PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

# DIPLOMOVÁ PRÁCE

Vyhledávač založený na formální konceptuální analýze

2011 Lukáš Havrlant

**Anotace**

*Napovídáním souvisejících dotazů může vyhledávač pomoci uživateli rychleji najít dokumenty, které potřebuje. Práce se zabývá tvorbou vyhledávače s webovým rozhraním, který pracuje nad uzavřenou sadou dokumentů. Po položení dotazu dokáže napovědět konkrétnější, obecnější a podobný dotaz, což je realizováno pomocí formální konceptuální analýzy.*

Děkuji Mgr. Janu Outratovi, Ph.D. za vedení této diplomové práce a za rady při konzultacích.

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

Současné vyhledávače si uchovávají obsah, nad kterým mají vyhledávat, ve formě indexu, což je struktura v principu podobná indexu v knize. Uživateli stačí vložit do vyhledávacího pole svůj dotaz a vyhledávač z indexu získá potřebná data a zobrazí uživateli výsledek, obvykle ve formě nějakého uspořádaného seznamu odkazů na dokumenty. Pokud není uživatel spokojený s výsledky, musí přeformulovat svůj dotaz tak, aby lépe vystihoval to, co chce najít.

Tento problém je typický pro slova, která mají několik významů. Například pokud ve webovém vyhledávači vyhledáme slovo „jaguár“, tak vyhledávač nemůže vědět, zda chceme hledat auto, zvíře nebo ještě něco jiného. Pokud má přístup k historii hledání daného uživatele, může pomocí ní přizpůsobit výsledky. Ale může také uživateli zobrazit návrhy na nový dotaz, například „jaguár auto“ nebo „jaguár zvíře“, po jejichž vyhledání se výsledky velmi zpřesní.

Otázkou je, jak tyto návrhy získat. Pokud si vyhledávač uchovává historii všech hledání, která uživatelé provádějí, může se je pokusit vytáhnout právě z této historie. Pokud ovšem tato data vyhledávač nemá, nebo jich má málo, musí se použít jiná metoda.

Práce popisuje metodu, jak získat podobné dotazy pomocí formální konceptuální analýzy (anglicky Formal concept analysis, dále jen FCA) pouze na základně znalosti obsahu dokumentů. FCA pracuje s tabulkovými daty, ve kterých hledá nějaké potenciálně zajímavé shluky dat. Tyto shluky pro nás budou představovat množiny dokumentů, které jsou nějakým způsobem podobné a které sdílí nějaká zásadní klíčová slova. Tato klíčová slova poté dále využijeme při generování návrhů na nové dotazy.

# 1. Information retrieval

Tato sekce se zabývá budováním samotného vyhledávače, jehož výsledky budou zobrazeny uživateli a také budou použity jako vstup do formální konceptuální analýzy.

Cílem je popsat tvorbu vyhledávače, který bude umět stáhnout z webu požadované dokumenty, zaindexovat je a následně v nich vyhledávat. Dokumenty mohou být buď obyčejné webové stránky nebo složitější soubory, například PDF.

## 1.1. Jak funguje vyhledávač

V této části si popíšeme, jak může obecný vyhledávač fungovat.

### 1.1.1. Pojmy související s vyhledávačem

**Sada dokumentů** Na začátku máme nějakou sadu dokumentů, nad kterými chceme vyhledávat. Tato sada může být libovolná – může se jednat o webové stránky, o lokální dokumenty na osobním počítači nebo o e-maily na serveru. V případě webových vyhledávačů typu Google je pak sadou dokumentů „všechno, co lze nalézt na webu“, v případě vyhledávačů v moderních operačních systémech je sadou „všechno, co lze nalézt na počítači“.

Sada dokumentů může být statická, nebo dynamická. Statická sada je taková sada, která se nemění buď nikdy nebo málokdy. Může to být například vyhledávač PSČ. Dynamická sada se naopak mění poměrně často, například webové stránky. Velké vyhledávače častěji pracují s dynamickou sadou dokumentů.

**Uživatel** Uživatel je obecné označení toho, kdo pracuje s vyhledávačem. Typicky se jedná o nějakého člověka, ale může to být i program, který například každý den kontroluje počet dokumentů na dané klíčové slovo a při zvýšení tohoto počtu pošle někomu informační e-mail.

**Dotaz** Pomocí dotazů komunikuje uživatel s vyhledávačem. Dotaz je nějaký seznam slov, který reprezentuje to, co uživatel chce najít. Může se jednat o jednoduché dotazy typu „auto“, ale také o složitější otázky jako „Jaký je smysl života?“ Slovům v dotazu říkáme klíčová slova.

Vyhledávač může uživatelům umožnit používat v dotazu různé operátory, kterými lze zpřesnit dotaz. Typické operátory jsou logické operátory AND, OR a NOT. Webové vyhledávače umožňují například vyhledávat na určité doméně pomocí SITE operátoru, vyhledat odkazující stránky pomocí LINK operátoru a podobně.

**Výsledky vyhledávání** Po položení dotazu zobrazí vyhledávač výsledky. Obecně se jedná o nějaké zkrácené zobrazení dokumentů, které vyhovují danému dotazu. Typicky jde o seznam odkazů na dané dokumenty, ale v některých případech může vyhledávač přímo zodpovědět položenou otázku. Například na otázku o smyslu života může odpověď 42.

Samotný seznam dokumentů bývá obvykle seřazen podle nějakých kritérií. Těchto kritérií bývá poměrně hodně a vyvážit je tak, aby vyhledávač vracel co možná nejlepší výsledky je obtížný úkol, kterým se tato diplomová práce nebude více zabývat.

Mezi výsledky může být kromě seznamu odkazů i nějaká další informace. Současné webové vyhledávače například umí opravovat překlepy, přehrát video odpovídající klíčovým slovům, vypočítat jednoduché matematické operace nebo napovědět dotaz, který by vedl k přesnějším výsledkům.

Dále si představíme dva modely, jak může vyhledávač fungovat. Naivní model a model používající index.

### 1.1.2. Naivní model vyhledávače

Naivní model vyhledávače funguje tak, že po položení dotazu začne vyhledávač procházet všechny dokumenty a kontroluje, jestli některý z dokumentů odpovídá dotazu. Pokud ano, vrátí tento dokument mezi výsledky.

Tento model je velice jednoduchý, ale není příliš efektivní. V současné době můžeme mít sady dokumentů, které obsahují miliony či miliardy položek, takže chceme-li znát výsledky řádově za desetiny sekund, není možné je při každém dotazu všechny procházet.

### 1.1.3. Vyhledávač používající index

Mnohem efektivnější je použít index. Jedná se o podobnou strukturu, kterou můžeme najít v některých, obzvláště odborných, knihách. Protože v klasické tištěné knize nemůžeme nijak „vyhledávat“, dává se na konec knihy seznam nejdůležitějších slov, která kniha obsahuje, spolu s čísly stránek, na kterých se pojem vyskytuje. Takže chce-li uživatel nalézt stránky obsahující slovo „derivace“, podívá se do indexu, kde hned zjistí, že slovo se vyskytuje na té a té stránce.

Podobný princip můžeme použít i ve vyhledávačích. Nebudeme ale vytvářet index z důležitých slov, ale ze všech slov, která se v dokumentech vyskytují. Vytvoříme tak slovníkovou strukturu, kde klíčem bude slovo a hodnotou bude seznam dokumentů, které dané slovo obsahují. Při položení dotazu pak může vyhledávač rychle zjistit jaké dokumenty obsahují dané klíčové slovo prostým nahlédnutím do tohoto slovníku.

Spolu s tím si můžeme uložit informace i o tom, kolik daných slov daný dokument obsahuje, což můžeme dále využít při řazení dokumentů.

Celou práci vyhledávače tak můžeme rozdělit do dvou částí – budování a upravování indexu a hledání odpovědí na dotaz. Zde je nutné dát pozor na to, jak velká je sada dokumentů a zda je statická nebo dynamická. Samotné budování indexu můžeme ještě rozdělit na dvě části: předzpracování dokumentů a technická realizace indexu.

## 1.2. Vyhledávač CLaSeek

Součástí této diplomové práce je naprogramovaný vyhledávač CLaSeek (Concept Lattice Seeker), který bude v dalších částech textu popsán. CLaSeek je vyhledávač napsaný v Pythonu 3. Pracuje se statickou sadou dokumentů; předpokládá se, že se sada dokumentů bude měnit pouze nárazově jednou za čas. Samotná sada dokumentů, nad kterou má vyhledávač pracovat, nebude příliš velká, řádově stovky dokumentů.

Jedná se o vyhledávač používající index. Při budování indexu je vstupem seznam URL adres, který si vyhledávač sám stáhne a zaindexuje obsahy dokumentů. Výsledný index je pak uložen do několika souborů, ke kterým pak CLaSeek přistupuje.

CLaSeek má rozumět logickým operátorům AND, OR a NOT. Po zadání dotazu má vrátit výsledky seřazené podle relevance, kterou spočítá pomocí klasického tf-idf algoritmu. Výstup bude textový ve formátu JSON, se kterým pak mohou pracovat další programy, v tomto případě webové rozhraní, které je napsáno v PHP.

Další částí vyhledávače je hledání souvisejících dotazů. Tato část bude podrobně rozebrána v další kapitole. V této kapitole je dále popsáno, jak CLaSeek buduje index a jak vrací výsledky.

## 1.3. Předzpracování dokumentů

Při budování indexu máme na vstupu sadu dokumentů a na výstupu strukturu, která reprezentuje index této sady. Během samotného budování indexu procházíme jednotlivé dokumenty a upravujeme je do takové podoby, která se hodí pro uložení. Budeme uvažovat pouze textové dokumenty jako je HTML nebo PDF a budeme předpokládat statickou sadu dokumentů, takže se nebudeme příliš zatěžovat aktualizací indexu.

Na samotném začátku tak musíme upravit jednotlivé textové dokumenty do nějaké kanonické podoby. To budeme dělat postupnými úpravami jednotlivých dokumentů. S každým dokumentem budeme provádět identické operace v identickém pořadí. Jednotlivé operace budou popsány v takovém pořadí, v jakém se aplikují ve vyhledávači. Všechny operace jsou popsány v knihách [3], kapitola 7 a [15], kapitola 2.

**Odstranění formátovacích prvků dokumentu** Vstupem do algoritmu budování indexu může být soubor, který kromě samotného textu obsahuje i různé formátovací a jiné prvky. Tyto prvky pro další zpracování nepotřebujeme, takže je všechny odstraníme. Jedná se například o HTML nebo XML značky. Po aplikaci tohoto bodu už bychom měli mít pouze čistý text.

**Ponechání písmen** Text obsahuje spousty různých znaků, které jsou vyhledávači téměř k ničemu. Jedná se o interpunkci, která typicky doplňuje text, ale při samotném vyhledávání se bez ní obejdeme. Odstraníme tak všechny znaky, které nejsou písmeny.

Při tomto odstraňování můžeme narazit na určité problémy a může zde dojít k první ztrátě informace. Například pokud ze slova „chcete-li“ odstraníme spojovník, můžeme dostat buď slovo „chceteli“ nebo dvě slova „chcete“ a „li“.

**Odstranění bílých znaků** Pod bílé znaky spadají mezery, tabulátory a nové řádky. To jsou opět informace, které jsou zbytné a které můžeme odstranit. Namísto libovolně dlouhé posloupnosti bílých znaků tak vždy vložíme právě jednu mezeru. Tím dosáhneme toho, že všechna slova v celém dokumentu budou v jednom řádku a budou oddělena právě jednou mezerou.

**Převod na malá písmena** Velikost písmen má pouze minimální vliv na hodnocení dokumentů, takže můžeme všechna písmena převést na jednotný tvar. V tomto případě na malá písmena. Můžeme sice nalézt případy, kdy ztratíme jistý kousek informace, například „Prokop Buben“ vs. „prokop buben“, ale tento kousek je natolik zanedbatelný, že si to můžeme dovolit.

**Odstranění stop slov** Stop slova jsou slova, která se vyskytují téměř v každém dokumentu, jsou příliš obecná a nicneříkající, a tak nemá příliš velký smysl je indexovat. Typicky se jedná o předložky jako „ke“, „u“, „na“ a podobně. Pokud je odstraníme, neztratíme příliš mnoho informací, ale můžeme tím i poměrně výrazně zredukovat velikost výsledného indexu. Obecně můžeme odstranit všechna slova délky jedna nebo dva bez větší ztráty informace.

**Převod na stemy** Jedna z nejdůležitějších části vyhledávače je převod slov na stemy. Stem je kořen, základ slova. Smyslem je, abychom si v indexu neuchovávali všechny tvary každého slova, ale abychom si od každé slova uchovávali ideálně jen jeden, základní tvar – místo „strom“, „stromy“, „stromu“ tak budeme mít pouze jeden tvar „strom“.

Důvody pro to máme v zásadě dva: opět se velmi výrazně sníží velikost indexu, protože namísto například pěti tvarů, budeme mít uchovaný pouze jeden tvar. Druhým důvodem je, že pokud uživatel hledá slovo „strom“, pravděpodobně bude spokojený i s dokumentem, který obsahuje slova „stromy“ a „stromu“, ale neobsahuje samotné slovo „strom“.

Algoritmus, který by k libovolnému slovu vrátil jeho základ, je poměrně komplikovaný a většinou ani není, alespoň v případě českého jazyka, příliš úspěšný. Přesto se stále vyplatí nějaký stemmer použít a ve vyhledávači použit je.

**Odstranění diakritiky** Po převodu na stemy můžeme odstranit diakritiku. Odstraněním diakritiky může vzniknout několik konfliktů, které před odstraněním neexistovaly, ale je to jednoduchý způsob, jak umožnit vyhledávat i lidem, kteří diakritiku vůbec nepoužívají. Navíc ani autor nějakého dokumentu diakritiku nemusel používat a z některých dokumentů se kvůli problémům s kódováním ani písmena s diakritikou nepodaří získat. Odstraňovat veškerou diakritiku je tak nejmenší zlo.

## 1.4. Uložení indexu

V tuto chvíli máme vstupní dokumenty převedeny do tvaru, se kterým můžeme pracovat dále a můžeme vytvořit základní index. Do něj si uložíme informace o tom, ve kterých dokumentech se vyskytuje daný stem. Pokud pak budeme chtít získat dokumenty, které obsahují stem „strom“, vyhledáme si v indexu záznam o stemu „strom“ a okamžitě získáme množinu dokumentů, která stem obsahuje.

Bude se nám také hodit informace o počtu stemů v daném dokumentu. To využijeme během řazení dokumentů od nejvíce relevantních. Nakonec si můžeme ještě uložit informaci o počtu výskytů daného stemu ve všech dokumentech. Všechny tyto informace můžeme uložit do struktury, která vypadá takto:

```
stem => (počet výskytů slova ve všech dokumentech,
         [(doc1, počet výskytů v doc1),
          (doc2, počet výskytů v doc2),
          ...,
          (docN, počet výskytů v docN)])
```

Proměnné `doc1`, `doc2`, ... představují ID dokumentů, které obsahují daný stem a každé slovo tak může obsahovat různou sadu docID. Pokud dokument `docA` stem neobsahuje, pak ve struktuře vůbec řádek `(docA, počet výskytů v docA)` není. Myšlenka indexu je popsána v [15], kapitola 1 a 2.

## 1.5. Vyhledání atributů dokumentu

V druhé části vyhledávače, která se zabývá nalezením souvisejících dotazů, budeme potřebovat znát množinu slov, která nejvíce charakterizuje daný dokument. Těmto slovům budeme říkat „atributy dokumentu“.

### 1.5.1. Jak by měly atributy vypadat

Atributy jsou poměrně běžnou součástí různých vědeckých článků, kde je ale obvykle vyplňuje sám autor pod názvem „klíčová slova“. V případě vyhledávače stojíme před problémem, jak získat atributy z libovolného dokumentu nějakým obecným způsobem.

Jednoduchým způsobem je seřazení všech stemů v daném dokumentu podle jejich četnosti od nejvíce častého. Za atributy pak můžeme vzít ty stemy, které překročí nějakou absolutní hranici („alespoň 10 výskytů v dokumentu“) nebo nějakou relativní hranici („poměr počtu daného stemu ku počtu všech stemů v dokumentu je větší než 0,05“).

Tento postup může být úspěšný v případě, kdy máme pouze jeden dokument. Pokud ale máme sadu dokumentů, můžeme ještě zjistit vztah s dalšími dokumenty. Atribut pro daný dokument by totiž mělo být takové slovo, které je mezi ostatními dokumenty co možná nejvíce unikátní.

Pokud máme sadu dokumentů, která se zabývá analýzou dat, je možné, že by v každém dokumentu bylo nejčastější slovo právě „analýza“. Tím bychom dostali pro každý dokument stejný atribut a to není to, co bychom chtěli.

Tento problém vyřešíme tím, že při hledání atributů pro dokument vezmeme v potaz i to, jak často se daná slova vyskytují v ostatních dokumentech. Budeme tak hledat taková slova, která se v daném dokumentu vyskytují co nejčastěji a v ostatních dokumentech co nejméně často.

