                p {Y}
```

SeRQL dovoluje použít tzv. volitelné trojice. Je to obdoba left join z relačních databází. Pokud zadaná volitelná trojice neexistuje, místo ní jsou použity hodnoty NULL.

```
select X, Y
from {X} rdf:type {trip:Trip},
        [{X} trip:OndrasRating {Y}]
using namespace
        rdf  =<http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
        rdfs  =<http://www.w3.org/2000/01/rdf-schema#>,
        trip =<http://www.muaddib.wz.cz/>
```

Tento dotaz vrátí seznam výletů a případné Ondrovo hodnocení, pokud existuje. Pokud ne, bude na místě proměnné Y hodnota NULL.

4.5.2. SPARQL

SPARQL používá klauzule "SELECT", "FROM" a "WHERE". Dále zavádí klauzuli "PREFIX", která umožňuje využití jmenných prostorů. Klauzule "FROM" ovšem není použita v tradičním smyslu, ale označuje zdroj dat. V našem případě ji nebudeme uvažovat.

Typický SPARQL dotaz může vypadat takto

```
PREFIX trip:    <http://www.muaddib.wz.cz/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?y
WHERE { ?x rdf:type trip:Trip.
        ?x ?p ?y.
        FILTER (REGEX(str(?p) ,".*City.*")).
         }
```

Příklad je stejný, jako u jazyku SeRQL. Místo výrazu like používá SPARQL obecný výraz FILTER, který značí podmínku a dále REGEX, který porovnává proměnnou ?p převedenou na řetězec proti regulárnímu výrazu.

Také SPARQL umožňuje volitelné trojice.

```
PREFIX trip:    <http://www.muaddib.wz.cz/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?x ?y
WHERE { ?x rdf:type trip:Trip.
        OPTIONAL {?x trip:OndrasRating  ?y}
}
```

Oproti SeRQL nabízí SPARQL možnost setřídit výsledky podle hodnot nějaké proměnné

```
PREFIX trip:    <http://www.muaddib.wz.cz/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:    <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?x ?y
WHERE { ?x rdf:type trip:Trip.
        OPTIONAL {?x trip:OndrasRating  ?y}
        }
        ORDER BY DESC(?y)
```

Řádky, kde proměnná ?y má hodnotu NULL, jsou umístěny na konci výsledků.

4.5.3. Shrnutí

Ve stručnosti jsme si přiblížili dva dotazovací jazyky pro RDF. Oba jazyky používají známe klauzule SELECT, FROM, WHERE, byť SPARQL dává klauzuli FROM jiný význam.

Také je zde možnost omezovat hodnoty např. pomocí regulárních výrazů nebo aritmetických porovnání. Chybí zde agregace výsledků, např. součet hodnot, průměr apod.

Největší rozdíl oproti SQL je zavedení pojmu proměnné. Z důvodu reprezentace dat jako grafu je dotazování prováděno na nejmenších částech grafu - na trojicích. Jednotlivé složky trojic jsou proměnné. Na tyto proměnné můžeme klást podmínky podobně, jako na buňky tabulek v relačním modelu.

Pro naše účely bude stačit takto stručný úvod, protože budeme využívat pouze elementární dotazy.

4.6. Závěr kapitoly

Formát dat RDF je častý způsob uložení dat v doméně sémantického webu. Proto se jím budeme dále zabývat. Zde jsme si ukázali základy syntaxe a některé nástroje, které umožňují RDF data ukládat a poté k nim přistupovat pomocí dotazovacího jazyka.

5. OWL

5.1. Základy OWL

OWL nabízí bohatší slovník a sémantiku pro popis vztahů v nějaké oblasti než RDFS. Vychází z RDFS a využívá jeho vlastnosti, dále však přidává nové možnosti pro přesnější definici tříd a vlastností.

Navíc OWL rozšiřuje koncept definovaných tříd oproti RDFS o možnost přidání podmínek. Tyto podmínky jsou přiřazeny ke třídě *C*. Pokud je podmínka *p* nutná, každé individuum třídy ji musí splňovat. Nutné podmínky jsou určeny vlastností `<rdfs:subClassOf>`. Pokud je to podmínka *p* nutná a postačující, všechny zdroje, které podmínku *p* splňují, jsou označeny jako individua této třídy *C*. Nutné a postačující podmínky jsou určeny vlastností `<owl:equivalentClass>`.

OWL je rozděleno do třech úrovní

1) OWL Lite
2) OWL DL
3) OWL Full

OWL Lite má nejmenší výrazovou schopnost, OWL Full největší.

Budeme používat jmenný prostor owl:, což je zkratka za http://www.w3.org/2002/07/owl #.

5.1.1. Jazyk OWL Lite

5.1.2. Jazyk tříd

Třídy jsou v OWL převzaty z RDFS, čímž je zachována zpětná kompatibilita. Platí, že všechny korektní OWL soubory jsou korektní RDFS soubory.

<rdfs:Class>

<rdfs:subClassOf> se používá také pro označení nutných podmínek.

<rdf:Property>

<rdfs:domain>

<rdfs:range>

<owl:Individual> je třída individuí.

5.1.3. Jazyk rovnocennosti

<owl:equivalentClass> vyjadřuje rovnocennost dvou tříd. Množina individuí obou tříd je stejná. Tato vlastnost je použita pro označení postačující podmínky.
<trip:Human> **<owl:equivalentClass>** <foaf:Person>.

<owl:equivalentProperty> vyjadřuje rovnocennost dvou vlastností.

<owl:sameAs> označuje dvě shodná individua. Tato vlastnost může být použita při odkazu do jiných ontologií. Tímto způsobem lze vytvářet distribuované ontologie, které pak mohou být mezi sebou provázány právě vlastnostmi **<owl:sameAs>**.

Pokud již máme ontologii vytvořenou a zjistíme, že některá již použitá individua se vyskytují i v jiných ontologiích, můžeme je pomocí vlastností **<owl:sameAs>**, **<owl:equivalentProperty>** a **<owl:equivalentClass>** propojit.

<owl:differentFrom> označuje, že podmět je rozdílný od předmětu.

<owl:AllDifferent> je seznam množin individuí, která jsou po dvou rozdílná. Je to zkratka za více vlastností <owl:differentFrom>.
<owl:distinctMembers> je množina individuí, která jsou po dvou rozdílná. Tato množina je prvkem seznamu <owl:AllDifferent>.

5.1.4. Jazyk typů vlastností

<owl:ObjectProperty> je třída vlastností, které mají jako předmět individua.
<trip:hasTerrain> <rdf:type> <**owl:ObjectProperty**>

<owl:DatatypeProperty> je třída vlastností, které mají jako předmět literál.
<trip:numberOfDays> <rdf:type> <**owl:equivalentClass**>

<owl:inverseOf> označuje dvě inverzní vlastnosti.
<trip:isEndingCityOf> <**owl:inverseOf**> <trip:hasEndingCity>

<owl:TransitiveProperty> označuje transitivní vlastnost.
<owl:SymmetricProperty> označuje symetrickou vlastnost.

<owl:FunctionalProperty> označuje vlastnost, která je funkcí. Jeden podmět může být touto vlastností spojen s nejvýše jedním předmětem.
<trip:hasStartingCity> <rdf:type> <**owl:FunctionalProperty**>

<owl:InverseFunctionalProperty> označuje vlastnost, jejíž inverzní vlastnost je funkce.
<trip:isStartingCityOf> <rdf:type> <**owl:InverseFunctionalProperty**>

5.1.5. Jazyk průniku tříd

<owl:intersectionOf> definuje novou třídu jako průnik stávajících tříd.

5.1.6. Jazyk omezení vlastností

<owl:Restriction> je základní třída všech omezení. Všechna omezení jsou individui této třídy.
<_:A18> <owl:type> <owl:Restriction>
<owl:onProperty> označuje vlastnost, která má být omezena.
<_:A18> <**owl:onProperty**> <trip:hasFoods>

<owl:allValuesFrom> vyžaduje, aby všechny předměty omezované vlastnosti náležely do množiny definované předmětem.
<_:A18> <**owl:allValuesFrom**> <trip:FoodWithoutPreparation>

<**owl:someValuesFrom**> vyžaduje, aby alespoň jeden předmět omezované vlastnosti náležel do množiny definované předmětem.

5.1.7. Jazyk četností

<**owl:minCardinality**> stanovuje nejmenší počet individuí připojených touto vlastností. V OWL Lite toto omezení může být pouze 0 nebo 1.

<**owl:maxCardinality**> stanovuje největší počet individuí připojených touto vlastností. V OWL Lite toto omezení může být pouze 0 nebo 1.

<**owl:cardinality**> stanovuje přesnou četnost této vlastnosti. Opět v OWL Lite toto omezení může být pouze 0 nebo 1.

5.1.8. Jazyk informací o ontologii

<**owl:Ontology**> je třída, která definuje ontologie. Jedna instance této třídy by měla být na začátku každé ontologie.
<**owl:Imports**> umožňuje načíst jinou ontologii. Načítaná ontologie je specifikována jejím URI, předpokládá se, že je to typ `<owl:Ontology>`. `<owl:Imports>` je transitivní vlastnost, tedy všechny ontologie, načtené načítanou ontologií, budou také načtené.

Ontologie jsou importovány jako celek. Není možné převzít pouze část ontologie.

Použitelnost vlastnosti `<owl:Imports>` je zpochybněna v [ECONNECTIONS]. Autoři zde navrhují nové vlastnosti a třídy, které umožňují načtení pouze části ontologie. Navíc je zde popsán algoritmus pro rozdělení existujících ontologií na vzájemně provázané části. Tyto části jsou mezi sebou provázány právě novými vlastnostmi.

Editor ontologií SWOOP již má v sobě integrované tyto vlastnosti a dokonce i algoritmus pro rozčlenění existujících ontologií. Na [ECONNECTIONS] lze také nalézt odkaz na velké známé ontologie, např. [GALEN], rozdělené pomocí SWOOP na více částí.

Další výrazy jsou dostupné již pouze v OWL DL a Full.

5.1.9. Definice tříd

<**owl:oneOf**>, `<owl:dataRange>` umožňují definovat třídy výčtem prvků.
<**owl:disjointWith**> označuje dvě třídy jako různé.
`<trip:MenuForOneDay>` <**owl:disjointWith**> `<trip:MenoForOneWeek>`

5.1.10. Definice tříd množinovými operacemi

<**owl:unionOf**> umožňuje definovat novou třídu jako sjednocení dvou stávajících tříd.
<**owl:complementOf**> umožňuje definovat novou třídu jako doplněk stávající třídy.
<**owl:intersectionOf**> umožňuje definovat novou třídu jako průnik dvou stávajících tříd.

5.1.11. Omezení předmětů vlastnosti

<**owl:hasValue**> omezuje možné hodnoty vlastnosti na jedno individuum.

5.1.12. Příklad

Ukážeme si část ontologie, která definuje podmínku a tuto podmínku poté aplikuje na třídu. Podmínka nám říká, že v jídelníčku na jednodenní výlet mohou být pouze jídla, která nevyžadují přípravu.

Podmínka je definovaná tímto RDF grafem

```
<_:A18>  <owl:type> <owl:Restriction>
<_:A18>  <owl:onProperty> <trip:hasFoods>
<_:A18>  <owl:allValuesFrom> <trip:FoodWithoutPreparation>
```

Nyní podmínku aplikujeme na třídu pomocí vlastnosti `<rdfs:subClassOf>`

```
<trip:MenuForOneDay>  <rdfs:comment> "Jídelníček na jeden den.
Nepředpokládá se, že by se vařilo, proto jsou povolena pouze jídla bez
přípravy."^^xsd:string
<trip:MenuForOneDay> <rdfs:subClassOf> <_:A18>
<trip:MenuForOneDay> <rdf:type> <owl:Class>
<trip:MenuForOneDay> <rdf:subClassOf> <trip:FoodMenu>
<trip:MenuForOneDay> <owl:disjointWith> <trip:MenoForOneWeek>
<trip:MenuForOneDay> <owl:disjointWith> <trip:MenuForWeekend>
```

5.2. Vytváření ontologií

Nyní již jsme připraveni vytvořit novou ontologii. Tato ontologie bude pokrývat obor aktivit turistického oddílu. Její hlavní motivací je vytvořit prostor pro definování výletních lokalit a jejich atributů. Tato ontologie bude následně použita jako základ pro hledání optimální výletní lokality pro tento oddíl.

Pokud budeme chtít najít podobnou ontologii na internetu, např. [TRAVEL], zjistíme, že takové existují. Bohužel ovšem popisují problematiku výletů z pohledu turisty, který se chce někde ubytovat a očekává, že mu dané místo nabídne nějaký program. Specifik uvažovaných turistických oddílů je několik. První z nich je, že se neubytovává v hotelech, ale spí ve volné přírodě, převážně pod celtou, zřídka v kempu. Další je, že turistické oddíly nevyhledávají rozptýlení v daném místě, ale zajímá je v jakém terénu dané místo nachází. Všechna tato specifika se musí promítnout do výsledné ontologie.

Pokud chceme vytvořit komplexní ontologii, práci nám velmi ušetří použití některého z nástrojů pro tvorbu ontologií. Mezi nejznámější patří Protégé [PROTEGE] a SWOOP [SWOOP]. Protégé má velmi mnoho zásuvných modulů a je velmi hezky graficky navržen. SWOOP má přímo zabudovaný OWL reasoner a podporuje zmiňované dělení ontologií.

Jelikož má OWL pevnou syntax, není problém ontologii vytvořenou v jednom nástroji otevřít v jiném. Tím je zajištěna velká přenositelnost a modularita ontologií.

Níže uvedený postup není rozepsán do detailů. Výslednou ontologii lze nalézt na přiloženém CD. Při tvorbě ontologie jsme použili editor Protégé, SWOOP pak pouze pro kontrolu konzistence.

Při vytváření ontologie je vhodné začít definicí tříd. Základní třídy vyplynou poměrně jasně z prvního prozkoumání problému. V našem případě to budou třídy výletních lokací, terénu, kde se lokace nachází, měst, které se poblíž nacházejí, a o možnostech získání pitné vody ze studánek a potoků. Dále budeme chtít uchovávat informace o jednotlivých uskutečněných výletech, o počasí a lidech, kteří na výletu byli. Bude užitečné mít informace o obchodech ve městech (o jejich otevírací době) a o tom, jakým dopravním prostředkem se půjde do daného města dostat. Nakonec ještě přidáme strukturu jídelníčku na výletech.

Zanesením těchto tříd do Protégé získáme toto úvodní schéma.

5.2.1. Schéma

Nyní vytvoříme podtřídy k některým třídám. První z nich bude terén, který rozdělíme na hory, louky, lesy, města a vodní plochy. Dále rozdělíme třídy jídla, jídelních menu, přepravy a podnebních podmínek. Vhodnost rozdělení na podtřídy může vyplynout až v průběhu vytváření ontologie, případně až při jejím používání. Např. v našem případě se rozdělení počasí ukázalo jako vhodné až při samotném vytváření individuí. V původním schématu bylo nutno psát dohromady teplotu i oblačnost, což neúměrně zvyšovalo počet nutných individuí, z nichž většina by zůstala nevyužita.

Po návrhu tříd zjistíme, jaké jsou mezi třídami vztahy. Tyto vztahy zapíšeme pomocí vlastností. Vytváření vlastností většinou probíhá současně s tvorbou tříd, ideální postup je však oba procesy oddělit. Vyhneme se tím dvojitým změnám, kdy změna v definici třídy si vynutí změny ve vlastnostech.

V závěrečné fázi potom vytvoříme omezení na třídy, případně vytvoříme nové definované třídy. Tyto budou hrát roli uživatelských pohledů. Náležení do těchto tříd bude zjišťovat odvozovací stroj.

5.3. Závěr kapitoly

Jazyk OWL jsme studovali hlavně proto, že umožňuje přesnější zachycení vztahů v určité oblasti. Proto je vhodnější pro vytváření ontologií než čisté RDFS, ze kterého OWL vychází. Dále jsme si ukázali příklad vytvoření ontologie výletních míst, která bude určena turistickému oddílu. Aplikaci, která tuto ontologii využívá, ukážeme v kapitole 11 v praktické části práce.

6. Zavedení vícehodnotové predikátové logiky do RDF, RDFS a OWL

Chceme použít RDF a RDFS pro vyjádření ohodnocených faktů a pravidel z logického programování. Navrhneme dva možné přístupy a zvážíme možné přínosy, které by RDFS a OWL mohlo do logického programování přinést.

Logické programování s vícehodnotovou logikou je jeden z možných modelů uživatelských preferencí. Zavedení tohoto modelu do RDF nám umožní provádět dotazy s uživatelskými preferencemi. Tyto preference budou modelované pravdivostními hodnotami.

Navíc bychom chtěli, aby se níže vytvořený model logického programu dal vnořit do již existujících RDF dat. Tím se umožní využití logického přístupu k existujícím datům. Konkrétní způsob zanoření může být využití vlastnosti `<owl:Imports>` nebo pomocí E-Connections. V příloze 4 uvedeme RDFS soubor popisující logický program, čímž tuto vlastnost zajistíme.

6.1. Přímé modelování pravidel

První přístup pro zavedení logického programování do RDF je vytvoření nových vlastností a tříd, které odpovídají ohodnoceným faktům a pravidlům. Jde v podstatě o doslovné přepsání pravidel a faktů do RDF, budeme ho označovat jako doslovný. Tyto nové vlastnosti a třídy umístíme do virtuálního jmenného prostoru mvlp: . Individua budou prvky nové třídy `<mvlp:U>`, pravdivostní hodnoty pak `<mvlp:TruthValue>` a proměnné `<mvlp:Variable>`.

Druhý přístup bude využívat stejné třídy a vlastnosti jako doslovný přístup. Ovšem bude umožňovat využití již existujících RDF dat. Označíme ho uchovávající.

Rozhodli jsme se nové vlastnosti a třídy pojmenovat anglicky, jelikož všechny standardy jsou psané v angličtině. Tím se zajistí určitá mezinárodní přenositelnost.

6.1.1. Modelování predikátů

Chceme zapsat fakt $P(\alpha_1,...,\alpha_k):r$ do RDF.

Zavedeme novou třídu `<mvlp:Predicate>` pro predikáty. Jednotlivé predikáty budou podtřídy třídy `<mvlp:Predicate>`, ohodnocené fakty potom individua těchto podtříd. Tato třída bude splňovat následující požadavky

```
<mvlp:Predicate> <rdf:type> <rdfs:Class>
<mvlp:Predicate> <rdf:_1> <mvlp:U>
<mvlp:Predicate> <rdf:_1> <mvlp:Variable>
<mvlp:Predicate> <rdf:_2> <mvlp:U>
<mvlp:Predicate> <rdf:_2> <mvlp:Variable>
<mvlp:Predicate> <rdf:_3> <mvlp:U>
<mvlp:Predicate> <rdf:_3> <mvlp:Variable>

...
```

6.1.2. Modelování hodnocení

Dále zavedeme novou vlastnost `<mvlp:Rating>`, která bude realizovat vlastní ohodnocení faktu trojicí

```
<mvlp:Predicate> <mvlp:Rating> <mvlp:T>
```

Vlastnost `<mvlp:Rating>` definujeme těmito trojicemi

```
<mvlp:Rating> <rdf:type> <rdf:Property>
<mvlp:Rating> <rdfs:domain> <mvlp:Predicate>
```

Dále pokud je T konečná, přidáme
`<mvlp:Rating> <rdfs:range> <mvlp:TruthValue>`

Pokud je T nekonečná množina, obor hodnot vlastnosti `<mvlp:Rating>` bude literál vhodného typu, např. xsd:decimal
`<mvlp:Rating> <rdfs:range> <xsd:decimal>`

Tento přístup nám umožní lehce modelovat i více uživatelských hodnocení. Všechny vlastnosti, jejichž nadtřída je `<mvlp:Rating>`, budou také považovány za hodnocení.
Např. trojicí
`<mvlp:OndrasRating> <rdfs:subPropertyOf> <mvlp:Rating>`
jsme nadefinovali hodnocení Ondřeje. Všechny trojice, které vyjadřují Ondřejovo hodnocení budou brány jako trojice vyjadřující hodnocení.

6.1.3. Modelování pravidel

Chceme zapsat v RDF pravidlo $(H \leftarrow B, r)$, kde $H \in AF, r \in T$ a B je formule ve tvaru $A \,\&_1\, B \,\&_2\, C..., A, B, C \in AF$. Tuto implikaci budeme modelovat pomocí nové třídy `<mvlp:Rule>`.
Konjunkci $A \,\&_1\, B \,\&_2\, C...$ zapíšeme pomocí nové třídy `<mvlp:Conjunctor>`.
Pro tyto třídy bude platit

```
<mvlp:Rule> <mvlp:head> <mvlp:Predicate>
<mvlp:Rule> <mvlp:body> <mvlp:Conjunctor>

<mvlp:Conjunctor> <rdf:_1> <mvlp:Predicate>
```
nebo
```
<mvlp:Conjunctor> <rdf:_1> <mvlp:Conjunctor>

<mvlp:Conjunctor> <rdf:_2> <mvlp:Predicate>
```
nebo
```
<mvlp:Conjunctor> <rdf:_2> <mvlp:Conjunctor>
```

Použitím této definice se zpřehlední uzávorkování těla konjunktoru.

Dále budeme muset rozšířit i definiční obor `<mvlp:Rating>` o
`<mvlp:Rating> <rdfs:domain> <mvlp:Rule>`

Podtřídy třídy `<mvlp:Rule>` budou konkrétní implikátory, např. produktový implikátor. To samé platí pro třídu `<mvlp:Conjunctor>`.
Jednotlivá ohodnocená pravidla potom budou instance těchto podtříd.

Nakonec vytvoříme třídu `<mvlp:AggregatingOperator>` pro agregační operátory. Agregační operátory jsou velmi volně definované. Jejich hierarchie zřejmě bude složitější než u implikátorů a konjunktorů. Lze např. definovat podtřídu `<mvlp:WeightAverageOperator>`, jejíž další podtřídy budou vážené průměry s různými váhami. Dále by bylo možné definovat další podtřídy podle arity agregátoru. Tuto problematiku necháváme otevřenou pro další zkoumání.
Jednotlivé parametry agregačního operátoru budou definované opět pomocí vlastností `<rdf:_1>,<rdf:_2>,...` takto
`<mvlp:AggregatingOperator> <rdf:_1> <mvlp:Predicate>`

```
<mvlp:AggregatingOperator> <rdf:_2> <mvlp:Predicate>
<mvlp:AggregatingOperator> <rdf:_3> <mvlp:Predicate>
...
```

Tím jsme namodelovali ohodnocené fakty i pravidla vícehodnotového logického programu do RDF a RDFS.

6.2. Využití sémantiky RDF/RDFS

Pokud bychom nechtěli zavádět nové vlastnosti do RDF, musíme využít pouze ty výrazové prostředky, které jsme zmínili v kapitole o RDF, RDFS a OWL. Nicméně přidání některých vlastností se zřejmě nevyhneme.

Množina individuí nyní bude třída `<rdfs:Resource>`, od které jsou všechny další třídy zděděné.

Pro konečnou množinu pravdivostních hodnot *T* můžeme opět zavést třídu `<mvlp:TruthValue>` nebo používat literál pro nekonečnou množinu.

6.2.1. Modelování ohodnocení

Ohodnocené fakty budeme modelovat stejně jako v doslovném přístupu. Zde nemůžeme využít žádnou z vlastností RDF, musíme vytvořit novou vlastnost `<mvlp:Rating>`. Změníme ovšem definici definičního oboru

```
<mvlp:Rating> <rdfs:domain> <rdfs:Resource>.
```

Oproti definici faktu v predikátové logice $P(x,...,z):r$ jsme rozšířili definici hodnocení i na jednotlivá individua. Toto rozšíření může být relevantní, jelikož vyjadřuje oblíbenost konkrétního objektu *x* pro uživatele. Ohodnocení výletu

```
<trip:Brdy050501> <mvlp:OndrasRating> "0.9"
```

vyjadřuje přímou oblíbenost výletu pro Ondru. To, jak se zdál Ondrovi výlet dlouhý, může, ale také nemusí, souviset s hodnocením výletu. Takto rozšířená definice ohodnocení umožňuje zaznamenání obecnějších uživatelských preferencí. Uživatel může mít názor na libovolnou věc, tedy třeba i na predikát *P*. Hodnocení predikátu potom vyjadřuje, jak je pro uživatele důležitý.

Jiný způsob nazírání na hodnocení je považovat hodnocení jako vícehodnotový unární predikát. Přesné rozebrání smyslu a různých použití této definice bude podrobeno dalšímu detailnímu zkoumání, zde se spokojíme s tím, že tato definice poskytuje dostatečnou flexibilitu, které obecně preference mohou nabývat.

6.2.2. Modelování predikátů

Pokud bychom se omezili na binární predikáty ve tvaru $P(x,y)$, potom bychom vystačili s modelování faktů pomocí reifikací. Opět si uvědomíme, že nyní je univerzum U pro nás množina všech zdrojů. Binární predikáty $P(x,y)$ budeme chápat jako individua třídy `<rdf:Property>`. RDF trojice tedy pro nás jsou neohodnocené fakty

```
<foo:X> <foo:P> <foo:Y>
```

Reifikací této trojice pomocí vlastnosti `<mvlp:Rating>` bychom dodali hodnocení *r* a tím celý ohodnocený fakt.

Unární predikáty $P(x):r$ můžeme modelovat přímo trojicí

```
<mvlp:X> <mvlp:P> <T:r>
```

V závěru si ukážeme modelování více-árních predikátů pomocí třídy `<mvlp:Predicate>`.

6.2.3. Modelování pravidel

Ohodnocené pravidlo se skládá ze tří částí - těla, hlavy a ohodnocení. Smysl pravidel je v tom, že nám dovolují spočítat ohodnocení hlavy pravidla na základě ohodnocení jednotlivých částí těla pravidla.

Toto můžeme převést do RDF - zde jsou jednotlivé objekty provázány vlastnostmi. Vezměme si např. třídu `<trip:SunnyTrip>` z příkladu v kapitole o OWL. Tato třída je spojena se třídami `<trip:Weather>, <trip:Human>,<trip:TripLocation>,<trip:City>` a dalšími.

Pro vypočítání míru náležení individua o do třídy `<trip:SunnyTrip>` je možné použít hodnocení individuí, která jsou s individuem o spojená. Ovšem neznáme typ konjunktorů ani implikátorů, tyto informace budeme muset do RDF dodat. Navíc nemusíme chtít používat všechny vlastnosti, které mají definiční obor `<trip:SunnyTrip>`.

Chtěli bychom zapsat pravidlo

$$Type(x, SunnyTrip) \leftarrow hasWeather(x, y) \,\&\, Type(y, SunnyWeather) : 0.7$$

Chtěli bychom se nejlépe zbavit i nutnosti používat proměnné. Problém nastává při unifikaci proměnných v pravidle a jeho hlavě. Pravidla nepůjde zapsat pouze jako trojice

```
<trip:RuleSunnyTrip> <mvlp:head> <trip:SunnyTrip>
<trip:RuleSunnyTrip> <mvlp:body> <mvlp:Conjunctor1>
<mvlp:Conjunctor1> <rdf:_1> <trip:hasWeather>
<mvlp:Conjunctor1> <rdf:_2> <trip:SunnyWeather>.
```

Zde nám chybí informace o tom, jaká proměnná má mít vlastnost SunnyWeather. Z tohoto důvodu musíme využít proměnné. Zde si uvědomíme vztah s deskripční logikou, která proměnné nepoužívá, a tvoří teoretický základ pro popis omezení v jazyce OWL. Náš zápis logického programu proměnné využívat musí. Oproti deskripční logice v logickém programování chybí kvantifikátory. Deskripční logiku i logické programování lze použít pro odvozování nových znalostí, mají tedy v jistém smyslu stejný cíl. Více o deskripční logice lze nalézt na [DL].

Abychom se vyhnuli dlouhým zápisům pravidel, budeme využívat reifikace.

Nejprve zapíšeme hlavu pravidla takto

```
<_:Triple1> <rdf:type> <rdf:Statement>
<_:Triple1> <rdf:Subject> <mvlp:X>
<_:Triple1> <rdf:Predicate> <rdf:type>
<_:Triple1> <rdf:Object> <trip:SunnyTrip>
<trip:RuleSunnyTrip> <mvlp:head> <_:Triple1>
```

Výše uvedenými trojicemi jsme řekli, že v hlavě pravidla bude tvrzení, že proměnná X je typu SunnyTrip.

```
<_:Triple2> <rdf:type> <rdf:Statement>
<_:Triple2> <rdf:Subject> <mvlp:X>
<_:Triple2> <rdf:Predicate> <trip:hasWeather>
<_:Triple2> <rdf:Object> <mvlp:Y>
<mvlp:Conjunctor1> <rdf:_1> <_:Triple2>

<_:Triple3> <rdf:type> <rdf:Statement>
<_:Triple3> <rdf:Subject> <mvlp:Y>
<_:Triple3> <rdf:Predicate> <rdf:type>
<_:Triple3> <rdf:Object> <trip:SunnyWeather>
<mvlp:Conjunctor1> <rdf:_2> <_:Triple3>
```

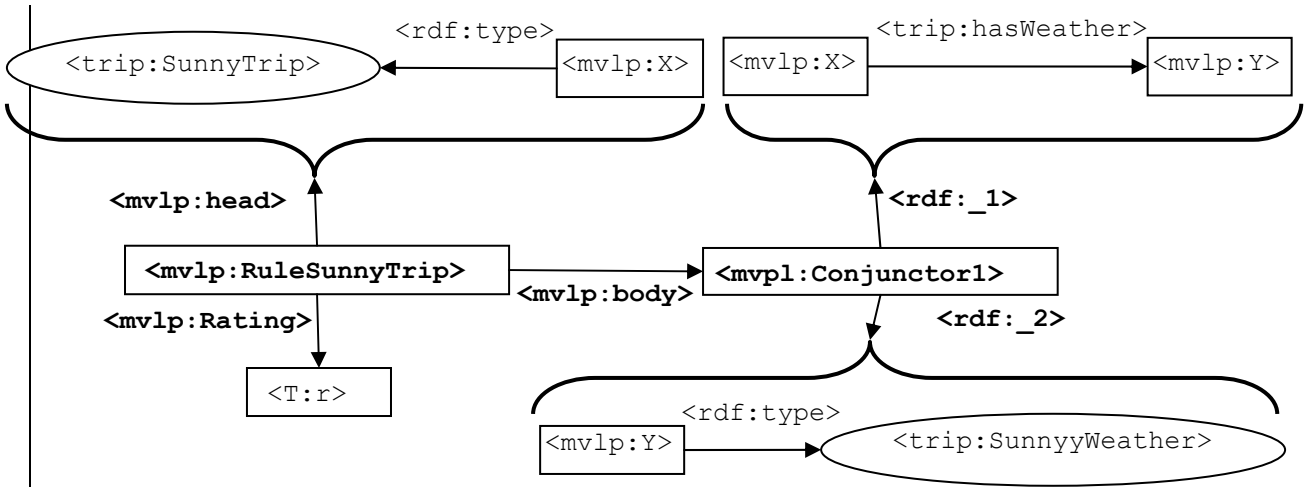
Tímto jsme definovali tělo pravidla. Jde o konjunkci tvrzení, že X má počasí Y a Y je typu SunnyWeather.

Nakonec přiřadíme implikátoru tělo pravidla a hodnocení pravidla.

```
<trip:RuleSunnyTrip> <mvlp:body> <mvlp:Conjunctor1>

<trip:RuleSunnyTrip> <mvlp:Rating> "0.7"
```

Celá struktura výsledného pravidla je na obrázku 6.2.3.1



6.2.3.1. Obrázek

Reifikované trojice samozřejmě mohou být libovolného tvaru, nemusí být použity pouze proměnné. Místo proměnné lze použít libovolné individuum.

Obecná definice ohodnocených pravidel tedy bude

```
<mvlp:Rule> <mvlp:head> <rdfs:Statement>

<mvlp:Rule> <mvlp:body> <mvlp:Conjunctor>
<mvlp:Rule> <mvlp:body> <mvlp:AggregatingOperator>
```

Pro binární predikáty použijeme

```
<mvlp:Conjunctor> <rdf:_1> <rdf:Statement>
<mvlp:Conjunctor> <rdf:_2> <rdf:Statement>
```

Nakonec pro složené konjunktory přidáme

```
<mvlp:Conjunctor> <rdf:_1> <mvlp:Conjunctor>
<mvlp:Conjunctor> <rdf:_2> <mvlp:Conjunctor>
```

Pro implementaci agregačních operátorů pouze rozšíříme jejich definici na

```
<mvlp:AggregatingOperator> <rdf:_1> <rdfs:Statement>
<mvlp:AggregatingOperator> <rdf:_2> <rdfs:Statement>
<mvlp:AggregatingOperator> <rdf:_3> <rdfs:Statement>
...
```

Dále je možné agregátory rozšířit o parametry datových typů. Např.

```
<mvlp:AggregatingOperator> <rdf:_1> <xsd:Decimal>
```

6.2.4. Shrnutí

Pokud se v programu omezíme na binární predikáty, lze tento program modelovat přímo v existujících RDF datech. Vytvoříme konjunktory a implikátory, které budou modelovat pravidla.

Ohodnocené fakty budou vlastní trojice v RDF datech, reifikované vlastností `<mvlp:Rating>`. Pokud reifikace není přítomná, můžeme předpokládat, že trojice platí na 1. Tím umožníme využití existujících dat pro výpočty logického programu.

6.2.5. Příklad

Přepíšeme příklad z kapitoly 3.4.6 do RDF.
Program měl následující strukturu:

a) $SunnyTrip(x) \leftarrow_p @_1(hasWeather(x,y), SunnyWeather(y)) : 0.9$

b) $SunnyWeather(x) \leftarrow_G hasAtmosphericConditions(x, Sunny) : 0.9$

c) $hasWeather(Brdy050501, OvercastsMildTemperature) : 1$

d) $hasAtmosphericConditions(OvercastsMildTemperature, Overcasts) : 0.9$

e) $hasAtmosphericConditions(OvercastsMildTemperature, Sunny) : 0.1$

f) $hasWeather(Polsko030810, SunnyHighTemperature) : 1$

g) $hasAtmosphericConditions(SunnyHighTemperature, Sunny) : 0.9$

Nejdříve převedeme fakty c)-g). Pro stručnost převedeme pouze první dva fakty.

```
<_:Triple1> <rdf:type> <rdf:Statement>
<_:Triple1> <rdf:Subject> <trip:Brdy050501>
<_:Triple1> <rdf:Predicate> <trip:hasWeather>
<_:Triple1> <rdf:Object> <trip:OvercastsMildTemperature>
<_:Triple1> <mvlp:Rating> "1"

<_:Triple2> <rdf:type> <rdf:Statement>
<_:Triple2> <rdf:Subject> <trip:OvercastsMildTemperature>
<_:Triple2> <rdf:Predicate> <trip:hasAtmosphericConditions>
<_:Triple2> <rdf:Object> <trip:Overcasts>
<_:Triple2> <mvlp:Rating> "0.9"
```

Pravidlo a) odpovídá pravidlu použitém v příkladě v textu výše. Převedeme tedy pouze pravidlo b)

```
<trip:RuleSunnyWeather> <rdf:type> <mvlp:Rule>

<_:Triple8> <rdf:type> <rdf:Statement>
<_:Triple8> <rdf:Subject> <mvlp:X>
<_:Triple8> <rdf:Predicate> <rdf:type>
<_:Triple8> <rdf:Object> <trip:SunnyWeather>
```

Triple8 odpovídá hlavě pravidla - proměnná X je typu SunnyWeather.