### 1.5.2. Algoritmus tf-idf

Tento postup má své jméno, jedná se o tf-idf algoritmus. Ten je rozdělený do několika částí. První je funkce $\mathrm{tf}_{t,d}$, která v základním nastavení vrací počet výskytů slova $t$ v dokumentu $d$. Dále máme funkci $\mathrm{df}_t$, která vrací počet dokumentů, které obsahují slovo $t$. Tuto funkci využijeme k tomu, abychom snížili skóre těch slov, která se vyskytují v příliš mnoha dokumentech.

Označme $N$ počet všech dokumentů v naší sadě. Pak vydělením $N/\mathrm{df}_t$ získáme koeficient, který značí, jak moc je slovo $t$ unikátní. Pokud se vyskytuje jen v jednom dokumentu, získáme maximální hodnotu $N$. Při vyšším výskytu slov v dokumentech by tento koeficient klesal příliš rychle, proto ještě použijeme logaritmus. Získáme funkci $\mathrm{idf}_t$

$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t}.$$

Složením funkcí tf a idf získáme funkci tf-idf′ (za chvíli tuto funkci ještě vylepšíme, prozatím si ji označíme s apostrofem) definovanou jako

$$\text{tf-idf}'_{t,d} = \mathrm{tf}_{t,d} \cdot \mathrm{idf}_t.$$

Tato funkce přiřazuje slovu $t$ a dokumentu $d$ hodnotu, která je

- vysoká, pokud se slovo $t$ často vyskytuje v malé množině dokumentů,
- nízká, pokud se slovo vyskytuje málo v dokumentu $d$ nebo pokud se vyskytuje hodně v ostatních dokumentech.

Algoritmus tf-idf je popsán v [15], kapitola 6.

### 1.5.3. Vylepšení algoritmu tf-idf

Problémem tohoto přístupu je, že příliš preferuje velké dokumenty, které obsahují mnoho slov. Máme-li například učebnici středoškolské matematiky, je pravděpodobné, že bude několikrát, řekněme 50krát, obsahovat slovo „kombinace“. Vedle toho můžeme mít desetistránkový dokument pojednávající čistě o kombinacích, ale slovo „kombinace“ obsahuje pouze 25krát. Podle stávající $\mathrm{tf}_{t,d}$ funkce bude učebnice na klíčové slovo „kombinace“ dvakrát relevantnější než článek přímo se zaměřující na kombinace.

Tento problém můžeme zkusit vyřešit tím, že hodnotu $\mathrm{tf}_{t,d}$ ještě vydělíme celkovým počtem slov v dokumentu. Získáme tak relativní zastoupení slova $t$ mezi všemi slovy v dokumentu. Učebnice z předchozího příkladu pak bude mít mnohem nižší hodnotu $\mathrm{tf}_{t,d}$, protože je mnohonásobně větší než článek. Vzorec funkce tf-idf by pak vypadal takto:

$$\text{tf-idf}''_{t,d} = \frac{\mathrm{tf}_{t,d}}{|d|} \cdot \mathrm{idf}_t,$$

kde $|d|$ je počet slov v dokumentu $d$.

Problém jsme tím ale ve skutečnosti nevyřešili, jen jsme ho obrátili – už nejsou preferované velké dokumenty, ale malé dokumenty, které dané klíčové slovo obsahují. Pokud bychom zpracovali dokument, jehož obsahem by byla pouze věta „Sázky a kurzy na severskou kombinaci.“, pak by hodnota $\mathrm{tf}_{t,d}$ pro slovo „kombinace“ byla $\frac{1}{4}$ – v dokumentu jsou čtyři slova (slova „a“ a „na“ nepočítáme, jsou to stop slova) a jedno z nich je právě „kombinace“ (po převedení na stem). Pokud by měl mít zmíněný článek, který obsahuje 25krát slovo „kombinace“, alespoň stejnou hodnotu $\mathrm{tf}_{t,d}$, nesměl by mít více než sto slov.

CLaSeek nakonec hodnotu $\mathrm{tf}_{t,d}$ ještě dělí logaritmem počtu všech slov v dokumentu. Hodnota $\mathrm{tf}_{t,d}$ u velkých dokumentů tak bude vydělena větší hodnotou než u malých dokumentů. Zároveň ale tato hodnota, kterou dělíme, nebude růst lineárně, takže desetkrát větší dokument nepotřebuje i desetkrát více klíčových slov, aby dosáhl na stejné hodnocení.

Nyní můžeme napsat finální verzi funkce tf-idf:

$$\text{tf-idf}_{t,d} = \frac{\mathrm{tf}_{t,d}}{\log |d|} \cdot \mathrm{idf}_t,$$

kde $|d|$ je počet slov v dokumentu $d$. Tato modifikace je vlastní, v [15] jsou popsány jiné možné modifikace tf-idf algoritmu.

### 1.5.4. Vygenerování atributů dokumentu

Nyní už máme všechny prostředky k tomu, abychom pro každý dokument vygenerovali jeho atributy. Pro každý dokument $d$ vezmeme množinu všech stemů, které obsahuje a pro každý stem $s$ spočítáme jeho tf-idf$_{s,d}$ skóre.

Za atributy pak můžeme vzít buď určitý počet stemů, které dosáhly nejvyššího skóre, nebo všechny stemy, které překročily určitou hranici tf-idf skóre. CLaSeek v základním nastavení pracuje podle druhého způsobu, takže mu můžeme nastavit minimální hranici tf-idf skóre pro atributy dokumentu a všechny stemy, které dosáhnou alespoň takového skóre, budou považovány za atributy pro daný dokument.

Dále můžeme ve vyhledávači nastavit:

- Minimální počet atributů. Pomocí tohoto nastavení můžeme specifikovat, že pro každý dokument chceme například alespoň tři atributy. Pokud se přes hranici dostanou pouze dva stemy, třetí stem s největším tf-idf skóre bude také považován za atribut dokumentu.
- Maximální počet atributů pro každý dokument.
- Vyhledávač umožňuje přepnout do druhého modelu, kdy se za atributy berou například tři stemy s největším tf-idf skórem. Žádná hranice se pak neuplatňuje.

Podrobnější informace o nastavení je v dokumentaci, viz kapitolu 5.3.

## 1.6. Další funkce vyhledávače

### 1.6.1. Inverzní stemovací funkce

Během zpracování textů jsme používali stemmer, který bral na vstup slovo a na výstupu vrátil základ slova. Bohužel pro poměrně velkou část slov vrací funkce takový základ slova, který je sám o sobě nesmyslný.

Například pro slovo „množina“ získáme s použitým stemmerem stem „mnoh“. Toto chování nám nevadí v případě budování indexu, ale vadilo by nám v druhé části. Nemůžeme uživateli napovědět, že má k dotazu přidat klíčové slovo „mnoh“, protože nebude vědět, jaké slovo ve skutečnosti přidává.

Vyhledávač vylepšíme tím, že se pokusíme najít inverzní funkci ke stemovací funkci, abychom mohli ke každému podobně špatnému stemu přiřadit nějaké reálné slovo. Bohužel inverzní funkce neexistuje, protože stemovací funkce není prostá. Sestrojíme takovou funkci, která bude mít na vstupu stem $s$ a na výstupu jedno ze slov, které má $s$ jako svůj stem. Mějme tak původní stemmer, který označíme jako funkci stem, která nám pro slovo vrací jeho stem. Budeme chtít sestavit funkci stem$^{-1}$, která nám pro stem vrací nějaké přirozené slovo.

Při používání funkce stem si tak u každého výsledného stemu $s$ budeme uchovávat množinu původních slov $P_s$, která se zobrazují právě na stem $s$. Tedy pokud $s = stem(w)$, pak $P_s = P_s \cup \{w\}$. Množina $P_s$ pak splňuje $\forall x \in P_s : \mathrm{stem}(x) = s$.

Zároveň si uložíme počet jednotlivých slov ve všech dokumentech. K tomu účelu definujeme funkci $\mathrm{sf}(w)$, která vrací počet výskytů slova $w$ ve všech dokumentech. Budeme chtít, aby nám funkce $\mathrm{stem}^{-1}$ vrátila z množiny $P_s$ takové slovo, které se v původní sadě dokumentů vyskytovalo nejčastěji.

Nejprve zadefinujeme pomocnou funkci $\mathrm{stem}^{-1}_{max}$:

$$\mathrm{stem}^{-1}_{max}(s) = \left\{ x \in P_s \mid \mathrm{sf}(x) = \max_{y \in P_s} \mathrm{sf}(y) \right\}.$$

Tato funkce nám vrátí množinu všech slov, která se zobrazují na stem $s$ a jejichž počet výskytů v sadě dokumentů je shodný, ale maximální mezi všemi slovy z $P_s$. Funkce $\mathrm{stem}^{-1}$ může z této množiny vrátit libovolný prvek. Aby byla funkce jednoznačně definována, vrátí takové slovo, které je nejmenší vzhledem k lexikografickému uspořádání:

$$\mathrm{stem}^{-1}(s) = w \quad \Leftrightarrow \quad w = \min_{x \in \mathrm{stem}^{-1}_{max}(s)} x.$$

Pokud nyní použijeme funkci $\mathrm{stem}^{-1}$ například na stem „mnoh“, měla by vrátit slovo „množina“, protože to je slovo, které má stem „mnoh“ a pravděpodobně se v dokumentech vyskytuje nejčastěji.

Idea inverzní stemovací funkce je popsána v [11], konkrétní implementace je vlastní.

### 1.6.2. Zjištění názvu a popisku dokumentu

Moderní vyhledávače typu Google nebo Seznam zobrazují ve výsledcích vyhledávání název dokumentu a popisek dokumentu. V případě HTML stránek se většinou jedná o obsah `TITLE` elementu. Jako popisek je buď zvolen obsah elementu `META`, který obsahuje popisek, nebo část stránky, která obsahuje klíčová slova, která uživatel hledal.

CLaSeek pracuje podobně a jako název HTML stránky volí obsah `TITLE` elementu. Pokud stránka nemá `TITLE` element nebo vyhledávač nezpracovává HTML stránku, použije se název souboru.

Jako popisek stránky se použije element `META` obsahující popisek, pokud existuje. V opačném případě vyhledávač v dokumentu nalezne několik vět, které obsahují atribut dokumentu, a tyto věty pak uloží jako popisek dokumentu. Oproti řešení, které volí Google nebo Seznam, je nevýhoda v tom, že se popisek nemění s dotazem uživatele, je vždy stejný.

### 1.6.3. Kontrola překlepů

CLaSeek má implementovanou jednoduchou kontrolu překlepů. V případě, že uživatel položí vyhledávači dotaz, na který vyhledávač odpoví prázdnou množinou výsledků, pokusí se zjistit, zda uživatel neudělal v některém klíčovém slovu v dotazu chybu.

Aby vyhledávač mohl tuto kontrolu provádět, musí znát množinu „správných“ slov. Můžeme buď sehnat jednu univerzální množinu všech validních slov v daném jazyku, nebo můžeme použít slova z dokumentů, která zpracováváme. Stačí si tak během zpracování a budování indexu uložit všechna slova, na která jsme v dokumentech narazili. Druhý přístup má dvě zásadní výhody.

- Kontrola překlepů bude rychlejší, protože množina slov z dané databáze bude pravděpodobně menší než množina všech slov daného jazyka. Vyhledávač tak bude hledat správná slova v menší množině.

- Nabídnuté opravy jistě povedou k nějakým výsledkům. Pokud bychom použili množinu všech slov jazyka, vyhledávač by mohl nabídnout slovo, které je sice platné, ale v databázi se nevyskytuje. Pokud se omezíme je na slova, která se reálně vyskytují ve zpracovaných dokumentech, povede nabídnutá oprava vždy k nějakému výsledku.

Nyní potřebujeme algoritmus, který dokáže porovnat dvě slova a vrátit hodnotu reprezentující jejich podobnost. Jedním z klasických algoritmů je Levenstheinova vzdálenost, který počítá počet překlepů mezi dvěma slovy. Takže pro slova „kotel“ a „hotel“ vrátí hodnotu 1, protože se liší pouze v prvním písmenu.

CLaSeek používá složitější algoritmus, který je založen na myšlence hledání největších společných subsekvencí. V prvním kroku nalezneme nejdelší společný řetězec mezi dvěma slovy. Uložíme si indexy, na kterých řetězec v jednotlivých slovech začíná a délku tohoto řetězce. Poté aplikujeme stejný postup na zbylé kousky řetězce. Nakonec dostaneme seznam všech společných řetězců, pomocí nichž můžeme vypočítat stupeň podobnosti, což je číslo z intervalu $[0, 1]$. CLaSeek na to používá funkci ze standardní knihovny Pythonu, viz [1].

### 1.6.4. Doplňující informace

Kromě samotného indexu a slovníku na překlad stemů zpět na slova si potřebujeme uchovávat ještě některé další informace. Jmenovitě tyto:

- Seznam všech atributů všech dokumentů v sadě spolu s jejich tf-idf skórem vzhledem k danému dokumentu. V dalších algoritmech budeme potřebovat vědět, zda dokument $d$ obsahuje atribut $a$, případně i jaké je jeho tf-idf$_{a,d}$ skóre.

  Na rychlosti získání těchto informací už bude přímo záležet rychlost celého algoritmu, takže by čtení a následné získání dat mělo být rychlé.

- Meta data ke všem dokumentům. Jedná se o:
  - Název dokumentu.
  - Popisek dokumentu.
  - URL dokumentu.
  - Atributy dokumentu spolu s jejich tf-idf skórem.
  - Číselný identifikátor dokumentu.
  - Počet slov v dokumentu.

## 1.7. Odpovídání na dotazy

Hlavním účelem vyhledávače je samozřejmě odpovídání na dotazy. Uživatel vloží do rozhraní vyhledávače svůj dotaz, který je dále zpracován vyhledávačem, který vrátí nějaký seznam výsledků. Celý princip fungování boolean dotazů je popsán v [15], kapitola 1.

### 1.7.1. Syntax dotazu

Dotazem může být libovolný text, přičemž může obsahovat jisté řídicí příkazy, kterým říkáme operátory. CLaSeek podporuje tři operátory: AND, OR a NOT. Pokud se tato tři slova objeví v dotazu, budou vyhledávačem pochopena jako operátory, nikoli jako obyčejná text. Musí být psány velkými písmeny, takže pokud chce uživatel použít daná slova jako obyčejný text, stačí použít jejich variantu s malými písmeny.

Během vyhodnocování dotazu budou na dotaz aplikovány stejné postupy na úpravy textu, jaké byly popsány v kapitole 1.3. Dojde tak k odstranění interpunkce, diakritiky a stop slov, text bude převeden na malá písmena a podobně. Dotaz „Hele, jak se máš?!“ tak bude vyhodnocen stejně jako dotaz „hele mas“, pokud budou slova „jak“ a „se“ v množině stop slov.

Pokud nevložíme do textu žádný operátor, použije se implicitně mezi každým slovem AND operátor. To znamená, že pokud hledáme „analýza dat“, vyhledávač bude vidět dotaz jako „analýza AND dat“ a vyhledá takové dokumenty, které obsahují slovo „analýza“ a zároveň obsahují slovo „dat“, respektive jejich stemy.

Spojíme-li dvě slova operátorem OR, budou se hledat dokumenty, které obsahují alespoň jedno z těchto slov. Hledáme-li „spočetné OR nespočetné“, vyhledávač bude hledat dokumenty, které obsahují slovo „spočetné“ nebo „nespočetné“.

Operátor NOT vylučuje ty dokumenty, které obsahují slovo, které se nachází za NOT. Hledáme-li „spočetné NOT nespočetné“, bude vyhledávač hledat ty dokumenty, které obsahují slovo „spočetné“, ale neobsahují slovo „nespočetné“.

Tyto operátory můžeme různě kombinovat a s pomocí závorek můžeme tvořit složitější dotazy, například „(spočetné OR nespočetné) množiny NOT (komplexní OR přirozená)“.

Každý operátor má jinou prioritu. Seřazeno od nejmenší priority: OR, AND, NOT. Znamená to, že dotaz „NOT a AND b OR c“ je ekvivalentní s dotazem „((NOT a) AND b) OR c“.

### 1.7.2. Parsování dotazu

Samotný dotaz zapsaný uživatelem musí CLaSeek přeložit do nějaké interní formy, které bude rozumět. K tomu využívá jednoduchý parser.

V prvním kroku převedeme holý text na lexémy. Lexém je nějaká základní jednotka dotazovacího jazyka. V našem případě může být lexém slovo, operátor nebo závorky. Takže lexém je například slovo „matematika“, operátor „AND“ nebo otevírací závorka „(“. Lexém není „matematika)“, protože na konci slova máme závorku – tento výraz by měl být rozdělen do dvou lexémů. Po aplikaci této části tak dostáváme seznam lexémů.

V tuto chvíli budeme jednotlivé lexémy převádět na syntaktický strom, což je stromová struktura, která obsahuje v každém uzlu informaci o typu uzlu a obsahuje odkaz na nula až dva potomky.

Typem může být buď slovo nebo operátor. V případě, že je uzel operátorem, uložíme si ještě, o jaký operátor se jedná. V případě, že je uzel slovem, uložíme si do uzlu samotné slovo. Pokud jde o binární operátor, tj. AND nebo OR, musí obsahovat ještě levého a pravého potomka. Tam může být uložen opět uzel libovolného typu. Pokud jde o uzel typu NOT, tak musí mít právě jednoho potomka, který může být libovolného typu.

Například dotazu „(matematika AND (spočetné OR nespočetné)) OR (NOT (množiny AND funkce))“ by odpovídal syntaktický strom zobrazený na obrázku 1.

Nakonec celý výraz zjednodušíme – odstraníme uzly, které obsahují stop slova, protože stop slova nepoužíváme při vyhledávání a převedeme slova na stemy.

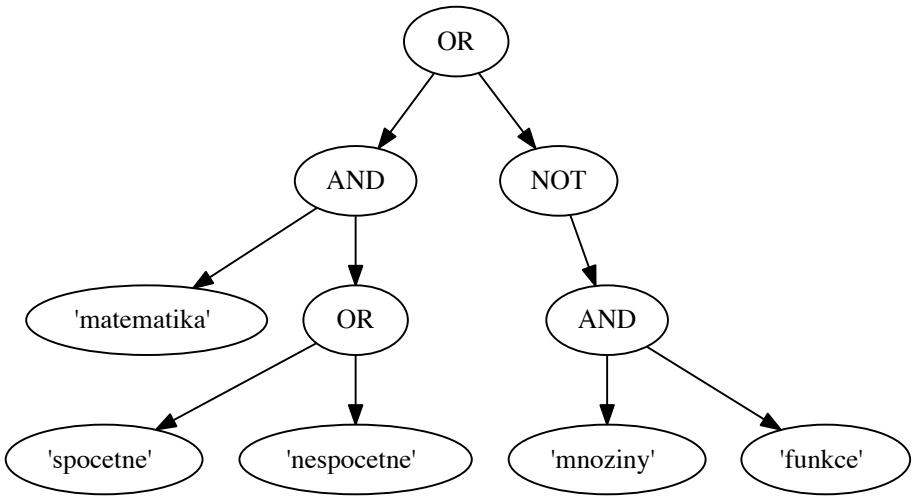
Více informací o parsování dotazu je například v [2].

### 1.7.3. Vyhledání odpovídajících dokumentů

Ve chvíli, kdy máme syntaktický strom, se můžeme pustit do hledání dokumentů. K tomu využijeme vybudovaný index, který nám umožňuje snadno zjistit, v jakých dokumentech se vyskytuje dané slovo. Pro účely popisu mechanismu si definujeme funkci R, která bere na vstupu dotaz (syntaktický strom) $Q$ a na výstupu vrací množinu dokumentů, které odpovídají zadanému dotazu. Tato funkce se bude chovat odlišně v závislosti na tom, jak vypadá dotaz $Q$.

Dále označme $\mathbb{D}$ množinu všech dokumentů a $\alpha$ a $\beta$ nechť označují nějaké dotazy.

- Je-li $Q = s$, kde $s$ je nějaký stem, pak funkce R vrátí množinu dokumentů, které obsahují daný stem, což vyčteme z indexu. Formálně to můžeme za-




Obrázek 1. Syntaktický strom

psat jako:

$$\mathrm{R}_s = \{d \in \mathbb{D} \,|\, s \in d\}\,.$$

- Je-li dotaz ve tvaru Q = „$\alpha$ AND $\beta$“, pak $\mathrm{R}_Q = \mathrm{R}_\alpha \cap \mathrm{R}_\beta$.
- Je-li dotaz ve tvaru Q = „$\alpha$ OR $\beta$“, pak $\mathrm{R}_Q = \mathrm{R}_\alpha \cup \mathrm{R}_\beta$.
- Je-li dotaz ve tvaru Q = „NOT $\alpha$“, pak $\mathrm{R}_Q = \mathbb{D} \setminus \mathrm{R}_\alpha$.

Postupnou aplikací těchto pravidel dostaneme množinu výsledných dokumentů $R$. V dalším kroku tyto dokumenty seřadíme podle relevance.

### 1.7.4. Seřazení dokumentů

V současné chvíli máme množinu dokumentů $\mathrm{R}_Q$. Aby byl vyhledávač smysluplný, měl by tyto dokumenty seřadit podle toho, jak relevantní dané dokumenty vzhledem k položenému dotazu jsou. To je obecně nelehký úkol. Ve vyhledávači je pak použit standardní tf-idf algoritmus popsaný v části 1.5.2.

Abychom mohli ohodnotit jednotlivé dokumenty, potřebujeme znát slova, vůči kterým máme dokumenty ohodnocovat. Odstraníme tak z dotazu všechny operátory a dostaneme množinu všech slov $S$. Vůči těmto slovům budeme dokumenty hodnotit.

K tomu už využijeme tf-idf algoritmus – pro každé slovo z množiny $S$ a pro každý dokument z množiny $\mathrm{R}_Q$ spočítáme jeho tf-idf skóre; poté tato skóre sečteme a dokumenty seřadíme sestupně podle tohoto skóre. Skóre $\mathrm{sc}_{d,S}$ dokumentu

```
1: function RANK(d, S)
2:     score ← sc_{d,S}
3:     for all s ∈ S do
4:         if s ∈ title(d)  ∨  s ∈ url(d) then
5:             score ← score · 3
6:         end if
7:         if s ∈ description(d) then
8:             score ← score · 2
9:         end if
10:    end for
11:    return score
12: end function
```

$d$ tak udává vzorec:

$$\mathrm{sc}_{d,S} = \sum_{s \in S} \text{tf-idf}_{s,d}\,.$$

Toto základní skóre ještě dále upravíme podle toho, zda se klíčová slova z dotazu nevyskytují v důležitých částech dokumentu, konkrétně jde o název, adresu a popisek stránky. Za každé klíčové slovo, které se vyskytuje v názvu stránky $d$ nebo v adrese stránky, ztrojnásobíme hodnotu $\mathrm{sc}_{d,S}$. Za každé klíčové slovo, které se vyskytuje v popisu stránky $d$, zdvojnásobíme hodnotu $\mathrm{sc}_{d,S}$. Ve výsledku tak dostáváme funkci rank, která nejprve vypočte hodnotu $\mathrm{sc}_{d,S}$ a poté aplikuje pravidla o ztrojnásobení a zdvojnásobení. Funkce rank bere na vstupu dva parametry: dokument $d$, pro který počítáme skóre a slova $S$ z dotazu $Q$.

Nyní seřadíme dokumenty $\mathrm{R}_Q$ do $n$-tice $\langle d_1, d_2, \ldots, d_n \rangle$, kde

$$n = |\mathrm{R}_Q| \quad \text{a} \quad \bigcup_{i=1}^{n} d_i = \mathrm{R}_Q$$

tak, aby platilo $\mathrm{rank}_{d_1,S} \geq \mathrm{rank}_{d_2,S} \geq \ldots \geq \mathrm{rank}_{d_n,S}$. Tato seřazená $n$-tice je výstupem algoritmu vyhledávání.

Idea a postup řazení dokumentů je popsán v [15], včetně zvýhodňování určitých částí dokumentu.

# 2. Formální konceptuální analýza

Formální konceptuální analýzu budeme používat jako matematický základ pro hledání souvisejících dotazů. V této kapitole bude popsána nezbytná teorie FCA.

Základní teorie kolem FCA byla poprvé představena roku 1982 v článku „Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts“, jehož autorem je německý matematik Rudolf Wille, viz [16].

## 2.1. Neformální úvod do formální konceptuální analýzy

Formální konceptuální analýza pracuje s tabulkovými daty. V řádcích máme objekty, ve sloupcích máme atributy objektů. Obsahem každé buňky je pak informace, zda daný objekt má daný atribut. Tato informace je binární, daný objekt buď daný atribut má, nebo nemá.

V tabulce 1. je příklad takových tabulkových dat. V řádcích se nachází objekty – živočichové či rostliny. Ve sloupcích pak jednotlivé atributy, což jsou nějaké vlastnosti, které dané objekty mohou mít. Pod tabulkou je legenda k jednotlivým atributům. Například první atribut $a$ je „potřebuje k životu vodu“. Každý živočich a každá rostlina potřebuje k životu vodu, takže v celém sloupečku je křížek. Druhý atribut $b$ je „žije ve vodě“. Zde už není křížek všude, takový pes ve vodě nežije.

Takové tabulce s daty říkáme *kontext*. V tomto kontextu pak hledáme shluky potenciálně zajímavých dat. Shlukem může být například množina $\{\text{Cejn, Žába, Pes}\}$, protože všechna tato zvířata sdílí společné atributy $\{a, g, h\}$ – potřebují vodu k životu, mohou se pohybovat a mají končetiny. Zároveň platí, že cejn, žába a pes žádné další společné atributy nesdílí a podobně u atributů: žádný další objekt už nemá právě tyto tři společné vlastnosti. Takové dvě množiny objektů a atributů pak tvoří *koncept*. Množinu objektů konceptu nazýváme *extent* a množinu atributů *intent*.

Když se podíváme na tabulku, zjistíme, že koncept je tvořený křížky, které dohromady tvoří „rozházený obdélník“. Pokud bychom ale sloupce, které reprezentují množinu společných atributů, přesunuly k sobě, získali bychom hezký obdélník (v tomto případě dokonce čtverec). Koncept je zvýrazněný v tabulce 2. Vždy nás zajímá maximální obdélník, tj. pokud existuje objekt nebo atribut, který můžeme přidat a opět získáme obdélník, přidáme ho.

To samozřejmě není jediný koncept, který kontext obsahuje. Ve skutečnosti tento kontext obsahuje 19 konceptů. Zkusíme si najít další. Existuje něco, co žije na zemi i ve vodě? Hledáme tak objekty, které mají atributy $\{b, c\}$. Pohledem do tabulky zjistíme, že tyto vlastnosti má žába a rákosí. Nicméně ještě jsme nenalezli koncept, protože nevíme, jestli náhodou tyto objekty nesdílí ještě nějaký další atribut. Vzhledem k tomu, že všechny objekty mají atribut $a$, musíme ho přidat do množiny atributů. Tím už získáme koncept s objekty {žába, rákosí} a atributy $\{a, b, c\}$.

| | a | b | c | d | e | f | g | h | i |
|---|---|---|---|---|---|---|---|---|---|
| Pijavice | × | × | | | | | × | | |
| Cejn | × | × | | | | | × | × | |
| Žába | × | × | × | | | | × | × | |
| Pes | × | | × | | | | × | × | × |
| Bodlák | × | × | | × | | × | | | |
| Rákosí | × | × | × | × | | × | | | |
| Fazole | × | | × | × | × | | | | |
| Kukuřice | × | | × | × | | × | | | |

Tabulka 1. Formální kontext, zdroj: [4]

$a$: potřebuje k životu vodu, $b$: žije ve vodě, $c$: žije na zemi,
$d$: potřebuje chlorofyl, $e$: dvouděložná rostlina, $f$: jednoděložná rostlina,
$g$: může se pohybovat, $h$: má končetiny, $i$: kojí potomky

| | b | c | d | e | f | g | h | a | i |
|---|---|---|---|---|---|---|---|---|---|
| Pijavice | × | | | | | × | | × | |
| Cejn | × | | | | | × | × | × | |
| Žába | × | × | | | | × | × | × | |
| Pes | | × | | | | × | × | × | × |
| Bodlák | × | | × | | × | | | × | |
| Rákosí | × | × | × | | × | | | × | |
| Fazole | | × | × | × | | | | × | |
| Kukuřice | | × | × | | × | | | × | |

Tabulka 2. Formální kontext se zvýrazněným konceptem

Řekli jsme, že formální konceptuální analýza zároveň řeší i hierarchii shluků. Ta se řeší pomocí relace inkluze. Můžeme říci, že koncept, který se skládá z extentu {pijavice, cejn, žába a pes} a intentu $\{a, g\}$ je obecnější než koncept {cejn, žába a pes} a $\{a, g, h\}$.

První koncept můžeme pojmenovat „pohybující se živočichové“, druhý koncept by byl „živočichové, kteří k pohybu využívají končetiny“. Druhý koncept obsahuje všechny atributy jako první koncept, plus jeden navíc – ten nějakým způsobem konkretizuje tento koncept. U objektů to platí naopak. Druhý koncept má více atributů, ale méně objektů. To je logické, pokud zvýšíme počet atributů, které objekty musí mít, zvýšíme tím na objekty nároky a objektů ubude.

Tím už se blížíme k definici uspořádání konceptů. Pokud je nějaký koncept konkrétnější než jiný, řekneme, že je menší. Koncept $A$ je konkrétnější než koncept $B$, zapíšeme $A < B$, pokud je extent konceptu $A$ vlastní podmnožinou extentu konceptu $B$. Tím máme definováno uspořádání, kterým můžeme uspořádat všechny koncepty. Přesnější popis bude v následující kapitole.

## 2.2. Formální zavedení FCA

Všechny definice a všechny věty z této kapitoly jsou citovány z [4]. Další materiály o FCA: [12], [6], [7], [8].

### 2.2.1. Základní pojmy

**Definice 1** (Formální kontext)**.** *Formální kontext je trojice $\langle X, Y, I\rangle$, kde $X$ je neprázdná množina objektů, $Y$ je neprázdná množina atributů a $I$ je binární relace mezi $X$ a $Y$, tj. $I \subseteq X \times Y$.*

V předchozím příkladě byla množina $X$ rovna živočichům a rostlinám a množina $Y$ byla rovna vlastnostem těchto živočichů a rostlin. Relace $I$ pak vyznačuje, zda má objekt $x$ atribut $y$. Objekt $x$ má atribut $y$ právě tehdy, když $\langle x, y\rangle \in I$. Formální kontext, jak jsme viděli, může být reprezentován pomocí tabulky.

**Definice 2** (Šipkové operátory)**.** *Pro kontext $\langle X, Y, I\rangle$ definujeme operátory $^{\uparrow} : 2^X \to 2^Y$ a $^{\downarrow} : 2^Y \to 2^X$ tak, že pro každé $A \subseteq X$ a $B \subseteq Y$:*

$$A^{\uparrow} = \{y \in Y \mid \textit{pro všechna } x \in A : \langle x, y\rangle \in I\} \quad (1)$$

$$B^{\downarrow} = \{x \in X \mid \textit{pro všechna } y \in B : \langle x, y\rangle \in I\} \quad (2)$$

Tyto operátory umožňují zjistit společné vlastnosti. Operátor $^{\uparrow}$ nám zjistí všechny atributy, které mají všechny objekty v $A$ společné. Operátor $^{\downarrow}$ nám zjistí všechny objekty, které sdílí všechny atributy z $B$.

Pokud se vrátíme ke kontextu v tabulce 1., tak:

$$
\begin{aligned}
\{\text{Žába}\}^{\uparrow} &= \{a, b, c, g, h\} \\
\{\text{Fazole, Bodlák}\}^{\uparrow} &= \{a, d\} \\
\{\text{Pijavice, Pes, Kukuřice}\}^{\uparrow} &= \{a\} \\
X^{\uparrow} &= \{a\} \\
\{c, d\}^{\downarrow} &= \{\text{Rákosí, Fazole, Kukuřice}\} \\
\{f, g\}^{\downarrow} &= \emptyset \\
\{b\}^{\downarrow} &= \{\text{Pijavice, Cejn, Žába, Bodlák, Rákosí}\}
\end{aligned}
$$

**Definice 3** (Formální koncept)**.** *Formální koncept v kontextu $\langle X, Y, I \rangle$ je dvojice $\langle A, B \rangle$, $A \subseteq X$, $B \subseteq Y$ tak, že:*

$$A^{\uparrow} = B \wedge B^{\downarrow} = A.$$

*Množině $A$ říkáme „extent“ a množině $B$ „intent“.*

Takto definovaný formální koncept odpovídá naší předchozí představě o konceptu. Pokud jsme měli danou množinu objektů, pak jsme našli sdílené atributy. K těmto sdíleným atributům jsme následně nalezli objekty, které všechny tyto atributy sdílí, což mohlo být více objektů, než jaké jsme měli na začátku. Ve chvíli, kdy se k těmto objektům pokusíme najít sdílené atributy, nenalezneme již žádný nový.

V tabulce 1. tak můžeme najít například koncept

$$\langle \{\text{Cejn, Žába}\}, \{a, b, g, h\} \rangle,$$

protože

$$\{\text{Cejn, Žába}\}^{\uparrow} = \{a, b, g, h\} \quad \wedge \quad \{a, b, g, h\}^{\downarrow} = \{\text{Cejn, Žába}\}.$$

### 2.2.2. Galoisova spojení

Évariste Galois byl francouzský matematik 19. století. Je známý svou prací v oblasti moderní algebry a považuje se za zakladatele teorie grup. Bohužel zemřel již ve svých dvaceti letech, když se účastnil střeleckého souboje. My využijeme Galoisovo spojení jako základní matematickou strukturu pro FCA.