```
<trip:RuleSunnyWeather> <mvlp:head> <_:Triple8>


<_:Triple9> <rdf:type> <rdf:Statement>
<_:Triple9> <rdf:Subject> <mvlp:X>
<_:Triple9> <rdf:Predicate> <trip:hasAtmosphericConditions>
<_:Triple9> <rdf:Object> <trip:Sunny>
```

Triple9 odpovídá tělu pravidla - proměnná X má povětrnostní podmínky slunečno.

```
<trip:RuleSunnyWeather> <mvlp:body> <_:Triple9>
```

Nakonec přidáme hodnocení celé implikace

```
<trip:RuleSunnyWeather> <mvlp:Rating> "0.9"
```

Poznámka

Ternární a více-ární predikáty budeme modelovat pomocí třídy `<mvlp:Predicate>`. Toto řešení je ekvivalentní rozdělení těchto predikátů na více binárních predikátů. Ukážeme si to na příkladu ternárního predikátu $P(x,y,z)$. Chceme jej vyjádřit pomocí binárních predikátů P_1, P_2 a P_3. Vytvoříme nové virtuální individuum $a_{P(x,y,z)}$. Potom predikát $P(x,y,z)$ zapíšeme pomocí třech predikátů $P_1(a_{P(x,y,z)},x)$, $P_2(a_{P(x,y,z)},y)$ a $P_3(a_{P(x,y,z)},z)$.

Porovnejme tří výše uvedené predikáty s trojicemi

```
<mvlp:P_x_y_z> <rdf:type> <mvlp:Predicate>
<mvlp:P_x_y_z> <rdf:_1> <mvlp:x>
<mvlp:P_x_y_z> <rdf:_2> <mvlp:y>
<mvlp:P_x_y_z> <rdf:_3> <mvlp:z>
```

Tvrzení

Rozložení ternárního predikátu $P(x,y,z)$ na tři binární je ekvivalentní zápisu pomocí třídy `<mvlp:Predicate>`.

Důkaz

Vytvoříme mapování mezi třemi binárními predikáty a RDF grafem. Mapování je zapsáno v tabulce

P_1	`<rdf:_1>`
P_2	`<rdf:_2>`
P_3	`<rdf:_3>`
$a_{P(x,y,z)}$	`<mvlp:P_x_y_z>`

6.2.5.1. Tabulka

Tvrzení i důkaz lze přímo zobecnit na více-ární predikáty.

Z uvedené ekvivalence plyne, že je zbytečné zavádět nové konstrukce pro rozložení více-árních predikátů. Budeme tedy využívat již zavedenou třídu `<mvlp:Predicate>`. Změníme ovšem její definici na

```
<mvlp:Predicate> <rdf:type> <rdfs:Class>
<mvlp:Predicate> <rdf:_1> <rdf:Resource>
<mvlp:Predicate> <rdf:_2> <rdf:Resource>
<mvlp:Predicate> <rdf:_3> <rdf:Resource>
...
```

Pro více-ární predikáty přidáme do axiomů tato pravidla

```
<mvlp:Rule> <mvlp:head> <rdfs:Predicate>
<mvlp:Conjunctor> <rdf:_1> <mvlp:Predicate>
<mvlp:Conjunctor> <rdf:_2> <mvlp:Predicate>
<mvlp:AggregatingOperator> <rdf:_1> <mvlp:Predicate>
...
```

6.3. Srovnání obou přístupů

Doslovný přístup mechanicky přepisuje fakty a pravidla vícehodnotového logického programu do RDF. Tím ovšem přicházíme o všechny možnosti, jež nám RDF nabízí. Jiná práce než výpočet logického programu s takto uloženými daty by byla nesnadná. Dále se plně ztrácí sémantika dat a požadavek na použitelnost modelu pro již existující data zde není splněn. Pro jeho splnění bychom museli pro všechna individua vytvořit nové fakty.

Uchovávající přístup vychází z možností RDF. Umožňuje přímé využití všech mechanismů RDF a RDFS. Požadavek na vnoření do již existujících RDF dat je zde splněn. Stačí importovat níže popsanou ontologii a vytvořit pravidla. Ohodnocené fakty jsou trojice v existujícím RDF reifikované vlastností `<mvlp:Rating>`. Pokud reifikace chybí, předpokládá se plná platnost.

Hlavní výhoda uchovávajícího přístupu spočívá v zachování sémantiky RDF. Pokud navíc vytváříme program nad ontologií OWL, všechny bohaté prostředky OWL nám jsou přímo přístupné v programu. Mezi nejpodstatnější patří tzv. inference. Jde o proces, kdy se náležení instance do nějaké třídy určuje dynamicky za běhu programu, přičemž toto náležení nemusí být explicitně zaneseno v datech. Tuto inferenci většinou provádí úložiště dat a tím se vlastně automaticky generují nové fakty.

Na závěr musíme zdůraznit, že RDF a RDFS slouží pouze k uchování vlastního programu. Výpočet dotazu se provádí ve vyšší vrstvě, která tato data bude využívat. Ovšem logický program nemusí být jediný, kdo daná data využívá. Druhý přístup ponechává data v sémantičtější formě.

6.4. Mapování na RuleML

RuleML je způsob zápisu pravidel logického (dvouhodnotového) programu do XML, bližší informace lze nalézt na [RULEML]. Převod RuleML do RDF je vytvořen v [RDFRULEML], nicméně není dokumentován a je značně nepřehledný.

Syntax RuleML je podobná doslovnému přístupu zápisu pravidel do RDF, definovaném v kapitole 6.1, nicméně použité značení je stručnější. V této kapitole si ukážeme vztah mezi dvouhodnotovým RuleML a zápisem vícehodnotových pravidel s využitím sémantiky RDF. Při převodu z RDF do RuleML se omezíme na binární predikáty.

Použijeme opět příklad pravidel z kapitoly 3.4.6. Převedeme ho do RuleML a poté si ukážeme mapování jednotlivých XML značek na RDF vlastnosti.

a) $SunnyTrip(x) \leftarrow_p @_1(hasWeather(x,y), SunnyWeather(y)) : 0.9$

b) $SunnyWeather(x) \leftarrow_G hasAtmosphericConditions(x, Sunny) : 0.9$

c) $hasWeather(Brdy050501, OvercastsMildTemperature) : 1$

d) $hasAtmosphericConditions(OvercastsMildTemperature, Overcasts) : 0.9$

e) $hasAtmosphericConditions(OvercastsMildTemperature, Sunny) : 0.1$

f) $hasWeather(Polsko030810, SunnyHighTemperature) : 1$

g) $hasAtmosphericConditions(SunnyHighTemperature, Sunny) : 0.9$

6.4.1. Mapování faktů

Fakty se v RuleML zapisují pomocí značky Atom. Jeho podelementy jsou Rel, který označuje predikát a Ind a Var. Ind označuje individuum, Var potom proměnnou.

Ukážeme si, jak vypadá v RuleML fakt c)

```
<Atom>
        <Rel>hasWeather</Rel>
        <Ind>Brdy050501</Ind>
        <Ind>OvercastsMildTemperature</Ind>
</Atom>
```

RuleML pracuje s dvouhodnotovou logikou, proto není nikde zanesena pravdivostní hodnota. Pro fakty tedy použijeme tuto tabulku -

RuleML	RDF
Atom	libovolná trojice
Rel	název vlastnosti v trojici
Ind	podmět nebo předmět trojice, pokud nejsou typu `<mvlp:Variable>`
Var	zdroj typu `<mvlp:Variable>`

6.4.1.1. Tabulka

6.4.2. Mapování pravidel

Nejprve si ukážeme, jak vypadají pravidle a) a b) v RuleML.

a) $SunnyTrip(x) \leftarrow_p @_1(hasWeather(x,y), SunnyWeather(y)) : 0.9$

b) $SunnyWeather(x) \leftarrow_G hasAtmosphericConditions(x, Sunny) : 0.9$

Začneme pravidlem b)

```
<Implies>
        <head>
                <Atom>
                        <Rel>SunnyWeather</Rel>
                        <Var>x</Var>
                </Atom>
        </head>
        <body>
                <Atom>
                        <Rel>hasAtmosphericConditions</Rel>
                        <Var>x</Var>
                        <Ind>Sunny</Ind>
                </Atom>
        </body>
</Implies>
```

A nyní pravidlo a)

```
<Implies>
        <head>
                <Atom>
                        <Rel>SunnyTrip</Rel>
                        <Var>x</Var>
                </Atom>
        </head>
        <body>
            <And>
                  <Atom>
                    <Rel>hasWeather</Rel>
                    <Var>x</Var>
                    <Var>y</Var>
                  </Atom>
                  <Atom>
                    <Rel>SunnyWeather</Rel>
                    <Var>y</Var>
                  </Atom>
                </And>
        </body>
</Implies>
```

Zde vidíme, že RuleML nemá podporu pro více druhů implikace, konjunkce ani agregační funkce. Proto musíme pravidla redukovat na prostou implikaci a konjunkce. Pro binární predikáty tedy použijeme tuto tabulku -

RuleML	RDF
Implies	zdroj typu `<mvlp:Rule>`
Rel	název vlastnosti v trojici
Ind	podmět nebo předmět trojice, pokud nejsou typu `<mvlp:Variable>`
Var	zdroj typu `<mvlp:Variable>`

6.4.2.1. Tabulka

Rozšíření RuleML do vícehodnotové logiky není předmětem této práce, jelikož formát RDF je řádově více využívaný pro různé aplikace. Tím je umožněno zavedení vícehodnotových logických programů do aplikací, které s RDF pracují. RuleML je určeno výhradně k zápisu logických programů a nemá příliš jiné využití.

6.5. Závěr kapitoly

Navrhli jsme dva způsoby zápisu vícehodnotového logického programu do RDF a RDFS. Srovnali jsme tyto způsoby a vybrali jsme způsob, který ponechává existující RDF dokumenty a interpretuje je jako fakty. Pravidla programu je třeba zapsat pomocí navržených vlastností a zdrojů.

Dále jsme probrali vztah navržené notace a RuleML. Ujasnili jsme si analogické zdroje, ovšem největší nekompatibilita těchto dvou zápisů je dvouhodnotovost RuleML, které není navržené ve vícehodnotové logice. Z tohoto důvodu je také možné brát námi navrženou notaci jako rozšíření RuleML o vícehodnotovou logiku.

Na přiloženém CD je RDF soubor, kde jsou zaneseny vztahy, které jsme výše popsali. Je zde i příklad zapsaného vícehodnotového logického programu.

7. Uživatelské preference

7.1. Definice uživatelských preferencí

7.1.1. Uživatelské preference ve vícehodnotové logice

V kapitole 1 jsme si přiblížili vícehodnotovou logiku a vícehodnotové logické programování, s jejich pomocí budeme modelovat uživatelské preference. Preference jsou komplexní pojem, v různých oblastech má různý význam. Obecně můžeme preference chápat jako zobrazení z libovolné entity na množinu pravdivostních hodnot. My se zde omezíme na dvě části, které můžeme ve vícehodnotové logice nalézt.

První jsou ohodnocené fakty. Ve výrokové logice nám ohodnocený fakt říká, nakolik je daná výroková proměnná uživateli sympatická. V predikátovém počtu potom určuje míru vlastnosti nějakého individua, ovšem tuto míru mu přisuzuje uživatel. Příklad vlastnosti může být KrátkýVýlet, DobrýTerén apod.

Druhá část je zaznamenání procesu složitějšího rozhodování. Pokud máme nějaký komplexní objekt, který se skládá z více atributů, celkové hodnocení objektu bude závislé na každém z těchto atributů. Váha atributů se může různit. Proces počítání celkového hodnocení lze zaznamenat pomocí ohodnocených pravidel. Tato pravidla jsou ovšem pro každého uživatele jiná.

V ohodnocených pravidlech máme velkou volnost pro zaznamenání rozhodovacího procesu. V ohodnoceném pravidle můžeme použít libovolnou implikaci, která určuje způsob agregace hodnoty pravidla *r* a hodnoty těla pravidla *b*. Dále v samotném těle pravidla pomocí agregační funkce můžeme v podstatě použít libovolnou funkci, která splňuje mírné požadavky na agregační operátor. Tím je zajištěna velká flexibilita nastavení ohodnocených pravidel.

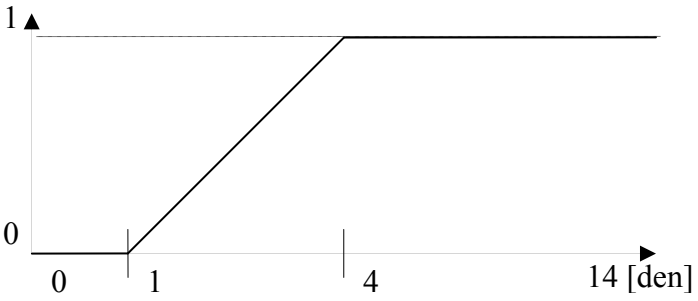
7.1.2. Uživatelské preference na datových typech

Pokud je jeden z atributů komplexního objektu číselného typu, lze na něm zavést tzv. fuzzy funkci. Fuzzy funkce je zobrazení $f : R \to T$. Dále budeme uvažovat pro přehlednost T=[0,1]. Bližší rozebrání fuzzy funkcí lze nalézt např. ve [VANEKOVA].

Logické programování ani zápis logického programu do RDF nemají prostředky pro zaznamenání uživatelských fuzzy funkcí. Tyto funkce budou muset být definované ve vyšším programovacím jazyce, jejich definice ovšem lze zapsat jako predikáty a uložit do RDF. Funkci na schématu 7.1.2.1 bychom pojmenovali např. Schod_1_4_Vzestupne. Tento řetězec lze jednoduše rozkódovat a interpretovat ve funkcionálním programu.

Fuzzy funkce nám vlastně modeluje náležení do predikátu *P*. Vezměme si predikát DlouhýVýlet. Tento predikát je závislý pouze na atributu délka výletu.

Ukážeme si dva příklady fuzzy funkce predikátu DlouhýVýlet



7.1.2.1. Schéma

Toto může být fuzzy množina průměrného člověka. Do jednoho dne to není dlouhý výlet, pak během druhého a třetího dne se subjektivní pocit délky zvětšuje, až od čtyřech dnů dál je každý výlet dlouhý.

7.1.2.2. Schéma

Schéma 7.1.2.2 je fuzzy funkce dlouhého výletu pro dítě. Hned od začátku se pocit délky výletu zvětšuje a jakmile délka přeroste jeden den, je to dlouhý výlet a rozdíl proti pětidennímu už je zanedbatelný.

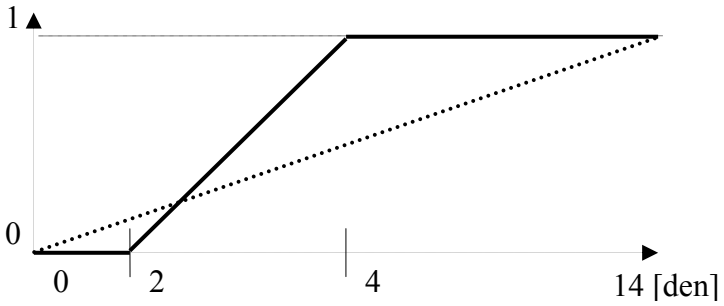
7.1.3. Interpretace fuzzy funkce

Rozlišíme dva případy interpretace fuzzy funkce.

První případ je hodnotový – tedy záleží nám na hodnotě, kterou uživatel danému bodu domény přiřazuje. Např. pokud modelujeme fyzikální nebo technologický proces a fuzzy regulátor má řídit konkrétní nastavení ovládacích prvků.

Druhý případ je uspořádávací – fuzzy funkci chápeme jako nové uspořádán domény atributu. Na konkrétních hodnotách již nezáleží, využíváme pouze porovnání hodnot fuzzy funkce. Takto se modeluje relevance, kvalita, uživatelské preference, hodnota 1 je "nejlepší" a 0 je "nejhorší".

Uvažme fuzzy funkci 7.1.2.1. a prosté uspořádání domény atributu.



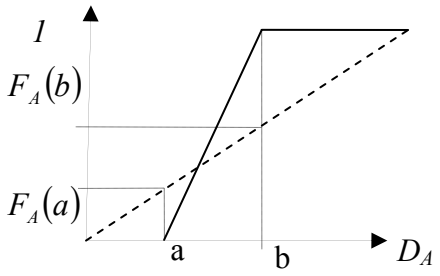
7.1.3.1. Schéma

Omezíme se na interval [2,4]. Na tomto intervalu se v uspořádávacím přístupu fuzzy funkce od prostého uspořádání neliší – obě funkce uspořádávají interval vzestupně. Naproti tomu v hodnotovém modelu se liší – každá funkce přiřazuje stejnému číslu jinou hodnotu.

7.1.4. Převod fuzzy funkcí na prosté uspořádání

Pokusíme se převést pravidla využívající fuzzy funkce na pravidla, kde je použito pouze prosté uspořádání. Použijeme změnu agregačního operátoru. Toto převedení jsme v žádné literatuře nenašli, jde o vlastní náznak možného směru dalšího výzkumu.

Budeme převádět fuzzy funkci ve tvaru, který je znázorněn na schématu 7.1.4.1.

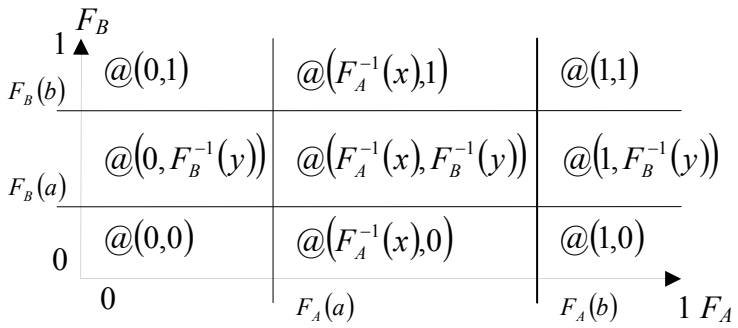


7.1.4.1. Schéma

Q^U je uživatelská fuzzy funkce a F_A je prosté uspořádání domény atributu D_A, které je nezávislé na uživateli. F_A chápeme jako univerzální fuzzy funkci pro atribut A.

Pravidlo ve tvaru $\left(P(x,y) \leftarrow @^{\mathrm{U}}\left(Q_A^U(x), Q_B^U(y)\right), r\right)$, kde predikát Q_A a Q_B jsou fuzzy funkce $D_A \rightarrow [0,1]$ a $D_B \rightarrow [0,1]$, převedeme na pravidlo $\left(P(x,y) \leftarrow @'^{\mathrm{U}}\left(F_A(x), F_B(y)\right), r\right)$. Chceme tedy posunout vliv uživatele až do agregační funkce a jako fuzzy funkci použít obecnou funkci, která je na uživateli nezávislá.

V tabulce je znázorněna situace $@(Q(x), Q(y))$ pro dvě proměnné.



7.1.4.2. Tabulka

Hodnoty $@'(x,y)$ odpovídají údajům v tabulce. Pro více proměnných je situace analogická. Přímý výpočet $@(F_A(x), F_B(y), ...)$ se omezí pouze na jeden případ, v ostatních alespoň v jedné proměnné je použita krajní hodnota.

Tímto způsobem jsme nahradili uživatelskou funkci Q obecnou funkcí F.

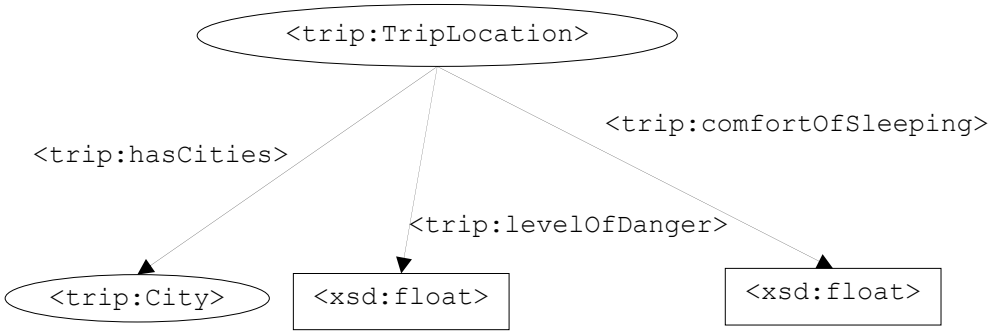
Tento způsob lze použít, pokud používáme uspořádávací přístup k fuzzy funkcím. Pokud nás zajímá přímo hodnota fuzzy funkce, nelze tento přístup použít.

7.2. Generování uživatelských preferencí

V této kapitole se budeme zabývat zmenšením počtu ohodnocení, která musí uživatel provést. Chceme generovat nová hodnocení daného uživatele na základě jím již provedených hodnocení. Nová hodnocení budou strojově vypočtena a je vhodné je odlišit od skutečných hodnocení přímo provedených uživatelem. Proto jsme zavedli novou vlastnost `<mvlp:ComputedRating>`, která bude označovat vygenerovaná hodnocení. Opět podvlastnosti této vlastnosti budou vypočtená hodnocení pro konkrétního uživatele. Tato část je naším příspěvkem do této problematiky. Myslíme si, že může ulehčit uživatelům práci se zadáváním hodnocení, navíc může sloužit i pro odhalování nesrovnalostí v hodnoceních. Přikládáme tomuto tématu velkou důležitost.

7.2.1. Motivační příklad

Nejprve si přiblížíme problematiku na následujícím příkladu. Uvažujme tento fragment OWL ontologie.



7.2.1.1. Ontologie

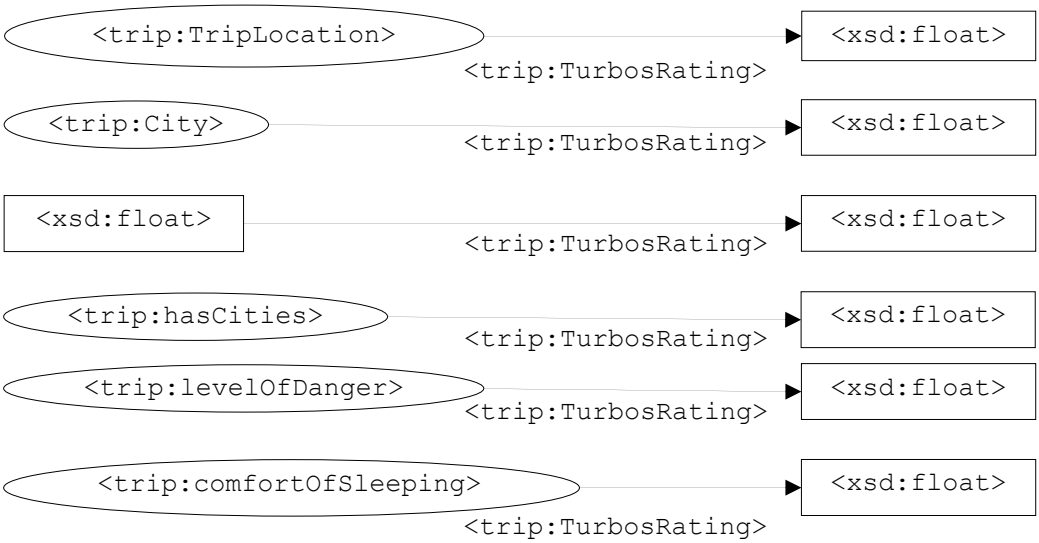
Toto schéma vyjadřuje vztahy týkající se výletních lokalit. Jsou zde zachycené **atributy** výletní lokality a **vlastnosti**, které výletní lokalitu s těmito atributy spojují.

Nyní podle definice z kapitoly 6. přidáme uživatelské hodnocení. Toto hodnocení se bude týkat individuí výše uvedených tříd a vlastností.

Implementace fuzzy funkcí zůstane na vyšším programovacím jazyku, my ji zde budeme zapisovat pomocí hodnocení individuí. To se shoduje s myšlenkou, že preference jsou zobrazení z entity do množiny pravdivostních hodnot.

Dále budeme uvažovat uživatelskou agregační funkci $@_U$, která bude agregovat hodnocení jednotlivých atributů výletní lokality do jediného hodnocení samotné výletní lokality. Předpokládáme, že agregační funkce bere v úvahu hodnocení jednotlivých atributů a navíc i hodnocení vlastností, které spojují atributy s výletní lokalitou. Míra hodnocení vlastnosti říká, jak je daný atribut pro uživatele důležitý při hodnocení výletní lokality.

Budeme předpokládat, že každý objekt může mít nejvýše jeden atribut pro každou vlastnost.



7.2.1.2. Schéma hodnocení

Tato hodnocení rozdělíme do tří skupin -

1) hodnocení individuí třídy `<trip:TripLocation>` - tuto skupinu budeme nazývat hodnocení **objektů** (budeme také používat "komplexní objekt").

2) hodnocení individuí třídy `<trip:City>` a fuzzy funkce pro vlastnosti `<trip:levelOfDanger>` a `<trip:comfortOfSleeping>` - tuto skupinu budeme nazývat hodnocení **atributů**

3) hodnocení vlastností `<trip:hasCities>, <trip:levelOfDanger>` a `<trip:comfortOfSleeping>` - tuto skupinu budeme nazývat hodnocení **vlastností**. Tato skupina je úzce svázána s agregační funkcí. Zjišťování vlastní agregační funkce je mimo rámec této práce a bude na ně zaměřena pozornost v dalším studiu. Zde se omezíme na zjištění váhy daného atributu pro hodnocení komplexního objektu.

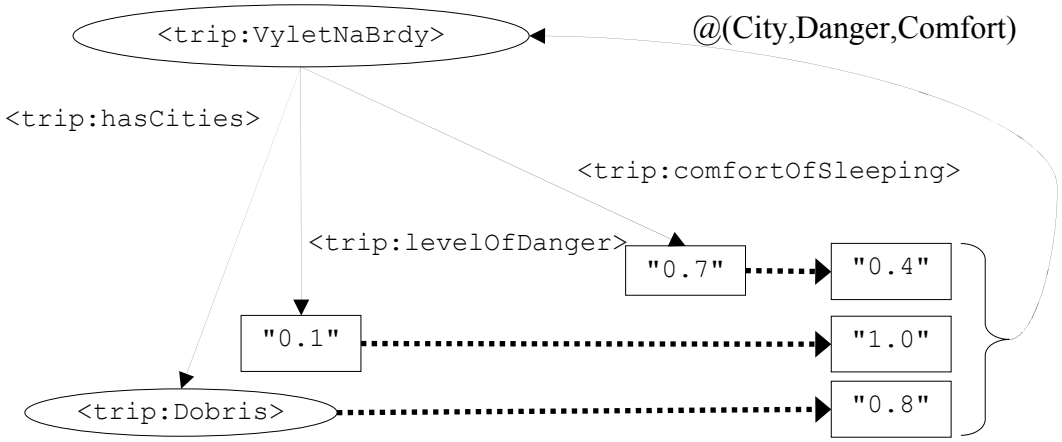
7.2.1.3. Poznámka

Pro přehlednost již nebudeme ve schématech používat predikát `<trip:TurbosRating>`. Šipku odpovídající vlastnosti `<trip:TurbosRating>` budeme kreslit tečkovanou tlustou čárou.

Přiblížíme si tři případy, v každém z nich nám jedna skupina hodnocení bude chybět.

Chybějící hodnocení komplexních objektů

Uvažujme případ, že Turbo ohodnotil vlastnosti a atributy. Hodnotit výletní lokality se mu už nechtělo. Náš úkol bude na základě znalostí hodnocení vlastností a atributů vypočítat hodnocení objektů, v tomto případě výletních lokalit. Uvažujme nyní výlet na Brdy.



7.2.1.4. Schéma agregace atributů

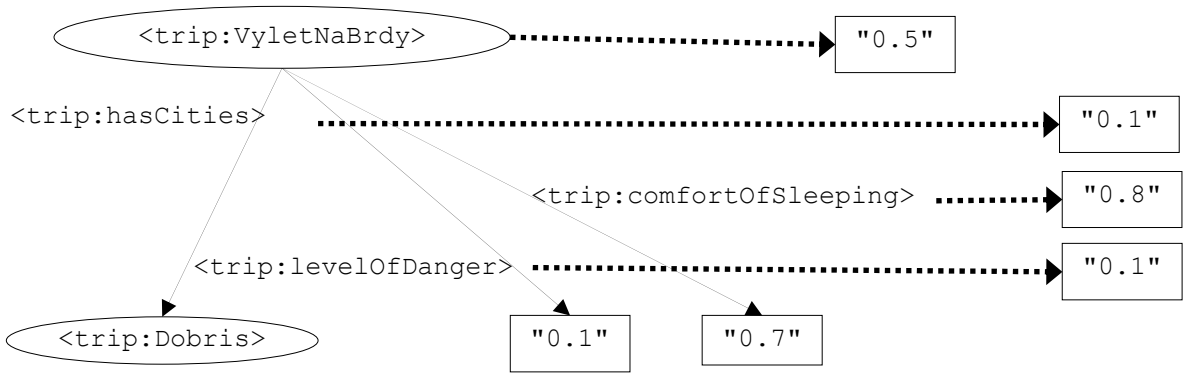
Chceme nyní ze tří atributů[1] spočítat hodnocení výletní lokality.

Tento případ dále opustíme, budeme se mu věnovat v kapitole 8. Při znalosti agregační funkce a hodnocení atributů budeme počítat hodnocení výletní lokality přímo jako agregaci všech tří atributů. Zde není žádný principiální problém, budeme se spíše zabývat rychlostí nalezení k (k>0) nejlepších objektů. Tento model je převzat z [FLN].

[1] Všimněme si použití fuzzy funkcí pro míru nebezpečí a komfort spaní. Pro míru nebezpečí jsou lepší menší hodnoty, 0.1 je převedeno na 1.0. Navíc je možné usoudit, že Turbo má velké požadavky na dobré spaní, protože hodnota 0.7 pro něj znamená hodnocení pouze 0.4. Hodnota 0.7 je objektivní situace spaní v místě a reprezentuje např. četnost kamenů, míru hrbolatosti terénu apod.

Chybějící hodnocení atributů

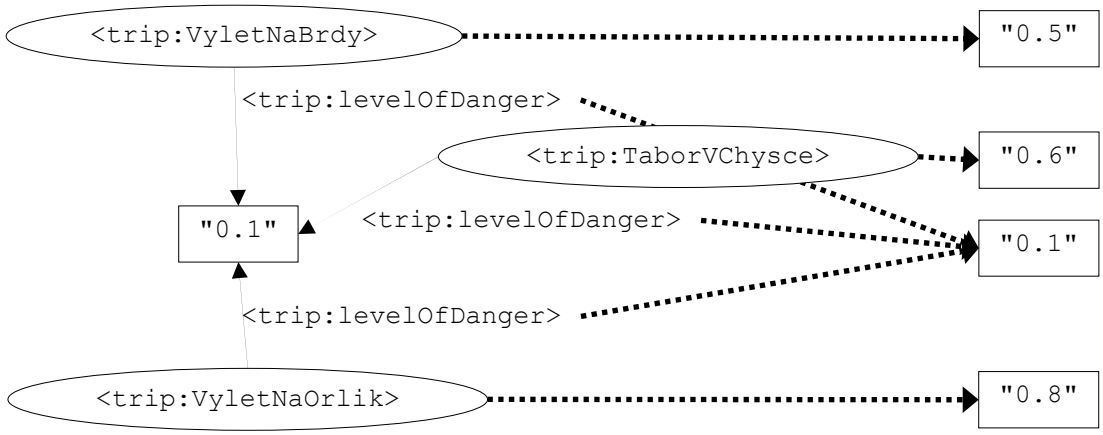
Dále uvažujme případ, kdy nám chybí hodnocení atributů. Známe tedy hodnocení jednotlivých lokalit a agregační funkci, ale neznáme hodnocení měst, komfortu ke spaní a nebezpečí dané lokality.



7.2.1.5. Schéma

V tomto případě budeme chtít použít hodnocení výletní lokality a znalosti o váze atributu na výpočet hodnocení jednotlivých atributů. Jelikož atribut `<trip:comfortOfSleeping>` má největší váhu, jeho hodnota by měla nejvíce odpovídat hodnocení lokality, tedy měla by být blízká 0.5.

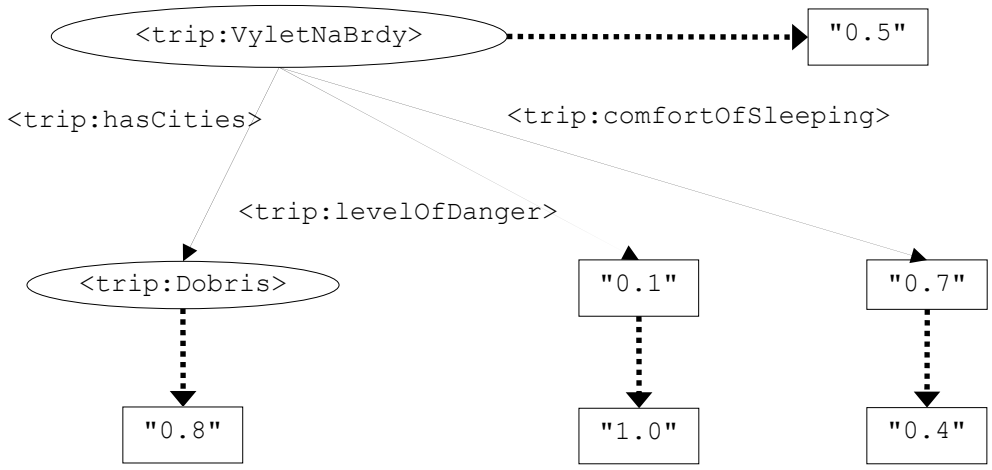
Zde je potřeba si uvědomit, že např. atribut "0.1" vlastnosti `<trip:levelOfDanger>` může být spojený s více lokalitami než pouze s Brdy. Bude proto nutné hodnoty všech lokalit, se kterými je daný atribut spojen, nějakým způsobem agregovat. Navíc může být spojen i se zdroji z jiných tříd než je `<trip:TripLocation>`. Tento případ bude rozebrán níže.



7.2.1.6. Schéma

Chybějící hodnocení vlastností

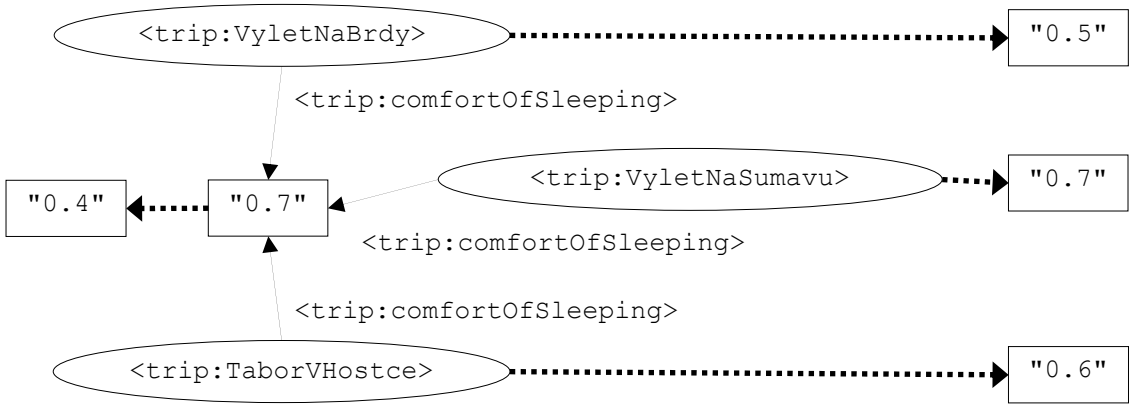
Poslední případ je chybějící hodnocení vlastností. Máme k dispozici hodnocení komplexního objektu, např. lokality, a jeho atributů. Z těchto údajů chceme zjistit, jak je který atribut důležitý pro celkové hodnocení lokality.



7.2.1.7. Schéma

V tomto případě porovnáváme hodnocení atributu a hodnocení lokality. Hodnocení komfortu spaní zde odpovídá nejvíce hodnocení lokality Brdy, tudíž vlastnost `<trip:comfortOfSleeping>` zřejmě dostane nejvyšší váhu ze všech zmíněných vlastností.

Stejně jako v předchozím případě je potřeba vzít do úvahy všechny možné lokality a jejich komforty spaní.



7.2.1.8. Schéma

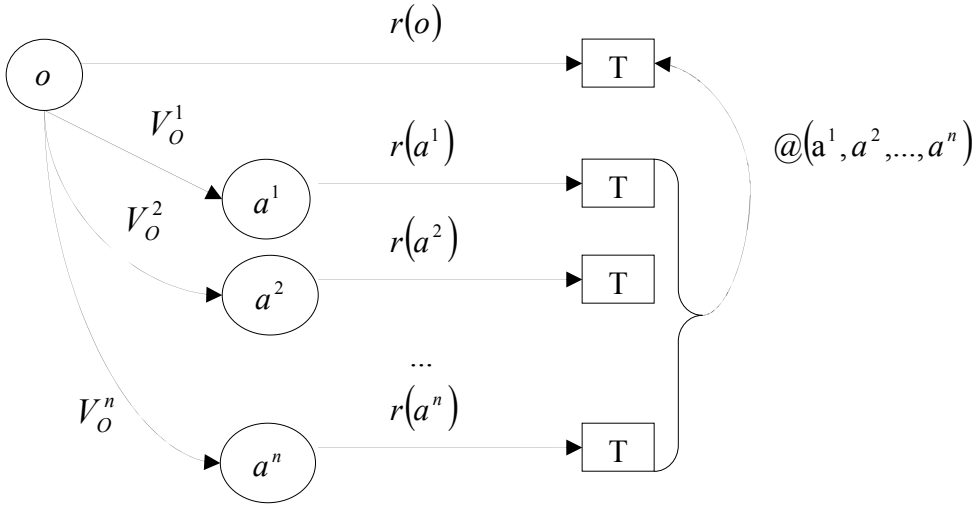
Výsledné hodnocení vlastnosti `<trip:comfortOfSleeping>` bude vygenerováno na základě dílčích výsledků všech předmětů vlastnosti `<trip:comfortOfSleeping>`.

7.2.2. Značení pro obecný případ

Pro běh algoritmu výběru nejlepších k objektů potřebujeme atributy seřazené sestupně podle uživatelského hodnocení r a uživatelskou agregační funkci $@$. Budeme používat stejné značení jako v kapitole 8. Objekt, jehož hodnocení chceme najít, značíme o, jeho atributy pak a^i.

Rozšíříme chápání této definice o OWL. Objekt o bude nyní individuum třídy O, atribut a^i bude individuum třídy A^i. Pro přehlednost a stručnost budeme psát třídy a individua v abstraktní notaci místo standardní notace `<trip:o>` apod. Třídy A^i jsou spojené s třídou O vlastnostmi, které budeme značit V_O^i. Výraz $V_O^i(o, a^i)$ bude reprezentovat trojici `<foo:o>` `<foo:`V_O^i`>` `<foo:`a^i`>`, tedy že objekt o je spojen s atributem a^i vlastností V_O^i.

Ostatní třídy budeme značit velkými písmeny $P, Q, R, ...$



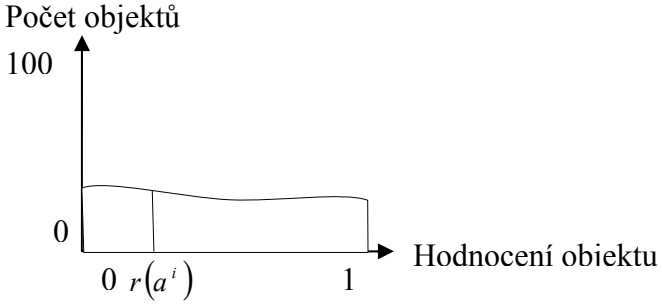
7.2.2.1. Schéma

Na schématu 7.2.2.1 je znázorněna situace pro objekt o. Nyní si nadefinujeme tři úkoly, které jsme nastínili v příkladu výše.

7.2.3. Způsob výpočtu hodnocení vlastností V_O^i

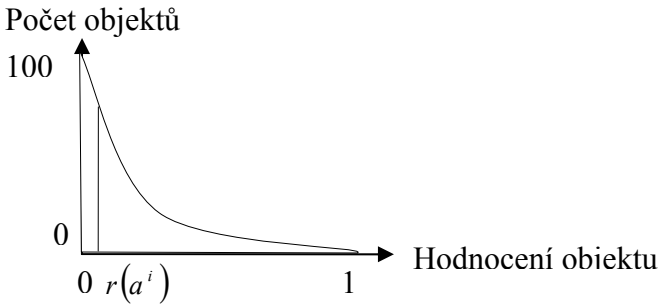
Ohodnocení vlastnosti V_O^i vyjadřuje, jak je důležitý parametr A^i v celkovém hodnocení individuí ze třídy O. Předpokládáme, že již máme ohodnocená nějaká individua ze tříd A^i i ze třídy O.

Zafixujme si individuum $a^i \in A^i$. Dále uvažujme množinu $I(a^i) = \{o : V_O^i(o, a^i)\}$. Je to množina objektů, jejichž i-tý atribut je a^i.



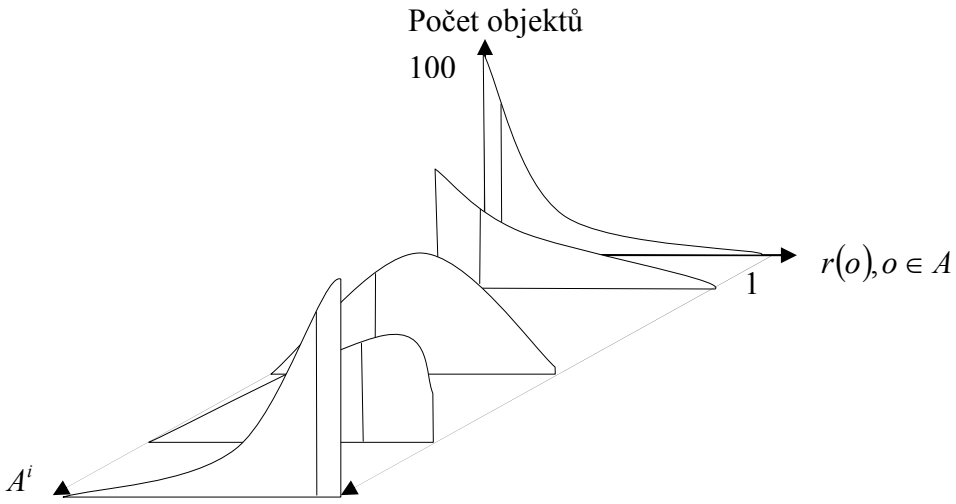
7.2.3.1. Schéma pro jeden atribut a^i

Schéma 7.2.3.1 vyjadřuje rozložení hodnocení r na množině $I(a^i)$. Svislá čára označuje hodnocení $r(a^i)$. Hodnoty jsou rozložené rovnoměrně po celém intervalu, což ukazuje na fakt, že hodnota atributu a^i významně neovlivňuje hodnocení objektů $o \in I(a^i)$. U ostatních individuí třídy A^i může situace vypadat odlišně.



7.2.3.2. Schéma pro konkrétní atribut a^i

V případě 7.2.3.2 naopak jsou hodnoty z větší části u hodnoty 0. To znamená, že většina objektů s atributem a^i má nízké hodnocení. Navíc má atribut a^i také nízké hodnocení, je zde tedy vztah mezi hodnocením atributu a^i a celkovým hodnocením objektů $o \in I(a^i)$.



7.2.3.3. Schéma pro celou třídu A^i

Schéma 7.2.3.3 vyjadřuje rozložení hodnocení r na celé množině A^i.

Pro každý prvek $a^i \in A^i$ vypočteme míru shody mezi hodnocením atributu $r(a^i)$ a hodnocením objektů $o \in I(a^i)$. Tuto shodu budeme označovat jako korelaci a značit $cor(a^i)$. Navrhujeme tuto korelaci počítat podle vzorce

$$cor(a^i) = \frac{\sum_{o \in I(a^i)} |r(o) - r(a_i)|}{|I(a_i)|},$$

kde korelace $cor(a^i)$ je průměrná odchylka hodnocení objektů $o \in I(a^i)$ od hodnocení atributu a^i.

Tento vzorec lze dále pozměnit např. použitím jiné metriky než absolutní hodnoty. Případně lze využít robustnější funkci než je průměr pro eliminování extrémních hodnot. Konkrétní definovaní korelace je mimo rámec této práce.

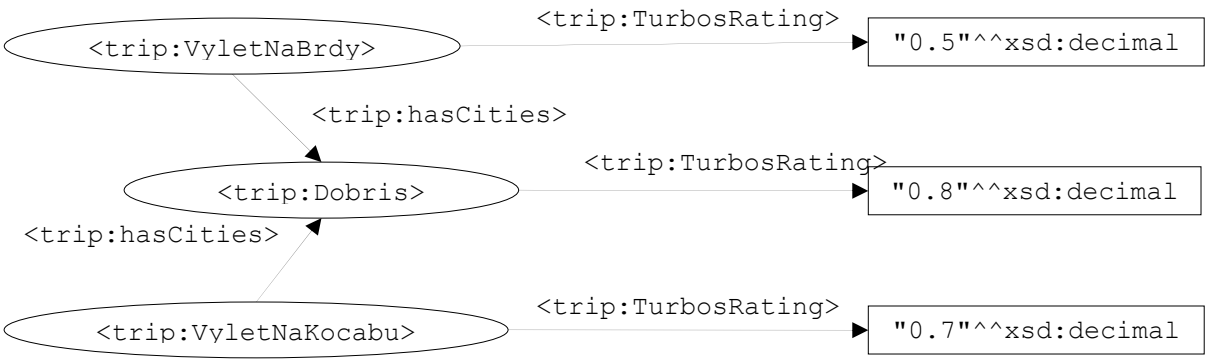
Celková důležitost spojující vlastnosti $r(V_O^i)$ bude závislá na korelacích všech individuí a^i. Výsledné hodnocení $r(V_O^i)$ tedy budeme počítat průměrem

$$r\left(V_O^i\right) = 1 - \frac{\sum_{a^i \in A^i} cor\left(a^i\right)}{\left|A^i\right|}.$$

Tato metoda je implementována ve třídě InducedRatings metodou getInducedRatingOnPredicate.

7.2.3.4. Příklad

Pro přiblížení si ukážeme jednoduchý příklad.



7.2.3.5. Schéma

Schéma 7.2.3.5 znázorňuje situaci na příkladu vlastnosti `<trip:hasCities>`. Tato vlastnost určuje, jaký dopravní prostředek byl použit na daném výletě.

Máme dva objekty, `<trip:VyletNaBrdy>` a `<trip:VyletNaKocabu>`, které jsou spojené s atributem `<trip:Dobris>`. Korelace mezi hodnocením objektů a atributem spočítáme pomocí vzorce

$$cor\left(a^i\right) = \frac{\sum_{o \in A} \left| r(o) - r(a_i) \right|}{\left|A\right|}.$$

V našem případě tedy

$$cor(< \text{trip} : \text{Dobris} >) =$$

$$\frac{\left| r(< \text{trip} : \text{VyletNaBrdy} >) - r(< \text{trip} : \text{Dobris} >) \right| + \left| r(< \text{trip} : \text{VyletNaKocabu} >) - r(< \text{trip} : \text{Dobris} >) \right|}{2} =$$

$$\frac{|0.5 - 0.8| + |0.7 - 0.8|}{2} = \frac{0.3 + 0.1}{2} = \frac{0.4}{2} = 0.2$$

Pokud bychom tedy měli pouze tyto údaje, vypočítáme hodnocení vlastnosti `<trip:hasCities>` takto

$$r(< \text{trip} : \text{hasCities} >) = 1 - \frac{cor(< \text{trip} : \text{Dobris} >)}{1} = 1 - 0.2 = 0.8$$

Zde je také vidět, že čím nižší je korelace atributů, tím vyšší bude potom hodnocení vlastnosti.

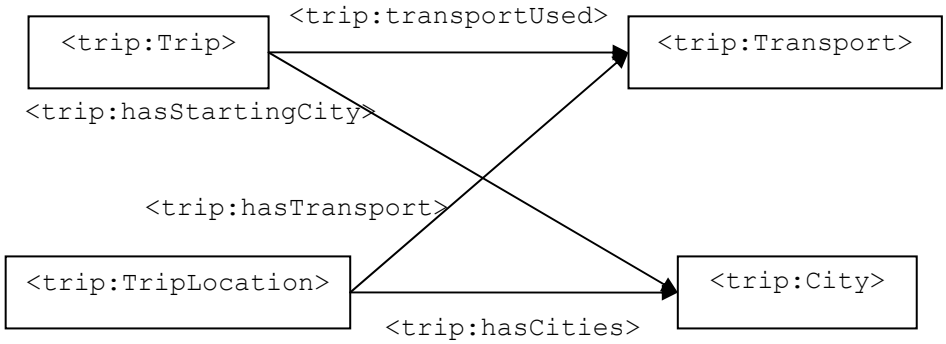
Nakonec poznamenáme, že uvedené hodnocení je platné pouze pro uživatele Turbo, pro ostatní se bude lišit.

7.2.4. Způsob výpočtu hodnocení individuí

Nyní se budeme zabývat případem 2) Chybějící hodnocení individuí. Chceme získat ohodnocení individuí a^i.

Vezmeme do úvahy zbytek ontologie, tedy třídy $P,Q,R,...$ Některé z těchto tříd mohou být spojeny s třídami atributů A^i vlastnostmi V_P^i, V_Q^i.

Schéma 7.2.5 vyjadřuje vztahy mezi třídami.



7.2.5. Schéma

Pokud uživatel ohodnotí individuum `<trip:Brdy050501>` ze třídy `<trip:Trip>`, budeme chtít hodnocení $r(<\text{trip}:\text{Brdy050501}>)$ použít a vypočítat z něj hodnocení atributů `<trip:Dobris>` a `<trip:CSAD>`. Myšlenka je taková, že mohou existovat jiné třídy, které jsou vhodnější pro hodnocení než třída O. Pokud třída O vyhovuje kritériím na objektivní hodnocení, můžeme následující postup použít i pro tuto třídu.

Při počítání ohodnocení individua a^i se budeme snažit použít všechny ohodnocené objekty *x* z libovolné třídy X, které jsou spojené s atributem a^i. Ohodnocení vlastnosti V_X^i bude určovat, s jakou vahou budeme hodnocení objektu *x* brát.

Nechť pro individuum a^i třídy A^i existuje *l* individuí $x_1,...,x_l$ ze tříd $X_1,...,X_l$ takových, že platí $\forall j = 1,..,l : V_{X_j}^i(x_j, a^i)$. Potom vypočítané hodnocení individua a^i bude $f_{A^i}(r(x_1),...,r(x_l))$.

Funkce f_{A^i} je monotónní v každé proměnné. Navíc by měla brát v potaz i váhy $r(V_O^i), r(V_P^i),...$. Čím větší přiřazuje uživatel váhu vlastnosti V_X^i, tím větší má váhu hodnocení individuí ze třídy X. V této funkci navrhujeme implementovat statistické metody pro odstranění nekonzistence v uživatelských hodnocení.

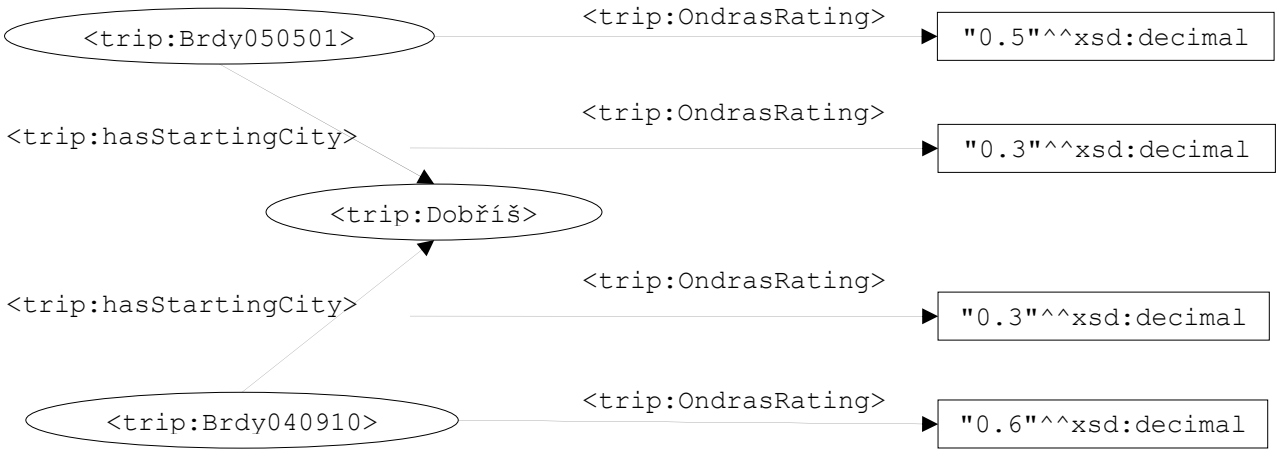
Výše uvedeným způsobem můžeme vypočítat hodnocení libovolného atributu a^i. Podmínka je, že máme již uživatelská hodnocení individuí jiných tříd. Tento mechanismus je možné použít v případě, že existují třídy vhodné k přímému hodnocení, tedy objekty, se kterými má uživatel přímou zkušenost. Navíc lze postup použít i pro „atributy atributů“.

Tato metoda je implementovaná ve třídě InducedRatings metodou getIducedRatingOnEntity. Jako funkce $f_{A^i}(r(x_1),...,r(x_l))$ je zde použit vážený průměr. Jeho výpočet odpovídá vzorci

$$f_{A^i}(r(x_1),...,r(x_l)) = \sum_{j=1,...,l} r(x_j) \cdot r(V_{X_j}^i) \Big/ \sum_{j=1,...,l} r(V_{X_j}^i)$$

7.2.6. Příklad

Ukážeme si příklad z ontologie výletních míst.



7.2.6.1. Schéma

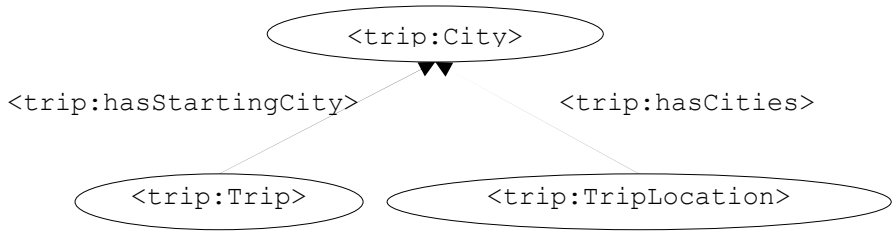
V tomto případě známe hodnocení vlastnosti `<trip:hasStartingCity>` a výletů na Brdy `<trip:Brdy040910>` a `<trip:Brdy050501>`. Chceme vypočítat hodnocení zdroje `<trip:Dobříš>`.

Můžeme zde použít např. vážený průměr.

$$r(<\text{trip}:\textit{Dobříš}>) = \frac{r(<\text{trip}:\text{Brdy040910}>)r(<\text{trip}:\text{hasStartingCity}>) + r(<\text{trip}:\text{Brdy050501}>)r(<\text{trip}:\text{hasStartingCity}>)}{r(<\text{trip}:\text{hasStartingCity}>) + r(<\text{trip}:\text{hasStartingCity}>)} =$$

$$\frac{0.6*0.3+0.2*0.3}{0.3+0.3} = \frac{0.18+0.06}{0.6} = \frac{0.24}{0.6} = 0.4$$

Tím jsme vypočítali hodnocení pro `<trip:Dobříš>` a toto hodnocení můžeme použít pro hledání výletních míst.



7.2.6.2. Schéma

Ve schématu 7.2.6.2 vidíme tři třídy – výletní místa, města a výlety. Chceme počítat hodnocení výletních míst, k tomu potřebujeme i hodnocení měst, která se v místě nacházejí.

Toto hodnocení můžeme nechat uživatele zadat ručně. Je tu ovšem riziko, že uživatel bude město hodnotit i z jiných hledisek než z pohledu problematiky výletů. Některá města mohou být krásná, ale cesta k nim trvá neúnosně dlouho, což jejich hodnotu snižuje. To si ale uživatel nemusí v dané chvíli uvědomit.

Nechme uživatele ohodnotit výlety, kterých se zúčastnil. Poté, co se vrátí domů z výletu, je plný zážitků a dojmů. Tím jeho hodnocení výletu bude živé a přesné, bude vystihovat přesně to, co od něj požadujeme - jak se mu výlet líbil. Bude zde zahrnuta např. i únava z dlouhé cesty vlakem.

Z ohodnocení výletů potom výše uvedeným způsobem vypočteme hodnocení měst. Zde je vhodné, aby byl ohodnocen i vztah mezi výletem a městem.

Nakonec při hledání místa pro další výlet již můžeme použít vypočítané hodnocení měst.

Z výše uvedeného příkladu je zřejmé, že některé třídy jsou vhodnější pro hodnocení než jiné. Můžeme tvrdit, že hodnocení komplexnějších objektů je užitečnější než hodnocení primitivních jevů. Ohodnocení komplexnějšího objektu může vyjadřovat přímou oblibu objektu, zatímco hodnocení abstraktního atributu může být pro uživatele obtížně představitelné.

7.3. Změny uživatelských preferencí v čase

V příkladu o fuzzy funkcích jsme zmínili dvě funkce pro průměrného dospělého a pro dítě. Zřejmě tyto dvě funkce byly diametrálně odlišné, stejně jako by byly např. pro predikát VysokýPlat. Uživatelské preference se tedy mohou měnit, a to jak v dlouhodobém, tak i v krátkodobém horizontu.