**Definice 4** (Galoisovo spojení)**.** *Galoisovo spojení mezi množinami $X$ a $Y$ je dvojice zobrazení $\langle f, g \rangle$, $f : 2^X \to 2^Y$ a $g : 2^Y \to 2^X$, která pro všechna $A, A_1, A_2 \subseteq X$ a $B, B_1, B_2 \subseteq Y$ splňují:*

$$A_1 \subseteq A_2 \Rightarrow f(A_2) \subseteq f(A_1) \tag{3}$$
$$B_1 \subseteq B_2 \Rightarrow g(B_2) \subseteq g(B_1) \tag{4}$$
$$A \subseteq g(f(A)) \tag{5}$$
$$B \subseteq f(g(B)) \tag{6}$$

Galoisovo spojení má jednoduchou interpretaci, kterou si můžeme ukázat na příkladu herců a filmů. Nechť množina $X$ je množina všech herců a hereček a množina $Y$ všech filmů.

Zobrazení $f$ nám pro danou podmnožinu herců $A$ vrací množinu filmů, ve kterých všichni herci z množiny $A$ hráli. Zobrazení $g$ naopak vrací pro podmnožinu filmů $B$ takovou množinu herců, kteří hráli ve všech filmech v množině $B$. Výsledkem $f(\{\text{Bruce Willis}\})$ je množina všech filmů, ve kterých hrál Bruce Willis. Výsledkem $g(\{\text{Pelíšky; Dobrá zpráva; Pasti, pasti, pastičky}\})$ je množina herců a hereček, kteří hráli ve všech těchto filmech: {Miroslav Donutil, Eva Holubová}.

Vlastnosti zmíněné v definici 4 už jsou pak velmi přirozené. Vlastnost 3 říká, že pokud máme skupiny herců $A_1 = \{\text{Johnny Depp}\}$ a $A_2 = \{\text{Johnny Depp, Bruce Lee}\}$, tak můžeme očekávat, že pouze Depp hrál ve více filmech než Depp spolu s Leem. Filmy, ve kterých spolu hráli Depp a Lee jsou určitě podmnožinou filmů, kde hrál pouze Depp.

Vlastnost 5 zase říká, že pokud nalezneme všechny filmy, ve kterých hrál Johnny Depp a poté nalezneme všechny herce, kteří hráli ve všech těchto filmech, tak mezi těmito herci musí být Johnny Depp. Ostatní dvě vlastnosti říkají duálně totéž.

**Definice 5** (Pevné body Galoisova spojení)**.** *Pro Galoisovo spojení $\langle f, g\rangle$ mezi množinami $X$ a $Y$ definujeme množinu*

$$\operatorname{fix}(\langle f, g\rangle) = \left\{\langle A, B\rangle \in 2^X \times 2^Y \mid f(A) = B, g(B) = A\right\}.$$

*Tuto množinu nazýváme množinu pevných bodů spojení $\langle f, g\rangle$.*

Pevným bodem je například dvojice

$$\begin{aligned} A &= \{\text{Miroslav Donutil, Eva Holubová}\}, \\ B &= \{\text{Pelíšky; Dobrá zpráva; Pasti, pasti, pastičky}\}. \end{aligned}$$

Donutil s Holubovou spolu nehráli v žádném dalším filmu, kromě těchto tří. A zároveň v těchto třech filmech spolu nehráli žádní jiní herci než Donutil a Holubová.

**Věta 6** (Zřetězení Galoisových spojení)**.** *Pro Galoisovo spojení $\langle f, g\rangle$ mezi $X$ a $Y$ a pro všechna $A \subseteq X$ a $B \subseteq Y$ platí:*

$$f(A) = f(g(f(A))) \tag{7}$$
$$g(B) = g(f(g(B))) \tag{8}$$

□

Toto je zajímavá vlastnost. Na začátku máme množinu herců $A$. Nalezneme filmy, ve kterých tito herci hráli. Tím získáme $f(A)$. Dále nalezneme herce, kteří hráli ve všech těchto nalezených filmech. Tam se určitě objeví herci z $A$, viz definici 4, ale mohou se tam objevit i další herci. Tím získáme hodnotu $g(f(A))$. Nakonec zpátky spočítáme filmy, ve kterých tito herci hráli. Musíme nutně získat množinu filmů $f(A)$, protože $g(f(A))$ jsou právě herci, kteří hrají všech filmech z $f(A)$.

**Definice 7** (Uzávěrový operátor)**.** *Uzávěrový operátor $C$ na množině $X$ je zobrazení $C : 2^X \to 2^X$, které pro každé $A, A_1, A_2 \subseteq X$ splňuje:*

$$A \subseteq C(A), \tag{9}$$
$$A_1 \subseteq A_2 \Rightarrow C(A_1) \subseteq C(A_2), \tag{10}$$
$$C(A) = C(C(A)). \tag{11}$$

**Definice 8** (Pevné body uzávěrového operátoru)**.** *Pro uzávěrový operátor $C : 2^X \to 2^X$ pojmenujeme množinu*

$$\mathrm{fix}(C) = \{A \subseteq X \mid C(A) = A\}$$

*jako množinu pevných bodů uzávěrového operátoru $C$.*

**Věta 9** (Od Galoisova spojení k uzávěrovému operátoru)**.** *Nechť $\langle f, g\rangle$ je Galoisovo spojení mezi $X$ a $Y$. Potom $C_X = f \circ g$ je uzávěrový operátor na $X$ a $C_Y = g \circ f$ je uzávěrový operátor na $Y$.* □

Zápis $f \circ g$ značí skládání množin ve smyslu $(f \circ g)(x) = g(f(x))$.

Pevným bodem uzávěrového operátoru nad množinou herců rozumíme takovou množinu herců, kteří spolu hráli v určitých filmech, ale *žádní jiní herci už v těchto filmech nehráli*. Poslední vlastnost je pro pevný bod klíčová.

Je například množina {Danny Glover, Mel Gibson} pevným bodem? Věta 9 nám říká, jak postupovat, pokud pevný bod chceme nalézt. Jako první vypočteme $f(\{\text{Danny Glover, Mel Gibson}\})$. Tím získáme množinu filmů, ve kterých herci hráli spolu, tj. {Smrtonosná zbraň 1, 2, 3, 4}. Nyní zpět zjistíme herce, kteří hráli v těchto filmech, spočítáme $g(\{\text{Smrtonosná zbraň 1, 2, 3, 4}\})$. Kromě Gibsona a Glovera se tam objeví ještě Mary Ellen Trainor: {Danny Glover, Mel Gibson, Mary Ellen Trainor}. Tato množina herců a hereček je už pevným bodem.

Uzávěrový operátor si můžeme ukázat i na úplně jiném příkladě, doplnění rovinného útvaru na nejmenší konvexní útvar. Máme-li tak nějaký útvar v rovině, který není konvexní, například klasický obrázek hvězdy, uzávěrový operátor ho doplní tak, aby z něj vznikl konvexní útvar a aby tento nový útvar byl nejmenší. Na tomto operátoru si můžeme ilustrovat podmínky v definici 7.

První vlastnost říká, že původní útvar bude vždy podmnožinou uzavřeného útvaru. Původní rovinný útvar bude vždy celý obsažen v konvexním útvaru, který

ho má obalovat – uzavřením pouze „přidáme“ původnímu útvaru body tak, aby vznikl konvexní útvar. Druhá vlastnost říká, že pokud máme dva různé útvary, přičemž jeden je zanořený ve druhém, tak i jejich obaly budou zanořené. Poslední vlastnost říká, že obalem obalu je tentýž obal. Pokud obalíme hvězdu konvexním útvarem, pro příklad pravidelným pětiúhelníkem, tak obalem tohoto pětiúhelníku bude tentýž pětiúhelník.

**Věta 10.** *Nechť $\langle f, g\rangle$ je Galoisovo spojení mezi $X$ a $Y$. Pak pro každé $A_j \subseteq X, j \in J$ a $B_j \subseteq Y, j \in J$ platí:*

$$f(\bigcup_{j\in J} A_j) = \bigcap_{j\in J} f(A_j) \tag{12}$$

$$g(\bigcup_{j\in J} B_j) = \bigcap_{j\in J} g(B_j) \tag{13}$$

□

**Věta 11** (Šipkové operátory tvoří Galoisovo spojení)**.** *Pro formální kontext $\langle X, Y, I\rangle$ tvoří dvojice $\left\langle {}^{\uparrow}, {}^{\downarrow} \right\rangle$ Galoisovo spojení mezi $X$ a $Y$.* □

Předchozí věta je důležitým výsledkem. Odpovídá nám na otázku, jak jednoduše počítat extenty a intenty k libovolným podmnožinám objektů či atributů. Protože pro kontext $\langle X, Y, I\rangle$ je $\left\langle {}^{\uparrow}, {}^{\downarrow} \right\rangle$ Galoisovo spojení, platí, že zobrazení ${}^{\uparrow}\circ{}^{\downarrow}$ je uzávěrový operátor na množině $X$, tj. na množině objektů.

Máme-li množinu objektů $A \subseteq X$, pak uzavřením $A^{\uparrow\downarrow}$ získáme pevný bod operátoru, což je zároveň extent nějakého konceptu. Podobně pokud máme množinu atributů $B \subseteq Y$, pak uzavřením $B^{\downarrow\uparrow}$ získáme intent nějakého konceptu.

Pokud si spočítáme extent jako $A^{\uparrow\downarrow}$, můžeme intent tohoto konceptu zjistit tak, že ještě jednou spočítáme ${}^{\uparrow}$, dostaneme tak množinu $A^{\uparrow\downarrow\uparrow}$. Nicméně z věty 6 víme, že $A^{\uparrow} = A^{\uparrow\downarrow\uparrow}$, takže intent je zároveň rovný $A^{\uparrow}$. Jakýkoliv pár tvaru $\left\langle A^{\uparrow\downarrow}, A^{\uparrow}\right\rangle$, kde $A \subseteq X$, tvoří formální koncept.

Můžeme si také označit množinu všech extentů jako Ext a množinu všech intentů jako Int:

$$\mathrm{Ext}(X, Y, I) = \mathrm{fix}({}^{\uparrow\downarrow}), \tag{14}$$

$$\mathrm{Int}(X, Y, I) = \mathrm{fix}({}^{\downarrow\uparrow}). \tag{15}$$

Dále si označíme množinu všech konceptů kontextu $\langle X, Y, I\rangle$:

$$\mathcal{B}(X, Y, I) = \left\{\left\langle A, A^{\uparrow}\right\rangle \mid A \in \mathrm{Ext}(X, Y, I)\right\}, \tag{16}$$

$$\mathcal{B}(X, Y, I) = \left\{\left\langle B^{\downarrow}, B\right\rangle \mid B \in \mathrm{Int}(X, Y, I)\right\}. \tag{17}$$

### 2.2.3. Konceptuální svaz

**Definice 12** (Uspořádání konceptů)**.** *O konceptech* $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle$ *kontextu* $\langle X, Y, I\rangle$ *řekneme, že*

$$\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle \quad \textit{právě když} \quad A_1 \subseteq A_2 \quad (B_2 \subseteq B_1).$$

Relaci menší nebo rovno mezi koncepty můžeme interpretovat jako relaci „být konkrétnější koncept“. Pokud pro koncepty platí $\langle A_1, B_1\rangle \leq \langle A_2, B_2\rangle$, pak koncept $\langle A_1, B_1\rangle$ je konkrétnější než koncept $\langle A_2, B_2\rangle$.

**Věta 13** (Extenty, intenty a formální koncepty)**.** *Některé důležité vlastnosti množin extentů, intentů a konceptů:*

1. $\langle \mathrm{Ext}(X, Y, I), \subseteq\rangle$ *a* $\langle \mathrm{Int}(X, Y, I), \subseteq\rangle$ *jsou (částečně) uspořádáné množiny.*
2. $\langle \mathrm{Ext}(X, Y, I), \subseteq\rangle$ *a* $\langle \mathrm{Int}(X, Y, I), \subseteq\rangle$ *jsou duálně isomorfní. Tj. existuje zobrazení* $f : \mathrm{Ext}(X, Y, I) \to \mathrm{Int}(X, Y, I)$ *takové, že* $A_1 \subseteq A_2$ *platí právě tehdy, když* $f(A_2) \subseteq f(A_1)$*.*
3. $\langle \mathcal{B}(X, Y, I), \leq\rangle$ *je isomorfní s* $\langle \mathrm{Ext}(X, Y, I), \subseteq\rangle$*.*

□

**Definice 14** (Svaz)**.** *Nechť* $\langle X, \leq\rangle$ *je uspořádaná množina. Pokud pro každé* $x, y \in X$ *existuje* $\sup(x, y)$ *a* $\inf(x, y)$*, pak* $\langle X, \leq\rangle$ *nazveme svaz. Pokud můžeme nalézt infima a suprema pro jakoukoliv podmnožinu množiny* $X$*, pak se jedná o úplný svaz.*

Uspořádaná množina tak je svazem, pokud dokážeme ke každým dvěma prvkům množiny nalézt jejich infimum a supremum. Příkladem jednoduchého svazu mohou být reálná čísla s klasickým uspořádáním. Pak platí, že pro $x, y \in \mathbb{R}$: $\sup(x, y) = \max(x, y)$ a $\inf(x, y) = \min(x, y)$.

Dalším příkladem svazu je $\langle 2^{\mathbb{N}}, \subseteq\rangle$, množina všech podmnožin přirozených čísel s uspořádáním podle množinové inkluze. Platí pak, že pro $X, Y \in 2^{\mathbb{N}}$ máme $\sup(X, Y) = X \cup Y$ a $\inf(X, Y) = X \cap Y$.

**Definice 15** (Konceptuální svaz)**.** *Pro kontext* $\langle X, Y, I\rangle$ *máme definovanou množinu všech formálních konceptů* $\mathcal{B}(X, Y, I)$ *z* $\langle X, Y, I\rangle$*. Tedy*

$$\mathcal{B}(X, Y, I) = \left\{\langle A, B\rangle \in 2^X \times 2^Y \mid A^{\uparrow} = B \wedge B^{\downarrow} = A\right\}.$$

*Dvojice* $\langle \mathcal{B}(X, Y, I), \leq\rangle$ *se nazývá konceptuální svaz.*

K tomu, abychom dokázali, že $\langle \mathcal{B}(X, Y, I), \leq\rangle$ je skutečně svaz, musíme dokázat, že pro každé dva koncepty $c_1, c_2 \in \langle \mathcal{B}(X, Y, I), \leq\rangle$ existuje v tomto svazu jejich supremum a infimum. Vrátíme se ještě k obecnému uzávěrovému operátoru.

**Věta 16** (Systém pevných bodů uzávěrového operátoru)**.** *Mějme uzávěrový operátor $C$ na množině $X$. Pak uspořádaná množina $\langle \mathrm{fix}(C), \leq \rangle$ pevných bodů tohoto operátoru je úplný svaz, kde infima a suprema získáme*

$$\bigwedge_{j \in J} A_j = \bigcap_{j \in J} A_j, \tag{18}$$

$$\bigvee_{j \in J} A_j = C(\bigcup_{j \in J} A_j). \tag{19}$$

□

Ukážeme si na příkladu myšlenku důkazu. Rovnice 18 říká, že pokud máme dva pevné body uzávěrového operátoru a provedeme jejich průnik, pak opět získáme pevný bod a tento bod bude zároveň infimem původních dvou bodů (rovnice je ve skutečnosti obecnější a povoluje libovolně velkou množinu pevných bodů, ne jen dva).

Vrátíme se k příkladu uzávěrového operátoru, který uzavíral rovinný útvar na nejmenší konvexní útvar. Představme si dva konvexní útvary, například dva kruhy, které se protínají. Jejich průnikem pak bude jistě konvexní útvar.

Zajímavější je v tomto případě rovnice 19, kde počítáme supremum. Očekávali bychom, že bude stačit pevné body pouze sjednotit a dostaneme opět pevný bod. Nicméně tomu tak není. Abychom dostali pevný bod, musíme sjednocení bodů ještě znovu uzavřít. Když se vrátíme ke kruhům – sjednotíme-li dva protínající se kruhy, získáme útvar, který není konvexní. Abychom získali konvexní útvar, musíme tyto dva sjednocenné útvary opět uzavřít.

Nyní bychom měli ukázat, že $\bigwedge_{j \in J} A_j$ je skutečně rovno $\bigcap_{j \in J} A_j$. Už víme, že $\bigcap_{j \in J} A_j$ je také pevný bod. Z principu platí, že $\bigcap_{j \in J} A_j$ je menší nebo rovno než všechna $A_j$. Pokud si dále vezmeme množinu všech $B \in \mathrm{fix}(C)$, která jsou menší než všechna $A_j$, tak platí, že $\bigcap_{j \in J} A_j$ je větší nebo rovno než všechna tato $B$. Je to tedy největší prvek spodní závory množiny $\{A_j\}$.

Pokud to vztáhneme na příklad: největší konvexní útvar, který je obsažen ve dvou rovinných útvarech, je právě průnik těchto dvou útvarů. Jakýkoliv větší útvar, nebo stejně velký, ale jinam umístěný, nebude obsažen v alespoň jednom z útvarů a jakýkoliv menší útvar bude ... menší.