Změna preference se může týkat dvou základních věcí. Buď se mění preference jednoho objektu nebo způsob agregace atributů do hodnocení komplexního objektu. S přibývajícím věkem např. můžeme více preferovat dopravu vlakem a méně dopravu na kole. Ovšem dobrý výlet pro nás stále znamená, že tam bylo hezké počasí a trasa procházela zajímavým územím, byť na to, aby počasí bylo pro uživatele hezké, bude potřeba více než zamlada.

Změna se bude častěji týkat preferencí individuí. Změny jsou způsobeny přirozeným vývojem jedince, ale také vnějšími okolnostmi. Uvažme vliv ročního období - výlet do Vysokých Tater v létě je jistě příjemnější než v zimě. Navíc v zimě je přístup do Vysokých Tater zakázán. Jiný příklad je výlet do zahraničí. Jakkoliv hodně se nám výlet do Bulharských hor líbil, v podstatě není možné ho podniknout během školního roku. Toto je také příklad relativity preferencí - sice se nám pohoří Rila v Bulharsku líbí opravdu nejvíce, zároveň víme, že tam nemůžeme. V celkovém pohledu tedy bude mít hodnocení velmi blízké nule.

Druhý způsob změn preferencí je v agregaci atributů. Tento druh změny bude méně častý, protože vyžaduje poměrně velkou změnu v přístupu jedince. Většinou se bude jednat o menší změny např. ve vahách jednotlivých atributů. Ovšem např. výhra v loterii může naprosto obrátit směr fuzzy funkcí. Typický případ je změna preferencí při nástupu do zaměstnání, kdy uživatel zjistí, že nemusí tak úpěnlivě hledat nejlevnější varianty.

Z výše uvedených důvodů vyplývá, že zaznamenávání uživatelských preferencí je dynamický proces. V našem případě, kdy evidujeme preference výletů, tato problematika není tak důležitá. Projekt [FILMTRUST] eviduje uživatelské hodnocení filmů. Pokud srovnáme filmy ze současnosti a např. z roku 1995, zjistíme, že staré filmy jsou diametrálně horší např. v kvalitě speciálních efektů. V této problematice bude nutno uvažovat i stáří hodnocení. Čím je hodnocení starší, tím méně je důvěryhodné.

7.4. Více uživatelských hodnocení

7.4.1. Úvod do více uživatelských hodnocení

Vícehodnotová logika počítá pouze s jedním ohodnocením faktů a pravidel. Pro naše potřeby je nezbytné uchovávat těchto hodnocení více, ať už hodnocení každého uživatele nebo vytvořených profilů uživatelů.

Více uživatelských hodnocení můžeme do predikátové vícehodnotové logiky zavést rozšířením definice realizace $\Re$ a realizací predikátů $P^k_{\Re}$, kdy každý uživatel bude mít jinou

realizaci predikátů. Jednodušší je pro každého uživatele zavést novou realizaci, ovšem všechny realizace by se měly shodovat na množině individuí *U*.

Stejným přístupem můžeme modifikovat i logický program. Ohodnocený fakt bude nést informaci o uživateli, pro nějž platí, stejně ji ponesou i pravidla. V zadání dotazu bude identifikace uživatele *u*, čímž se omezí množina faktů a pravidel pouze na ta, jež jsou platná pro uživatele *u*.

V predikátové logice se rozlišení uživatelů děje pouze na realizaci predikátů, u logického programu se liší v hodnocení faktů a pravidel, samotné fakty a pravidla zůstávají nezměněny. Z toho, že každý uživatel má různé hodnocení pravidel a faktů, plyne, že pro různého uživatele na stejný dotaz můžeme dostat různé odpovědi.

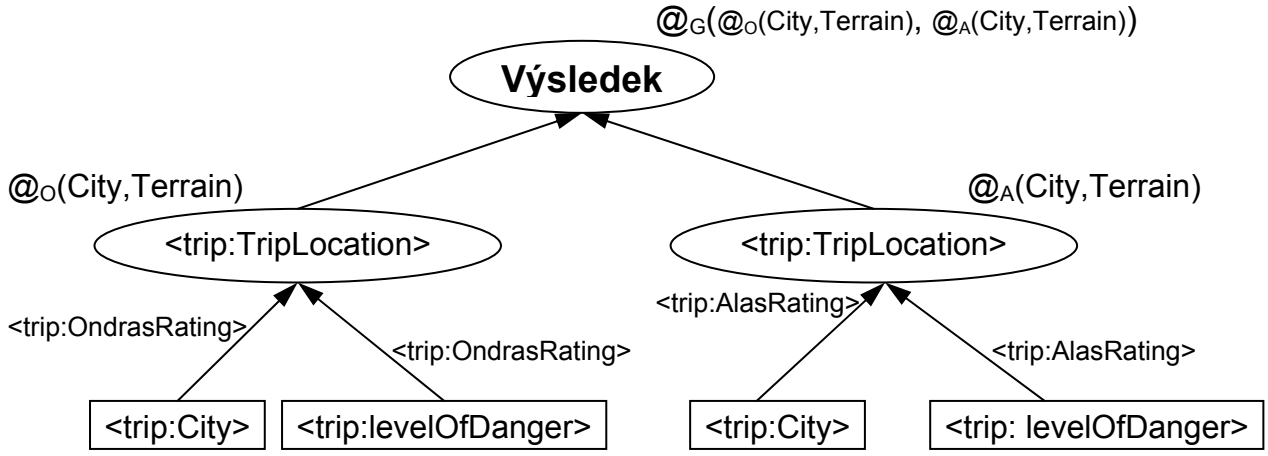
Zde musíme zdůraznit, že uživatel může pravidlo nebo fakt ohodnotit také hodnotou nula, čímž naznačí, že pro něj nemá žádnou relevanci.

7.4.2. Agregace hodnocení více uživatelů

Zanesení možnosti zaznamenat více uživatelských hodnocení do jednoho logického programu nám poté umožňuje s těmito hodnoceními dále pracovat. Mezi nejdůležitější možnosti patří podpora skupinového rozhodování. To je motivace, s kterou jsme vytvářeli ontologii výletních lokalit. Máme skupinu turistů, každého s jinými preferencemi a chceme vybrat místo, které je optimální skupinový kompromis.

Při skupinovém výběru výletní lokality budeme brát v úvahu atributy lokality, např. terén, města, která se v lokalitě nacházejí, četnost vodních zdrojů, nebezpečnost apod. Každý atribut, např. město Praha, je hodnocen různě dvěma různými uživateli. Atributu jako celku je přiřazována různá důležitost – někomu záleží spíše na terénu a jiný zase chce dostatek vody. Jinými slovy, uživatelé se liší jak v ohodnocení dat, tak i v agregačních funkcích, kterými uživatelé agregují jednotlivé atributy do hodnocení celku.

Uvažujme pro jednoduchost pouze dva atributy výletních míst - míru nebezpečnosti a města. Na následujících dvou schématech jsou dva různé postupy, jak agregovat preference více uživatelů.

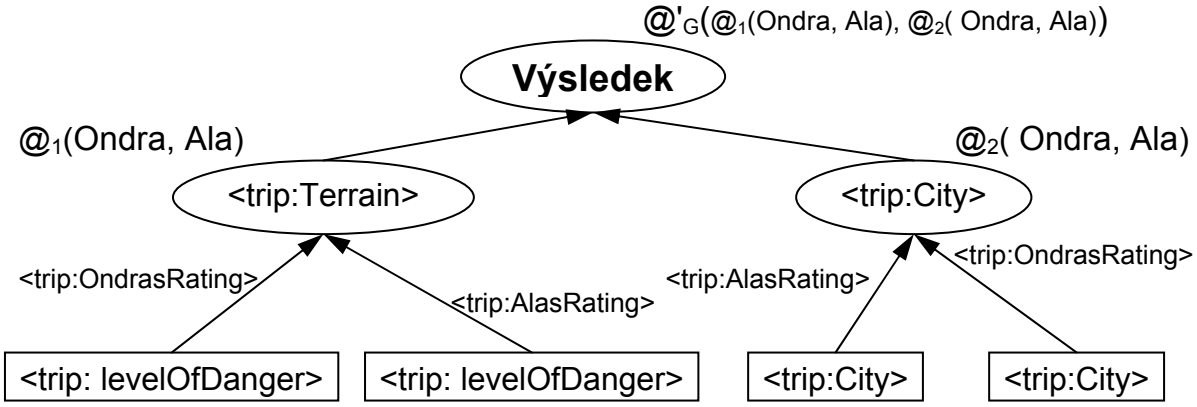


7.4.2.1. Schéma

Na schématu 7.4.2.1 je znázorněna typická situace, kterou známe např. ve volbách. Uživatelé si vyberou ty volební strany, které jim vyhovují, a přiřadí jim své preference. Uživatelské preference na těchto objektech se poté agregují globální agregační funkcí $@_G$, např. průměrem.

V tomto konkrétním případě se tedy Ondra a Ála nejprve sami rozhodnou, jak se jim která lokalita líbí. Poté se tato hodnocení zprůměrují do globálního hodnocení a vyberou se

výletní lokality s nejlepším globálním hodnocením. Z obrázku 7.4.2.1 je také patrné, že oba dva uživatelé používají jinou agregační funkci ($@_O$ a $@_A$).



7.4.2.2. Schéma

Na schématu 7.4.2.2 je znázorněn druhý způsob agregace více uživatelských preferencí. Vybíráme rovnou z uživatelsky ohodnocených atributů. Hodnocení atributů agregujeme globálními agregačními funkcemi $@_1$ a $@_2$. Takto dostaneme atributy ohodnocené globálním hodnocením. Z nich pak jinou globální funkcí $@'_G$ vypočteme objekty, které jsou vhodné pro všechny uživatele.

V politickém případě bychom tedy nehodnotili přímo jednotlivé politické strany, ale např. jednotlivé obory programů - rovná a progresivní daň v ekonomice, jaderné, hnědouhelné nebo větrné elektrárny v energetice apod. Z těchto ohodnocených problematik bychom pak agregovali hodnocení nejlepšího řešení toho kterého oboru. Vyšlo by tedy, že nejlepší řešení ekonomiky je např. progresivní daň a energetiky jaderné elektrárny. Strany, které by tato řešení podporovala, by měla větší hodnocení než strany, které prosazují rovnou daň.

Globální funkce $@_1$ a $@_2$ mohou být klasický průměr nebo je zde možno využít funkce pro sofistikovanější kombinaci hodnocení. Funkce $@'_G$ by pak měla brát v úvahu také uživatelské agregační funkce $@_O$ a $@_A$. Pokud jsou obě funkce vážený průměr, globální váha atributu bude průměr obou vah.

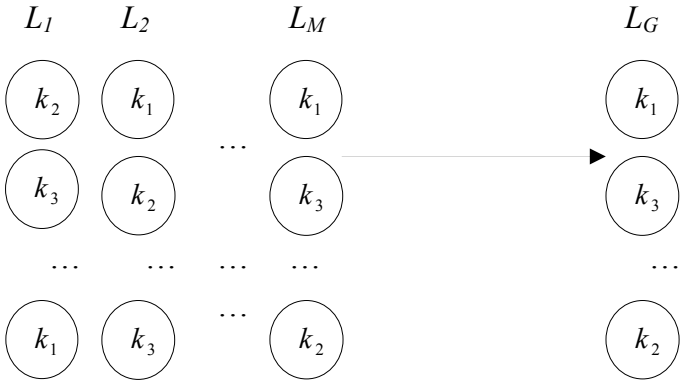
V tomto případě tedy nejdříve necháme uživatele Ondřeje a Alenu ohodnotit jednotlivá města a terény. Poté každému městu a terénu přiřadíme jedno globální hodnocení. Na základě těchto hodnocení nakonec vybereme nejlepší výletní lokality. Tento výběr bude záviset na důležitosti, jakou městům a terénu oba uživatelé přiřazují.

7.5. Návaznost na Arrowovu teorii

V této kapitole přiblížíme návaznost na teorii Kennetha Arrowa, která je přiblížena např. v [WIKIPEDIA]. Kenneth Arrow je americký ekonom, mezi jehož nejznámější práce patří teorie o nemožnosti nalezení výsledku hlasování, které by vyhovovalo pěti dále definovaným požadavkům, tzv. Arrowův paradox. Tento paradox nyní přeformulujeme do vícehodnotové logiky a nabídneme rozšíření.

Arrowův paradox pracuje s preferenčními seznamy. Mějme množinu N kandidátů (alternativ) $K = \{k_1,...,k_N\}$. Uspořádaný seznam kandidátů budeme značit $L_i = \langle k_{i_1},...,k_{i_N} \rangle$. To, že alternativa k_i je v seznamu před alternativou k_j budeme také značit $k_i \geq k_j$.

Každý z voličů $v_1,...,v_M$ uspořádá seznam kandidátů K podle jeho preferencí. Tím vznikne M uspořádaných seznamů $L_1,...,L_M$. Agregační funkce $s: L^M \to L$ potom z těchto seznamů vytvoří uspořádaný seznam L_G, jež vyjadřuje globální preference všech voličů.



7.5.1.1. Schéma

Požadavky na agregační funkci *s* jsou

1) **Všeobecnost** - Funkce *s* musí být deterministická, její obor hodnot je celý prostor uspořádaných seznamů pro daný počet voličů a kandidátů.

2) **Suverenita voličů** - Funkce *s* musí být na, tedy pro každý seznam *L* musí existovat vstup funkce tak, že výstup je seznam *L*.

3) **Absence diktátora** - Nesmí existovat uživatel, jehož vstupní seznam se vždy shoduje s výstupem funkce *s*.

4) **Monotonie** - Pokud volič změní svůj seznam tak, že zvýší hodnocení objektu *o*, v globálním seznamu L_G se pozice objektu *o* buď nezmění nebo zvýší.

5) **Nezávislost na irelevantních alternativách** – Pokud redukujeme množinu kandidátů na N', přičemž zachováme vzájemné pořadí zbylých kandidátů ve vstupních seznamech, vzájemné pořadí zbylých objektů v novém redukovaném globálním seznamu L_G' se také nezmění.

Věta

Neexistuje funkce, která splňuje všech pět výše uvedených požadavků.

Kenneth Arrow v roce 1951 tuto větu naformuloval a dokázal.

Důkaz věty

Ukážeme si ve stručnosti myšlenku důkazu, plné znění lze nalézt v [GEANAKOPLOS]. Dokážeme, že pokud s splňuje podmínky 1),2),4),5), potom existuje diktátor.

Vyberme jednoho alternativu *b*.

Lemma 1

Pokud je *b* ve všech seznamech buď první nebo poslední, potom i v L_G bude *b* buď první nebo poslední.

Důkaz lemmatu

Nechť platí opak, tedy existují *a,c* tak, že $a \geq b, b \geq c$ v L_G. Z podmínky 5) toto bude platit, i když se v seznamech $L_1,...,L_M$ posune alternativa *c* nad *a*. Protože *b* je buď první nebo poslední, nezmění tento přesun nijak lokální pořadí *a* a b ani *b* a *c*. Nyní tedy platí, že $a \geq b, b \geq c$ a zároveň $c > a$. Z transitivity tedy plyne, že $a \geq c$. Zároveň ale platí, že $c > a$, což je spor.

Nyní předpokládejme, že všichni voliči umístili *b* na konec svých seznamů. V L_G je tedy *b* také na konci.

Dále nechme postupně voliče přesouvat alternativu *b* z konce na začátek seznamů. Uvažujme okamžik, kdy se *b* přesune v L_G z posledního na první místo. Stav před přesunem označme I a stav po přesunu II. Voliče, který přesunul *b* z konce jeho seznamu na začátek mezi stavy I a II, označme n(*b*). Voliče n(*b*) označíme n*.

Nyní dokážeme že $\forall a,c : a \neq b, c \neq b \rightarrow (a >_{n*} c \Leftrightarrow a >_G c)$. Tedy pro *a* a *c* různá od *b* se pořadí *a* a *c* v globálním seznamu shoduje s pořadím *a* a *c* v seznamu uživatele n*.

Uvažme stav III, který zkonstruujeme ze stavu II tak, že uživatel n* přesune prvek *a* nad prvek *b*.

Porovnání *a* a *b* bude u všech uživatelů stejné jako ve stavu I, takže platí, že $a >_G b$. To plyne z toho, že ve stavu I byl *b* na konci seznamu L_G.

Porovnání *b* a *c* bude jako ve stavu II, protože mezi těmito dvěma alternativami se od stavu II nic nezměnilo. *b* byl ve stavu II na prvním místě seznamu, tedy $b >_G c$.

Z transitivity plyne, že $a >_G c$, což odpovídá hodnocení uživatele n*, který přesunul *a* na první místo svého seznamu.

Nakonec dokážeme, že i pro dvojice *a* a *b* se pořadí v seznamu shoduje s pořadím *a* a *b* v seznamu uživatele n*.

Uvažujme nyní voliče n(*c*), který vznikne stejnou konstrukcí, ovšem s prvkem *c*. n(c) může ovlivnit pořadí všech dvojic *xy*, kde *x* ani *y* nejsou *c*. Tedy konkrétně i dvojici *ab*. Ovšem volič n* ovlivnil pořadí dvojice *ab* mezi stavy II a III0. Ve stavu II platilo, že *b*>*a* a ve stavu III naopak *a*>*b*. Tedy n(*c*)=n* a tedy n* je diktátor.

□

7.5.2. Rozšíření Arrowovy teorie do vícehodnotové logiky

Uvažme nyní tuto teorii v kontextu vícehodnotové logiky. V kapitole 7.1.3 jsme si rozdělili interpretaci fuzzy funkce na dva přístupy - hodnotový a uspořádávací přístup. Arrowovy uspořádané seznamy mohou být chápány jako uspořádávající fuzzy funkce. Seznam *L* nám dává lineární uspořádání množiny *K*.

Arrow v [SOCIALCHOICE] zmiňuje možnost navrženou Neumannem a Mongerstrenem pro zavedení hodnocení alternativ. Uvažuje ovšem případ, kdy nejlepší kandidát má vždy hodnocení 1 a nejhorší má hodnocení 0. Hodnocení kandidáta je potom součet všech hodnocení. Tento přístup se ovšem neslučuje s podmínkou 5). Pro seznamy $L_1 = \{x:1, y:0.9, z:0\}$, $L_2 = \{x:1, y:0.9, z:0\}$ a $L_3 = \{y:1, x:0.5, z:0\}$ je nejlepší volba kandidát *y* s hodnocením 2.8 před kandidátem *x* s hodnocením 2.5 [1]. Pokud ovšem vyhodíme prvek z, dostaneme seznamy $L_1 = \{x:1, y:0\}$, $L_3 = \{x:1, y:0\}$ a $L_3 = \{y:1, x:0\}$. Nyní vítězí

[1] Pro seznamy bez hodnocení by vyhrál prvek *x*. Hodnocení prostředního prvku je tedy důležité pro celkové hodnocení. Tento případ také demonstruje rozdíl mezi hodnotovým a uspořádávacím přístupem fuzzy funkcí.

prvek *x* s hodnocením 2. V našem přístupu podmínka, že nejlepší kandidát má hodnocení 1 a nejhorší má hodnocení 0, není.

V případě, že budeme uvažovat hodnotový přístup, nejsou definice dostatečné. Rozšíříme proto definici seznamu následovně - uspořádaný seznam L_i bude $L_i = \{(k_{i_1} : r_i),...,(k_{i_N} : r_N)\}$ a bude platit, že $\forall i < N : r_i \geq r_{i+1}$. Máme tedy uspořádaný seznam, ve kterém je navíc uvedeno i hodnocení kandidáta.

Nyní se pokusíme přeformulovat požadavky do vícehodnotové logiky. Přeformulujeme požadavky na agregační funkci, aby odpovídaly vícehodnotové logice.

1) **Všeobecnost** – Tento požadavek zůstává nezměněn.

2) **Suverenita voličů** – Tento požadavek zůstává nezměněn.

4) **Monotonie** - Pokud volič změní svůj seznam tak, že zvýší hodnocení objektu *o*, v globálním seznamu se hodnocení objektu *o* buď nezmění nebo zvýší.

5) **Nezávislost na irelevantních alternativách** – Pokud redukujeme množinu kandidátů na N‘, přičemž zachováme hodnocení zbylých kandidátů ve vstupních seznamech, vzájemné pořadí zbylých objektů v novém redukovaném globálním seznamu L'_G se také nezmění. Hodnocení jednotlivých kandidátů v seznamu L'_G se změnit může.

Vynechali jsme bod 3) Absence diktátora. Tento bod je problematický z toho důvodu, že lze těžko určit shodnost dvou seznamů. Zde bude třeba zavést pojem "fuzzy" diktátora, který bude vyjadřovat míru diktátorství každého voliče. Tato definice bude předmětem dalšího výzkumu, vybočuje již mimo rámec této práce.

Nemůžeme tvrdit, zda tyto nové požadavky jsou také nekompatibilní, tedy zda neexistuje funkce agregující hodnocení uživatelů při zachování těchto požadavků. Nicméně zavedením hodnocení se již dá precizně kvantifikovat, nakolik je globální hodnocení kandidátů odlišné od hodnocení voliče *u*.

Arrowovu teorii lze rozšířit i o koncepci atributů kandidátů. Původní pojem kandidát vlastně odpovídá výrokové proměnné. Nyní rozšíříme kandidáty i o jejich atributy, převedeme tedy teorii do predikátové logiky.

Ve skutečných volbách volíme strany podle jejich volebních programů a dalších parametrů, jako je např. personální obsazení. V "duchu" provedeme agregaci všech těchto atributů stran a výsledkem této agregace je vlastní hodnocení té které politické strany. Stranu s naším nejlepším hodnocením poté volíme.

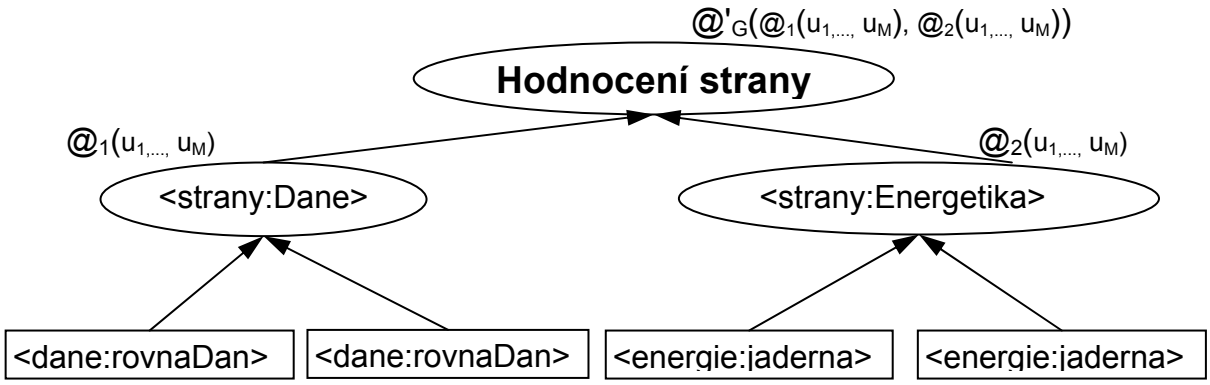
Tento přístup je více rozebrán v kapitole 7.4. Nevýhoda tohoto přístupu je ztráta procesu rozhodování. Pokud se mi na některé straně líbí, že prosazuje rovnou daň, ale nesouhlasím s její zemědělskou politikou, nikdo se již nedoví, proč jsem stranu volil.

Ve volbách, které by využívaly druhý přístup navržený v kapitole 7.4, bychom nevolili politické strany, nýbrž bychom vyplňovali dotazník týkající se jednotlivých problematik vládnutí - zemědělství, daní, školství, energetiky,.... V každé problematice by bylo několik alternativ, které bychom ohodnotili. Nakonec by také voliči vyplnili, na které problematice jim nejvíce záleží a na kterých méně.[1]

Ze všech hodnocení alternativy *a* by se vypočítalo globální hodnocení *a*. Hodnocení strany by byla agregace hodnocení řešení, které strana prosazuje ve sledovaných

[1] Tento způsob voleb by navíc výrazně zlevnil provoz voleb pro jednotlivé strany, protože volební kampaně by ztratily smysl. Na druhou stranu by byl výrazně časově náročnější pro voliče, kteří by navíc nad problematikami museli alespoň zčásti netriviálně přemýšlet. Proto by byl nutný perfektní informační servis pro všechny sociální vrstvy obyvatelstva.

problematikách. Situaci přibližuje následující pozměněné schéma z kapitoly 7.4, kde se sledují daňová a energetická problematika.



7.5.2.1. Schéma

Agregační funkce $@_1$ a $@_2$ by typicky byly průměr, závěrečná funkce $@'_G$ by brala v úvahu i váhy jednotlivých problematik, které voliči vyplnily.

Oba dva výše uvedené přístupy splňují požadavky 1), 2), 4) i 5). Body 1) a 2) jsou zjevné. Bod 4) plyne z monotonie agregačního operátoru. Bod 5) je splněn, protože výstup agregační funkce $@'_G$ pro objekt *o* není závislý na hodnocení ostatních objektů.

7.6. Závěr kapitoly

V této kapitole jsme si přiblížili uživatelské preference. V 7.2 jsme definovali tři modely pro výpočet preferencí, které uživatel nezadal (Chybějící hodnocení komplexních objektů, Chybějící hodnocení atributů a Chybějící hodnocení vlastností). V další kapitole se budeme podrobně zabývat rychlostí nalezení nejlepších *k* objektů vzhledem k uživatelské agregační funkci, tedy modelem Chybějící hodnocení komplexních objektů.

Zbylé dva modely výpočtu uživatelského hodnocení již přinesly komplikace v podobě nekonzistentnosti uživatelského hodnocení. Tyto modely se snaží tuto nekonzistenci zachytit a zohlednit.

Tyto metody je možné použít, i pokud uživatel hodnocení zadal. Vypočtené hodnocení může reflektovat další vztahy a může tedy být přesnější. Zde je důležité si uvědomit, že některé objekty lze jen těžko hodnotit přímo, protože jsou abstraktní povahy a obtížně si uživatel představuje všechny praktické důsledky.

Ukázali jsme, že je Arrowův paradox je vlastně vícehodnotový, kde se fuzzy funkce uvažují v uspořádávacím přístupu. Rozšířili jsme požadavky na sociální agregační funkci *s* i pro hodnotový přístup. Dále jsme rozšířili smysl kandidátů o jejich atributy, čímž jsme převedli teorii z výrokové do predikátové logiky. Nakonec jsme vytvořili paralelu mezi Arrowovým paradoxem a námi navrženým skupinovým rozhodováním navrženém v kapitole 7.4.

8. Problém top k

8.1. Úvod do problému

V této části si přiblížíme problém nalezení $k \geq 1$ nejlepších objektů vzhledem k uživatelské ohodnocující funkci $@^U$. Tento problém budeme označovat jako problém top-k, algoritmus řešící problém top-k potom jako algoritmus top-k. Model problému a struktura dat je převzata z [FLN]. Značení jsme ponechali stejné, jako je v kapitole 7. Je uvažován nejjednodušší model s lineárně uspořádanými doménami atributů. V této části již nevstupují do hry uživatelova nekonzistence nebo nepřesnost v hodnocení, snažíme se nalézt *k* nejlepších objektů co nejrychleji. Celý algoritmus se bude týkat uživatele *U*.

Algoritmus pracuje s objekty, které mají *M* atributů. Typicky budeme označovat objekt písmenem *o* a jeho atributy písmeny $a^1,...,a^M$ .V případě, že budeme potřebovat označit více objektů, budeme je označovat $o_1, o_2,...$, *i*-tý atribut objektu o_j potom a_j^i. Předpokládáme, že známe uživatelovu agregační funkci $@^U$ pro atributy $a^1,...,a^M$. Pro zkrácení a zpřehlednění notace budeme psát hodnocení objektu o s atributy a^i jako $@^U(o)$ místo korektnějšího $@^U(a^1,...,a^M)$.

V podstatě tedy hledáme uživatelovo hodnocení objektu $r^U(o)$, na základě jeho hodnocení atributů objektu $r_i^U(a^i)$ a jeho agregační funkce $@^U$.

Doména atributů bude množina *T* uživatelských hodnocení z kapitoly 1. V případě, že atributy a^i nejsou prvky množiny *T*, ale jiné množiny D_{A^i}, budeme využívat funkci $r_i^U : D_{A^i} \to T$. Tato funkce převádí prvky množiny D_{A^i} na prvky množiny *T*. Součástí této funkce bude uživatelská fuzzy množina daného atributu. Těchto převádějících funkcí budeme potřebovat *M*. Pokud množina *T* odpovídá tradičně intervalu [0,1], ale doména prvního atributu je množina {Nejlepší, Lepší, Neutrální, Horší, Nejhorší}, musíme použít převádějících funkci. Tu implementujeme např. následující tabulkou

D_{A^i}	T
Nejlepší	1
Lepší	0.7
Neutrální	0.5
Horší	0.3
Nejhorší	0

8.1.1. Tabulka

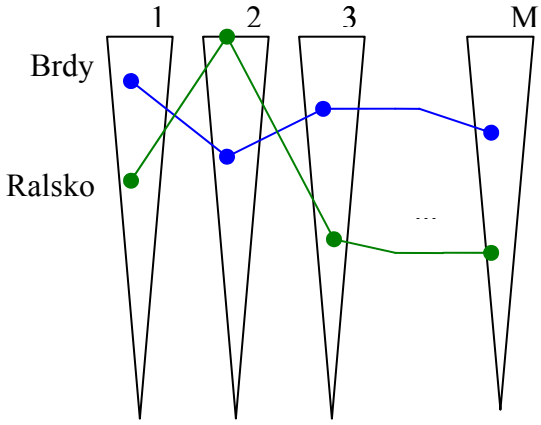
V dalším textu budeme již předpokládat, že $a^i \in T$.

Top-k algoritmus pracuje se setříděnými seznamy. Prvky seznamu L_i jsou atributy a^i spolu s objekty *o*, tedy $L_i^U = [(o_{i_1}, a_{i_2}^i), (o_{i_2}, a_{i_2}^i), ..]$, kde platí $a_{i_1}^1 \geq a_{i_2}^1 \geq ...$. Například seznam terénů bude vypadat takto: $L_{terén}^{Blesk} = [(\text{Brdy},0.9), (\text{Krkonose},0.8), ..]$. Vlastní terén je již převeden do množiny *T*. Seznamy jsou setříděné podle atributů sestupně, nejvhodnější atribut je tedy na vrcholu seznamu.

Pro čtení ze seznamů budeme používat sekvenční přístup. V průběhu běhu bude algoritmus postupovat ve všech seznamech směrem dolů. Hodnotu posledního viděného atributu v seznamu *i* budeme značit $\underline{a^i}$. $\underline{a^i}$ tedy označuje místo, kam jsme v daném seznamu

došli., všechny předchozí hodnoty byly větší nebo rovno $\underline{a^i}$ a všechny následující hodnoty budou menší nebo rovno $\underline{a^i}$.

Prvky typicky nebudou uspořádané ve všech seznamech stejně, některý bude nejlepší v jednom atributu, ale horší ve všech ostatních, další může mít všechny atributy průměrné.

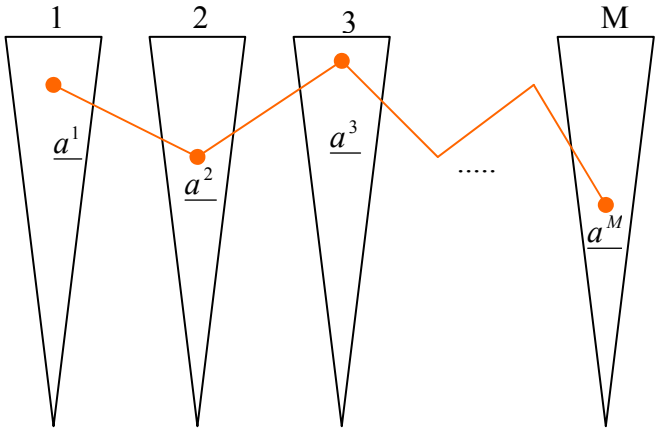


8.1.2. Schéma

Náš cíl je najít objekt *o* s největším uživatelským hodnocením $@^U(o)$ bez nutnosti procházet všechny seznamy až na konec. Jelikož všechny seznamy jsou setříděné sestupně a agregační funkce $@^U$ je monotónní ve všech proměnných, tento objekt se bude nalézat v horních částech všech seznamů. Se sestupem ve všech seznamech nalézáme objekty s menším a menším hodnocením. Z definice agregačního operátoru platí, že pro objekty o_1 a o_2, jejichž atributy splňují podmínku $a_1^1 \le a_2^1,...,a_1^M \le a_2^M$, platí nerovnost $@^U(o_1) \le @^U(o_2)$.

8.1.3. Definice

Virtuální prvek *t*, jehož atributy budou $\underline{a^1},...,\underline{a^M}$, nazveme ***práh***.



8.1.4. Schéma - prahová hodnota

Hodnocení $@^U(t)$ je největší hodnocení, jaké mohou nabývat objekty, které jsme zatím nenašli v žádném seznamu. Navíc pokud u některého objektu chybí *i*-tý atribut, jeho hodnota bude nejvýše hodnota $\underline{a^i}$.

8.1.5. Návaznost na teoretickou část

Problém top-k lze vyjádřit také v jazyku vícehodnotové logiky a uživatelských preferencí. Seznamy jsou setříděné podle uživatelského hodnocení atributu, tedy podle ohodnocení faktu r_i^U. Agregační funkce $@^U$ použitá pro výpočet hodnocení objektu *o* je agregační operátor.

8.2. Přehled algoritmů

V této části si přiblížíme pět algoritmů navržených ve [FLN], kde je také dokázána optimálnost těchto algoritmů. Algoritmy rozdělíme na tři kroky:

1) Zpracování aktuálních objektů
2) Postup ve všech seznamech
3) Kontrola konce algoritmu

Budeme se zabývat jen kroky 1) a 3). Krok 2) je pro všechny algoritmy stejný, v každém seznamu se posuneme o jednu pozici dolů. Množinu *k* nejlepších objektů budeme označovat jako TOPK. Spolu s objekty budeme uchovávat i jejich atributy, v případě, že chybí atribut na *i*-té pozici, bude na jeho místě hodnota null.

8.2.1. Naivní algoritmus

Toto je nejjednodušší ze všech čtyřech algoritmů. Musí si uchovávat informace o všech nalezených objektech, paměťové nároky jsou tedy úměrné počtu objektů.

Zpracování aktuálních objektů

Pokud prvek v seznamu *i* již máme v TOPK, přidáme k němu *i*-tý atribut, jinak objekt s atributem *i* přidáme do TOPK.

Kontrola konce algoritmu

Pokud jsme již ve všech seznamech na konci, algoritmus končí. Na závěr ohodnotíme všechny objekty, setřídíme je podle hodnocení a vrátíme prvních *k*.

8.2.2. Prahový algoritmus

Tento algoritmus využívá přímého přístupu do seznamu. Zadáním jména objektu o do seznamu *i* obdržíme hodnotu a^i. Dále algoritmus obsahuje práh.

Tento algoritmus vyžaduje uchovávat pouze množinu *k* nejlepších objektů, má tedy konstantní paměťovou náročnost.

Zpracování aktuálních objektů

Pro každý objekt nalezený v tomto kroku si přímým přístupem do ostatních seznamů najdeme hodnoty jeho atributů. Tím zajistíme, že u všech objektů známe všechny atributy.

Pokud hodnocení nově nalezeného objektu *o* je větší než hodnocení *k*-tého nejlepšího objektu, přidáme jej do TOPK. Poslední prvek TOPK vyhodíme. Tím zajistíme, že nám stačí seznam o velikosti *k*.

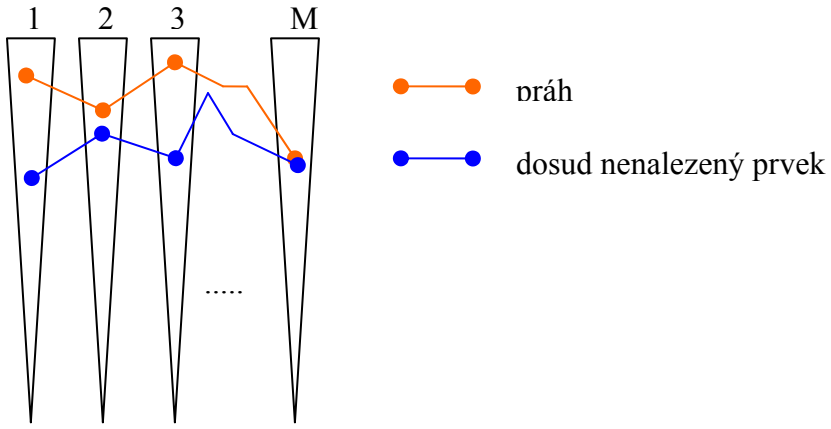
Pokud hodnocení nově nalezeného objektu *o* je menší větší než hodnocení *k*-tého nejlepšího objektu, objekt *o* vyhodíme.

Vypočteme nové hodnocení prahu.

Kontrola konce algoritmu

Pokud hodnocení *k*-tého objektu je větší než hodnocení prahu, algoritmus končí. Toto plyne ze dvou faktů:

1. Žádný nově nalezený objekt už nemůže mít hodnocení větší než práh, tedy ani větší než *k*-tý prvek. To je znázorněno na schématu 8.2.3.
2. V okamžiku, kdy rozhodujeme, zda objekt *o* vyhodíme nebo umístíme do TOPK, známe všechny atributy objektu *o* i objektů v TOPK. To znamená, že můžeme s konečnou platností rozhodnout, zda *o* patří do TOPK nebo ne.



8.2.3. Schéma

Pokud nějaký prvek nalezneme a vyhodíme, je pro nás informace o něm ztracená. Pokud bychom upustili od konstantní paměťové náročnosti, mohli bychom si uchovávat i informace o vyhozených prvcích, čímž by se zmenšil počet zbytečných prohledávání.

8.2.4. Algoritmus bez přímého přístupu

Tento algoritmus využívá pouze sekvenční přístup do všech seznamů.

V průběhu algoritmu typicky nebudeme znát všechny atributy všech objektů. Zavedeme proto dva nové pojmy.

Předpokládejme, že u objektu *o* známe pouze prvních *l* atributů. Zavedeme dvě nová hodnocení objektu pro případ, že nám zbylé atributy objektu chybí. V případě spodního ohodnocení nahradíme chybějící atributy hodnotou 0, v případě horního ohodnocení nahradíme chybějící atributy hodnotou 1, případně hodnotou $\underline{a^i}$.

8.2.4.1. Definice

Spodní ohodnocení objektu o budeme označovat $W_l(o)$ a počítat jako $W_l(o) = @(a_1,..,a_l,0,0,...,0)$.

Horní ohodnocení objektu o budeme označovat $B_l(o)$ a počítat jako $B_l(o) = @(a_1,..,a_l,1,1,...,1)$. Pokud známe hodnotu prahu, pak horní ohodnocení lze také počítat vzorcem $B_l(o) = @(a_1,..,a_l,\underline{a^{l+1}},\underline{a^{l+2}},...,\underline{a^M})$.

8.2.4.2. Pozorování

$W(o) \le @(o) \le B(o)$

Plyne přímo z monotonie agregační funkce $@^U$.

Dále zavedeme novou množinu kandidátů C. C je množina objektů, jejichž horní ohodnocení je větší než dolní ohodnocení *k*-tého nejlepšího prvku. Zde se uchovávají informace o všech nalezených objektech, které nejsou v TOPK, paměťové nároky jsou tedy úměrné počtu objektů. Množina kandidátů obsahuje ty prvky, které se ještě mohou dostat do TOPK.

Prvky v TOPK řadíme podle dolního ohodnocení. V případě rovnosti dolních ohodnocení prvků *x* a *y* rozhoduje horní ohodnocení, v případě rovnosti i horních ohodnocení je vybrán náhodný prvek *x* nebo *y*.

Zpracování aktuálních objektů

Vypočteme nové hodnocení prahu.

Pro každý nově nalezený objekt *o* spočteme jeho horní a spodní ohodnocení. Pokud jsme již objekt viděli, přidáme nově nalezený atribut do seznamu stávajících.

Pokud je dolní ohodnocení mezi *k* nejlepšími, přidáme *o* do TOPK, odebereme jej z množiny C a vyhodíme poslední prvek *x* z TOPK. Pokud je horní hodnocení vyhozeného prvku *x* větší než spodní hodnocení *k*-tého prvku, přidáme *x* do seznamu kandidátů.

Pokud je horní ohodnocení *o* větší než spodní hodnocení *k*-tého prvku, přidáme *o* do seznamu kandidátů.

Kontrola konce algoritmu

Pokud je množina C prázdná, neexistuje prvek, jež by mohl mít lepší hodnocení než *k*-tý nejlepší prvek. Algoritmus může tedy skončit, pokud TOPK obsahuje *k* prvků a prahová hodnota je menší než spodní hodnocení *k*-tého prvku.

Příklad průběhu algoritmu bez přímého přístupu

Mějme seznamy

Terén	Město
Brdy, 0.9	Krkonoše, 0.7
Orlík, 0.4	Brdy, 0.4
Krkonoše, 0.2	Orlík, 0.1

Dále agregační funkci $@^{U}(x,y) = {(x+2y)}/{3}$ a položíme *k*=1.

1) Nejprve vezmeme Brdy, které jsou na začátku seznamu s terény. W(Brdy)= (0.9+2*0)/3= 0.3. Jelikož je množina TOPK prázdná, umístíme Brdy do TOPK.

TOPK = { Brdy, 0.3 }, C = 0, práh = (0.9, 0.7)

2) Nyní vezmeme Krkonoše v seznamu měst. W(Krkonoše)= (0+2*0.7)/3= $0.4\overline{6}$, což je víc, než mají Brdy a proto Krkonoše umístíme do TOPK. B(Brdy)=(0.9+2*0.7)/3= $0.7\overline{6}$. Brdy tedy ještě mohou být v TOPK, umístíme je do množiny kandidátů C.

TOPK = { Krkonoše, $0.4\overline{6}$ }, C = { Brdy }, práh = (0.9, 0.7)

3) Vezměme Orlík v seznamu terénů. W(Orlík)=(0.4+2*0.0)/3= $0.1\overline{3}$, což je méně, než mají Krkonoše. B(Orlík) = (0.4+2*0.7)/3= 0.6 . To je ale více než spodní hodnocení Krkonoš, tedy Orlík umístíme do množiny kandidátů

TOPK = { Krkonoše, $0.5\overline{3}$ }, C = { Brdy, Orlík }, práh = (0.4, 0.7)

4) Dále jsou Brdy v seznamu měst. W(Brdy) = (0.9+2*0.4)/3= $0.5\overline{6}$. To je více než spodní hodnocení Krkonoš, Brdy tedy půjdou do TOPK. B(Krkonoše) = (0.4+2*0.7)/3= 0.6 ,

tedy více než spodní hodnocení Brd, Krkonoše tedy přemístíme do kandidátů. Spočítáme horní ohodnocení Orlíku B(Orlík) = (0.4+2*0.4)/3= 0.4 , což je méně než dolní hodnocení Brd, takže Orlík vyhodíme i z množiny kandidátů.

V tomto okamžiku je ohodnocení prahu @(práh)= (0.4+2*0.4)/3= 0.4. Již tedy nemá smysl kontrolovat nové objekty, které nalezneme dále, všechny budou mít hodnocení menší než 0.4 a tedy nebudou mít naději se dostat do TOPK. Nyní se do TOPK mohou dostat pouze objekty z množiny kandidátů C.

TOPK = { Brdy, $0.5\overline{6}$ }, C = { Krkonoše }, práh = (0.4, 0.4)

5) Dále jsou Krkonoše v seznamu terénů. W(Krkonoše) = (0.2+2*0.7)/3= $0.5\overline{3}$. To je méně než spodní hodnocení Brd. Dále B(Krkonoše) = (0.2+2*0.7)/3= $0.5\overline{3}$, tedy méně než spodní ohodnocení Brd, vyhodíme tedy Krkonoše z množiny kandidátů.

TOPK = { Brdy, $0.5\overline{6}$ }, C = { }, práh = (0.2, 0.4)

6) Množina kandidátů C je prázdná a v TOPK máme k=1 prvků, algoritmus tedy končí.

8.2.5. Kombinovaný algoritmus

Kombinovaný algoritmus vychází z algoritmu bez přímého přístupu. Rozdíl je v tom, že každých h kroků si z množiny objektů, kterým chybí některé atributy, vybere objekt o s nejvyšším horním ohodnocením B(o) a nalezne přímým přístupem všechny jeho chybějící atributy.

Tento algoritmus tedy využívá hlavně sekvenční přístup, občas si pomůže přímým přístupem. Konstanta h bude záviset na ceně přímého a sekvenčního přístupu. Čím dražší bude přímý přístup, tím bude h větší.

Zpracování aktuálních objektů

Jako algoritmus bez přímého přístupu.

Každých h kroků najdi objekt o s chybějícími atributy a s nejvyšším B(o) a pro něj nalezni přímým přístupem chybějící atributy.

Kontrola konce algoritmu

Stejná jako u algoritmu bez přímého přístupu.

9. Obecný nástroj pro top-k algoritmus Xoda

9.1. Průběh vývoje nástroje Xoda

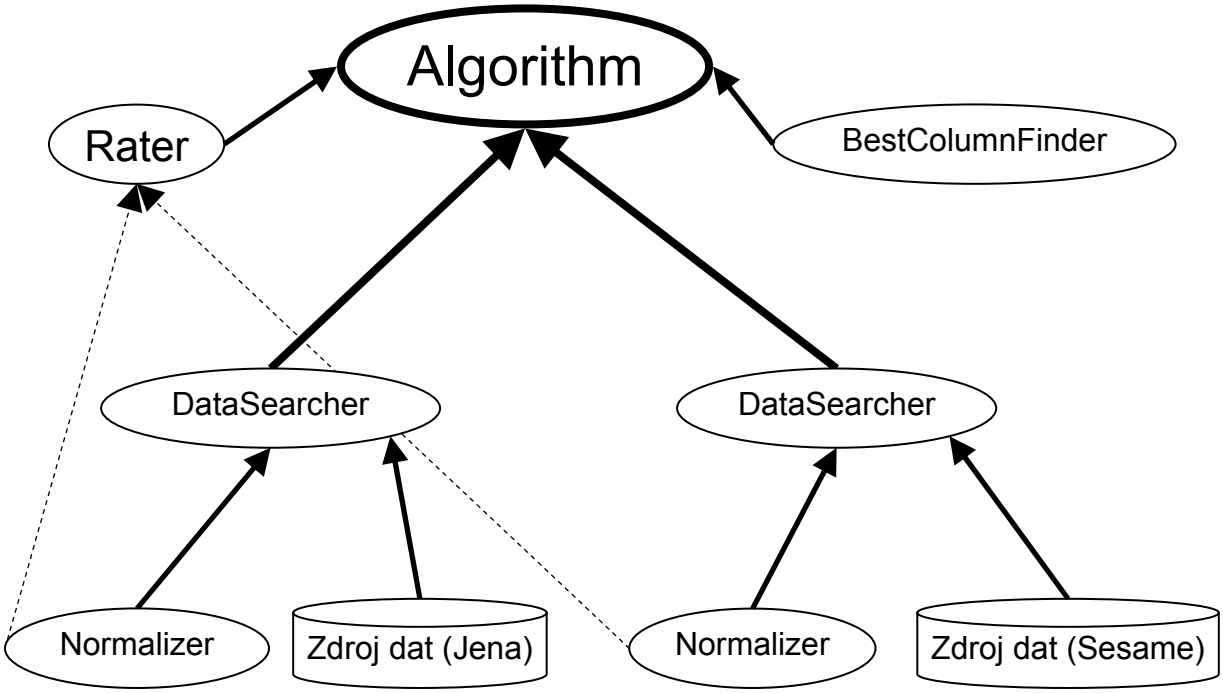
Prvotní motivací bylo vytvořit nástroj pro algoritmus hledající k nejlepších objektů v prostředí RDF. RDF bude použito jako zdroj dat pro algoritmus. Jedna z nejrozšířenějších RDF databází je Sesame, více v [SESAME].

Po krátkém studiu této databáze jsme objevili, že její dotazovací jazyk SeRQL nepodporuje operaci třídění. Výsledky dotazů se tedy musí třídit přímo v paměti počítače. V jiných ohledech se Sesame ukázal jako kvalitní RDF databáze, s příjemným grafickým web-prostředím přes http server Apache.

Další známá RDF databáze je Jena ([JENA]). Jena byla vybrána proto, že podporuje dotazovací jazyk SPARQL, který obsahuje order by klauzuli. Tím je umožněno plnohodnotné nasazení algoritmu top-k. Není již nutné třídit data v programu, ale přímo v databázi Jena. Bohužel zkoumáním velmi slabé výkonnosti databáze Jena jsme zjistili, že Jena třídění vykonává v paměti po načtení všech výsledků do paměti. Navíc trvání dotazu u Jeny s order by klauzulí je řádově delší než bez order by se setříděním dat v programu.

Z těchto technických důvodů jsme se rozhodli navrhnout program tak, aby byl co nejvíce nezávislý na použitých technologiích. Mělo by být možné lehce doprogramovat moduly pro přístup k jiné databázi, moduly s jinými agregačními funkcemi apod. Navíc ale musí být snadné použití tohoto algoritmu se stávajícími moduly.

9.2. Přehled komunikace datových typů nástroje



2.1.1. Schéma

Výše uvedené schéma reprezentuje komunikaci datových typů použité v Xodě. Šipky značí využívání funkcí komponent. Třída DataSearcher získává data, která předává třídě Algorithm. Ta je zpracovává, krom jiného pomocí třídy Rater, která reprezentuje agregační

funkci uživatele. Pro výběr sloupce se používá interface BestColumnFinder. K normalizaci hodnot z databáze na interval [0,1] se používá interface Normalizer.

Všechny výše uvedené komponenty jsou buď rozhraní nebo abstraktní třídy. Tím je zaručena velká obecnost – lze používat libovolné nové implementace nebo již existující zakomponovat do jiného programu.

Xoda obsahuje další třídy, které usnadňují některé časté operace nebo naopak komplexní výpočty.

9.3. TopKElement

Třída TopKElement představuje objekty, jejichž pořadí chceme zjistit.

TopKElement obsahuje jméno objektu a jeho atributy, které jsme již nalezli, spolu s jejich hodnocením, dále potom data o nalezení objektu - jeho pozice v každém seznamu a krok algoritmu, kdy byl objeven v daném seznamu. Pokud chybí některý atribut, je na jeho místě hodnota null.

9.4. Normalizer

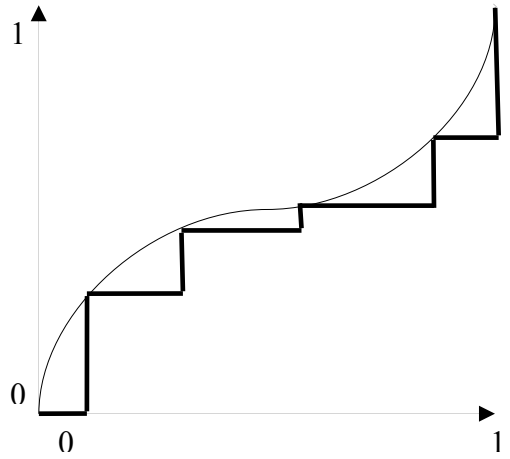
Normalizer reprezentuje funkci $r_i^U : D_{A^i} \to T$ z definice top-k algoritmu. Převádí data na množinu *T*. V našem případě jsme se rozhodli použít interval $[0,1]$. Normalizer má dvě funkce. Za prvé převádí data na číselný typ a za druhé se zde aplikuje uživatelská fuzzy množina.

Pokud data pro převedení již jsou z intervalu $[0,1]$, pak se vezme toto číslo a upraví se pomocí fuzzy funkce uživatele. V základním Normalizeru, třídě SimpleNormalizer, je tato funkce reprezentována třídou Ordering. Vytvořili jsme tři ukázkové podtřídy Ordering. Třída AscendingOrdering vrací číslo, které dostala. Tím vytváří vzestupné řazení. Třída DescendingOrdering vrací 1-vstup, čímž vytváří sestupné řazení. Nakonec OnePeekOrdering vrací vzdálenost od stanoveného ideálního bodu. Tato vzdálenost je normalizovaná na interval $[0,1]$.

Pokud data nejsou v intervalu $[0,1]$, musí se nejdřív na tento interval převést. To je již věc jednotlivých implementací Normalizeru.

Zmíníme zde HashMapNormalizer, který dostane na vstupu typ HashMap, což je asociativní pole. Klíče tohoto pole budou prvky a hodnoty jsou hodnocení prvků. Tento Normalizer je využit v pomocné třídě MultipleRatings. Vstup HashMapNormalizeru může být totiž i výstup nějakého předchozího top-k algoritmu. Potom tento Normalizer dostane objekt a vrátí jeho uživatelské hodnocení, spočítané v předchozím top-k algoritmu. Díky tomu lze lehce naimplementovat vícestupňový top-k, jak je znázorněn na schématu 7.4.2.1 nebo 7.4.2.2.

Nakonec jsme vytvořili LevelNormalizer, který výstup nějakého jiného Normalizeru zaokrouhlí. Hodnoty, na něž má zaokrouhlovat, dostane na vstupu. Příklad funkce LevelNormalizeru je na schématu 9.4.1. Výstup LevelNormalizeru je zvýrazněn tučně.



9.4.1. Schéma

Seznamy, jež jsou vstupem top-k algoritmu, by měly být setříděny podle použitého Normalizeru. Pokud je to obyčejný SimpleNormalizer s AscendingOrdering nebo DescendingOrdering, je vše v pořádku. Ovšem setřídění, odpovídající složitějšímu Normalizeru, již v běžném dotazovacím jazyku nenapíšeme a pak nám nezbývá nic jiného než seznamy setřídit v paměti.

9.5. Rater

Abstraktní třída Rater hraje roli agregační funkce $@^U$. Používá se pro počítání hodnocení objektu z hodnocení jeho atributů. Tuto funkci zajišťuje metoda getRating.

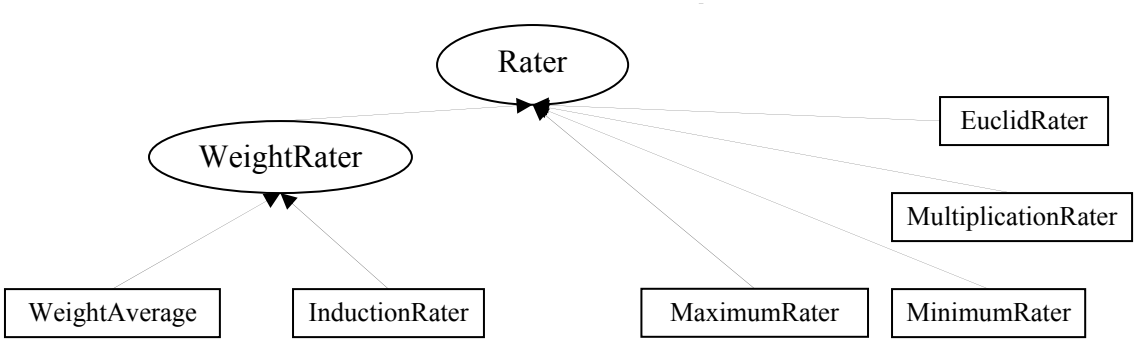
Metoda getRating tedy dostane na vstup TopKElement a sadu DataSearcherů spolu s jejich Normalizery. Atributy objektu převede pomocí jednotlivých Normalizerů na interval $[0,1]$ a z těchto čísel nakonec vypočítá hodnocení objektu. Navíc může dostat i hodnoty, které se mají použít v případě chybějícího atributu.

Jak rychle algoritmus top-k skončí, je z velké části dáno i agregační funkcí. Agregační funkce vystihuje, jak moc se hodnocení objektu zhorší, pokud je nějaký atribut špatně hodnocený. Vezměme si extrémní případ - minimum. Pokud tato agregační funkce bude použita v algoritmu bez přímého přístupu, většina objektů bude mít dlouho spodní ohodnocení 0, protože typicky nám nějaké atributy chybějí.

Z tohoto důvodu jsme zavedli další abstraktní třídu WeightRater, která je podtřídou třídy Rater. Tato funkce bere v úvahu váhy přiřazené jednotlivým atributům.

Další agregační funkce je EuclidRater. Počítá euklidovskou vzdálenost bodu [1,…,1] od objektu, jehož souřadnice jsou určeny hodnocením jeho atributů. Je pravda, že pro vyhledávání objektů podle vzdálenosti existují i jiné metody, ovšem top-k algoritmus nemůže předpokládat žádné další vlastnosti agregačních funkcí krom monotonie.

InductionRater je použit při generování hodnocení atributů a vlastností, pomocí metody popsané v kapitole 7.2.



9.5.1. Schéma

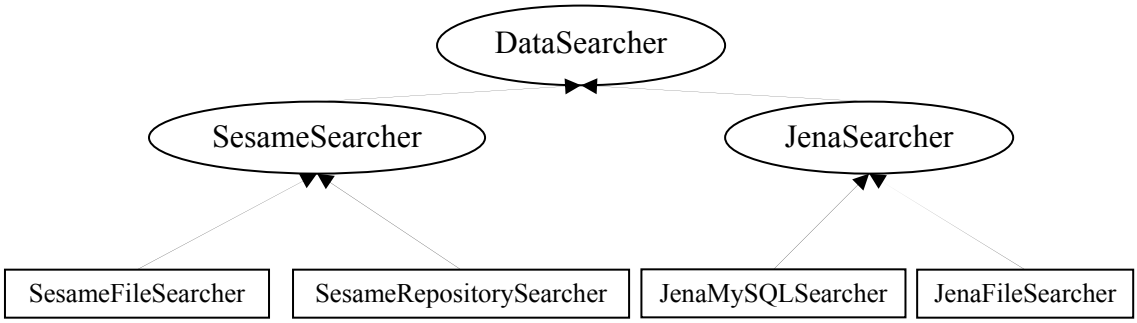
9.6. DataSearcher

Abstraktní třída DataSearcher předává data z nějakého zdroje. Tím, že se zdroj v algoritmu nepoužívá přímo, je zajištěna přenositelnost mezi různými databázemi.

Implementovali jsme dva DataSearchery pro dvě RDF databáze (Sesame a Jenu), jeden pro sekvenční přístup do souboru a jeden pro přímý přístup do paměti.

Posledně jmenovaný dostane jako vstupní parametr nějaký seznam, ze kterého pak vrací prvky. Tato třída je zde zavedena proto, že je umožněno použít výstup několika top-k algoritmů jako vstup dalšího top-k algoritmu. Tento mechanismus je použit např. ve třídě MultipleRatings, která usnadňuje nalezení *k* objektů, které vyhovují více uživatelům. Blíže je problematika rozebrána v kapitole 7.4.

DataSearchery pro RDF databáze jsme rozdělili do několika podtříd, jak je znázorněno na následujícím schématu

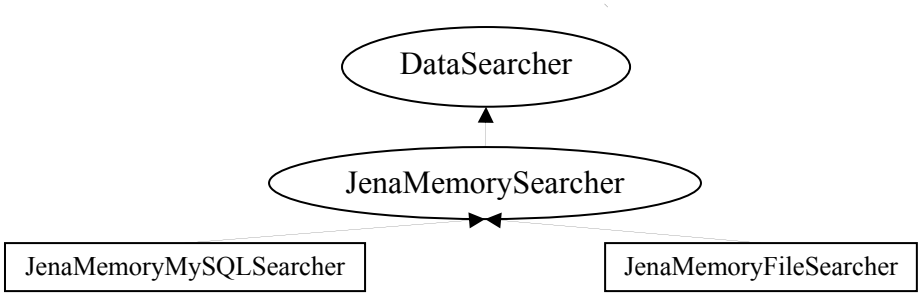


9.6.1. Schéma

SesameSearcher neumožňuje setřídění výsledků přímo v úložišti. Pracuje tak, že nejdříve načte všechny výsledky vrácené Sesamem do paměti a tam je setřídí. Pro uložení výsledků je použita struktura ArrayList.

Z důvodu špatné rychlosti třídění JenaSearchu jsme se rozhodli implementovat ještě jeden DataSearcher založený na RDF databázi Jena. JenaMemorySearcher načte výsledky z Jeny do paměti, kde je setřídí a tím se ušetří čas za třídění v Jeně. Při testech se navíc ukázalo, že přeuložené výsledky ve struktuře ArrayList pracují rychleji než samotné uložení dat v Jeně.

Jena teoreticky podporuje třídění pomocí jazyka SpaRQL. Po provedení dotazu se výsledky vrací sekvenčně po jednom. Jelikož ale Jena provádí třídění také v paměti, je celý výsledek dotazu uložen nějakým způsobem v Jeně, která z něj pak vrací další záznamy.



9.6.2. Schéma

Nakonec poznamenejme, že pro ideální funkci top-k algoritmu v prostředí internetu je možné implementovat DataSearcher, který bude získávat data z webovských služeb nebo jiných internetových zdrojů. Tím bude zajištěno, že data budou aktuální.

9.7. BestColumnFinder

Interface BestColumnFinder reprezentuje heuristiku, která určuje sloupec, ve kterém se bude dále postupovat. Jako vstup dostane objekt typu Algorithm, může tedy využívat všechny parametry aktuálního stavu algoritmu.

Heuristiky zmíněné např. ve [GLV] berou v úvahu derivaci agregační funkce, případně vlastní hodnotu dat v daném seznamu. Dále je zde navržena heuristika, která střídá dva výše uvedené přístupy.

Zde poznamenejme, že heuristiky musí splňovat podmínku, že pokud algoritmus ještě v některých seznamech nedošel až na konec, heuristika vrací index některého z těchto seznamů. Pokud už ve všech seznamech byly prohledány všechny prvky, heuristika vrací číslo -1. Tím se zajistí korektní ukončení algoritmu.

9.7.1. MissingValuesFinder

Motivace pro vytvoření nové heuristiky

V kapitole 10.8 jsme testovali heuristiky založené na derivaci agregační funkce. Zjistili jsme, že pro prahový algoritmus jsou tyto heuristiky vhodné, ale algoritmu bez přímého přístupu naopak zpomalují jeho běh.

Proto jsme se rozhodli vytvořit heuristiku přímo pro tento algoritmus. Výše uvedené heuristiky sestupují rychleji v seznamech, kde je větší derivace agregační funkce. V seznamech, kde je tato váha malá, ale postupují velmi pomalu. Proto bude v množině TOPK velmi mnoho chybějících atributů právě u seznamů s malou vahou. To se při testování konce algoritmu promítne následovně :

1) *k*-tý nejlepší prvek bude mít malé spodní ohodnocení, protože mu budou chybět některé atributy

2) prvky z kandidátské množiny budou mít vysoké horní ohodnocení, protože jim sice také chybějí stejné atributy jako *k*-tému prvku, ale tyto atributy se nahradí atributy prahu. Práh má v těchto atributech velmi vysoké hodnocení.

Ukazuje se, že použití těchto heuristik zpomaluje algoritmus, který musí prozkoumat více řádek než při sekvenčním přístupu.

Popis heuristiky

Heuristika MissingValuesFinder bere v úvahu možné chybějící atributy *k* nejlepších objektů a snaží se tento počet minimalizovat. Je to motivováno tím, že chceme co nejvyšší spodní hodnocení *k*-tého nejlepšího prvku. Jakmile hodnota prahu překročí hodnotu *k*-tého

prvku, uzavře se nám množina kandidátů. Proto chceme co nejrychleji dosáhnout stavu, kdy práh bude hodnocen hůře než *k*-tý prvek.

Jedna možnost je snižovat hodnotu prahu, což lze například využitím výše zmíněných heuristik. Ukazuje se, že lepší způsob je minimalizování počtu chybějících atributů u *k*-tého objektu, proto budeme hledat v těch sloupcích, kde *k*-tý objektu nemá atribut.

Tuto myšlenku ještě rozšíříme – po nalezení chybějícího atributu *k*-tého objektu se jeho ohodnocení zvýší a s velkou pravděpodobností opustí *k*-té místo v seznamu. Na toto místo spadne (*k-1*)-ní objekt. Budeme tedy hledat počet chybějících atributů pro *h* prvků od *k*-tého objektu, tedy pro (*k-h*)-tý,…,*k*-tý objekt.

Tato heuristika se ukázala jako velmi užitečná, algoritmus bez přímého přístupu se při jejím použití zrychlil, což je významné mimo jiné i proto, že tento algoritmus nepoužívá přímý přístup, který může být velmi drahý.

Navíc, pokud již žádný objekt v TOPK nemá chybějící atribut, další seznam bude vybraný některou z výše uvedených heuristik. To je užitečné, protože tím se bude rychleji snižovat prahová hodnota.

9.8. Algorithm

Abstraktní třída Algorithm provádí vlastní top-k algoritmus. K běhu potřebuje pole DataSearcherů, které mu dodají data, Rater pro vypočtení ohodnocení prvků a BestColumnFinder pro vybrání dalšího seznamu. Po skončení vrátí seznam k nejlepších objektů.

Algoritmus obsahuje krom jiného i určité informace o běhu programu. Jsou to statistiky o tom, jak dlouho které části trvaly. Tyto statistiky se dají použít při ladění výkonu algoritmu.

9.8.1. Algoritmus bez přímého přístupu

Vybrali jsme tento algoritmus pro kvalitní implementaci. Tato volba vyplynula z faktu, že přímý přístup do databáze je řádově dražší než sekvenční přístup k výsledkům dotazu. Tím pádem prahový algoritmus nedosahuje rychlosti naivního algoritmu, byť je řádově lepší v počtu prozkoumaných řádek. Pokud by existovala databáze, kde by byl plnohodnotný sekvenční přístup, tak by se zřejmě naivní algoritmus řádově zpomalil. Tato problematika je více rozebrána v kapitole 10.10 Problémy testování.

Implementace přímo podle popisu ve [FLN] přinesla funkční verzi, která ovšem byla pomalejší než naivní algoritmus.

Problém - Jako nejpomalejší část se ukázala nutnost třídit seznam objektů podle dolního ohodnocení.

Řešení - Upravili jsme algoritmus tak, že nově přidávaný prvek se vsune přímo na místo, kam patří podle jeho dolního ohodnocení. Navíc se prohledává pouze prvních *k* objektů. V případě, že se shoduje dolní ohodnocení, použije se prvek s vyšším horním ohodnocením. To se ale mění spolu se změnou hodnot prahu, proto je nutné udržovat setříděný seznam *k* prvků a navíc všech, které mají stejné dolní ohodnocení jako *k*-tý prvek.

Touto úpravou se algoritmus řádově zrychlil, takže byl přibližně stejně rychlý jako naivní algoritmus. Nejdelší čas nyní zabírala část testování konce algoritmu.

Problém 1) Při testování konce algoritmu se procházejí prvky mimo prvních *k* a počítá se pro ně horní ohodnocení. Pokud má prvek větší ohodnocení než je dolní ohodnocení *k*-tého prvku, přerušíme hledání a algoritmus pokračuje. Pokud ne, prohledává se dále. Pokud dojdeme až na konec seznamu, algoritmus končí, protože neexistuje prvek s vyšším horním ohodnocením než je spodní ohodnocení *k*-tého prvku.

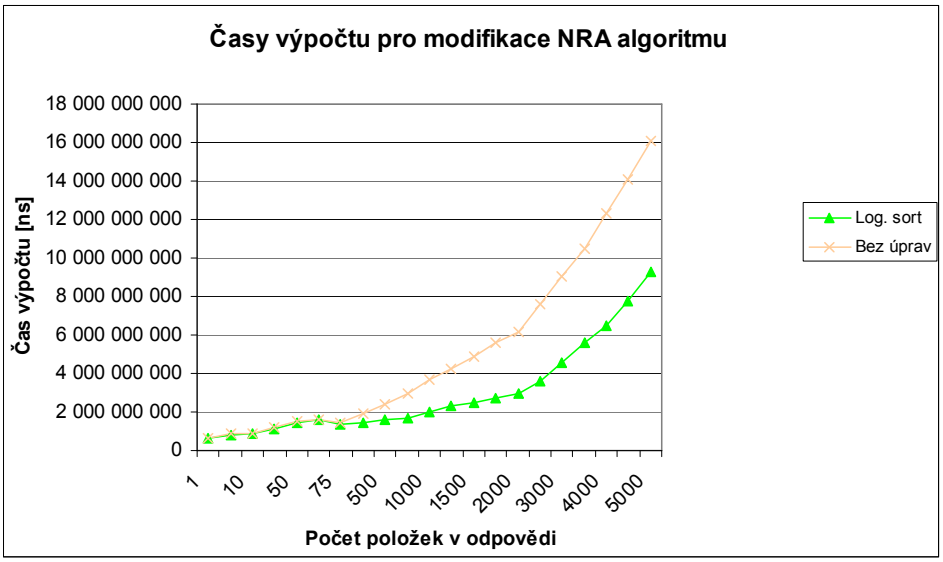
Řešení 1) Prohledáváme seznam tak dlouho, dokud nenarazíme na prvek s větším horním ohodnocením než je dolní ohodnocení *k*-tého prvku. V tom okamžiku se vyhodí všechny předcházející prvky ze seznamu kandidátů, takže se již nebudou příště prohledávat.

Problém 2) Pokud je hodnota prahu menší než je hodnota *k*-tého prvku, již nemá smysl přidávat nově nalezené prvky.

Řešení 2) Jakmile hodnota prahu klesne pod hodnotu *k*-tého prvku, zastaví se přidávání nových prvků. Od tohoto okamžiku se prvky z množiny kandidátů pouze odebírají.

Navrhujeme tři přístupy odebírání prvků ze seznamu. První neodebírá vůbec, druhý nejdříve najde prvek s větším horním ohodnocením než je dolní ohodnocení *k*-tého prvku a následně odebere všechny předchozí prvky a třetí odebírá pokaždé, když narazí na prvek s menším horním ohodnocením než je dolní ohodnocení *k*-tého prvku. V testování přístupů se ukázal nejlepší druhý přístup, tedy nalezení prvku, jehož horní ohodnocení je větší než dolní ohodnocení *k*-tého prvku, a následné odstranění všech předchozích záznamů.

Poslední úprava se týkala setřiďování seznamů. Jelikož je prvních *k* objektů uspřádáno podle hodnocení, můžeme místo nového prvku nalézt pomocí binárního hledání, které je logaritmické vzhledem k velikosti dat. Tím se nám opět algoritmus zrychlil.



9.8.1.1. Graf

Tímto způsobem jsme zrychlili algoritmus bez přímého přístupu na přijatelnou úroveň. Bližší srovnání s dalšími algoritmy je uvedeno v kapitole 10.5 Testování algoritmů.

Stejné heuristiky jsou použity v kombinovaném algoritmu, který se liší od algoritmu bez přímého přístupu pouze tím, že každých *h* kroků si přímým přístupem dohledá všechny atributy prvku s největším horním ohodnocením.

9.9. Pomocné třídy

Xoda také obsahuje sadu pomocných tříd, které usnadňují programátorovi integraci Xody do svého programu.

QueryFinder

Načítá dotazy pro databázi z XML souboru. Tyto dotazy potom využívá DataSearcher pro získání dat. Pokud chceme nějakou aplikaci přenést na jinou databázi, stačí přepsat tento XML soubor. Nahradíme dotazy v původním jazyce ekvivalentními dotazy v jazyce nové databáze.

MultiUserRating

Umožňuje aplikovat top-k algoritmus v prostředí s více uživatelskými hodnoceními. Proces je blíže popsán v kapitole 7.4. Oba dva výše zmíněné přístupy tato třída realizuje pomocí funkcí getFirstGlobal a getFirstLocal. Dále umožňuje získat i mezivýsledky procesů. Funkce getTypeByGlobalPreferences vrátí zadanou třídu setříděnou podle agregace hodnocení jednotlivých uživatelů. Funkce getTypeBySubtypes pak vrátí seznam objektů seřazených podle agregace hodnocení atributů.

UserNormalizers

Ukládá informace o uživatelském Normalizeru do databáze, případně je poté z databáze načítá. Normalizery jsou navázány k nějaké vlastnosti.

Příklad