Dále ukážeme, že $C(\bigcup_{j \in J} A_j)$ je skutečně rovno $\bigvee_{j \in J} A_j$. Víme, že $C(\bigcup_{j \in J} A_j)$ je pevný bod. Dále víme, že $C(\bigcup_{j \in J} A_j)$ je větší nebo roven než všechna $A_j$, takže musí být větší nebo roven než supremum $\bigvee_{j \in J} A_j$, tj. $\bigvee_{j \in J} A_j \subseteq C(\bigcup_{j \in J} A_j)$.

Dále platí, že $\bigvee_{j \in J} A_j \supseteq A_j$ pro všechna $j$ – supremum systému množin musí být větší nebo rovno než každá množina systému. Z toho vyplývá, že $\bigvee_{j \in J} A_j \supseteq \bigcup_{j \in J} A_j$. Pokud je supremum větší nebo rovno než každá množina, pak je větší nebo rovno než sjednocení systému. Využijeme faktu, že uzavřením pevného bodu získáme tentýž pevný bod: pokud $B \in \mathrm{fix}(C)$, pak $B = C(B)$. Protože $\bigvee_{j \in J} A_j$

je pevným bodem, můžeme napsat $\bigvee_{j\in J} A_j = C(\bigvee_{j\in J} A_j)$. Upravíme předchozí inkluzi:

$$\bigvee_{j\in J} A_j \supseteq \bigcup_{j\in J} A_j \tag{20}$$

$$C(\bigvee_{j\in J} A_j) \supseteq C(\bigcup_{j\in J} A_j) \tag{21}$$

$$\bigvee_{j\in J} A_j \supseteq C(\bigcup_{j\in J} A_j) \tag{22}$$

Nejprve oba výrazy uzavřeme, čímž nezměníme platnost inkluze. Poté z levé strany odstraníme $C$, protože supremum už pevným bodem je. Dostali jsme tak

$$\bigvee_{j\in J} A_j \supseteq C(\bigcup_{j\in J} A_j) \quad \text{a} \quad \bigvee_{j\in J} A_j \subseteq C(\bigcup_{j\in J} A_j)$$

Z toho vyplývá, že $\bigvee_{j\in J} A_j = C(\bigcup_{j\in J} A_j)$.

**Definice 17** (Supremálně a infimálně hustá množina)**.** *Množinu $K \subseteq V$ nazveme supremálně hustou ve $V$ právě tehdy, když pro všechna $v \in V$ existuje $K' \subseteq K$ takové, že $v = \bigvee K'$. Každý prvek množiny $V$ je tak supremem nějakých prvků z $K$.*

*Duálně pro infimum: když pro všechna $v \in V$ existuje $K' \subseteq K$ takové, že $v = \bigwedge K'$.*

**Věta 18** (Hlavní věta konceptuálních svazů)**.** *Věta má dvě části:*

1. *Nechť $\mathcal{B}(X, Y, I)$ je kompletní svaz, kde suprema a infima získáme*

$$\bigwedge_{j\in J} \langle A_j, B_j\rangle = \left\langle \bigcap_{j\in J} A_j, \left(\bigcup_{j\in J} B_j\right)^{\downarrow\uparrow} \right\rangle, \tag{23}$$

$$\bigvee_{j\in J} \langle A_j, B_j\rangle = \left\langle \left(\bigcup_{j\in J} A_j\right)^{\uparrow\downarrow}, \bigcap_{j\in J} B_j \right\rangle. \tag{24}$$

2. *Dále, libovolný úplný svaz $\boldsymbol{V} = \langle V, \leq\rangle$ je isomorfní k $\mathcal{B}(X, Y, I)$ právě tehdy, když existují zobrazení $\gamma : X \to V, \mu : Y \to V$ tak, že*

   - *$\gamma(X)$ je supremálně hustá množina ve $V$ a $\mu(Y)$ je infimálně hustá množina v $V$,*
   - *$\gamma(x) \leq \mu(y)$ právě tehdy, když $\langle x, y\rangle \in I$.*

□

$C_0 = \langle \{1,2,3,4,5,6,7,8\}, \{a\} \rangle$, $C_1 = \langle \{1,2,3,4\}, \{a,g\} \rangle$, $C_2 = \langle \{2,3,4\}, \{a,g,h\} \rangle$,
$C_3 = \langle \{5,6,7,8\}, \{a,d\} \rangle$, $C_4 = \langle \{5,6,8\}, \{a,d,f\} \rangle$, $C_5 = \langle \{3,4,6,7,8\}, \{a,c\} \rangle$,
$C_6 = \langle \{3,4\}, \{a,c,g,h\} \rangle$, $C_7 = \langle \{4\}, \{a,c,g,h,i\} \rangle$, $C_8 = \langle \{6,7,8\}, \{a,c,d\} \rangle$,
$C_9 = \langle \{6,8\}, \{a,c,d,f\} \rangle$, $C_{10} = \langle \{7\}, \{a,c,d,e\} \rangle$, $C_{11} = \langle \{1,2,3,5,6\}, \{a,b\} \rangle$,
$C_{12} = \langle \{1,2,3\}, \{a,b,g\} \rangle$, $C_{13} = \langle \{2,3\}, \{a,b,g,h\} \rangle$, $C_{14} = \langle \{5,6\}, \{a,b,d,f\} \rangle$,
$C_{15} = \langle \{3,6\}, \{a,b,c\} \rangle$, $C_{16} = \langle \{3\}, \{a,b,c,g,h\} \rangle$, $C_{17} = \langle \{6\}, \{a,b,c,d,f\} \rangle$,
$C_{18} = \langle \{\}, \{a,b,c,d,e,f,g,h,i\} \rangle$.

Obrázek 2. Všechny koncepty z kontextu v tabulce 1., zdroj: [4]

### 2.2.4. Zobrazení konceptuálního svazu

Už máme definované uspořádání na množině konceptů a víme, jak počítat suprema a infima, tak si ukážeme, jak takový konceptuální svaz vypadá. Vrátíme se k tabulce 1., kde byl zobrazen kontext s živočichy a rostlinami. Všechny koncepty v tomto kontextu jsou vypsány na obrázku 2.

Tento svaz můžeme zobrazit klasickým Hasseovým diagramem, viz obrázek 3.

Můžeme si na jednoduchých příkladech ukázat, jak se počítají infima a suprema. Vezměme si množinu konceptů $A = \{C_6, C_{15}\}$. Z obrázku můžeme vyčíst, že infimum $A$ je rovno $C_{16}$ a supremum $C_5$. Nejdříve si zapíšeme koncepty včetně všech objektů a atributů

$$\bigwedge \{\langle \{3,4\}, \{a,c,g,h\} \rangle, \langle \{3,6\}, \{a,b,c\} \rangle\}$$

a nyní spočítáme infimum množiny extentů těchto konceptů. Počítáme tak

$$\bigwedge \{\{3,4\}, \{3,6\}\} = \bigcap \{\{3,4\}, \{3,6\}\} = \{3\}$$

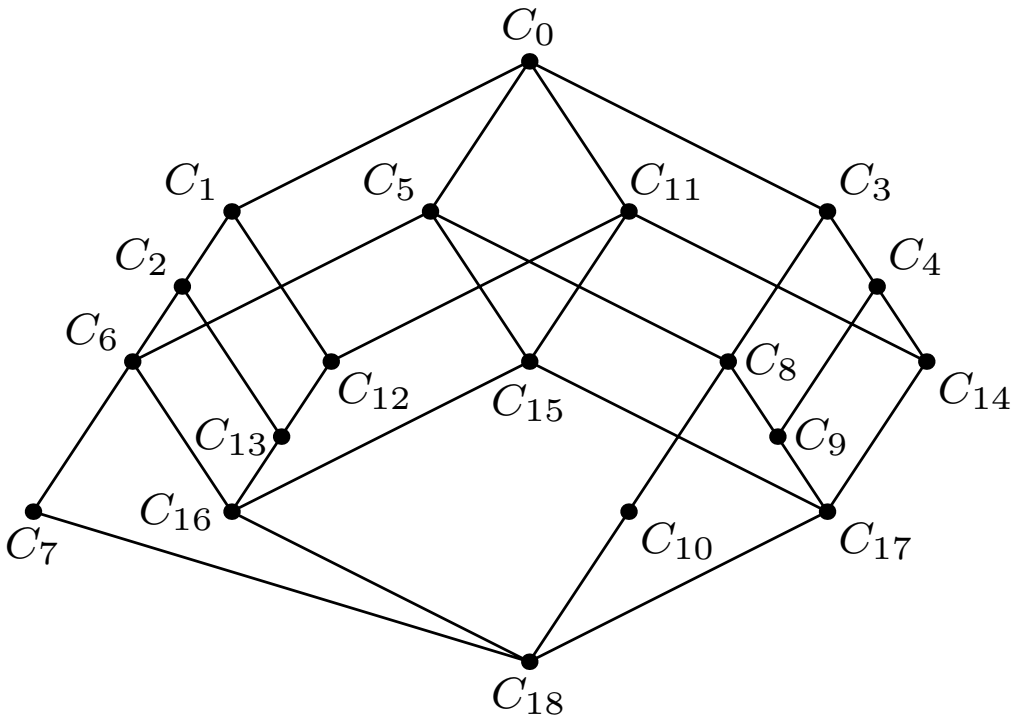
Extent infima bude obsahovat pouze objekt číslo tři. Intent spočítáme podobně, pouze použijeme supremum a jeden uzávěrový operátor navíc:

$$\bigvee \{\{a,c,g,h\}, \{a,b,c\}\} = \bigcup \{\{a,c,g,h\}, \{a,b,c\}\} = \{a,b,c,g,h\}$$

Tuto množinu ještě musíme uzavřít:

$$\{a,b,c,g,h\}^{\downarrow\uparrow} = \{a,b,c,g,h\}$$

Žádný další atribut už ale nepřibyl. Získali jsme tak koncept $\langle \{3\}, \{a,b,c,g,h\} \rangle$. Když se podíváme na výčet všech konceptů na obrázku 2., zjistíme, že to opravdu odpovídá konceptu $C_{16}$, jak jsme očekávali. Supremum spočítáme podobně.




Obrázek 3. Svaz, který vznikne z kontextu v tabulce 1., zdroj: [4]

# 3. Hledání souvisejících dokumentů

V této kapitole bude popsána druhá část vyhledávače, která se stará o nalezení souvisejících dotazů.

## 3.1. O co nám půjde

### 3.1.1. Motivace

Pokud uživatel položí vyhledávači nějaký dotaz, vyhledávač odpoví nějakým seznamem dokumentů, které jsou podle něj nejvíce relevantní. Pokud má uživatel štěstí, bude v tomto seznamu dokument, který zrovna potřebuje najít. Pokud bude mít velké štěstí, pak bude tento dokument na předních místech v seznamu.

Pokud ale toto štěstí mít nebude a dokument se mu nepodaří nalézt, musí uživatel nějakým způsobem přeformulovat svůj dotaz tak, aby vyhledávač vrátil jinou sadu výsledků. Obecně může upravit dotaz třemi různými způsoby. Může

1. přidat k dotazu jedno či více slov, díky čemuž obdrží méně výsledků,
2. změnit jedno či více slov, díky čemuž obdrží podobné výsledky,
3. odebrat jedno či více slov, díky čemuž obdrží více výsledků.

V různých situacích se hodí různé postupy. Pokud jsme zadali příliš konkrétní dotaz, na který vyhledávač odpověděl málo dokumenty, bude vhodné odebrat některá klíčová slova dotazu, abychom získali více výsledků. Pokud jsme naopak zadali příliš obecný dotaz, můžeme přidat nějaká klíčová slova, abychom obdrželi méně dokumentů, která ale lépe odpovídají na náš dotaz.

### 3.1.2. Hlavní cíl

Hlavním cílem vyhledávače je nacházet zmíněné úpravy dotazu automaticky. Tyto úpravy můžeme hledat například pomocí historie dotazů, pokud danou historii máme. Pokud uživatel položí dotaz „hosting php“, můžeme se podívat do historie vyhledávání a nalézt všechny dotazy, které obsahují alespoň jedno ze slov v dotazu a z této množiny dotazů pak můžeme nějakým postupem vybrat související dotazy pro všechny tři kategorie úprav. Můžeme například zjistit, že z konkrétnějších dotazů je nejvíce hledaný dotaz „hosting php mysql“, z podobných „server php“ a podobně.

My ale použijeme jiný postup. Všechny související dotazy budeme hledat pouze na základě znalostí sady dokumentů. Nebudeme k tomu využívat žádné dodatečné informace, všechno si spočítáme pouze ze samotných dokumentů.

K tomu využijeme formální konceptuální analýzu, která v daných datech vyhledává jisté shluky potenciálně zajímavých dat a zároveň je ukládá do hierarchie,

se kterou můžeme dále pracovat. Pro základní představu si můžeme představit výstup FCA jako Hasseův diagram, kde jeden z uzlů – ten musíme nějak najít – představuje aktuální výsledky, které nám vyhledávač zobrazil, „otcové“ tohoto uzlu jsou obecnější dotazy, „synové“ jsou konkrétnější dotazy a „sourozenci“ jsou podobné dotazy.

### 3.1.3. Výstup vyhledávače

Jak by měl vypadat výstup vyhledávače? Vyhledávač by měl zobrazit seřazenou sadu klasických výsledků, na tom se nic nemění. Dále by měl spočítat návrhy na úpravu dotazů ve všech třech kategorií – konkrétnější, podobný a obecnější dotaz. Pokud tyto návrhy existují, což není vždy, měl by je uživateli zobrazit. Tyto návrhy by měl vyhledávač opět seřadit podle relevance, tj. aby první úpravy dotazů byly nejsmysluplnější. Výstup pro dotaz „příležitostné příjmy“ by měl vypadat přibližně takto:

```
1. Příležitostná činnost
2. Ostatní příjmy a daňové přiznání
3. ...

+ sleva | + potvrzení
+/- sleva, příjmy | +/- příjmy, paušální
- příležitostné | -příjmy
```

Nejprve máme klasický seznam dokumentů setříděných podle relevance. Následující tři řádky jsou hlavním výsledkem vyhledávače, jsou to návrhy, jak upravit dotaz.

Na prvním řádku jsou klíčová slova, která můžeme k dotazu přidat. Tj. první návrh, `+sleva`, nám říká, abychom zkusili vyhledat dotaz „příležitostné příjmy sleva“, čímž bychom měli získat dokumenty, které nám řeknou, zda můžeme u příležitostných příjmů uplatnit nějakou slevu. To zní jako smysluplný dotaz.

Na druhém řádku máme podobné dotazy. Nabízí nám to vyhledat „sleva příjmy“, čímž bychom měli dostat dokumenty, které se věnují celkově slevám, které můžeme během danění příjmů uplatnit. Druhým dotazem, „paušální příjmy“, bychom se měli dostat k informacím o paušálních výdajích, které můžeme uplatnit při danění příjmů.

Na posledním řádku nám vyhledávač nabízí odstranit slova, abychom získali více dokumentů. Vizualizace nalezených návrhů je přebrána z vyhledávače SearchSleuth, viz kapitolu 5.4.

## 3.2. Stručný popis funkčnosti celého vyhledávače

Následující seznam ukazuje postup vyhledávače po zadání dotazu a nastiňuje spojení FCA s Information retrieval.

1. Uživatel vloží do vyhledávače dotaz. Vyhledávač odpoví seřazenou sadou výsledků. To zůstává stejné.

2. Pokud se jedná o dotaz s více než jedním klíčovým slovem, provedeme ještě jeden dotaz, kde všechna klíčová slova z dotazu spojíme operátorem OR.

3. Nyní přichází na řadu FCA část vyhledávače. Jako první se vytvoří kontext $\langle X, Y, I \rangle$. Objekty budou dokumenty, které nám vyhledávač vrátil v předchozím bodě. Atributy budou atributy těchto dokumentů plus klíčová slova z dotazu.

4. Relaci $I$ definujeme takto: $\langle x, y \rangle \in I$ právě tehdy, když dokument $x$ obsahuje slovo $y$.

5. Nyní musíme nalézt koncept dotazu. To je koncept, který odpovídá dotazu od uživatele. Můžeme ho nalézt tak, že vezmeme množinu dokumentů, které vyhledávač vrátil v prvním bodě, a spočítáme koncept, který těmto dokumentům odpovídá.

6. Jednotlivé návrhy poté nalezneme v intentech okolních konceptů. Specializaci nalezneme v dolních sousedech, generalizaci v horních sousedech a podobné dotazy v sousedních konceptech. Pokud například koncept dotazu má intent {limita, funkce} a nějaký dolní soused má intent {limita, funkce, vlastní}, odvodíme z toho specializaci `+vlastní`.

Tato idea není nijak nová a už byla implementována v několika prototypech jako například CREDO, FooCA a SearchSleuth, podrobnější popis těchto vyhledávačů, včetně popisu odlišností, se nachází v kapitole 5.4. Myšlenka vyhledávače, který spolupracuje s FCA je pak popsána například v [5]. Zde popisovaný vyhledávač je velice podobný vyhledávači SearchSleuth, přebírá některé vizuální prvky a FCA část je také téměř stejná. Vyhledávač CLaSeek je vlastní pokus o ověření funkčnosti celé ideje.