```
<trip:hasCitiesBlesksNormalizer> <rdf:type>  <mvlp:SimpleNormalizer>
<trip:hasCitiesBlesksNormalizer> <mvlp:hasOrdering>  <mvlp:Ascending>
<trip:hasCities> <trip:BlesksNormalizer> <trip:hasCitiesBlesksNormalizer>
```

UserRatings

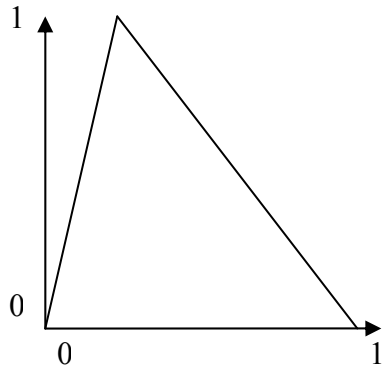
Usnadňuje práci s uživatelskými hodnoceními. Jednoduchým voláním funkce může přidat, odebrat či upravit existující uživatelské hodnocení nějaké entity. Také jsou zde implementovány postupy z kapitoly 7.2 pro generování uživatelských hodnocení. Je možné použít model pro generování hodnocení na vlastnosti i na atributy.

9.10. Závěr kapitoly

Vytvořili jsme nástroj, který dovoluje provádět top-k algoritmus na libovolných datech. Dále je možné změnit agregační funkce, fuzzy funkce pro atributy a heuristiky použité pro výběr sloupce. Naprogramovali jsme základní sestavu, která umožňuje top-k algoritmy pro dvě rozšířené RDF databáze, spolu se základními agregačními funkcemi.

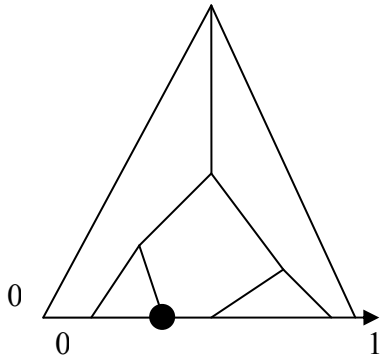
Druhá část se týká teorie z kapitoly 7. a to generování nových vypočítaných uživatelských preferencí a skupinové rozhodování.

Ideální by pro top-k algoritmus byla databáze, která by umožňovala vytvářet vlastní indexy, které by pak odpovídaly uspořádání definovaném různými Normalizery. Při použití B+ stromů by šlo např. naimplementovat i setřídění OnePeakOrdering, které dostane ideální hodnotu atributu, té přiřadí 1 a pak hodnocení klesá oběma směry dolů. B+ strom by pak představoval obecnou fuzzy funkci F_A a F_B z kapitoly 7.1.4.



9.10.1. Schéma

Pomocí B+ stromu se můžeme dostat do vybraného ideálního bodu. Poté budeme zpětně prohledávat další listy, přičemž budeme preferovat nejbližší sousedy ve stromě. Čím později narazíme na list, tím dále bude od ideálního bodu a tím menší bude mít hodnocení.



9.10.2. Schéma

Databáze pro top-k algoritmus může být poměrně jednoduchá - stačí základní podpora dotazů na vlastnost objektu, seřazených podle určitého kritéria. Poté pro přímý přístup by vracela hodnotu vlastnosti nějakého objektu. Největším úskalím je setřiďování podle zadaného kritéria.

10. Testování top-k algoritmu

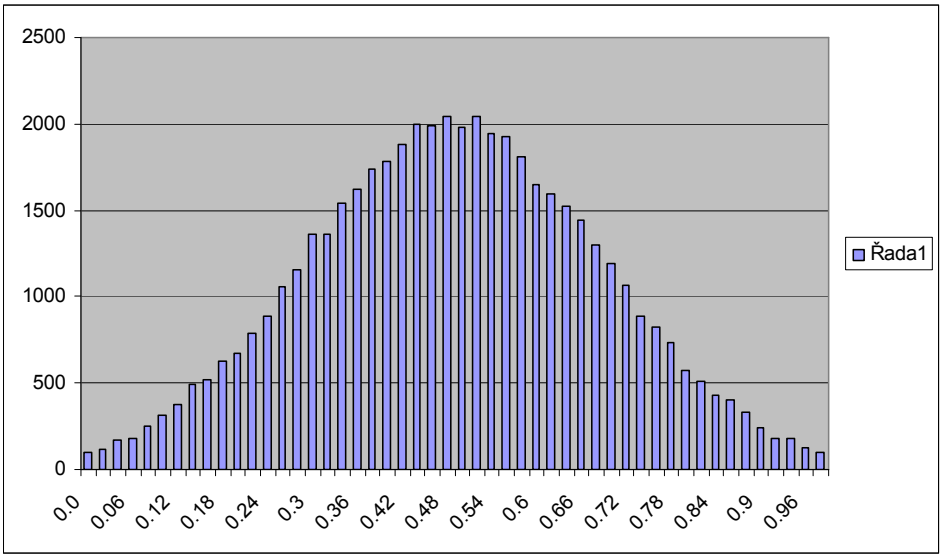
10.1. Struktura dat

Pro testování algoritmů jsme použili testovací data. Testovací data obsahují N objektů každý s M vlastnostmi. Vlastnosti jsme pojmenovali val1,..,val10, objekty potom item0, item1,…, itemX. Vygenerovali jsme různé soubory pro různá M i N. Nakonec jsme ve většině případů použili 5 atributů a 100 000 zdrojů.

Obor hodnot vlastností je desetinné číslo v rozsahu [0,1]. Zvolili jsme tři různá rozložení hodnot těchto vlastností.

1) Normální rozdělení

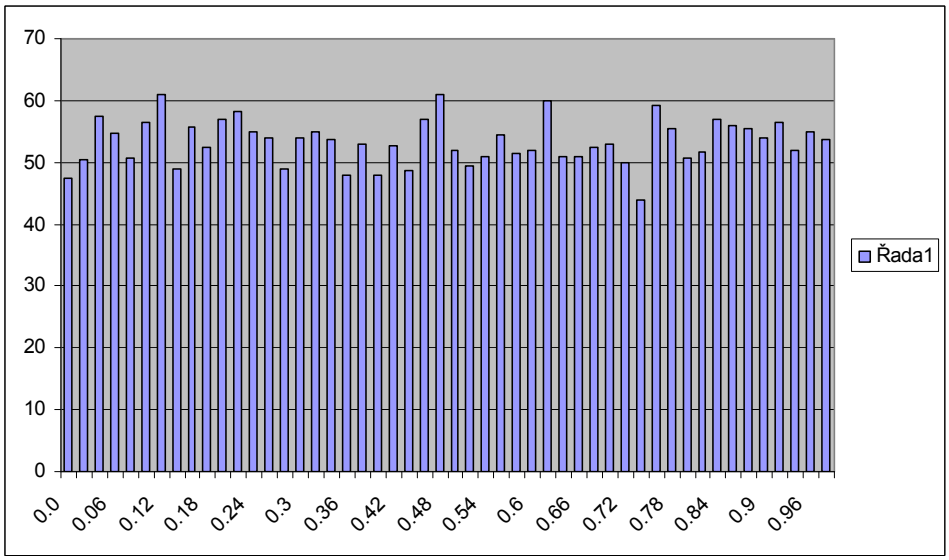
Grafické znázornění rozložení je na grafu 10.1.1.



10.1.1. Graf

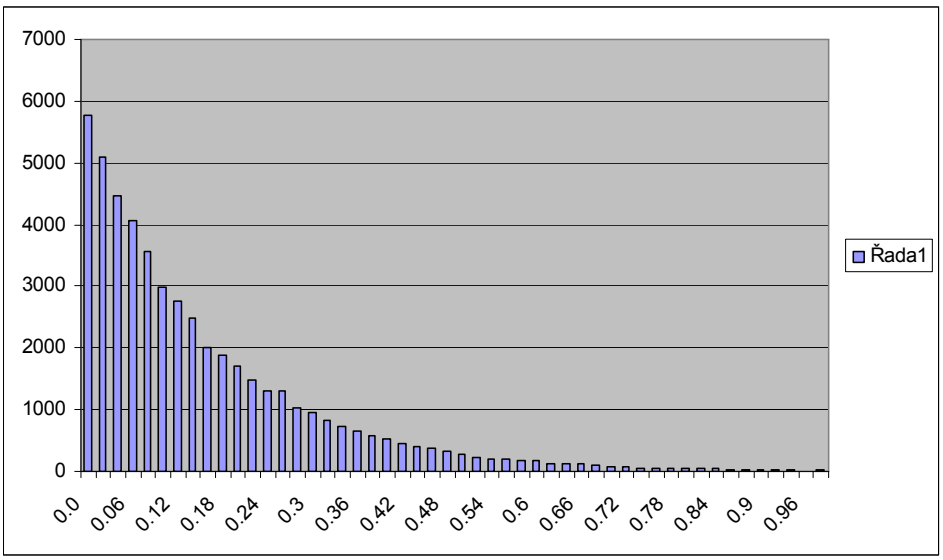
2) Rovnoměrné rozdělení

Grafické znázornění rozložení je na grafu 10.1.2.



10.1.2. Graf

1) Exponenciální rozdělení
Grafické znázornění rozložení je na grafu 10.1.3.



10.1.3. Graf

10.2. Testované veličiny

Z mnoha různých parametrů top-k algoritmu jsme vybrali několik pro testování.

1) Vliv rozložení hodnot vlastností na rychlost algoritmu
2) Porovnání algoritmů mezi sebou
3) Porovnání RDF databází
4) Vliv agregační funkce na průběh algoritmu
5) Vliv heuristik na rychlost algoritmu
6) Speciálně budeme studovat heuristiku MissingValuesFinder

Sledovat budeme :

1) Čas běhu algoritmu
2) Počet prohledaných řádek
3) Pozice, při které hodnota prahu klesla pod hodnotu *k*-tého prvku (pro NRA algoritmus)
4) Časy jednotlivých částí algoritmů – kontrola konce, čas pro získání dat, atd.

10.3. Způsob měření

Všechny testy probíhaly na počítači s 512MB RAM, 1,7 GHz Intel Pentium, několik let starém. Z tohoto pohledu lze říci, že se jedná o spíše pomalejší počítač. Systém byl použit Microsoft Windows XP.

Budeme sledovat několik charakteristik algoritmu. První bude doba běhu algoritmu, další počet prozkoumaných řádek, trvání jednotlivých částí algoritmu a informace o pozici v seznamech při překročení prahu. Doba běhu algoritmu se bere jako čistá doba běhu – neuvažuje se vytváření RDF databází v paměti ani setřiďování výsledků dotazů.

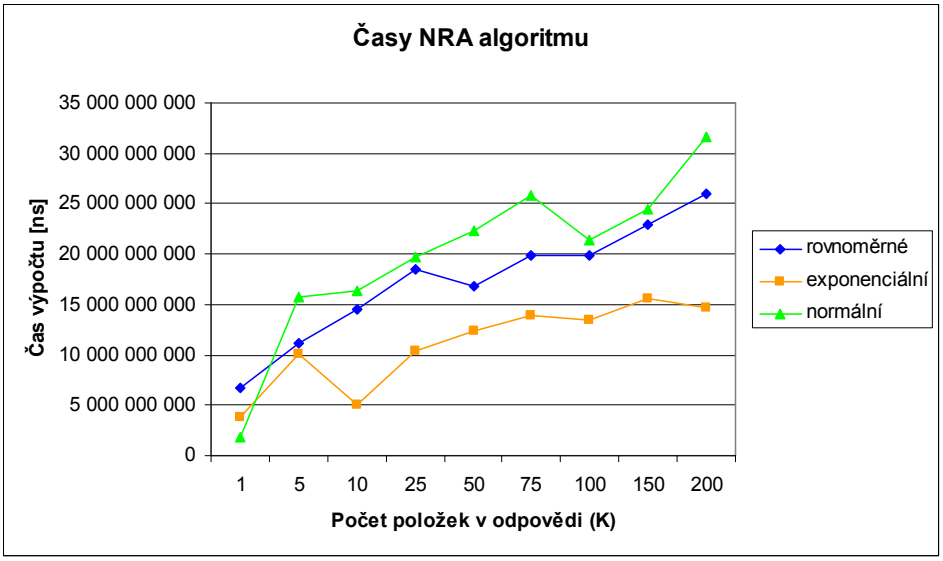
Pro testování Sesamu a Jeny jsme použili Sesame verze 1.2.4 a Jenu 2.4.

10.4. Testování různých rozložení dat

Nejdříve otestujeme různá rozložení dat pomocí algoritmů NRA a prahového algoritmu. Očekáváme, že algoritmy skončí nejrychleji na exponenciálním rozdělení, poté na rovnoměrném a nakonec na normálním. Testované rozdělení je na všech atributech stejné.

Počet atributů jsme stanovili na pět, což je rozumný počet pro uživatele.V testovací množině bylo padesát tisíc zdrojů s pěti atributy. Celkem tedy dvě stě padesát tisíc trojic.

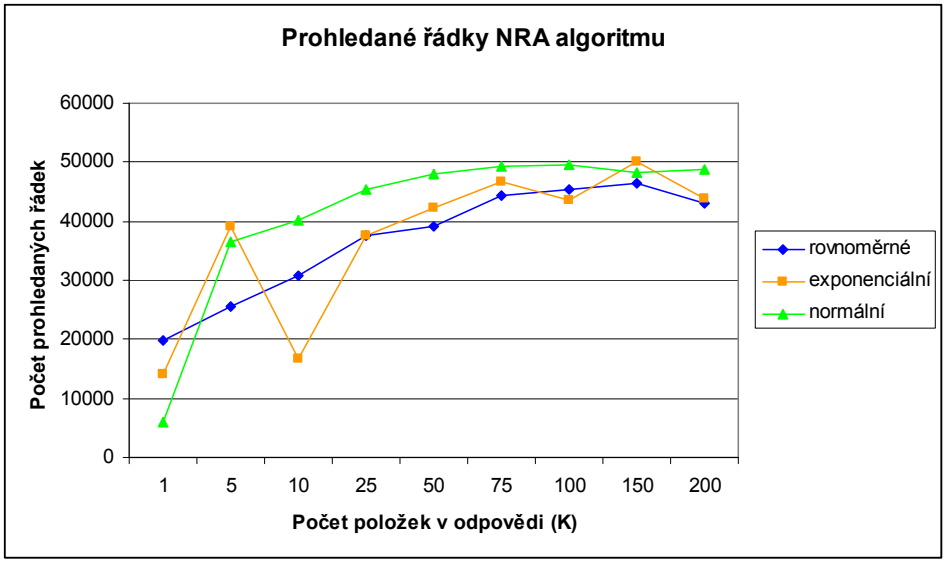
Algoritmus bez přímého přístupu



10.4.1. Graf

Podle předpokladu algoritmus běžel nejrychleji na datech s exponenciálním rozložením hodnot.

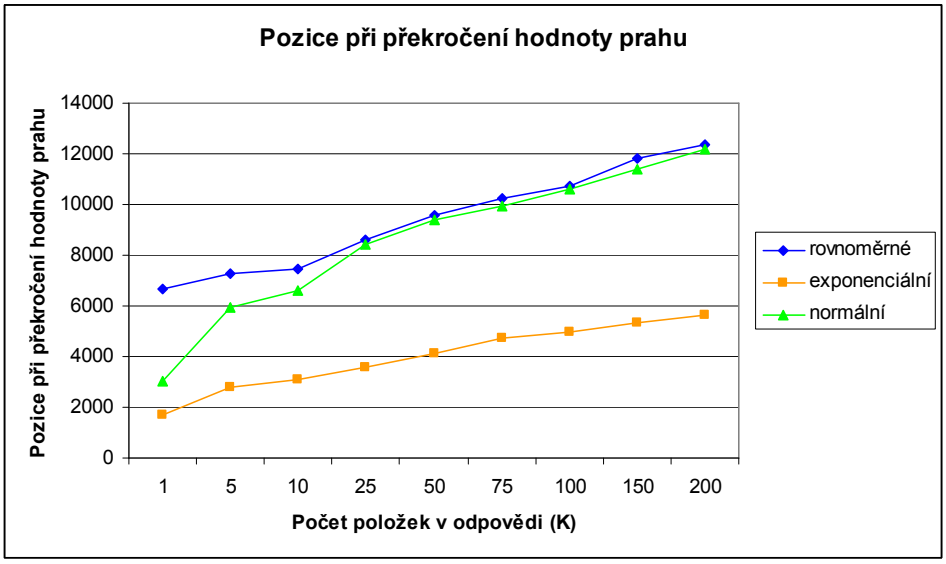
Nyní ještě zkontrolujeme, kolik bylo prohledáno řádek.



10.4.2. Graf

Na grafu 10.4.2 vidíme, že na exponenciálních datech nebylo kromě k=10 prozkoumáno výrazně méně řádek. Data pro k=10 můžeme považovat za shodu okolností, ostatní hodnoty vyjadřují obecný trend dostatečně.

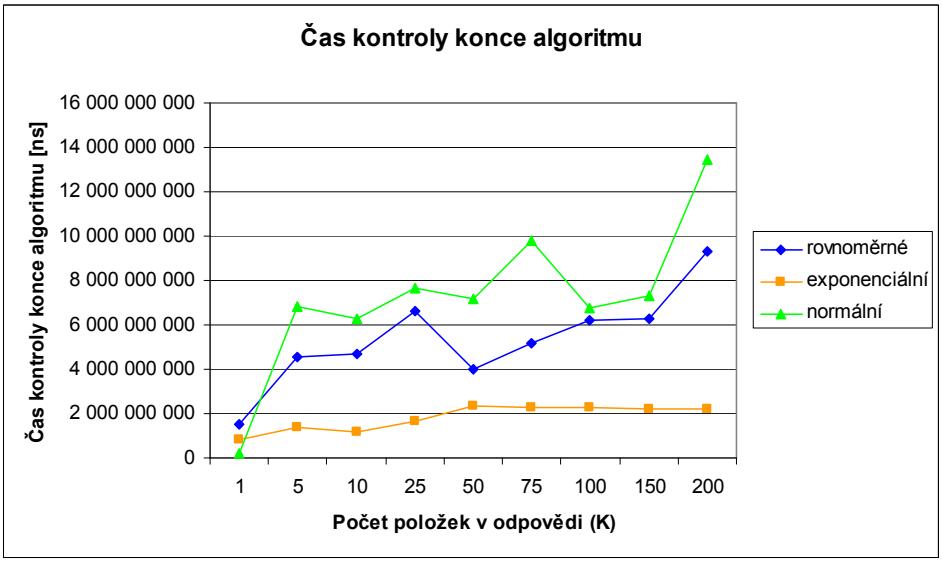
Zajímalo nás, proč je algoritmus na exponenciálních datech o tolik rychlejší, i když počet řádek je přibližně stejný jako u ostatních rozložení. Zkusili jsme sledovat, kdy hodnocení *k*-tého prvku přesáhne hodnocení prahu. To je okamžik, kdy se již nepřidávají nové prvky do množiny kandidátů, naopak už z této množiny pouze odebíráme.



10.4.3. Graf

Zde již je vidět, že při exponenciálním rozložení je prahová hodnota překročena řádově dříve než u ostatních rozložení.

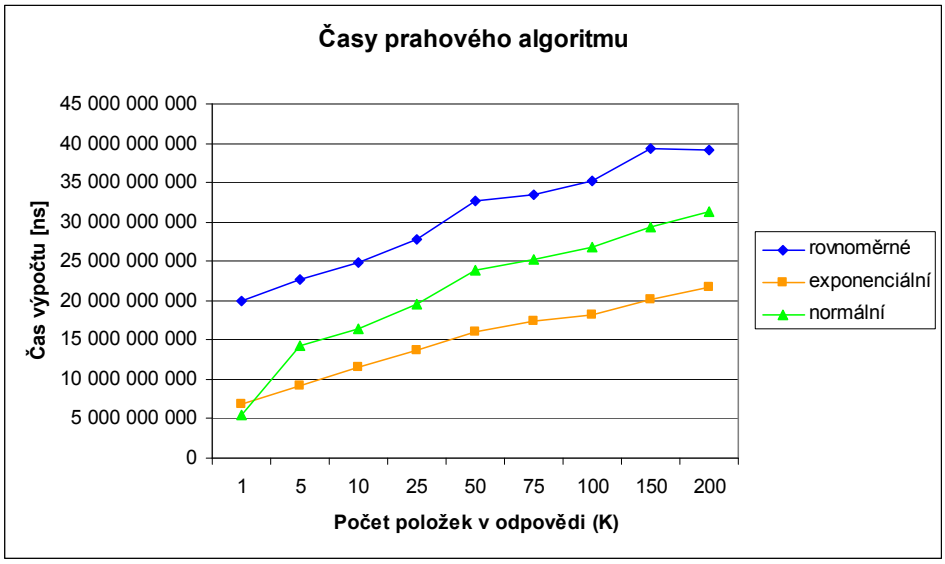
Nakonec ještě potvrdíme domněnku změřením času, který algoritmus potřebuje na zjištění, zda již může skončit. V tomto kroku se prochází celá množina kandidátů, dokud se nenajde prvek s horním ohodnocením větším než je dolní ohodnocení k-tého prvku.



10.4.4. Graf

Prahový algoritmus

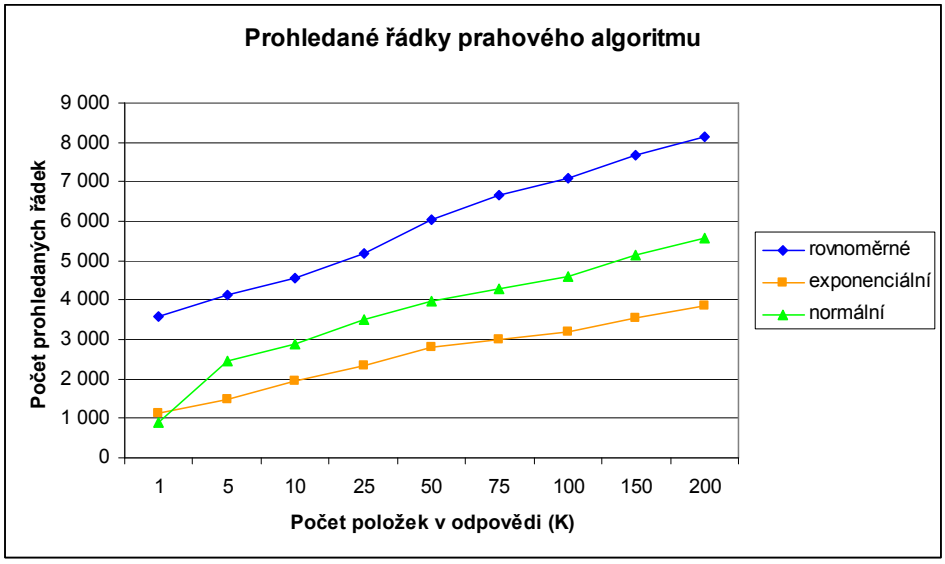
Nyní otestujeme prahový algoritmus. Začneme opět časem algoritmu.



10.4.5. Graf

Zde se zajímavě prohodilo pořadí normálního a rovnoměrného rozložení. Jelikož prahový algoritmus má řádově méně prohledaných řádek, vysvětlujeme to tím, že v případě normálního rozložení se algoritmus ještě nedostal do prostřední části, kde jsou nahuštěna data s podobnými hodnotami.

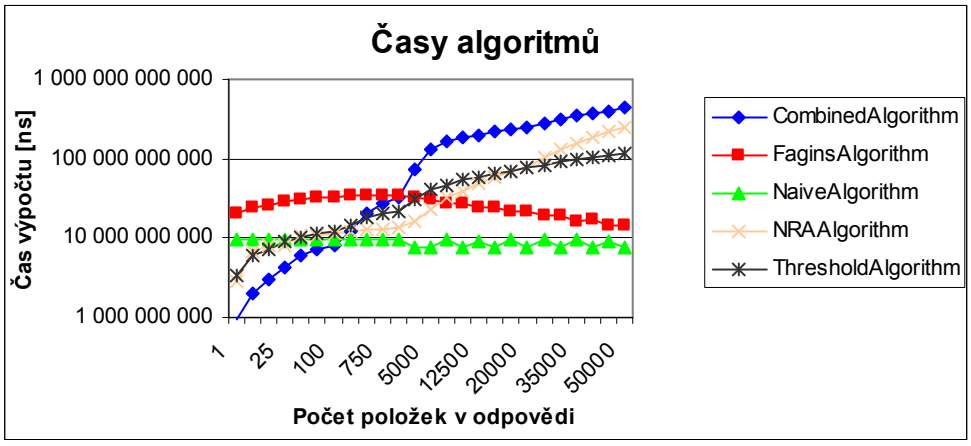
Náš předpoklad potvrzuje následující graf s počtem prohledaných řádek.



10.4.6. Graf

10.5. Testování různých algoritmů

Nyní otestujeme jednotlivé algoritmy v jejich základní podobě. Agregační funkce je vážený průměr. Nebyly použity žádné heuristiky.



10.5.1. Graf

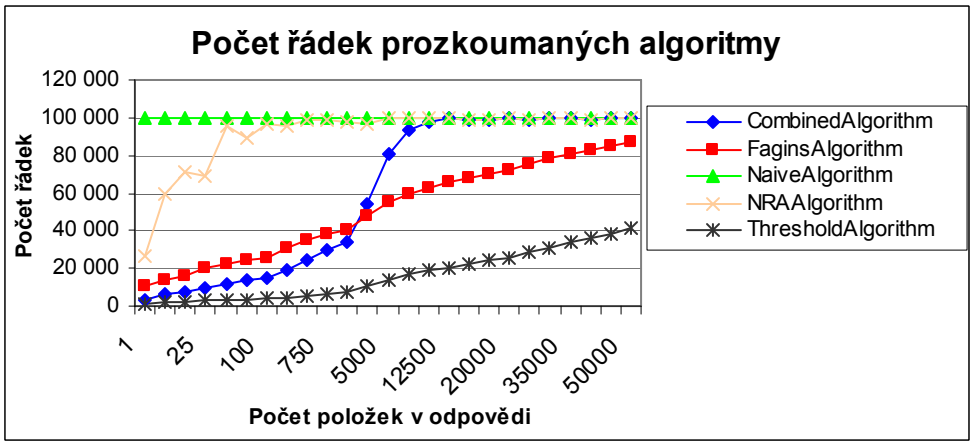
Graf lze rozdělit na dvě části – časy pro malá k a časy pro velká k.

Pro malá k je nejlepší kombinovaný algoritmus, jehož čas výpočtu ovšem stoupá rychleji než u ostatních algoritmů. Druhý nejrychlejší je algoritmus bez přímého přístupu, pak prahový a Faginův algoritmus.

Pro $k \geq 750$ je nejlepší naivní algoritmus, algoritmus bez přímého přístupu se do k=10000 drží na srovnatelné úrovni a pak se také prudce zhoršuje. Zde je nejprve předstižen Faginovým algoritmem a posléze i prahovým algoritmem. Kombinovaný algoritmus zůstává stále nejhorší.

Zajímavé je zlepšování Faginova algoritmu pro větší k. To se děje proto, že se prodlužuje část hledání k prvků se všemi atributy. Pro k=počet zdrojů je potom průběh Faginova algoritmu stejný jako naivního algoritmu.

Naivní algoritmus je zde jako jakási norma, ke které se ostatní algoritmy vztahují, má pro všechna k stejný čas výpočtu.



10.5.2. Graf

Graf potvrzuje prudký nárůst počtu řádek pro $k \geq 1000$ u kombinovaného algoritmu. Dále je vidět, že algoritmus bez přímého přístupu velmi rychle dosáhl skoro plného počtu prohledaných řádek. Pro k=100 to již bylo 96 924 ze 100 000.

10.6. Testování RDF databází

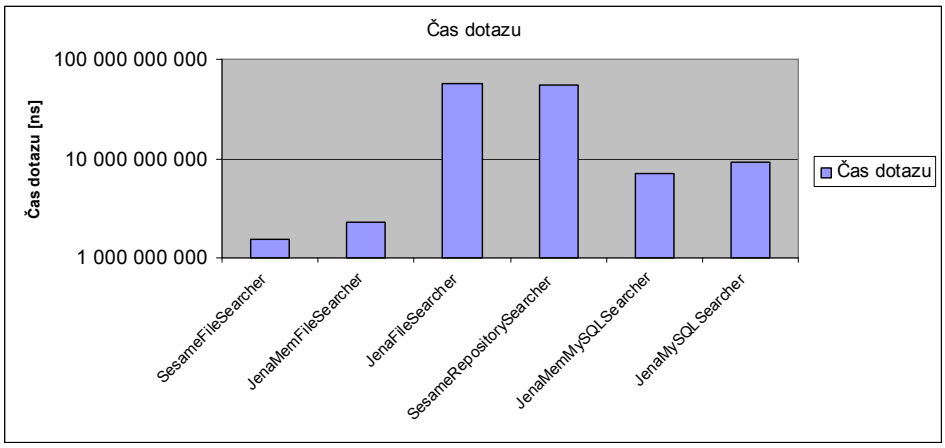
V této části jsme testovali rychlost Jeny a Sesamu ve dvou režimech. První je vytvoření virtuální databáze v paměti, obsah databáze je načten z určeného souboru. Druhý je přístup do relační databáze, v tomto případě MySQL, a získávání dat z ní.

Použijeme opět pět atributů a dvacet tisíc zdrojů, celkem pět set tisíc trojic.

Zde musíme poznamenat, že Sesame pracuje s obecným pojmem "repository" (úložiště), které může uchovávat data buď v relační databázi nebo v paměti. Přístup k úložištím je uskutečněn přes server Apache Tomcat.

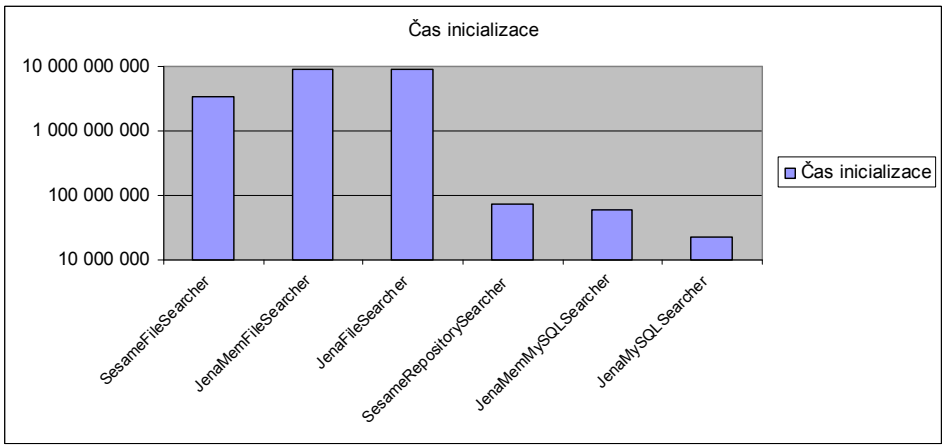
Nahrávání dat do databáze MySQL probíhá u Sesamu přes webovské rozhranní serveru Apache, u Jeny přes program v Javě. Uložení testovací sady s dvaceti tisíci objekty a pěti vlastnostmi, tedy celkem sto tisíci trojicemi, trvalo Sesamu pět minut, Jeně pak padesát jednu minutu.

Pro data v paměti si obě dvě RDF databáze vytvoří v paměti virtuální databázi, do které se potom dotazují. Obsah databáze je naplněn ze souboru. Pro úplnost uvedeme grafy doby vytvoření této virtuální databáze a také časy pro vykonání dotazu do databáze.



10.6.1. Graf

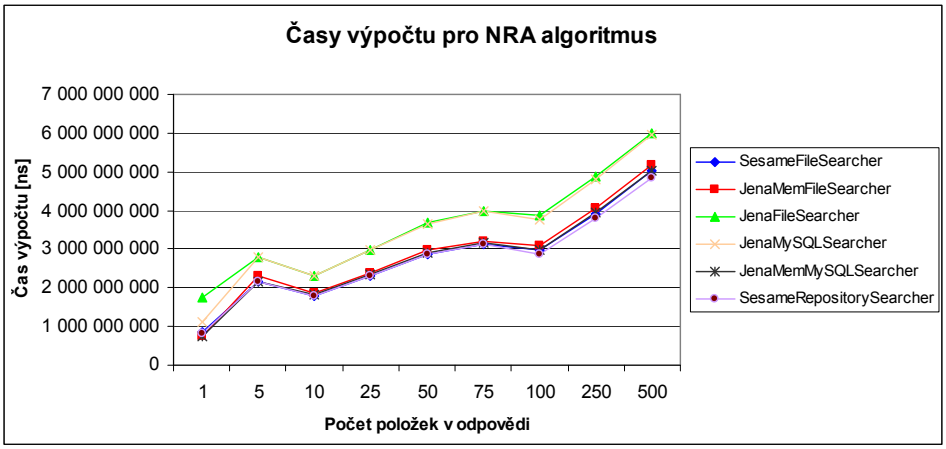
Graf je v logaritmickém měřítku. Zde vidíme, že při práci v paměti je velmi výrazně nejpomalejší v provádění dotazu Jena. Sesame zase ztrácí při přístupu k databázi, to je způsobeno již zmíněným Apachem.



10.6.2. Graf

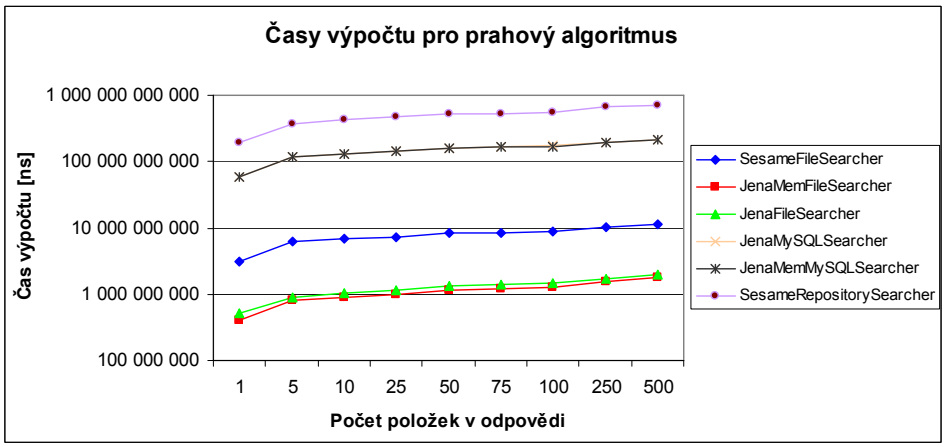
Opět je graf vynesen v logaritmickém měřítku.Vytvoření virtuální databáze ze souboru zabírá řádově více času než přístup do databáze, kde jsou již data připravena.

Na grafu 10.6.3 vidíme, že rychlost NRA algoritmu v podstatě nezáleží na druhu přístupu k datům. Je to dáno tím, že ani Sesame ani Jena nepodporují plnohodnotné třídění dat. Třídění se děje v paměti na již načtených datech. JenaSearcher je oproti JenaMemorySearcher a SesameSearcher o konstantu pomalejší. V průměru zpomalení je 0.7 sekundy.



10.6.3. Graf

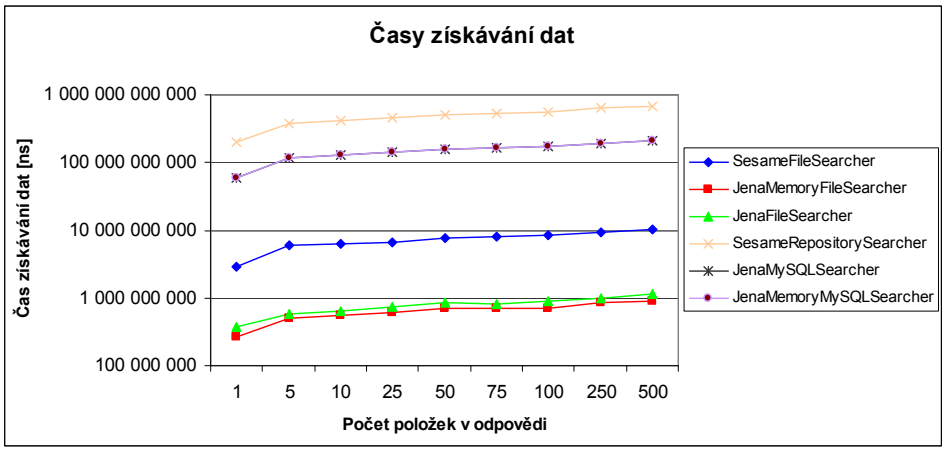
Zajímavější situace bude pro prahový algoritmus, který využívá přímého přístupu. Zde by se mělo objevit větší zpoždění u relační databáze.



10.6.4. Graf

Na grafu 10.6.4 vidíme poměry časů pro Jenu a Sesame. Časy jsou zaneseny v logaritmickém měřítku. Pro Sesame jsou časy pro relační databázi oproti datům v paměti šedesátkrát horší, pro Jenu řádově stokrát.

Ovšem je zde vidět, že Sesame je výrazně horší než Jena. To je způsobeno cenou přímého přístupu do databáze.



10.6.5. Graf

Na grafu 10.6.5 jsou zaneseny časy, které potřeboval prahový algoritmus pro získání dat. Algoritmy pracující s relační databází jsou výrazně horší než ty, pracující s virtuální databází v paměti. V případě relační databáze je Sesame třikrát horší než Jena, v případě virtuální databáze v paměti je dokonce desetkrát horší.

Zajímavé je, že při NRA algoritmu se poměr časů pro získání dat obrací.

Časy pro získání dat				
	NRA		Prahový algoritmus	
	Soubor	MySQL	Soubor	MySQL
Sesame	245 078 729	235 082 134	7 282 376 402	488 601 149 795
Jena	698 005 889	702 142 154	777 296 360	148 771 973 263
JenaMemory	247 523 843	242 769 315	644 342 349	148 615 501 968

10.6.6. Tabulka

Pro algoritmus bez přímého přístupu se úložiště použilo pouze na začátku pro provedení dotazu. V dalším běhu algoritmu jsou již výsledky u všech úložišť uloženy v paměti. Pro SesameSearcher a JenaMemorySearcher jsou uloženy v datové struktuře ArrayList, v případě JenaSearcher to je vlastní úložiště Jeny. Toto vlastní úložiště je řádově třikrát pomalejší než přístup pomocí struktury ArrayList.

Prahový algoritmus využívá úložiště i v průběhu algoritmu pro přímý přístup. V případě virtuálních databází v paměti je Sesame nyní řádově desetkrát horší než Jena. Jena poskytuje přímo funkci v Javě, která pro zadaný podmět a vlastnost vrátí předmět trojice. Sesame podporuje získávání dat pouze přes dotazy. Zpracování dotazu a jeho provedení zřejmě způsobuje daný rozdíl.

V případě databáze MySQL je Sesame třikrát pomalejší, což si vysvětlujeme tím, že k databázi přistupuje přes server Apache, čímž se zpomaluje přístup.

10.7. Testování různých agregačních funkcí

V této části budeme sledovat, jaký vliv má agregační funkce na rychlost algoritmu. Zřejmě nejrychleji proběhne algoritmus s více benevolentní agregační funkcí, např. maximem než s více omezující funkcí, např. minimem. Důvod je ten, že v případě minima bude mít práh dlouho vysokou hodnotu, ale prvky typicky budou mít hodnotu malou – alespoň jeden z atributů má většinou malé hodnocení. Naopak v případě maxima budou mít hned první prvky, které nalezneme, velmi vysoké hodnocení.

Budeme testovat následující agregační funkce:

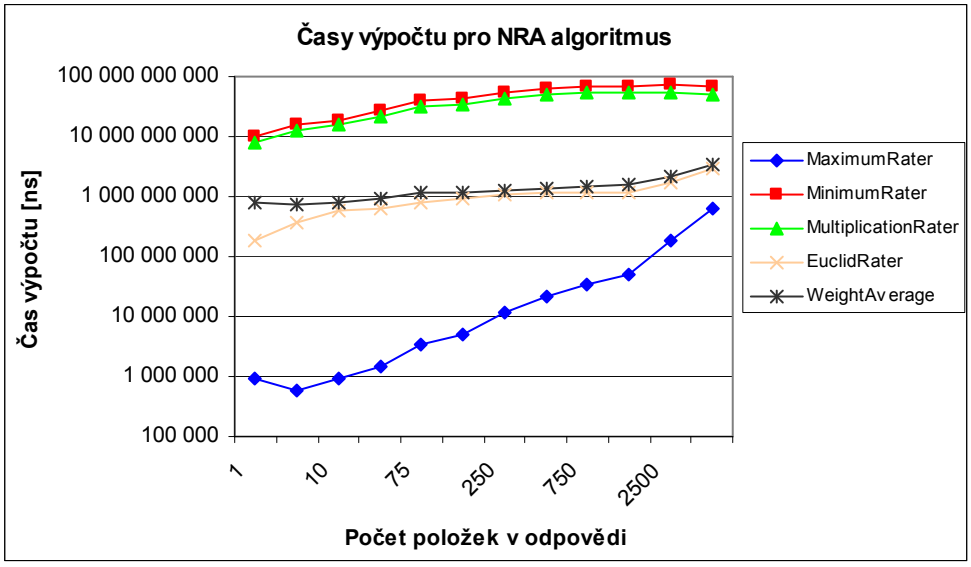
1) WeightAverage – vážený průměr hodnocení atributů. Váha i-tého atributu je 1/(i+1)
2) EuclidRater – Euklidovská vzdálenost bodu, jehož souřadnice jsou hodnocení atributů objektu, od bodu [1,…,1].
3) Minimum – minimum všech hodnocení atributů
4) Maximum – maximum všech hodnocení atributů
5) Multiplication – součin hodnocení všech atributů

Z těchto agregačních funkcí předpokládáme, že nejpomalejší bude minimum a součin hodnocení. Obě dvě funkce při chybějícím hodnocení atributu dávají jako výsledek nulu. Z důvodu pomalosti těchto dvou funkcí jsme použili testovací množinu o deseti tisíci zdrojích a pěti atributech.

Pro testování jsme použili algoritmus bez přímého přístupu, kombinovaný algoritmus a prahový algoritmus. Ani Faginův ani naivní algoritmus nezávisí na použité agregační funkci, jelikož nevyužívají hodnotu prahu.

Nejprve se budeme zabývat algoritmem bez přímého přístupu.

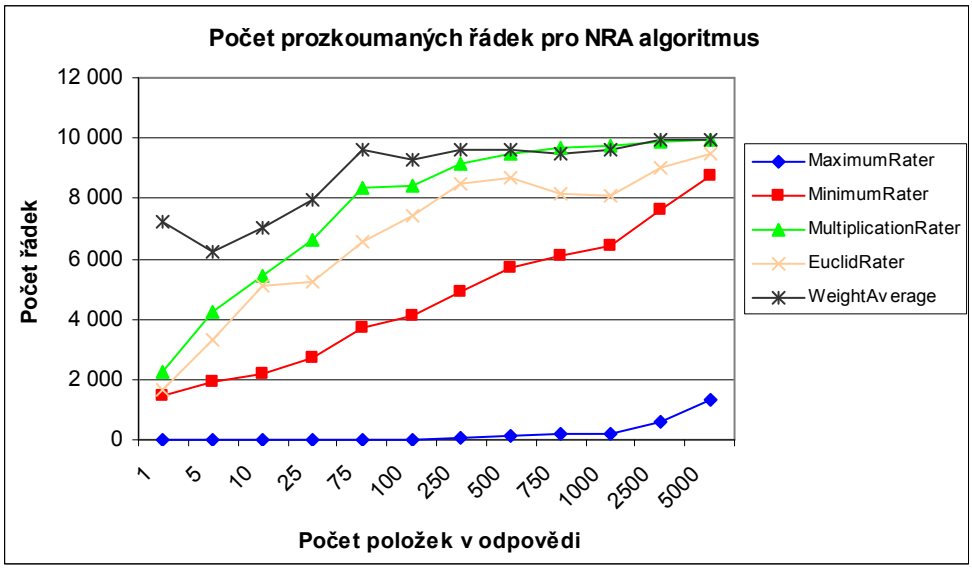
Algoritmus bez přímého přístupu



10.7.1. Graf

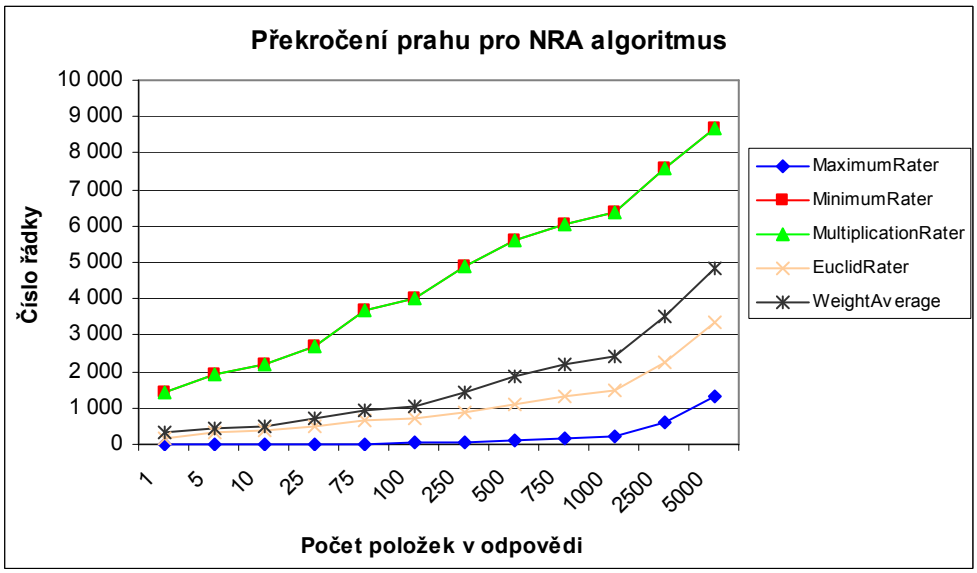
Zde se potvrdil předpoklad, že minimum a součin budou nejpomalejší. Maximum je nejrychlejší. Graf je v logaritmickém měřítku.

Dále nás bude zajímat počet prozkoumaných řádek.

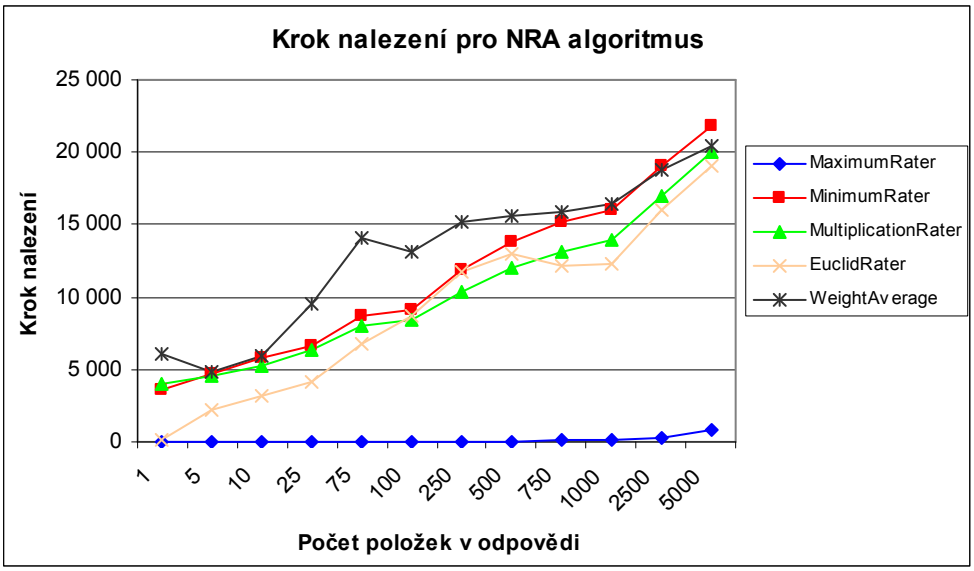


10.7.2. Graf

Na grafu 10.7.2 vidíme, že nejvíce řádek potřeboval vážený průměr. Abychom zjistili, proč jsou minimum a součin nejpomalejší přestože nemají nejvíce prozkoumaných řádek, musíme sledovat okamžik překročení prahu a nalezení odpovědi.

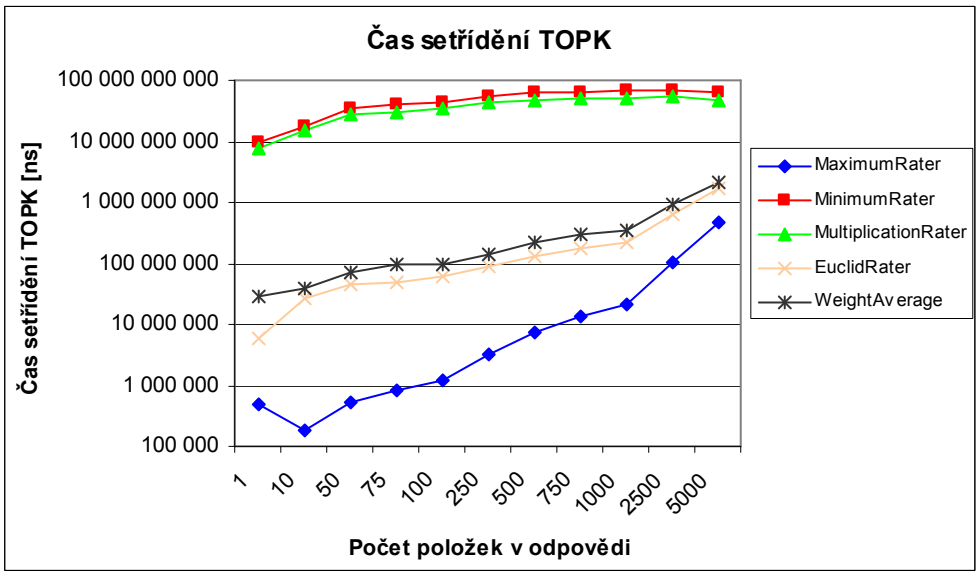


10.7.3. Graf



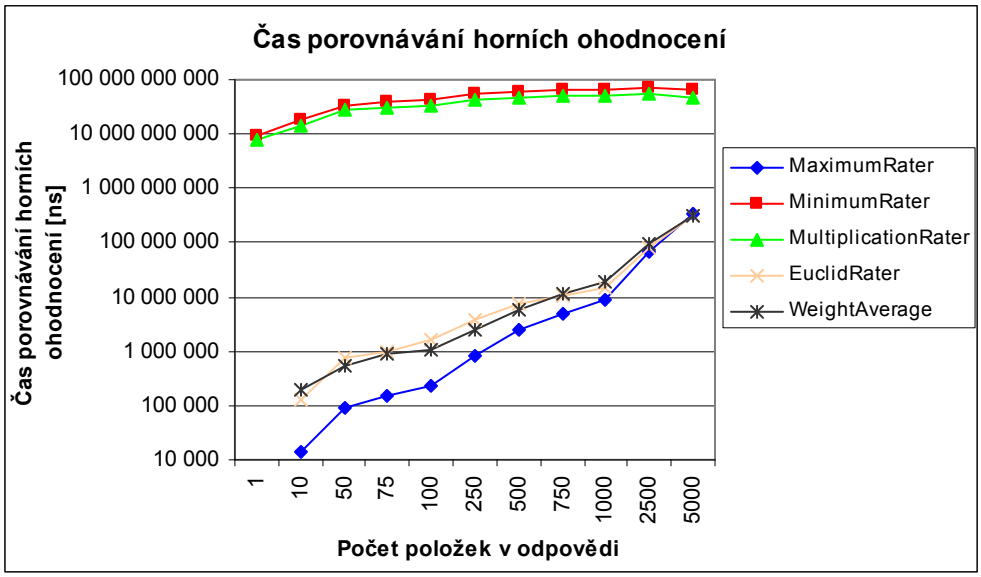
10.7.4. Graf

Jelikož i v kroku nalezení se minimum a součin nejsou horší než vážený průměr, vysvětlení leží v hodnotě překročení prahu. Než je překročena hodnota prahu, musíme přidávat nově nalezené prvky, čímž se zpomaluje setřiďování výsledků.



10.7.5. Graf

Zde již je patrná velká ztráta minima a součinu oproti ostatním agregačním funkcím. Navíc při minimu se bude mnoho hodnot objektů shodovat – budou mít dlouho hodnocení nula. Proto ještě změříme, jak dlouho trvá porovnávání horních ohodnocení prvků.

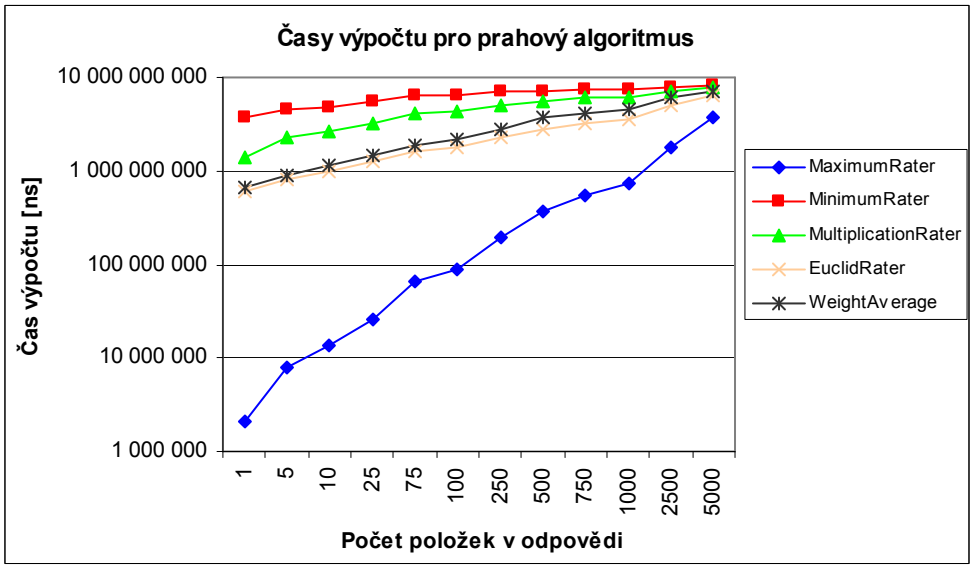


10.7.6. Graf

V tomto grafu je již zřejmé, že problém minima a součinu leží při porovnávání horních ohodnocení prvků se stejným dolním ohodnocením. Tato operace je poměrně drahá, jelikož se musí znovu počítat znovu celé hodnocení. Ostatní agregační funkce v podstatě čas na tuto činnost nepotřebují, nebo je o několik řádů menší.

Prahový algoritmus

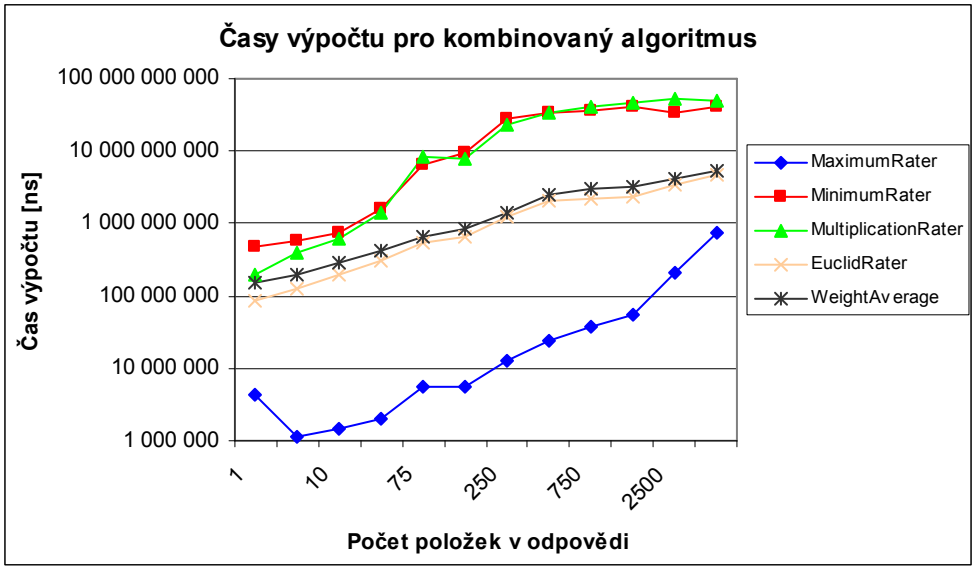
Nyní otestujeme agregační funkce na prahovém algoritmu.



10.7.7. Graf

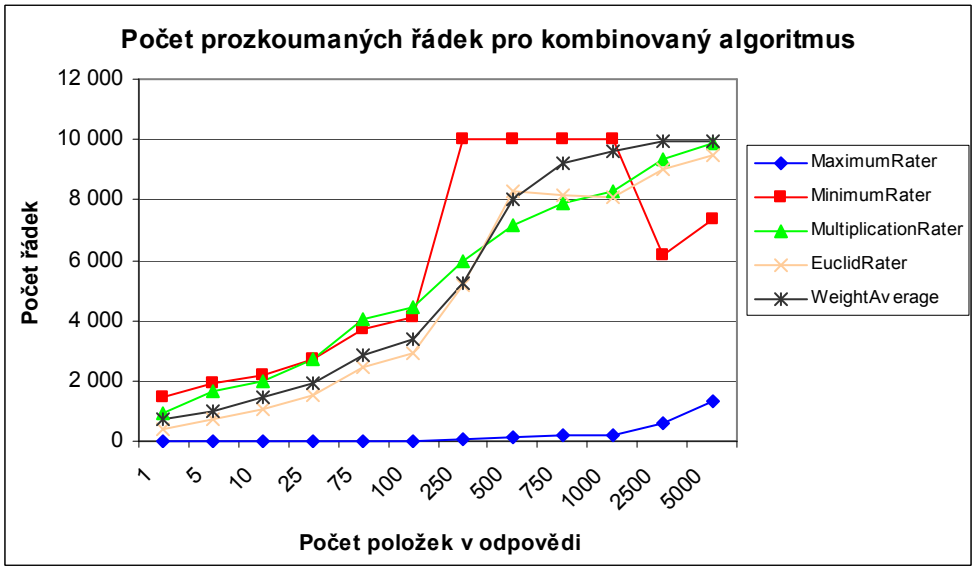
Vidíme, že rozdíl v časech již není tak markantní. Zde velmi vybočuje maximum, které je typicky nejrychlejší. Malý rozdíl mezi zbylými agregačními funkcemi je dán tím, že prahový algoritmus zjišťuje hned všechny atributy objektu, není zde tedy rozdíl mezi dolním a horním ohodnocením.

Kombinovaný algoritmus



10.7.8. Graf

U kombinovaného algoritmu je menší rozdíl mezi minimem a součinem a ostatními agregačními funkcemi. Pro malé *k* jsou časy velmi podobné, se zvyšujícím *k* se zvětšuje i rozdíl časů.



10.7.9. Graf

V počtu prozkoumaných řádek se pro malé *k* skoro shodují všechny agregační funkce krom maxima. Pro velká *k* již vidíme některé singulární případy u minima (*k*=2500 a *k*=5000).

Opět největší díl času zabírá pro součin a minimum porovnávání záznamů se stejným dolním ohodnocením. Pro ostatní agregační funkce je to čas přímého přístupu.

10.8. Testování různých heuristik

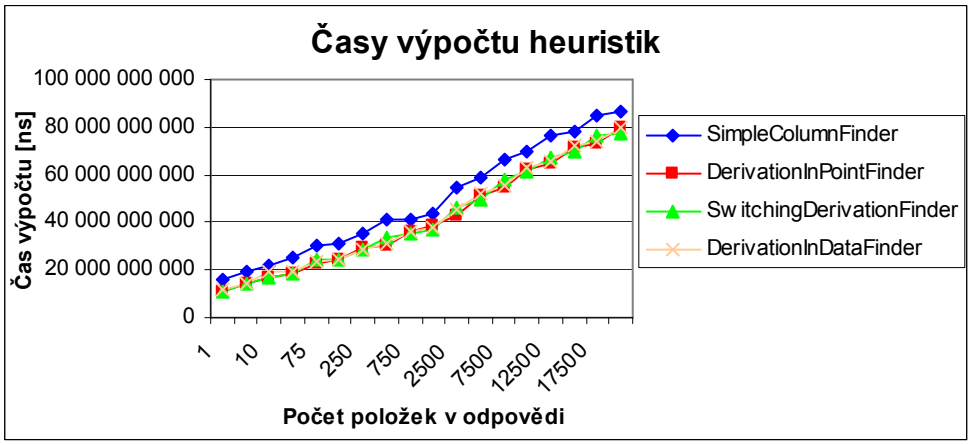
V této části otestujeme heuristiky popsané v [GLV]. Jde o heuristiky pro prahový algoritmus, které snižují počet prozkoumaných řádek.

SimpleColumnFinder vybírá postupně všechny sloupce tak, aby počet prozkoumaných řádek byl v každém sloupci stejný.

DerivationInPointFinder určuje následující sloupec podle parciální derivace agregační funkce. Derivace se počítá v bodě definovaném prahem v daném sloupci.

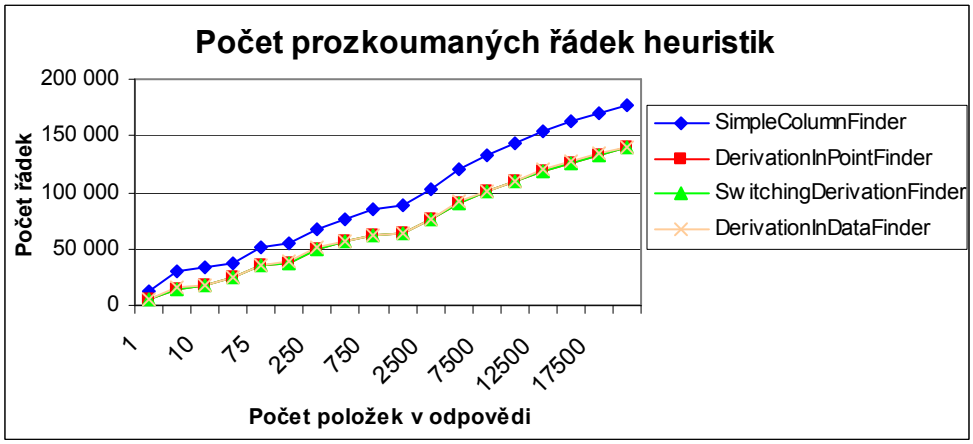
DerivationInDataFinder hledá sloupec s největším derivací v datech. Derivace se počítá jako rozdíl hodnocení aktuálního prvku v seznamu a hodnocení prvku, který byl viděn před *h* kroky. Tento rozdíl je poté vynásoben derivací agregační funkce jako v DerivationInPointFinder.

Nakonec SwitchingDerivationFinder střídá oba výše zmíněné přístupy.



10.8.1. Graf

Z grafu je patrné, že všechny heuristiky jsou lepší než paralelní přístup, řádově o patnáct procent. Mezi sebou se ovšem výrazně neliší.



10.8.2. Graf

Ani v počtu řádek se heuristiky neliší.

Oproti [GLV] jsme tedy nezjistili výraznější rozdíl mezi vlivem heuristik. To je zřejmě způsobeno použitím vygenerovaných testovacích dat. V [GLV] byla použita množina reálných dat, kde se v každém z parametrů mohlo lišit i např. rozložení hodnot.

V dalším studiu by mohlo být přínosné porovnat data nebo implementace algoritmů. Výsledky jednotlivých heuristik se v [GLV] liší velmi výrazně.

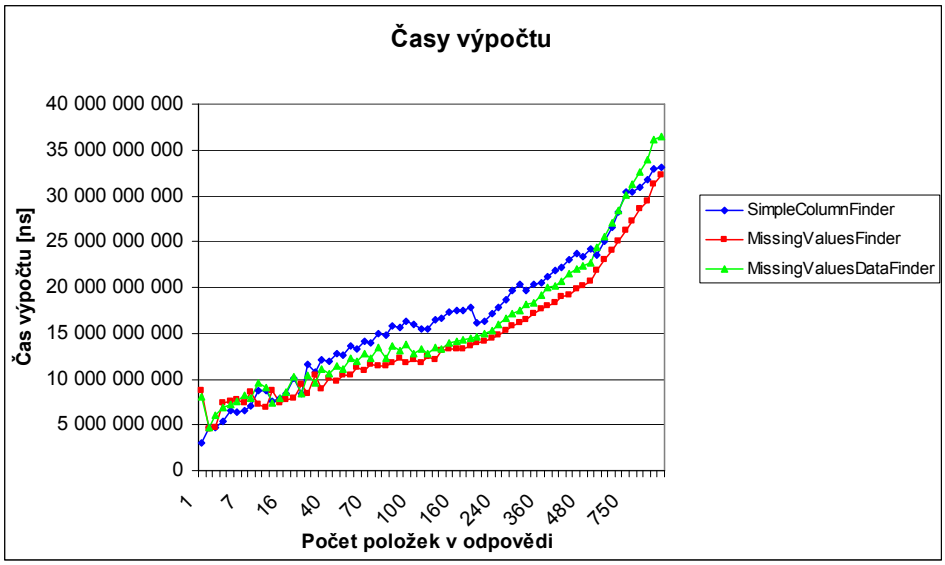
10.9. Testování heuristiky MissingValuesFinder

Nyní zkusíme pro NRA algoritmus použít heuristiku MissingValuesFinder. Tu jsme implementovali ve dvou verzích, které se liší svým chováním v případě, že v TOPK nejsou žádné chybějící hodnoty. Obě verze využívají jinou heuristiku, MissingValuesFinder používá DerivationInPointFinder a MissingValuesDataFinder používá DerivationInPointFinder.

Použijeme opět pět atributů a sto tisíc zdrojů, celkem pět set tisíc trojic.

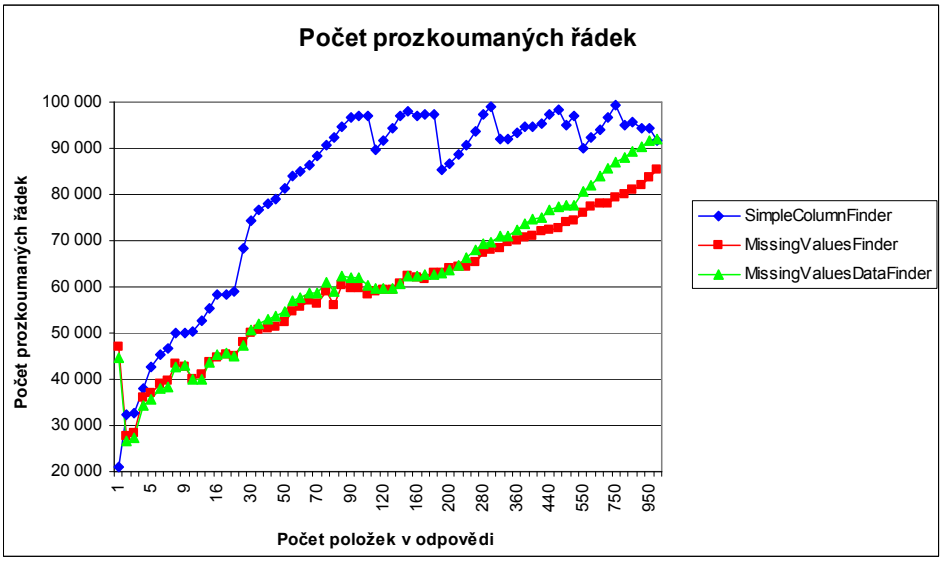
Testování heuristiky ukázalo, že není vhodné procházet celý seznam TOPK, ale pouze posledních *h*. Po testování různých hodnot *h* jsme ji stanovili na 10. Dále se ukázalo jako vhodnější uvažovat index posledního prvku s prázdnou hodnotou místo sčítání indexů všech prvků s prázdnou hodnotou nebo uvažování počtu prvků s prázdnou hodnotou.

Ze všech rozložení hodnot se heuristika nejlépe projevila na rovnoměrném rozložení.



10.9.1. Graf

Zajímavější je statistika počtu prozkoumaných řádek.



10.9.2. Graf

Zde je vidět, že heuristiky MissingValues mají mnohem pozvolnější nárůst počtu řádek.

Na závěr testování heuristiky MissingValuesFinder musíme podotknout, že míra zrychlení algoritmu klesala s tím, jak se zrychloval samotný algoritmus. Tento postup je rozebrán v kapitole 9.8.1.

10.10. Problémy testování

Testování top-k algoritmu má několik úskalí, která jsme již dříve naznačili.

Největší úskalí je absence třídění. Sesame třídění nepodporuje vůbec, Jena sice ano, nicméně je implementováno jako třídění výsledků v paměti. Obě implementace nejsou v souladu s myšlenkou top-k algoritmu - setříděnými seznamy, ke kterým lze přistupovat sekvenčně.

Proto je lepší interpretovat výsledky spíše vzhledem k počtu prozkoumaných řádek. To je údaj, který se nemění s použitou technologií uložení dat. Naopak čas výpočtu je na použité technologii silně závislý, jak bylo potvrzeno v testu 10.6 Testování RDF databází. Nakonec je potřeba zmínit potřebu správně analyzovat data a vybrat vhodné heuristiky před vlastním nasazením algoritmu. S tím je spojen problém agregačních funkcí, které se mohou měnit v čase. Spolu se změnou agregační funkce může být spojena i změna ideálního algoritmu.

10.11. Závěr kapitoly

Porovnali jsme jednotlivé implementace top-k algoritmů i jejich heuristik na různých rozložení hodnot v datech. Zjistili jsme, že pro velká k je nejrychlejší naivní algoritmus, pro malá k potom lze s úspěchem použít kombinovaný algoritmus. Doba výpočtu algoritmů radikálně závisí na použité agregační funkci. Nalezli jsme také výkonnostní rozdíly mezi RDF databázemi Sesame a Jena. V dalším výzkumu by bylo vhodné zjistit rozdíly v datech a implementacích algoritmů a heuristik proti [GLV].

11. Aplikace pro výběr výletních lokalit Potlatch

11.1. Motivace

Vytvoření aplikace pro výběr výletních lokalit Potlach bylo motivováno potřebou turistického oddílu evidovat své výlety. Evidence výletů měla sloužit pro podporu rozhodování během výběru lokality příštího výletu. Bylo potřeba uchovávat informace o tom, jak se kterému členovi oddílu na výletě líbilo, stejně tak i obecné informace o výletu – jaké bylo počasí a kdo se ho účastnil.

V této aplikaci je využita teorie z kapitoly 7.4 o více uživatelských hodnoceních. Jsou zde implementovány oba přístupy pro počítání optimálního místa pro výlet. Dále je použita metoda pro generování uživatelských hodnocení atributů na základě ohodnocení výletu.

Aplikace je postavena nad databází Sesame, která umožňuje snadné připojení více uživatelů přes internet pomocí serveru Apache.

Scénář použití se týká dvou situací – zadávání hodnocení a použití aplikace pro hledání výletu.

Zadávání hodnocení bude probíhat hned po návratu z výletu. Administrátor (vedoucí) oddílu založí výlet v aplikaci a přiřadí mu správné atributy. Ostatní členové potom tento výlet ohodnotí. Všichni pracují z domova, což zajišťuje lepší věrohodnost hodnocení, než kdyby hodnocení probíhalo veřejně na schůzce.

Hledání optimálního místa výletu by mělo probíhat tak, že se vytvoří virtuální výlet s očekávanými parametry. Těmito parametry jsou počasí, účastníci výletu apod. V ideálním případě by si systém některé hodnoty mohl zjistit sám. Na základě zadaných parametrů systém potom dohledá ideální výletní lokalitu. Tento způsob ovšem vyžaduje jiný přístup než použití top-k algoritmu. Jde vlastně o úlohu dolování znalostí z dat. Proto tento způsob opomíjíme a ponecháváme dalšímu zkoumání.

Budeme zde implementovat přímý způsob, kdy se lokalita hledá na základě uživatelského hodnocení atributů místa výletu, tedy situaci, kdy se využívá algoritmus top-k. Zde je použita teorie z kapitoly 7.4, tedy hledání optimálního objektu vzhledem k preferencím více uživatelů.

11.2. Ontologie

Tato aplikace využívá vytvořenou ontologii výletních míst. Zčásti jsme si tuto ontologii přiblížili v předchozím textu, celá je v příloze 1 a na přiloženém CD.

Zde musíme zmínit jazykovou problematiku. Z důvodu mezinárodnosti jsme psali ontologii v anglickém jazyce. Ovšem po ohlasech z řad turistů, kteří nejsou všichni anglicky mluvící, jsme se rozhodli ontologii přepsat do češtiny.

Ontologie obsahuje všechny potřebné třídy a vlastnosti, na některé jsme v průběhu práce narazili. Dále obsahuje několik omezení na třídy a několik definovaných tříd, jako např. zmíněné slunečné výlety.

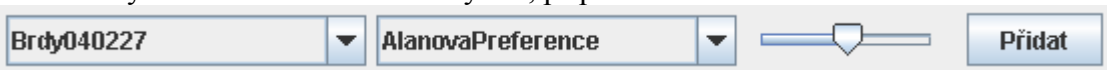
Ontologie byla vytvořena v programu Protégé, ve kterém byla i naplněna počáteční sadou individuí.

11.3. Uživatelské rozhraní

Uživatelské rozhraní je navrženo funkčně, bez zbytečných okras. Po startu programu uživatel zadá své jméno a heslo.

11.3.1.1. Obrazovka

Poté vykoná vlastní hodnocení výletu, případně dalších atributů.



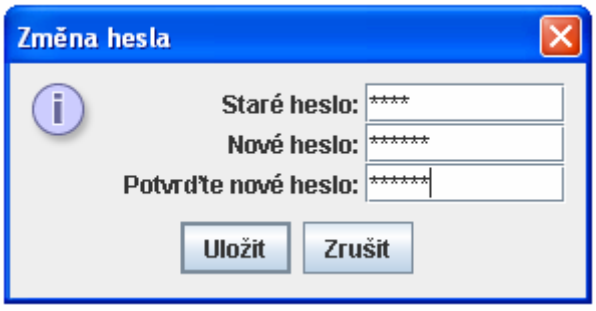
11.3.1.2. Obrazovka

Hodnocení se zadává v dolní liště, kde si nejdříve uživatel vybere výlet, který chce hodnotit a poté posuvníkem vpravo vybere hodnocení. Čím více vlevo, tím je hodnocení horší, čím více vpravo, tím lepší. Toto řešení je pro uživatele příjemnější než zadávat hodnocení číselně. Navíc v tabulce nad spodní lištou vidí již dříve hodnocené výlety, takže hodnocení nového výletu k nim může vztáhnout.

Výlet	Název hodnocení	Hodnocení
Rohace010805	AlanovaPreference	
SlaneniZdakovskehoMostu051104	AlanovaPreference	
Brdy040227	AlanovaPreference	
Polsko030810	AlanovaPreference	
Hostka020631	AlanovaPreference	
Brdy050501	AlanovaPreference	
CeskeStredohori040507	AlanovaPreference	
Chyska050714	AlanovaPreference	
Hostka030701	AlanovaPreference	
MokropeskaPekarna050410	AlanovaPreference	

11.3.1.3. Obrazovka

Také si zde může změnit své heslo. Hesla jsou zakódována pomocí algoritmu MD5 a uložena přímo v RDF databázi.

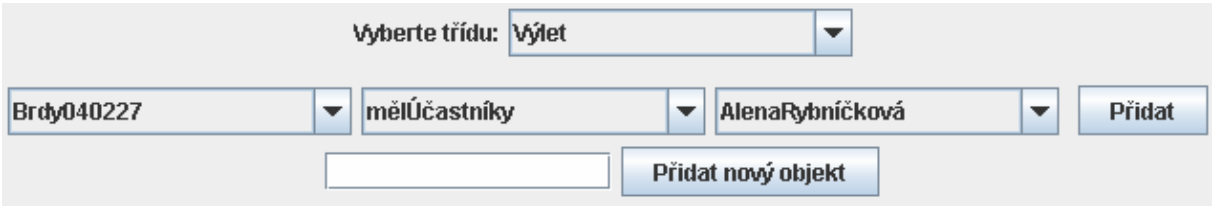


11.3.1.4. Obrazovka

Administrátorský režim umožňuje kromě přidávání vlastních hodnocení také vyváření nových individuí tříd a modifikovat existující. Navíc má administrátor možnost zjistit ideální výletní místo pomocí dvou výše zmíněných způsobů.

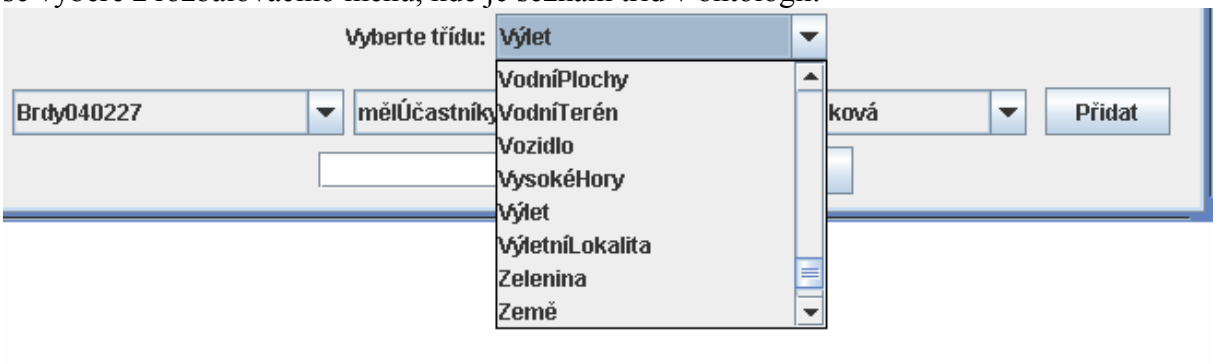
11.3.2. Administrátorská záložka

Zajímavou funkcí je tzv. administrátorská záložka. Umožňuje v administrátorském režimu přidávat nové entity a modifikovat existující. Zde se využívá vytvořená ontologie výletních míst.



11.3.2.1. Obrazovka

První krok k vytvoření nové entity je vybrat, do jaké třídy bude náležet. Zvolená třída se vybere z rozbalovacího menu, kde je seznam tříd v ontologii.

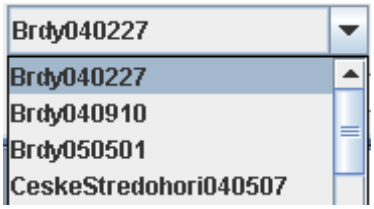


11.3.2.2. Obrazovka

Pokud chceme přidat nový objekt, vyplníme jeho název a stiskneme tlačítko "Přidej nový objekt".

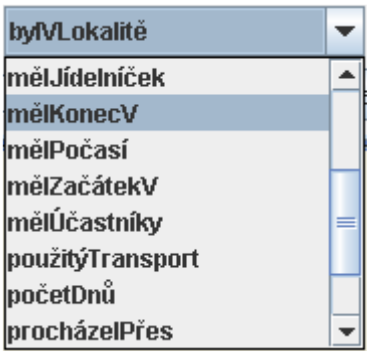
11.3.2.3. Obrazovka

Pokud chceme upravit stávající objekty, uděláme to pomocí horních rozbalovacích menu.



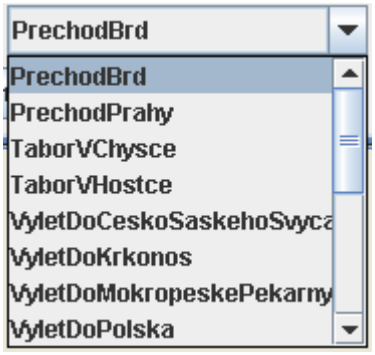
11.3.2.4. Obrazovka

V nich nejdříve vybereme individuum ze třídy.



11.3.2.5. Obrazovka

Dále vlastnost, kterou má mít.



11.3.2.6. Obrazovka

Nakonec zvolíme předmět vlastnosti. Nakonec trojici přidáme tlačítkem "Přidej".

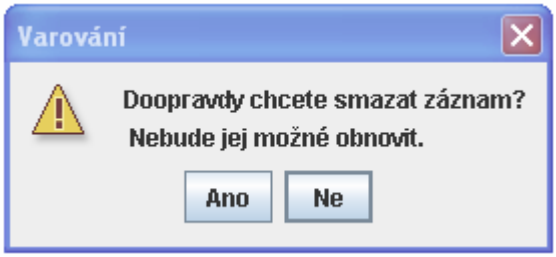
Jelikož se typ předmětu může měnit, mění se i komponenta, pomocí které se zadává.

Buď jde o vlastnost, jejíž předmět je nějaký zdroj a pak se možné zdroje zobrazí v rozbalovacím menu.

Pokud je to řetězec, vstup je klasické zadávací pole pro text.

Čas a datum má speciální zadávací pole, které je formátováno jako datum.

Pokud chceme nějakou trojici smazat, vybereme ji ze seznamu nahoře a zmáčkneme klávesu Delete. Po potvrzení uživatelské hlášky bude trojice smazána z RDF databáze.



11.3.2.7. Obrazovka

Jediný problém v tomto typu aplikací je nedostatečná kontrola úložiště. Chtěli bychom, aby se při přidávání trojice automaticky kontrolovala omezení kladená na třídy a

vlastnosti. Např. by bylo vhodné, aby uživatel nemohl hodnotit vícekrát stejný objekt. Vyzkoušeli jsme úložiště [KIM], které by mělo provádět odvozování pro OWL, nicméně se ukázalo jako nefunkční - potřebujeme přímou zpětnou vazbu po přidání trojice. Kontrolovat konzistenci celého úložiště po každé operaci je nemožné.

Problém kontroly OWL je kritický hlavně pro omezení vlastností, které mají být funkční. Bez této kontroly může uživatel zadat více hodnocení jednomu zdroji nebo přiřadit uživateli více křestních jmen apod.

11.4. Implementace

Uživatelské rozhraní se skládá ze záložek, kde jdou přidávat preference. Tyto záložky jsou definované v souboru settings.xml, kde navíc jsou názvy klíčových tříd a vlastností pro top-k algoritmus a manipulaci s uživatelskými preferencemi. Ukážeme si příklad pro ontologii výletních míst

```
 <Path>mem-rdfs-db</Path>
<Namespace>http://www.muaddib.wz.cz/</Namespace>
<UserClassName>Člověk</UserClassName>
<SuperRatingPredicate>preference</SuperRatingPredicate>
<SuperComputedRatingPredicate>vypočtenéPreference</SuperComputedRatingPredicate>
<UserRatingPredicate>máPreferenci</UserRatingPredicate>
<UserComputedRatingPredicate>máVypočtenouPreferenci</UserComputedRatingPredicate>
```

Definuje se zde název úložiště Sesame, typický jmenný prostor ontologie, třída, která obsahuje uživatele aplikace. Dále je to vlastnost, která je nadvlastností všech uživatelských hodnocení, nadvlastnost všech vypočtených hodnocení, vlastnost, která spojuje uživatele s jeho hodnocením a nakonec vlastnost, která spojuje uživatele s jeho vypočteným hodnocením.

V definici jedné záložky potom figurují tyto vlastnosti.

```
<Page name="Výlety" number="1">
    <Namespace>http://www.muaddib.wz.cz/</Namespace>
    <TypeOfPage>EditRDFSesamePanel</TypeOfPage>
    <TypeOfEntity>http://www.muaddib.wz.cz/Výlet</TypeOfEntity>
    <InducedType>http://www.muaddib.wz.cz/VýletníLokalita</InducedType>
    <InducedClasses>
            <InducedClass>http://www.muaddib.wz.cz/Výlet</InducedClass>
            <InducedClass>http://www.muaddib.wz.cz/Město</InducedClass>
            <InducedClass>http://www.muaddib.wz.cz/Terén</InducedClass>
    </InducedClasses>
    <InducedPredicates>
       <InducedPredicate>http://www.muaddib.wz.cz/bylaCílemVýletu</InducedPredicate>
       <InducedPredicate>http://www.muaddib.wz.cz/máMěsta</InducedPredicate>
       <InducedPredicate>http://www.muaddib.wz.cz/máTerén</InducedPredicate>
    </InducedPredicates>
    <ColumnNames>
            <ColumnName>Výlet</ColumnName>
            <ColumnName>Název hodnocení</ColumnName>
            <ColumnName>Hodnocení</ColumnName>
    </ColumnNames>
</Page>
```

Tato struktura definuje všechny potřebné údaje pro zobrazení záložky - název záložky, pořadí, jmenný prostor, typ záložky. Typ může být SimpleEditRDFSesamePanel, EditRDFSesamePanel nebo VaryingEditRDFSesamePanel. První povoluje pouze přidávání hodnocení, druhý typ dovoluje i přidávání nových entit a poslední je Administrátorská záložka, která je popsaná výše.

Poslední položka určuje názvy sloupců v tabulce.

Dále je zaznamenána třída, ze které se budou zobrazovat hodnocené entity. Další položka je třída, ke které chceme vypočítávat hodnocení. Tento mechanismus využívá třídy a vlastnosti definované ve značkách InducedClasses a InducedPredicates.

Postupuje se takto - změníme hodnocení Výletu X. Nalezneme lokalitu Y, která je s výletem spojená a za pomocí definovaných vlastností a tříd vypočítáme vypočtené hodnocení lokality Y. Toto hodnocení zapíšeme do databáze.

Aplikace umožňuje definovat pouze jednu třídu, ke které se bude definovat vypočtené hodnocení.

V ideálním případě by součástí aplikace byla i informace o každé třídě a o vlastnostech a třídách, které jsou použité při počítání vypočtené preference. U každé třídy bychom tedy hned věděli, jaké třídy použít pro vypočtení preference.

Tyto informace by bylo vhodné uložit do jiného souboru, ale to je již mimo implementační rámec této práce.

11.5. Závěr kapitoly

Aplikace Potlatch byla vytvořena jako jednoduché uživatelské prostředí pro zadávání preferencí s některými vylepšeními, zvláště co se týče příjemnosti zadávání hodnocení pro uživatele. Po zadání dostatečného množství hodnocení může být použita jako prostředek pro skupinové rozhodování o místě výletu, použitém dopravním prostředku, jídelníčku na výlet apod.

Díky rozmachu připojení k internetu není zadávání hodnocení velký problém, navíc pomocí teorie z kapitoly 7.2 se nám podařilo omezit počet zadávaných hodnocení na minimum.

12. Ukázková aplikace obecného nástroje pro Cocoon

Na závěr si ukážeme snadnost, jak lze nástroj Xoda použít v praxi. Navrhneme internetovou aplikaci pro hledání výletních lokalit. Hledat se bude podle stupně nebezpečnosti, pohodlí pro spaní a rozdělávání ohně a nakonec podle stupně ochrany lokality.

Aplikace pracuje na platformě Cocoon, která zabezpečuje předávání parametrů zadaných v prohlížeči. Dále potom umožňuje řetězit za sebe jednotlivé komponenty. První část řetězce má za úkol vytvořit data, která bude výsledná stránka obsahovat. To bude úkol SesameGeneratoru. Integraci SesameGeneratoru do Cocoonu popisuje příloha 2. Další komponenty poté upravují výsledek. Vývoj těchto modulů je ovšem již práce spíše pro návrháře vzhledu webových stránek.

12.1. Webovské rozhraní

Navrhnuté rozhraní je vyřešeno funkčně. Dají se zadat váhy jednotlivých atributů, normalizátory a případná ideální hodnota atributu. Hodnoty lze setřídit buď sestupně, vzestupně nebo zadat ideální hodnotu atributu. Poslední možnost má smysl např. při hledání výrobků s danou cenou, pozemků s určitou rozlohou apod. V těchto případech jsou příliš velké hodnoty špatné, ale moc malé hodnoty jsou také špatné.

Toto rozhraní by šlo jistě v mnoha aspektech vylepšit po stránce designu, to ovšem není cílem této práce.

Po odeslání požadavku se zadané parametry předají do třídy SesameGenerator. Ta je zpracuje a spustí top-k algoritmus, který seřadí byty podle zadaných kritérií.

Pro vlastní nastavení a spuštění top-k algoritmu je v kódu potřeba cca. dvacet řádek.

12.2. Výstup SesameGeneratoru

Na platformě Cocoon komponenty mezi sebou komunikují pomocí XML souborů. Proto i výstup SesameGeneratoru je XML soubor, který obsahuje entity nalezené top-k algoritmem.

Ideálně by tento XML soubor zpracoval další článek řetězce procesů, tzv. transformátor. Ten by mohl výsledky prezentovat např. v tabulce nebo jiné struktuře pro uživatele příjemnější než je XML strom.

13. Závěr

V naší práci jsme po prostudování vícehodnotové logiky a vícehodnotových logických programů navrhli dva způsoby zápisu těchto logických programů do RDF. Jeden způsob je ryze syntaktický, kdy doslovně přepisuje pravidla i fakty, druhý lze použít na stávající RDF data, kde se existující trojice interpretují jako fakty. Toto zavedení vícehodnotové logiky do RDF zatím nebylo nikde specifikováno. RuleML je zápis dvouhodnotových pravidel do XML a náš zápis se dá na RuleML převést při vypuštění vícehodnotovosti. Tímto jsme splnili jeden z hlavních cílů této práce.

Uživatelské preference jsou komplexní problematika, které jsme věnovali více místa. Přiblížili jsme si uživatelské fuzzy funkce a dva druhy přístupů při jejich interpretaci. Probrali jsme tři modely, které umožňují získat uživatelská hodnocení objektů, u kterých hodnocení chybí. Proces využívá hodnocení ostatních individuí, která jsou spojená s dosud nehodnocenými objekty. Dále jsme navrhli způsob počítání váhy určitého atributu v agregaci při hodnocení komplexních objektů. Tento způsob je relevantní, pokud se omezíme na agregaci váženým průměrem, v ostatních případech může dávat pouze přibližné výsledky.

Vyšli jsme z hodnocení jednoho uživatele, rozšířili jsme problematiku na preference více uživatelů, zvláště pak na hledání skupinového kompromisu. Uvedli jsme dva přístupy počítání kompromisu, které se vhodně dají převést na teorii K. J. Arrowa. Arrowův model jsme obohatili o číselné hodnocení kandidátů a také o jejich atributy. Použili jsme příklad z volebního prostředí, který vhodně ilustruje oba dva různé přístupy volby. Uživatelské preference a speciálně skupinové rozhodování bylo další klíčovým tématem.

Pro generování hodnocení komplexního objektu na základě hodnocení jeho atributů a agregační funkce @ jsme vycházeli z již existující literatury. Algoritmus top-k jsme rozdělili na více částí a vytvořili jsme nástroj Xoda, který je nezávislý na použité databázi, agregační funkci a uživatelských fuzzy funkcích. Tím je zajištěna maximální obecnost a jednoduchost jeho nasazení v různých prostředích. Vytvoření Xody umožňuje aplikaci top-k algoritmu v libovolné doméně, např. při implementaci vyhledávacího stroje. To je další přínos, nyní po praktické stránce.

Naimplementovali jsme algoritmy popsané v [FLN] a vylepšili jsme algoritmus bez přímého přístupu. Tím jsme radikálně zrychlili jeho čas výpočtu oproti přímé implementaci algoritmu tak, jak je popsán v [FLN]. Jednotlivé algoritmy jsme poté testovali na testovacích RDF datech. Zkoumali jsme také vliv rozložení hodnot v atributech, agregační funkce nebo použití relační databáze pro uložení dat. Také jsme porovnali systém Sesame a Jena.

Nakonec jsme vytvořili aplikaci Potlatch, která slouží k hodnocení výletů turistického oddílu. V této aplikaci je využita teorie z kapitoly 7., která umožňuje generování uživatelských hodnocení a skupinové rozhodování. Dále jsme vytvořili několik tříd, které usnadňují manipulaci s uživatelskými preferencemi a ukládání uživatelských fuzzy funkcí. Potlatch je ukázkovou aplikací, která využívá ontologii, pro uživatelsky příjemnou práci s daty. Systém sám nabízí možné vlastnosti a předměty tak, aby výsledná trojice odpovídala ontologii.

Apache Cocoon umožňuje snadné použití tříd jazyka Java pro tvorbu internetových stránek. Vytvořili jsme SesameGenerator, který provádí top-k algoritmus a výsledné objekty vrací jako XML soubor. SesameGenerator využívá Xodu, čímž se jeho implementace radikálně zjednodušila a zredukovala se v podstatě na oddělení parametrů.

Systém Xoda lze využít při libovolném vícekriteriálním hledání na internetu nebo v expertních systémech. Obecnost v definici třídy DataSearcher umožňuje dynamické získávání dalších dat z internetu v průběhu algoritmu a to z nejrůznějších zdrojů.

14. Bibliografie

[FLN]	Ronald Fagin, Amnon Lotem, Moni Naor. *Optimal Aggregation Algorithms for Middleware.* Extended abstract appeared in Proc. Twentieth ACM Symposium on Principles of Database Systems, 2001 (PODS 2001), pp. 102-113
[SESAME]	www.openrdf.com
[JENA]	http://jena.sourceforge.net/
[URI]	*RFC 2396 - Uniform Resource Identifiers (URI):Generic Syntax*, T. Berners-Lee, R. Fielding and L. Masinter, IETF, August 1998. http://www.isi.edu/in-notes/rfc2396.txt]
[ECONNECTIONS]	http://www.mindswap.org/2004/multipleOnt
[GALEN]	http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/galen.owl
[TRAVEL]	http://www.mindswap.org/2004/multipleOnt/FactoredOntologies/factoredTravel/
[PROTEGE]	http://protege.stanford.edu/
[SWOOP]	http://www.mindswap.org/2004/SWOOP/
[PELLET]	http://www.mindswap.org/2003/pellet/
[GLV]	P. Gurský, R. Lencses, P. Vojtáš. *Algorithm for User Dependent Integration of Ranked Distibuted Information*.
[FILMTRUST]	http://trust.mindswap.org/FilmTrust/
[BKS]	Stephan Börzsönyi, Donald Kossmann, Konrad Stocker. *The Skyline Operator.*
[KIM]	http://www.ontotext.com/owlim/
[RULEML]	http://www.ruleml.org
[VANEKOVA]	Veronika Vaněková. *Dopytovanie nad RDF dátami s užívatelskou preferenciou.*Diplomová práce 2006
[GEANAKOPLOS]	John Geanakoplos. *Three brief proofs of Arrow's impossibility theorem.* New Haven, Connecticut, http://cowles.econ.yale.edu/
[WIKIPEDIA]	http://en.wikipedia.org/wiki/Arrow%27s_Impossibility_Theorem
[LP]	Přednáška Preferenční dotazování DBI021 prof. Vojtáše na Matematicko-Fyzikální fakultě Univerzity Karlovy.
[W3C]	http://www.w3.org/rdf
[RDFRULEML]	http://www.ruleml.org/inrdf.html
[DL]	http://dl.kr.org/
[SOCIALCHOICE]	Kenneth J. Arrow. *Social Choice and Individual Values.* Cowles Foundation for research in economics at Yale University, 1963

Přílohy

1. Ontologie výletních míst

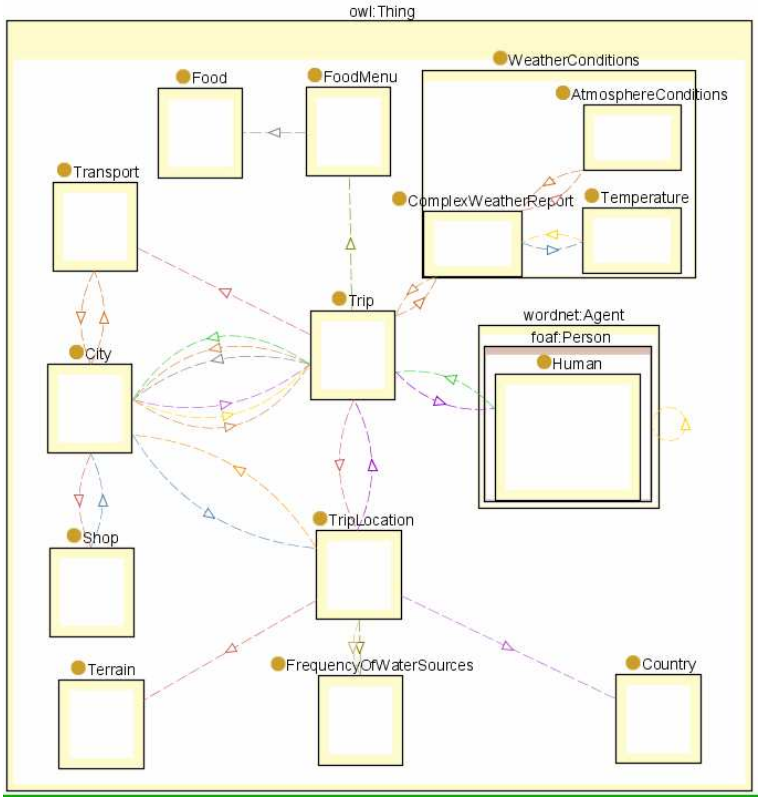
Ontologie je uložená na přiloženém CD v adresáři rdf/ontologie. Jsou zde obě verze - česká i anglická. Otevřít jdou v programech SWOOP nebo Protégé.

Třídy

Název třídy	Popis třídy
TripLocation	Místa, kam se dá jezdit na výlety
Trip	Výlety
Terrain	Terén místa výletu
Transport	Dopravní prostředek
WeatherConditions	Počasí
Shop	Obchod
City	Město v lokalitě
Country	Země
Food	Jídlo
FoodMenu	Jídelníček sestávající se z jídel
FrequencyOfWaterSources	Četnost zdrojů vody v lokalitě
Human	Účastníci výletu, členové skupiny

Vlastnosti

Hlavní dvě třídy jsou Trip, reprezentující výlety a TripLocation, reprezentující místa, kam se dá jezdit na výlety.



Na schématu je zobrazena celá struktura základních tříd, které jsou provázané vlastnostmi. Pokud je třída vnořena do jiné, znamená to, že vnitřní třída je podtřídou vnější. Pro přehlednost jsou podtřídy pouze tam, kde je to nezbytné. Další podtřídy si ukážeme dále.

Nyní si ukážeme podtřídy jednotlivých tříd.

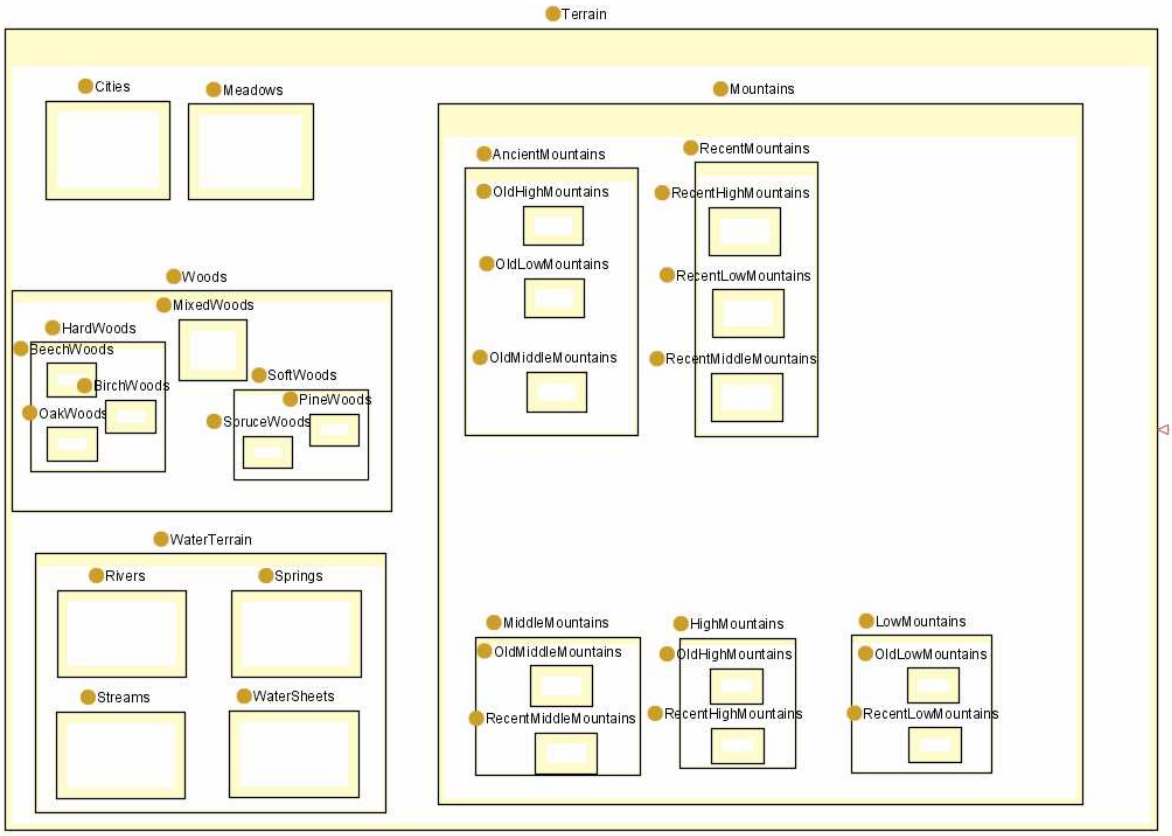


Schéma znázorňuje podtřídy třídy Terrain. Máme městský terén, louky, vodní terény, které se dále dělí na řeky, potoky, prameny a vodní plochy, lesy a hory. Lesy se dělí na jehličnaté, listnaté a smíšené. Ty se dále dělí podle převládajícího druhu stromu v lese. Horský terén se dělí podle výšky a stáří hor. Hory jsou rozděleny na staré a mladé, vysoké, nízké a středně vysoké. Každá z těchto tříd má dále podtřídu z druhého parametru - např. vysoké hory mají podtřídy vysoké a staré hory a vysoké a mladé hory.

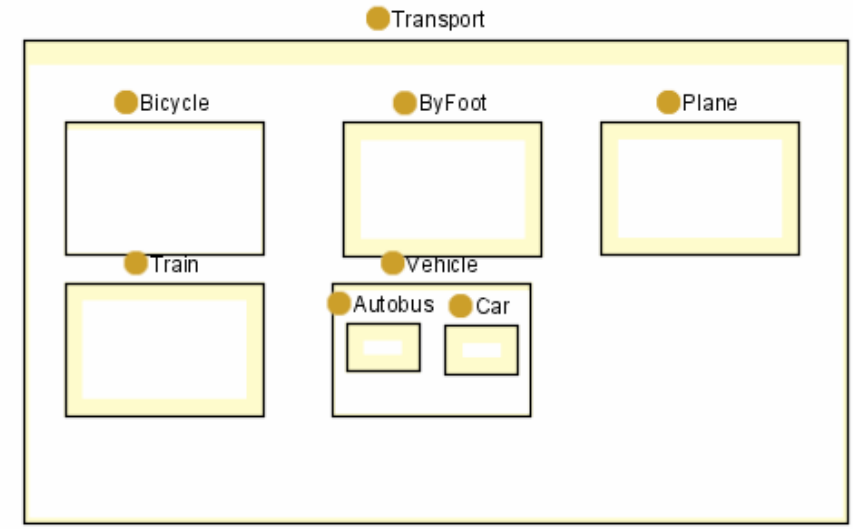
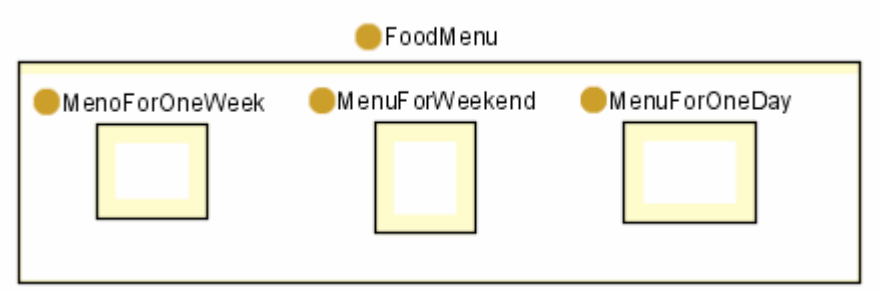
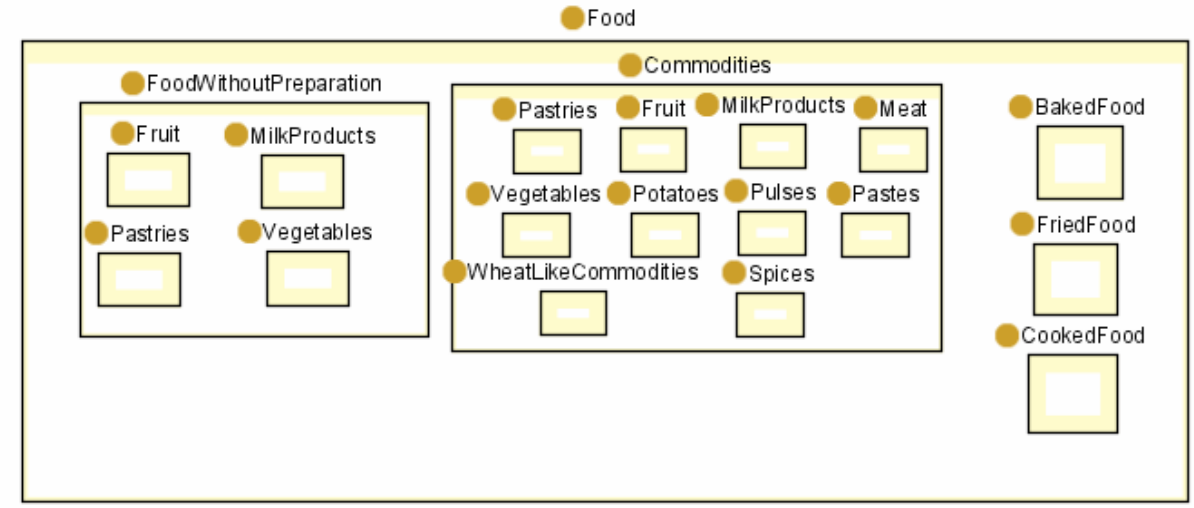


Schéma znázorňuje podtřídy třídy Transport.



Jídelníček se dělí na podtřídy podle doby, na kterou jídelníček plánuje jídlo. Jednotlivé podtřídy mají omezení - jídelníček na jeden den nesmí obsahovat žádná jídla, kde je nutné vaření, jsou povolena pouze jídla bez přípravy.

Jídelníček na celý týden musí obsahovat alespoň tři jídla a některé z jídel musí být tepelně zpracované.



Jídla se dělí podle způsobu přípravy - jídla bez přípravy, suroviny na vaření, pečená, smažená a vařená jídla.

2. Konfigurace SesameGeneratoru pro Apache Cocoon

Pro používání SesameGeneratoru v systému Cocoon musíme nejdříve samotný Cocoon nainstalovat do cesty %COCOON%. Poté do adresáře %COCOON%\build\webapp\WEB-INF\classes\ nakopírujeme SesameGenerator.class.

Do adresáře %COCOON%\build\webapp\WEB-INF\lib\ nahrajeme přiložené .jar soubory Sesamu a xoda.jar.

Dále v souboru %COCOON%\build\webapp\sitemap.xmap přidáme řádek