Všechny tři předchozí vyhledávače pracují, či pracovaly, nad dynamickou sadou dokumentů, typicky braly výsledky z nějakého webového vyhledávače. Naproti tomu vyhledávač popsaný v této práci pracuje primárně nad statickou sadou dokumentů; na vstupu bere seznam dokumentů, ty stáhne, zaindexuje a pak pracuje nad tímto statickým indexem. Nicméně skrze poskytované API je možné vytvořit vyhledávač, který bude pracovat nad dynamickou sadou dokumentů a bude se chovat prakticky stejně jako například SearchSleuth, viz kapitolu 5.1.2.

## 3.3. Jak vytvořit kontext

### 3.3.1. Sestavení kontextu

Označme dotaz, který uživatel položil vyhledávači, jako $Q$. Výsledný seřazený seznam dokumentů označíme $D$, přičemž $D_1$ je dokument na prvním místě, $D_2$ na druhém atp. Pro účely této kapitoly předpokládejme, že máme dokumenty v $D$ reprezentovány jako množinu všech stemů, které dokument obsahuje.

Během budování seznamu jsme každému dokumentu spočítali jeho atributy, což byla nějaká slova, která tento dokument nejvíce charakterizovala. Tyto atributy máme seřazené podle jejich tf-idf skóre. Označme $A_i$ seznam atributů pro dokument $D_i$.

Kontext $\langle X, Y, I \rangle$ sestavíme takto: ze seznamu dokumentů vezmeme prvních $n$ dokumentů, kde $n$ je hodnota daná nastavením vyhledávače. V základním nastavení platí $n = 50$. Tento seznam $n$ dokumentů bude představovat objekty kontextu

$$X = \{D_i \mid i \leq n\}.$$

Dále nalezneme atributy. Vyhledávač umožňuje nastavit maximální počet atributů, které má u každého dokumentu vzít. Tuto hodnotu pojmenujeme $m$. Nyní z každého dokumentu z množiny $X$ vezmeme maximálně $m$ atributů daného dokumentu a tato slova sjednotíme do jedné množiny. Nejprve si nadefinujeme funkci, která nám pro každý dokument vrací $m$ atributů dokumentu s nejvyšším tf-idf skórem

$$\alpha_m(A_i) = \left\{A_i^j \mid j \leq m\right\},$$

kde $A_i^j$ označuje $j$-tý atribut dokumentu $D_i$. Tedy $A_i^1$ je atribut dokumentu $D_i$, který má nejvyšší hodnocení. Nyní sjednotíme všechny atributy všech dokumentů do jedné množiny

$$Y_1 = \bigcup \{\alpha_m(A_i) \mid i \leq n\}.$$

K této množině ještě přidáme slova z dotazu $Q$ převedená na stemy. Nepřidáváme ale slova, která se vyskytují v NOT operátoru. Zavedeme tak funkci $\beta$, která bere na vstupu dotaz a na výstupu vrací množinu slov, které se vyskytují v dotazu, ale nevyskytují se v NOT. Tato funkce se bude chovat různě v závislosti na tom, jak vypadá dotaz $Q$:

1. pokud je $Q$ pouze jedno slovo, označme ho $s$, pak $\beta(Q) = \{s\}$,
2. pokud je $Q$ tvaru „$a$ AND $b$“ nebo „$a$ OR $b$“, pak $\beta(Q) = \beta(a) \cup \beta(b)$,
3. pokud je $Q$ tvaru „NOT $a$“, pak $\beta(Q) = \emptyset$.

Celou množinu atributů našeho kontextu tak získáme tak, že k současných atributům v $Y_1$ ještě přidáme klíčová slova dotazu

$$Y = Y_1 \cup \beta(Q).$$

Už zbývá pouze definovat relaci $I$. Křížek v tabulce bude tehdy, když daný dokument obsahuje dané slovo. Zapsáno formálně:

$$\langle x, y \rangle \in I \iff y \in x,$$

kde $x \in X$ a $y \in Y$.

### 3.3.2. Nalezení konceptu dotazu

Dalším úkolem je nalezení konceptu, který reprezentuje výsledek aktuálního dotazu. Máme v zásadě dvě možnosti, jak tento koncept nalézt. Vezmeme si slova z dotazu $\beta(Q)$ a spočítáme extent a zpět intent. Tím dostaneme koncept, který v intentu obsahuje slova z dotazu a zároveň je to nejmenší koncept, který tato slova obsahuje. Koncept dotazu budeme označovat $\mathbb{H}$:

$$\mathbb{H} = \left\langle \beta(Q)^{\downarrow}, \beta(Q)^{\downarrow\uparrow} \right\rangle.$$

Tento postup má ale zásadní nevýhodu v případě, kdy se snažíme nalézt koncept dotazu pro dotaz, který obsahuje jiný operátor než AND. Pokud má dotaz tvar $Q =$„derivace OR integrál“, nemůžeme hledat koncept dotazu tak, že spočítáme extent $\{\text{derivace, integrál}\}^{\downarrow}$, protože bychom tím získali dokumenty, které obsahují jak slovo „derivace“, tak slovo „integrál“. Což jistě nepředstavuje koncept dotazu „derivace OR integrál“.

Lepším postupem tak je začít počítat koncept dotazu přes dokumenty. Pokud vezmeme množinu dokumentů, které vyhledávač vrátil, přesněji tu podmnožinu, kterou jsme použili v kontextu, $X$, pokud jsme aplikovali omezení, a tuto množinu uzavřeme, dostaneme koncept, který přesně odpovídá našemu dotazu. Dostaneme tak koncept

$$\mathbb{H} = \left\langle X^{\uparrow\downarrow}, X^{\uparrow} \right\rangle = \left\langle X, X^{\uparrow} \right\rangle.$$

Problémem tohoto přístupu je, že vždy dostaneme ten největší koncept. Nemůžeme tak hledat obecnější dotazy v horních sousedech, ani související dotazy v okolních sousedech, protože ani horní, ani okolní sousedy tento koncept nemá. To vyřešíme jedině tak, že kontext, který používáme, sestavíme jinak.

### 3.3.3. Sestavení rozšířeného kontextu

Kontext, který byl popsán v kapitole 3.3.1. nebudeme vůbec používat. Sestavíme jiný kontext. Nicméně provedeme pouze malé změny.

Začneme opět tím, že vyhledáme dotaz $Q$. Dostaneme dokumenty $D$. První změnou bude, že v případě, že dotaz bude mít více než jedno slovo, $|\beta(Q)| > 1$, tak položíme vyhledávači ještě jeden dotaz. Nový dotaz označíme $Q'$ a bude mít tento tvar: vezmeme všechna slova z dotazu $Q$ a vložíme mezi každá dvě slova OR. Pokud byl původní dotaz $Q =$„formální konceptuální analýza“, tak nový

dotaz bude mít tvar $Q'$ =„formální OR konceptuální OR analýza“. Dotaz $Q'$ položíme vyhledávači, čímž získáme dokumenty $D'$.

Další postup už je stejný jako v kapitole 3.3.1., pouze do kontextu vložíme dokumenty a atributy z množiny $D'$, nikoliv z $D$. Za $X'$ vezmeme prvních $n$ dokumentů z $D'$, za $Y'$ jejich atributy sjednocené s $\beta(Q)$. Relaci $I'$ sestavíme stejně

$$\langle x, y\rangle \in I' \iff y \in x.$$

Dostaneme rozšířený kontext $\langle X', Y', I'\rangle$.

### 3.3.4. Nalezení konceptu dotazu v rozšířeném kontextu

Koncept dotazu nalezneme jednoduše v případě, že dotaz $Q$ se skládá pouze z AND operátorů. Pak je koncept dotazu roven

$$\mathbb{H} = \left\langle \beta(Q)^{\downarrow}, \beta(Q)^{\downarrow\uparrow} \right\rangle.$$

V případě, že dotaz obsahuje OR nebo NOT operátor, musíme koncept dotazu hledat jinak. Můžeme vzít všechny dokumenty, které vyhledávač vrátil během hledání prvního dotazu $Q$, tj. $D$. K těmto dokumentům nalezneme v kontextu $\langle X', Y', I'\rangle$ intent a zpět extent. Dostáváme tak koncept

$$\mathbb{H} = \left\langle D^{\uparrow\downarrow}, D^{\uparrow} \right\rangle.$$

Pokud bychom neomezovali velikost kontextu, postup by fungoval. Vzhledem k tomu, že množina objektů $X'$ může být menší než množina navrácených dokumentů $D'$, tak ještě provedeme průnik množin, abychom zajistili, že množina, ke které počítáme intent, je podmnožinou $X'$. Pak koncept dotazu je roven

$$\mathbb{H} = \left\langle (D \cap X')^{\uparrow\downarrow}, (D \cap X')^{\uparrow} \right\rangle.$$

Idea rozšířeného kontextu a hledání konceptu dotazu byla použita ve vyhledávači SearchSleuth a je popsána v [11]. SearchSleuth používá k tvorbě rozšířeného kontextu horní sousedy konceptu dotazu – nejprve sestaví nerozšířený formální kontext, spočítá koncept dotazu, nalezne horní sousedy a intenty těch sousedů znovu vyhledá a výsledné dokumenty připojí ke kontextu, čímž vznikne rozšířený kontext. CLaSeek používá jiný postup, hledá rozšířený koncept přes OR dotaz.

Zároveň také SearchSleuth nepřidává do kontextu klíčová slova z dotazu, protože předpokládá, že se slova vyskytnou v titulcích a popiscích jednotlivých odkazů, které vrátí webový vyhledávač. To nemusí vždy nastat, takže ve formálních kontextu mohou některá klíčová slova chybět. SearchSleuth dále odstraňuje atributy, které sdílí méně než 5 % objektů.

## 3.4. Lindigův algoritmus pro hledání horních sousedů

Lindigův algoritmus je algoritmus pro budování konceptuálního svazu z kontextu. Součástí je i algoritmus na spočítání horních sousedů nějakého konceptu. Tento algoritmus budeme potřebovat, takže si ho popíšeme. Nejprve si ale zadefinujeme, co je to horní soused konceptu.

**Definice 19** (Horní soused konceptu)**.** *Mějme kontext $\langle X, Y, I\rangle$ a jemu příslušný konceptuální svaz $\mathcal{B}(X, Y, I)$. Dále mějme koncept $\langle A, B\rangle \in \mathcal{B}(X, Y, I)$, který není největším konceptem svazu, tj. $\langle A, B\rangle \neq \langle X, X^{\uparrow}\rangle$.*

*Koncept $\langle A', B'\rangle$ nazveme horním sousedem konceptu $\langle A, B\rangle$, pokud platí, že $\langle A, B\rangle < \langle A', B'\rangle$ a zároveň platí, že neexistuje koncept $\langle A'', B''\rangle$ takový, že $\langle A, B\rangle < \langle A'', B''\rangle < \langle A', B'\rangle$.*

Jinými slovy, horní koncepty jsou ty koncepty, které se nachází právě o jednu úroveň výše než koncept $\langle A, B\rangle$.

Myšlenka algoritmu je založena na tom, že když přidáme k extentu $A$ konceptu $\langle A, B\rangle$ objekt z $X$, který není v $A$, a uzavřeme, získáme koncept, který je určitě větší. Pokud postupně přidáme všechny objekty z $X \setminus A$, dostaneme množinu konceptů, která obsahuje všechny horní sousedy. Množinu si pojmenujeme $S$:

$$S = \left\{ \left\langle (A \cup \{a\})^{\uparrow\downarrow}, (A \cup \{a\})^{\uparrow} \right\rangle \mid a \in X \setminus A \right\}$$

Tato množina sice obsahuje všechny horní sousedy, ale bohužel i některé navíc. Může se stát, že přidáním objektu $a \in X \setminus A$ „přeskočíme“ nějaký koncept a dostaneme koncept, který je o více než jednu úroveň výše. Abychom tak získali pouze množinu horních sousedů, musíme při každém přidání nového objektu zkontrolovat, zda jsme skutečně vygenerovali horního souseda.

To zkontrolujeme tak, že zjistíme, zda všechny objekty, které přibyly v novém konceptu $\langle A', B'\rangle$, generují tentýž koncept. Přesněji to popisuje následující věta.

**Věta 20** (Jak nalézt horní sousedy)**.** *Nechť $\langle A, B\rangle \in \mathcal{B}(X, Y, I)$ a $\langle A, B\rangle \neq \langle X, X^{\uparrow}\rangle$. Pak $(A \cup \{a\})^{\uparrow\downarrow}$, kde $a \in X \setminus A$ je extent horního souseda konceptu $\langle A, B\rangle$ právě tehdy, když pro všechna $x \in (A \cup \{a\})^{\uparrow\downarrow} \setminus A$ platí $(A \cup \{x\})^{\uparrow\downarrow} = (A \cup \{a\})^{\uparrow\downarrow}$.* □

Pomocí této věty už můžeme napsat algoritmus, kterým vygenerujeme všechny horní koncepty konceptu $\langle A, B\rangle$ ve svazu $\mathcal{B}(X, Y, I)$, viz obrázek 3.4.

Algoritmus používá efektivnější systém detekce, zda se jedná o horního souseda, nebo zda jde o vyšší koncept. Množina *candidates* obsahuje ty prvky, které budeme zkoušet přidávat k extentu $A$. Na řádku 4 vybereme prvek $a$ z této množiny a spočítáme nový extent a intent.

Na řádku 7 se nachází hlavní změna. Rozdíl množin $A' \setminus A$ nám dá ty objekty, které v novém konceptu přibyly. Od nich ještě odstraníme ten prvek $a$, který jsme právě přidali. Pokud se žádný ze zbývajících prvků nenachází v množině

```
1: function UpperNeighbors(⟨A, B⟩, ℬ(X, Y, I))
2:     candidates ← X \ A
3:     neighbors ← ∅
4:     for all a ∈ X \ A do
5:         B′ ← (A ∪ {a})↑
6:         A′ ← B′↓
7:         if candidates ∩ (A′ \ A \ {a}) = ∅ then
8:             neighbors ← neighbors ∪ (⟨A′, B′⟩)
9:         else
10:            candidates ← candidates \ {a}
11:        end if
12:    end for
13:    return neighbors
14: end function
```

Obrázek 4. Algoritmus UpperNeighbors na počítání horních sousedů konceptu

*candidates*, máme horního souseda, přidáme ho do množiny *neighbors* a prvek $a$ ponecháme v množině *candidates*, to je důležité. Pokud prvek $a$ negeneruje horního souseda, odstraníme ho z množiny *candidates*.

Pokud prvek $a$ generuje horního souseda, musí tak být nutně poslední z těch objektů, které tento koncept mohou generovat. Pokud má horní soused o tři objekty více, než koncept, ke kterému souseda hledáme, podmínkou na řádku 7 projde až ten třetí objekt, který zkusíme.

Pokud například máme koncept s extentem $\{1\}$ a horní soused má extent $\{1, 2, 3, 4\}$, tak pokud zvolíme poprvé $a = 2$, tak $\{1, 2, 3, 4\} \setminus \{1\} \setminus \{2\} = \{3, 4\}$. Množina *candidates* obsahuje přinejmenším prvky $\{2, 3, 4\}$, takže průnik bude neprázdný. Nyní bychom z *candidates* odstranili 2 a pokračovali s $a = 3$. To by neprošlo ze stejného důvodu, takže ho z *candidates* opět odstraníme. Na konci bychom zvolili $a = 4$, rozdíl by vyšel $\{1, 2, 3, 4\} \setminus \{1\} \setminus \{4\} = \{2, 3\}$. Z *candidates* jsme prvky 2 a 3 odstranili, takže tento prvek bude generovat horního souseda. Prvek 4 zároveň zůstane v množině *candidates*.

Nyní proč to nepropustí koncept, který je výše než horní koncept. Máme v zásadě dvě možnosti: pokud ten správný horní koncept ještě nebyl vygenerován, tak průnik na řádku 7 bude vždy neprázdný – vždy až poslední objekt generuje horního souseda. A pokud už jsme správného horního souseda vygenerovali, tak jsme ten poslední prvek $a$, který ho generoval, neodstranili z množiny *candidates* a v průniku na řádku 7 tak vždy zůstane minimálně tento prvek, čímž nebude splněna podmínka.

Například pokud bychom pokračovali v předchozím příkladu a snažili se ověřit koncept s extentem $\{1, 2, 3, 4, 5\}$, kde zvolíme $a = 5$, tak dojdeme k tomu, že $\{1, 2, 3, 4, 5\} \setminus \{1\} \setminus \{5\} = \{2, 3, 4\}$, ale prvek 4 v množině *candidates* zůstal,

takže průnik bude neprázdný. Množina *candidates* tak na konci obsahuje právě ty prvky, které generují horní sousedy.

Tento algoritmus můžeme jednoduše upravit tak, aby generoval dolní sousedy. Stačí místo objektů přidávat atributy a dostaneme algoritmus LowerNeighbors.

Algoritmus byl původně popsán v [14].