```
<map:generator name="SesameGenerator" src="SesameGenerator"/>
```

tak, aby byl obsažen ve značce <map:generators>

Nakonec do značky <map:pipelines> vložíme řádky

```
<map:match
      pattern="sesameTrips.xml">
   <map:generate
      type="SesameGenerator"/>
   <map:serialize type="xml"/>
</map:match>
```

čímž přiřadíme pro soubor sesameTrips.xml generátor SesameGenerator. To znamená, že pří požadavku souboru sesameTrips.xml na server Apache se použije generátor SesameGenerator a jeho výsledky budou zpracované standardním serializátorem pro XML soubory. Pokud bychom chtěli výsledky ještě nějak upravit, museli bychom zadefinovat nový Transformer a Serializer.

Poté když se při psaní formuláře HTML odkážeme na zdroj formuláře pomocí řádky

```
<FORM METHOD=GET ACTION="http://localhost:8888/sesameTrips.xml">
```

,

předají se vyplněné parametry do SesameGeneratoru.

Parametry by měly mít tuto strukturu -

1) váha vlastnosti má tvar property#NázevVlastnosti

2) použité uspořádání domény vlastnosti má tvar propertyNormalizer#NázevVlastnosti

3) pokud se použilo OnePeakOrdering, zadaná ideální hodnota má tvar propertyPeak#NázevVlastnosti

4) počet položek ve výsledku se zadává parametrem K

5) třída, kterou hledáme, se zadává parametrem Type

SesameGenerator si načte nejdříve konfigurační soubor a z něj získá typické hodnoty. Tyto hodnoty mohou být přepsány pomocí parametrů zadaných na formuláři (K je typicky 10, na formuláři lze zadat např. 1). Tento souboru je uložen v %COCOON%\build\webapp\resources\settings.xml

SesameGenerator používá standardně Sesamovské úložiště v Apache Tomcat, SesameGenerator lze ovšem nakonfigurovat pro čtení dat ze souboru.

Příklad HTML stránky je na přiloženém CD v adresáři programy/vytvorene/SesameGenerator/txt.

3. Konfigurace Apache Tomcat a Sesamu

Nejdříve nainstalujeme Apache Tomcat. V podadresáři webapps vytvoříme adresář sesame, kam nakopírujeme obsah sesame.war. Tento archiv umí otevřít např. Total Commander.

V adresáři sesame/WEB-INF je soubor system.conf, kde se nachází konfigurace jednotlivých úložišť.

Po spuštění služby Tomcat se můžeme k Sesamu připojit přes internetový prohlížeč zadáním adresy http://localhost:8080/sesame. Toto rozhraní umožňuje provádět základní operace nad jednotlivými úložišti - přidání souborů, provádění dotazů nebo mazání obsahu.

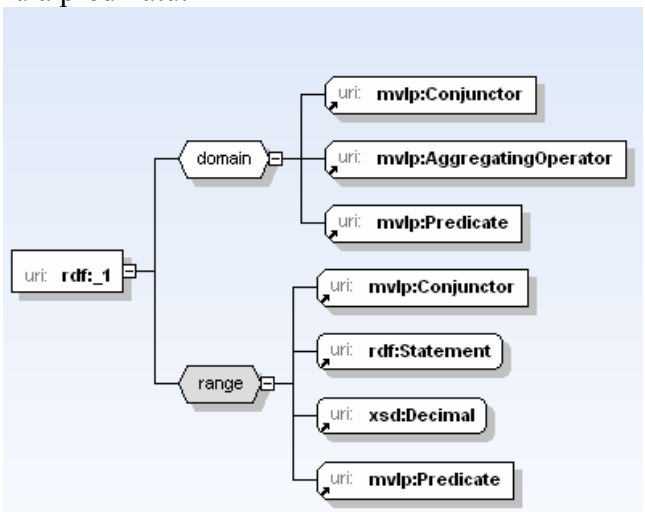
Pro úpravy úložiště se nejprve musíme k Sesamu přihlásit. Standardní přihlašovací jméno je testuser, heslo je opensesame. Uživatelé a hesla se definují v souboru system.conf.

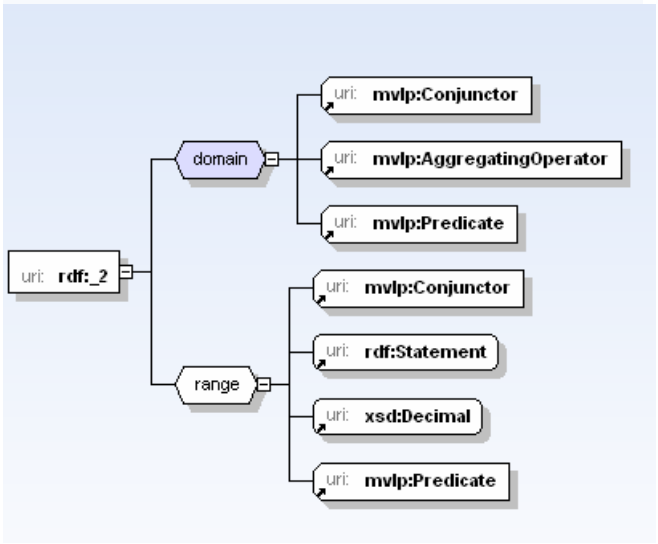
Pokud budeme chtít využívat Sesame s databází, např. MySQL, musíme tuto databázi nainstalovat a vytvořit uživatele Sesame s dostatečnými právy. Postup je blíže rozepsán na http://www.openrdf.org/doc/sesame/users/ch02.html#d0e351.

4. RDFS popisující vícehodnotový logický program

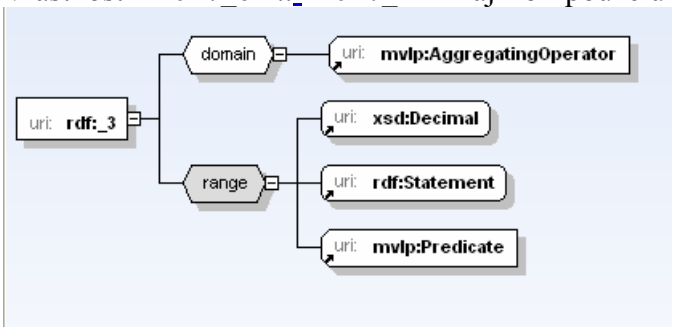
Vlastní RDFS je umístěno na CD v adresáři rdf/ontologie.

Zde si ukážeme schémata vlastností, tedy jejich definiční obory a obory hodnot. Nejdříve vlastnosti `<rdf:_1>` a `<rdf:_2>`, které hrají roli u konjunktorů, agregačních operátorů a predikátů.

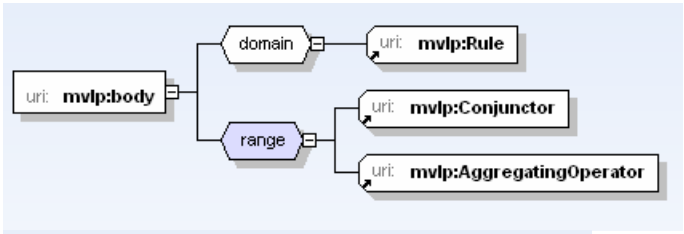


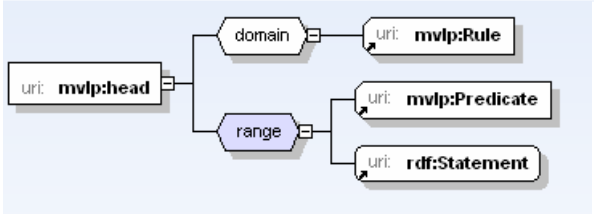


Vlastnosti `<rdf:_3>` .. `<rdf:_x>` hrají roli pouze u agregačních operátorů

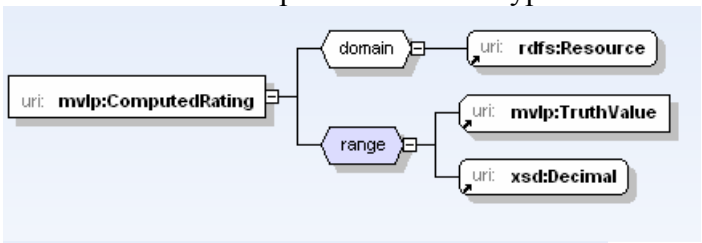


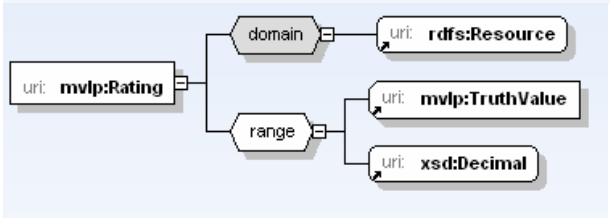
Dále vlastnosti pro pravidla `<mvlp:head>` a `<mvlp:body>`.





A nakonec vlastnosti pro hodnocení a vypočtené hodnocení





Bohužel RDFS nedovoluje zapisovat nové axiomy, které jsme nadefinovali v kapitole 6. Tyto požadavky bude muset kontrolovat program na vyšší úrovni.