## 3.5. Hledání návrhů ve svazu

V současné chvíli máme spočítaný koncept dotazu a víme, jak spočítat horní a dolní sousedy. To bude stačit k tomu, abychom zjistili všechny návrhy. Postup nalezení těchto návrhů bude vycházet z [11].

### 3.5.1. Specializace

Specializací budeme rozumět návrh, který konkretizuje dotaz tak, aby uživatel, z těch výsledků, které už má k dispozici, dostal nějakou významnou podmnožinu. Například pokud zadá slovo „jaguár“, aby ho specializace dovedla na podmnožinu dokumentů, které se zabývají autem, nebo zvířetem.

Jako první si spočítáme dolní sousedy našeho konceptu dotazu. Tím dostaneme množinu konceptů, kterou si označíme $\mathbb{L}_{\mathbb{H}}$

$$\mathbb{L}_{\mathbb{H}} = \text{LowerNeighbors}(\mathbb{H}).$$

Z každého konceptu budeme nejdříve potřebovat pouze intent, poté extent. Nechť tak funkce Int a Ext vrací z dané množiny konceptů $K$ intenty a extenty.

$$\text{Int}(K) = \{B \mid \langle A, B\rangle \in K\} \tag{25}$$
$$\text{Ext}(K) = \{A \mid \langle A, B\rangle \in K\} \tag{26}$$

V tuto chvíli máme koncept dotazu, který má například intent {Jaguár} a dva dolní sousedy, které mají intenty {Jaguár, Zvíře} a {Jaguár, Auto}. Abychom získali návrhy `+Zvíře` a `+Auto`, odečteme od intentů dolních sousedů intent konceptu dotazu. Označme $\mathbb{H} = \langle \mathbb{H}_E, \mathbb{H}_I\rangle$. Dostaneme množinu možných návrhů na specializaci dotazu.

$$Spec' = \{B \setminus \mathbb{H}_I \mid B \in \text{Int}(\mathbb{L}_{\mathbb{H}})\}.$$

Množina $Spec'$ obsahuje možné návrhy na rozšíření dotazu, ale některé z rozšíření mohou obsahovat více než jedno slovo. Pokud má koncept dotazu intent {Jaguár} a dolní soused má intent {Jaguár, rychlost, maximální}, pak v množině $Spec'$ bude návrh {rychlost, maximální}.

Nicméně je zbytečné navrhovat úpravu `+rychlost, maximální`, protože stačí, když uživateli navrhneme přidat pouze jedno z těchto slov a dovedeme ho ke

stejnému konceptu. Z každé množiny v $Spec'$ tak vybereme vždy jen to slovo, které se ve všech dokumentech vyskytuje nejčastěji. Dostaneme finální množinu

$$Spec = \{\max(s) \mid s \in Spec'\},$$

kde max v tomto případě vybere to slovo, které se v dokumentech $\mathbb{D}$ vyskytuje nejčastěji. Množina $Spec$ už je finální a obsahuje všechny návrhy, které můžeme uživatel předložit. Zbývá jen definovat uspořádání, abychom uživateli navrhovali ty nejrelevantnější návrhy.

Řekneme, že návrh $s_1 \in Spec$ je relevantnější než návrh $s_2 \in Spec$, zapíšeme $s_1 \geq s_2$, právě tehdy, když pro koncepty $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle \in \mathbb{L}_{\mathbb{H}}$, jejichž intenty generují $s_1$ a $s_2$, platí $|A_1| \geq |A_2|$.

V extentech $A_1$ a $A_2$ jsou jednotlivé dokumenty, ideou tak je, že nejrelevantnější návrh by měl zároveň vést na nejvíce dokumentů.

### 3.5.2. Generalizace

Generalizací budeme rozumět návrh, který zobecňuje dotaz tak, aby uživatel po této změně získal více dokumentů, které odpovídají nějakému obecnějšímu konceptu. Zobecnění bude prováděno odstraněním klíčových slov z dotazu.

Generalizace budeme hledat v horních sousedech konceptu dotazu, takže si je spočítáme

$$\mathbb{U}_{\mathbb{H}} = \text{UpperNeighbors}(\mathbb{H}).$$

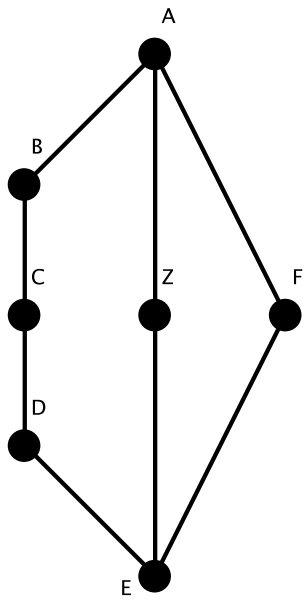
Nyní jsme v opačné situaci než během hledání specializací. Koncept dotazu může mít intent {Jaguár, Zvíře} a horní sousedé intenty {Jaguár} a {Zvíře}. Tím, že odečteme intenty horních sousedů od intentu konceptu dotazu, získáme slova, která se nachází v intentu konceptu, ale nenachází se v intentu horního souseda. Jsou to ta slova, která vyhledávač navrhne uživateli k odstranění.

$$Gen' = \{\mathbb{H}_I \setminus M \mid M \in \text{Int}(\mathbb{U}_{\mathbb{H}})\}$$

Množina $Gen'$ obsahuje množiny slov, která může uživatel odebrat. Ale může obsahovat i slova, která neobsahuje samotný dotaz a která tak nelze odstranit. Například pokud uživatel zadá dotaz „jaguár rychlost“, může být intent konceptu dotazu {jaguár, rychlost, maximální} a pokud má horní soused intent {jaguár}, pak by v množině $Gen'$ bylo i slovo „maximální“ a vyhledávač by tak uživateli nabídl odstranit slovo „maximální“, aniž by toto slovo v dotazu bylo. Proto ještě provedeme průnik všech prvků v množině $Gen'$ se slovy dotazu $Q$, abychom měli jistotu, že napovídáme odstranit jen ta slova, která uživatel skutečně do dotazu zadal.

$$Gen = \{g \cap \beta(Q) \mid g \in Gen'\}$$

Návrhy opět seřadíme tak, aby návrh na prvním místě vedl na co největší počet výsledků. Řekneme, že návrh $a_1 \in Gen$ je relevantnější než $a_2 \in Gen$, zapíšeme $a_1 \geq a_2$, právě tehdy, když koncepty pro koncepty $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle \in \mathbb{U}_{\mathbb{H}}$, jejichž intenty generují $a_1$ a $a_2$, platí $|A_1| \geq |A_2|$.




Obrázek 5. Sourozenci konceptu $Z$

### 3.5.3. Podobné dotazy

V této části popíšeme, jak vygenerovat dotazy, které jsou podobné současnému dotazu, který uživatel zadal do vyhledávače. Když jsme hledali konkrétnější dotazy, hledali jsme je v dolních sousedech; obecnější dotazy jsme hledali v horních sousedech. Podobné dotazy budeme hledat v konceptech, které se nachází na stejné úrovni. Můžeme tak říci, že je budeme hledat v sourozencích konceptu dotazu.

Otázkou je, jak definovat sourozence. Když se podíváme na obrázek 5., jaké koncepty by měly být sourozenci konceptu $Z$? Když se budeme držet terminologie, která je běžná u binárních stromů, pak by sourozenci byli potomci rodiče, což by odpovídalo sourozencům $\{B, F\}$.

U svazů ale můžeme zvolit i opačnou cestu – rodičové potomků. To by odpovídalo sourozencům $\{D, F\}$. Dále můžeme říci, že sourozenci jsou všichni mezi potomky a rodiči, tj. všechny koncepty mezi konceptem $A$ a $E$, tj. $\{B, C, D, F\}$.

Další rozumnou možností je vzít ty koncepty, které mají stejného rodiče jako koncept $Z$ a zároveň i stejného potomka. Tomu odpovídá koncept $F$.

Pro jednu z těchto definic bychom se měli rozhodnout. Není vhodné uživateli napovídat všechny dotazy z konceptů $B, C$ a $D$, protože $D$ je konkrétnější dotaz než $C$ a $C$ je konkrétnější dotaz než $B$. Z této řady stačí uživateli navrhnout jeden dotaz.

Pokud budeme za sourozence brát potomky rodičů, budeme mu nabízet spíše obecnější dotazy. Pokud zvolíme rodiče potomků, pak mu budeme nabízet spíše konkrétnější dotazy.

Zvolíme tak zlatou střední cestu – budeme mu nabízet dotazy, které jsou generovány koncepty mající společného rodiče i potomka. Zadefinujeme si funkce, které spočítají všechny horní a dolní sousedy všem prvkům množiny.

$$\mathrm{UN}(X) = \bigcup \{\mathrm{UpperNeighbors}(C) \,|\, C \in X\} \tag{27}$$

$$\mathrm{LN}(X) = \bigcup \{\mathrm{LowerNeighbors}(C) \,|\, C \in X\} \tag{28}$$

Nyní můžeme sourozence $\mathbb{S}$ konceptu $\mathbb{H}$ zadefinovat jako

$$\mathbb{S}_{\mathbb{H}} = [\mathrm{LN}(\mathrm{UN}(\{\mathbb{H}\})) \cap \mathrm{UN}(\mathrm{LN}(\{\mathbb{H}\}))] \setminus \{\mathbb{H}\}\,.$$

Z této množiny už jen vytáhneme intenty a dostaneme množinu podobných dotazů

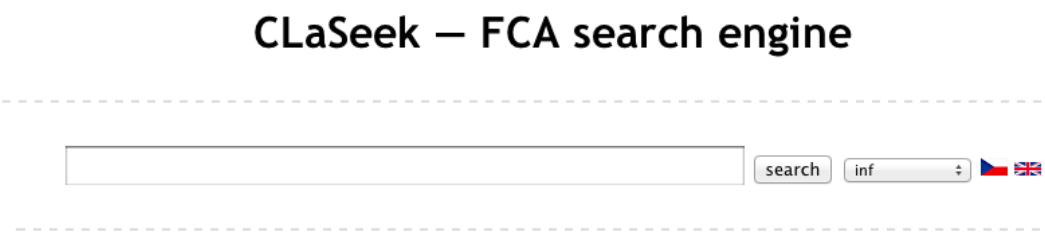
$$Sibl = \mathrm{Int}(\mathbb{S}_{\mathbb{H}}).$$

Tyto dotazy setřídíme tak, že spočítáme podobnost konceptů z množiny $\mathbb{S}_{\mathbb{H}}$ s konceptem dotazu. Podobnost dvou konceptů spočítáme přes podobnost jejich extentů a intentů. Podobnost dvou extentů pak jednoduše definujeme jako podíl počtu společných objektů a počtu všech objektů. Podobnost dvou konceptů $\langle A, B\rangle$ a $\langle C, D\rangle$ je rovna

$$\mathrm{sim}(\langle A, B\rangle, \langle C, D\rangle) = \frac{1}{2}\left(\frac{|A \cap C|}{|A \cup C|} + \frac{|B \cap D|}{|B \cup D|}\right).$$

Návrh $s_1 \in Sibl$ je relevantnější než $s_2 \in Sibl$, zapíšeme $s_1 \geq s_2$, právě tehdy, když pro koncepty $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle \in \mathbb{S}_{\mathbb{H}}$, jejichž intenty generují $s_1$ a $s_2$, platí $\mathrm{sim}(\langle A_1, B_1\rangle, \mathbb{H}) \geq \mathrm{sim}(\langle A_2, B_2\rangle, \mathbb{H})$. Jako první budeme zobrazovat dotaz, který je nejvíce podobný současnému dotazu $Q$.

Téma sousedních konceptů a jejich podobnosti je blíže rozebráno v článku [11] a [10].




Obrázek 6. Základní rozhraní vyhledávače

# 4. Výsledky vyhledávače

## 4.1. Uživatelské prostředí vyhledávače

Současná verze vyhledávače CLaSeek beží na adrese `http://phoebe.inf.upol.cz/claseek/`. Základní prázdné rozhraní je vidět na obrázku 6. Do textového pole se vloží dotaz, který chceme položit vyhledávači, vybereme jazyk, databázi, nad kterou chceme dotaz provádět a odešleme. Výsledek vyhledávání je vidět na obrázku 7.

Na prvních třech řádcích jsou zobrazeny návrhy na úpravu dotazu. Na prvním řádku jsou specializace, tj. jaká slova nabízíme uživateli k přidání k současnému dotazu. Na druhém řádku jsou podobné dotazy. Na třetím řádku je generalizace, tj. jaká slova by měl z dotazu odstranit. Tyto návrhy jsou odkazy, takže na ně stačí klepnout a CLaSeek vyhledá nový dotaz.

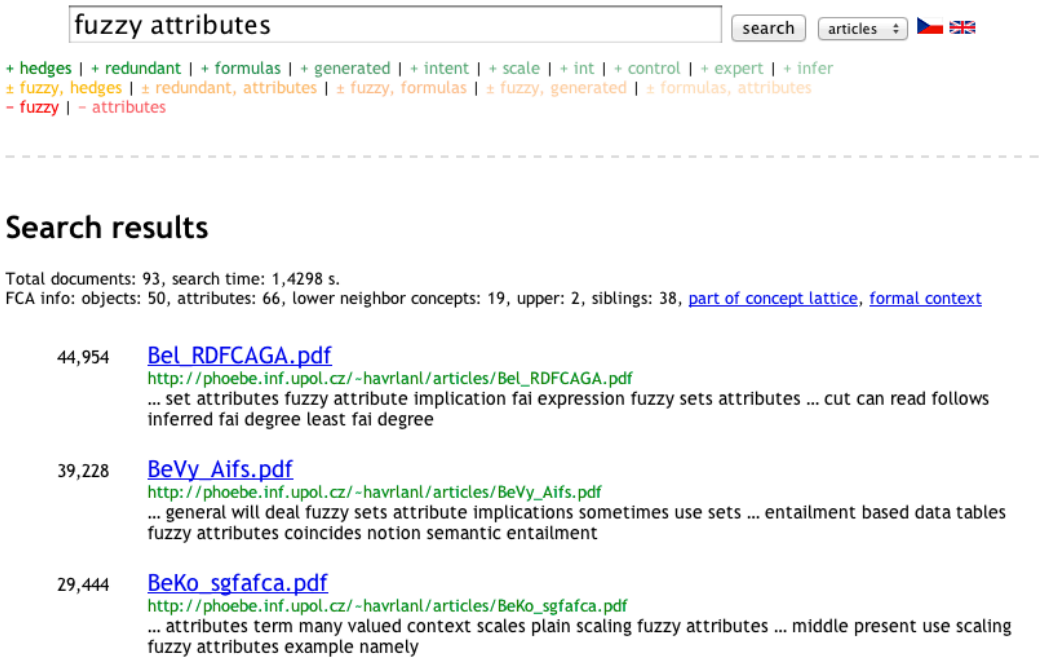
Níže se nachází část s výsledky vyhledávání a před nimi ještě nějaké informace o dotazu. Na prvním řádku je vypsán celkový počet nalezených dotazů a čas, který byl potřebný na jejich nalezení. Měří to čistě jen práci vyhledávače, jádra, větší množství času je obvykle ztraceno samotnou komunikací mezi jednotlivými vrstvami.

O řádek níže jsou informace o kontextu a konceptuálním svazu. První dvě položky jsou počty objektů a atributů, které obsahuje kontext, počet dolních a horních sousedů konceptu dotazu a počet jeho sourozenců. Na konci jsou odkazy na zobrazení části svazu, se kterým pracujeme, a na samotný kontext.

Pak už následují samotné výsledné dokumenty seřazené dle relevance. Vypočítaná hodnota tf-idf pro každý dokument je zobrazena nalevo od odkazů na dokumenty. U dokumentů je nejdříve zobrazen jejich název, pak URL a nakonec popisek.

## 4.2. Požadavky na zpracovávané dokumenty

Kvalita výsledků v prvé řadě závisí na kvalitě zpracovávaných dokumentů. V této části budou popsány některé vlastnosti, které by dokumenty měly mít, aby CLaSeek dobře fungoval.




Obrázek 7. Výsledek vyhledávače na dotaz „fuzzy attributes“

### 4.2.1. Kvalita obsahů dokumentů

Zásadním požadavkem na dokument je, aby měl dobrý obsah, ze kterého půjde snadno pomocí tf-idf algoritmu získat jeho atributy. Každý dokument by měl mít ideálně jedno hlavní téma a v jeho obsahu by se měla často vyskytovat slova, která toto téma charakterizují.

Pokud má vyhledávač dobře fungovat, nemělo by se stávat, že stránka „studijní obory“ obsahuje pouze seznam studijních oborů, ale samotná slova „studijní“ a „obor“ se tam nevyskytují buď vůbec, nebo jen v titulku. Vyhledávač se pak nemá čeho chytit, na dotaz „studijní obory“ tuto stránku zobrazí někde ke konci seznamu a jako atribut dokumentu zvolí nějaké nesmyslné slovo, které se tam objevuje poměrně často.

Například pro stránku o studijních oborech na `www.inf.upol.cz` zvolil vyhledávač jako hlavní atribut slovo „stupeň“, protože jednotlivé obory jsou tam rozděleny podle stupňů – 1. stupeň, 2. stupeň, 3. stupeň. Pro prezenční a kombinované studium zvlášť. Samozřejmě zůstává otázkou, jestli by zvýšený počet slov „obor“ pomohl i uživateli.

Vyhledávač si zároveň neumí efektivně poradit se stránkami, které obsahují příliš mnoho různých informací. Na webu katedry informatiky je například takovou stránkou kalendář nebo aktuality. Na těchto stránkách se vyskytují prakticky všechna důležitá klíčová slova. Nachází se zde informace o stážích jednotlivých

zaměstnanců katedry, o pořádaných seminářích, informace o oborech a podobně. Každá zpráva v aktualitách nebo v kalendáři je obvykle krátká a podepsaná.

Když pak uživatel vyhledá jméno nějakého zaměstnance z katedry, často se mezi prvními výsledky zobrazí právě aktuality nebo kalendář, protože samotná stránka není příliš velká, aby to srazilo hodnocení tf-idf a obvykle dané jméno obsahuje mnohokrát. Podobný výsledek se objeví, pokud vyhledáme slovo „seminář“. Pokud se seminář koná dvacetkrát do roka, má v kalendáři dvacet různých záznamů a kalendář tak obsahuje minimálně dvacetkrát slovo „seminář“. V hodnocení pak snadno předběhne stránku, která je pro informatický seminář přímo určena. Tyto problémy lze vyřešit různými způsoby:

- Napsat obsah stránky o informatickém semináři tak, aby obsahoval vícekrát slovo „seminář“.
- V hodnocení dokumentů zvýšit důležitost titulků nebo URL, které slovo „seminář“ obsahují.
- Využít další informace z celého webu, například text odkazů, které na stránku o informatickém semináři vedou.
- Vyjmout stránky kalendáře a aktualit z indexu, aby se ve výsledcích nezobrazovaly vůbec.

### 4.2.2. Technické požadavky na HTML stránky

Přestože CLaSeek podporuje další typy souborů jako PDF nebo ODT, je především stavěn pro HTML stránky. Aby vyhledávač dobře fungoval, měl by HTML dokument splňovat některá elementární pravidla.

Vyhledávač potřebuje u každého dokumentu znát jeho název. V případě HTML stránek ho zjišťuje v elementu `TITLE`. Tento element je dle specifikace HTML povinný, takže by ho měla obsahovat každá stránka. Nicméně pokud ho neobsahuje nebo je jeho obsah prázdný, tak se použije poslední část URL.

Obsah titulku by měl být pro každou stranu jiný. Není vhodné, aby mělo více stránek, nebo dokonce celý web, stejný titulek, protože pak je ve výsledcích vyhledávání nepůjde rozlišit jinak než jejich adresou. Toto pravidlo je důležité nejen pro tento vyhledávač, ale i pro běžné vyhledávače jako Google nebo Seznam. Zároveň se připravujeme o možnost zvýšit hodnocení stránky, pokud by klíčové slovo, které uživatel hledal, bylo v titulku.

Další informací, kterou může z webu získat, je popisek. Ten se zapisuje do elementu `<META NAME='description'>`. Jeho obsahem by měl být ne příliš dlouhý text, který popisuje danou stránku. Opět by měl být unikátní pro každou stránku. Pokud stránka neobsahuje popisek, vyhledávač se pokusí ze stránky vytáhnout nějaký popisek sám, ale téměř jistě to bude horší výsledek, než ručně psaný popisek pro každou stránku zvlášť.

Teoreticky by vyhledávač mohl získávat atributy dokumentů z elementu `<META NAME=keywords>`, který je k tomu určen. Ale během historie webu byl tento element poměrně hodně zneužívám vkládáním nesmyslných klíčových slov jako „sex, porno, nahá, pamela, anderson“, takže tento element postupně běžné vyhledávače začaly téměř ignorovat a většina běžných webů už ho ani nepoužívá. CLaSeek ho tak zcela ignoruje.

HTML kód nemusí být nutně validní, ale musí dodržovat syntaxi jazyka. Nehledí se například na to, jestli element `IMG` obsahuje povinný atribut `ALT`, ale pokud nebude hodnota atributu správně ukončena uvozovkami, parser si na tom vyláme zuby. Pro převod HTML stránky do obyčejného textu se ve vyhledávači používají dvě funkce – první používá vestavěný parser, který postupně parsuje stránku a neodpouští tak žádné chyby proti syntaxi jazyka; druhá funkce používá regulární výrazy a je tak benevolentnější, ale její výsledek je zase méně přesný. Druhá funkce se použije až v případě, kdy selže první funkce.

Web musí mít správně určené kódování stránek prostřednictvím elementu `<meta http-equiv='content-type'>`. Pokud se tento element nenajde, bude se předpokládat, že web je v kódování `utf-8`.

### 4.2.3. Zpracování PDF a ODT

CLaSeek si dále poradí s dokumenty ve formátu PDF a ODT. Formát ODT je pouze zabalené XML, takže daný soubor stačí pouze rozbalit, najít soubor, ve kterém je obsah celého dokumentu a zpracovat toto XML. To můžeme udělat podobně jako HTML stránky, stačí odstranit všechny XML značky a nechat pouze textové obsahy elementů.

S PDF už je větší problém. Vyhledávač používá externí aplikace na dolování textu z PDF souboru, ale žádná aplikace si neporadí se všemi PDF soubory. Největším problémem je diakritika; spousta převaděčů nedokázala po převodu na prostý text správně zobrazit diakritiku. V lepším případě převaděč zobrazil místo správného znaku nějaký nesprávný znak, v horším případě písmeno s dikritikou zcela vypustil, případně vypustil i následující písmeno. S takto převedeným textem už se toho moc dělat nedá.

Některé PDF dokumenty na webu katedry informatiky jsou v nějakém podivném stavu, kdy i nakopírování textu v některém ze známých PDF prohlížečů, například Adobe Reader, nefunguje jak má. Typická chyba je, že po vložení textu do jiné aplikace jsou zcela oddělené háčky a čárky, občas jsou ještě různě náhodně vloženy mezery. Vyhledávač se snaží alespoň o částečnou rekonstrukci textu, ale úplná rekonstrukce je nemožná.

### 4.2.4. Některé nedostatky vyhledávače

Podstatný požadavek je, aby se rozumné atributy nevyskytovala na všech stránkách webu. Tento problém může snadno nastat v případě, kdy jsou na všech

stránkách webu stejné odkazy v navigaci. Web katedry informatiky (ale i další zkoumané weby) má na všech stránkách stejnou navigaci, takže slova, která jsou v textech těchto odkazů, mají jen mimální šanci na to, že budou zvoleny atributem nějakého dokumentu.

Během výpočtu tf-idf funkce hledáme taková slova, která se často vyskytují v daném dokumentu, ale co nejméně ve všech ostatních dokumentech. Funkce bohužel nezkoumá, kolikrát se dané slovo v těch ostatních dokumentech nachází, takže stačí, když se v dokumentu nachází jen jednou v navigaci a už se to bude počítat jako výskyt.

Kódování HTML stránky lze definovat i HTTP hlavičkou. Samotná stránka pak nemusí obsahovat element `<meta http-equiv='content-type'>`. Vzhledem k tomu, že CLaSeek HTTP hlavičky nezkoumá, použil by v tomto případě výchozí kódování `utf-8`.

## 4.3. Hodnocení úspěšnosti vyhledávače

Ohodnotit, jak je CLaSeek úspěšný v řazení nebo v nabízených návrzích, nijak exaktně a automaticky nejde. Zbývá jen ruční zkoumání a slovní ohodnocení.

Ve vyhledávači je uloženo několik testovacích sad dokumentů. Jedná se o weby

- `jakpsatweb.cz`, který se zabývá tvorbou webových stránek a podobných věcí okolo,
- `matweb.cz`, který se zabývá především středoškolskou matematikou,
- `jakpodnikat.cz`, který se zabývá podnikáním, jak danit příjmy a podobně,
- `inf.upol.cz`, což je web katedry informatiky UP

a nakonec je ve vyhledávači použita i sada volně stažitelných odborných článků ve formátu PDF, které napsali zaměstnanci katedry informatiky UP.

Kromě webu katedry informatiky se jedná o weby, které mají jasné zaměření každé stránky a bohatý obsah, takže na jejich webech vypadá analýza nejlépe. Web katedry informatiky je stručnější, obsahuje jisté chyby (například titulky pro každou stranu zvlášť, jak je psáno v kapitole 4.2.1., byly vytvořeny až během psaní této práce) a obsahuje nepříjemnost v podobě opakujících se slov na každé stránce, viz kapitolu 4.2.4., takže jeho analýza neprobíhá nejlépe.

### 4.3.1. Příklady dobrých výsledků

Analýza webu jakpodnikat.cz dává několik dobrých výsledků, pro příklad:

- Na dotaz „živnostenský list“ vrací seznam specializací `+ studenti | + příjmy | + provozovna | + nemocenské | + účetnictví | + jednotný | + dědická | + odkazy | + nemovitostí | + auto`.

- Když k dotazu přidáme například slovo „provozovna“, vyhledávač odpoví novými specializacemi `+ studenti | + příjmy | + zahájení | + služby | + autorský | + nemovitostí`. Zároveň přidá i podobné dotazy `+/- list, příjmy, živnostenský | +/- list, zahájení, živnostenský | +/- autorský, list, živnostenský | +/- nemovitostí, list, živnostenský`.

- Po vyhledání dotazu „slevy na dani“ dostaneme specializace `+ příjmy | + studenti | + základ | + doklady | + tisíc | + příloha | + minimální`, z nichž všechny kromě slova „tisíc“ dávají nějaký smysl. Seznam podobných dotazů: `+/- dani, příjmy | +/- dani, základ | +/- dani, doklady | +/- tisíc, dani | +/- příloha, dani`. Opět kromě dotazu, který obsahuje klíčové slovo „tisíc“ to má nějaký smysl.

Dobré výsledky vrací CLaSeek i pro web jakpsatweb.cz:

- Pro dotaz „hosting“ vrátí seznam specializací `+ php | + serveru | + seo | + google | + weblogu | + zdarma | + nástroje | + statistiky | + domény | + anglicky`.

- Pokud přidáme slovo „php“, dostaneme nový seznam specializací `+ serveru | + knihy | + google | + zdarma | + faq | + díky | + weblogu | + xml | + seo | + statistiky` a seznam podobných dotazů: `+/- hosting, serveru | +/- google, hosting | +/- hosting, zdarma | +/- knihy, php | +/- php, zdarma`.

### 4.3.2. Příklady špatných výsledků

Ze současných databází dává většinou nejhorší výsledky web katedry informatiky. Příkladem budiž dotaz „studium“, na který získáme specializace `+ zimní | + absolvent | + letní | + skupina | + jan | + martin | + arnošt | + činnosti | + szz | + bartl`. Většina těchto dotazů nejspíše k žádnému významnému konceptu nepovede.

Dalším příkladem může být složitější dotaz „jak vložit obrázek na stránku“ nad databází jakpsatweb.cz. Vyhledávač odpoví specializacemi `+ obrázky | + soubor | + google | + mail`. Nabízí to dokonce slovo „obrázky“, protože použitý stemmer zvolí jiný základ pro slovo „obrázek“ a pro „obrázky“. Další návrhy nejsou o nic rozumnější.

## 4.4. Experimenty s nastavením

CLaSeek má poměrně bohaté možnosti nastavení. Některé z nich dokáží významně ovlivnit chování a výsledky celého vyhledávače. Všechny možnosti nastavení jsou popsány v kapitole 5.3.

### 4.4.1. Počet atributů dokumentu

Lze nastavit, jaký maximální počet atributů každého dokumentu vyhledávač vezme v potaz, když buduje rozšířený koncept, viz kapitolu 3.3.1. To do jisté míry ovlivní, jak budou vypadat jednotlivé návrhy u specializací a u podobných dotazů.

Čím vyšší limit nastavíme, tím větší bude kontext a tím více bude mít koncept dotazu dolních sousedů a sourozenců. Což v důsledku znamená, že vyhledávač má na konci více návrhů na specializaci a více podobných dotazů. Zvyšuje se tím šance, že se v této množině návrhů najde nějaký smysluplný návrh, ale zároveň se zvyšuje šance, že se tam dostane nějaký nesmysl, protože některý dokument nemusí mít tolik kvalitních atributů.

Brát v úvahu vysoký počet atributů dokumentů má smysl v případě, kdy víme, že k většině dokumentů máme alespoň tolik kvalitních atributů. Pokud není pravděpodobné, že bychom k většině dokumentů nalezli tolik kvalitních atributů, raději snížíme počet atributů, které bereme v potaz.

Zvýšení počtu atributů má samozřejmě za následek mírně zpomalení celého programu, protože se pracuje s větším kontextem a větším svazem.

### 4.4.2. Počet dokumentů v kontextu

V nastavení můžeme omezit počet dokumentů, které budou tvořit množinu objektů rozšířeného kontextu. Výsledek je podobný jako v předchozím případě – pokud zvýšíme počet dokumentů, které pustíme do kontextu, zvýšíme tím šanci, že se mezi návrhy objeví nějaký zajímavý, který by se tam jinak neobjevil. Současně tím ale zvyšujeme šanci, že se tam objeví nesmysl, protože jsme do kontextu pustili dokumenty, které jsou příliš vzdálené původnímu dotazu.

Zároveň při větším počtu dotazů roste výpočetní náročnost, protože se zvětšuje kontext a konceptuální svaz. Zvyšuje se více než v předchozím případě, protože tím, že zdvojnásobíme počet dokumentů, zdvojnásobí se i počet atributů.

### 4.4.3. Limit pro atributy dokumentu

Během budování indexu je možné zvolit dvě hlavní kritéria, jak se budou hledat atributy dokumentů. Jednodušší verze je, že vyhledávač spočítá tf-idf skóre pro všechna slova v dokumentu a poté vezme $n$ slov, která mají toto skóre nejvyšší. Druhý způsob je, že vezmeme jen ta slova, která překročila určitou hranici skóre.

První způsob je přímočarý, ale není příliš efektivní v případě, kdy máme různě kvalitní dokumenty. Pro ty nekvalitní dokumenty, u kterých lze jen těžko najít tolik smysluplných atributů, se budou za atributy postupně volit čím dál tím větší nesmysly. Výhodou je, že pro každý dokument máme stejný počet atributů.

U druhého způsobu je o něco těžší nalézt ideální hranici. Hranice je číslo z intervalu $\langle 0, 100 \rangle$ a bude aplikována takto: spočítáme hodnotu tf-idf skóre pro

všechna slova ve všech dokumentech. Tato čísla seřadíme a dostaneme uspořádaný seznam ohodnocených slov.

Ze začátku a z konce tohoto seznamu odstraníme setinu slov. Pokud seznam obsahuje 500 slov, pak ze začátku i z konce seznamu odstraníme pět slov. Tím odstraníme slova, která by mohla mít příliš vysoké nebo příliš nízké hodnocení. Dále spočítáme rozdíl největšího a nejmenšího hodnocení. Tento rozdíl si označíme $\delta$. Hranici vydělíme stem, abychom získali číslo v intervalu $\langle 0, 1\rangle$, označme tuto novou hodnotu $l$. Dále ještě označme nejmenší hodnotu skóre ze všech slov symbolem $m$.

Pak každé slovo, které má hodnotu tf-idf skóre alespoň $l \cdot \delta + m$ bude považováno za atribut daného dokumentu.

# 5. Dodatky

## 5.1. API vyhledávače

API aplikace je rozhraní, pomocí něhož může uživatel s danou aplikací komunikovat. Uživatel aplikaci posílá nějaké dotazy ve specifickém formátu a aplikace na ně opět ve specifickém formátu odpovídá. Uživatel tak může danou aplikaci používat, ačkoliv ji přímo nezkompiluje do své aplikace.

Jak jádro vyhledávače, tak webové rozhraní poskytuje API pro uživatele. Webové API je přístupné komukoliv, konzolové API je přístupné tomu, kdo si nainstaluje vyhledávač k sobě. Webové API je RESTové, to znamená, že využívá HTTP protokol a běžné metody jako GET nebo POST. Všechna data, která vyhledávač vrací na výstupu, jsou ve formátu JSON. Pro detaily viz kapitolu 5.3.

### 5.1.1. Získávání dat z indexu

CLaSeek umožňuje získávat data o existujících indexech. Ta se získávají pomocí HTTP GET požadavku s požadovanými parametry. Je možné zjistit například informace o daném dokumentu, jaký má titulek, jakou adresu a podobně. Zároveň lze zjisit další informace o indexu, například počet všech zaindexovaných dokumentů nebo seznam URL adres všech dokumentů.

Například dotazem `/api.php?d=jpw&docinfo=147&title` zjistíme název dokumentu z databáze `jpw`, který má ID 147.

### 5.1.2. Posílání vlastních dat na server

API vyhledávače umožňuje poslat na server vlastní dokumenty a provést nad nimi FCA analýzu. To se děje HTTP POST požadavkem. Poslaná data musí být ve formátu JSON. CLaSeek tyto dokumenty přijme, zpracuje je, tj. provede úplně stejné operace, jako by budoval klasický index, vytvoří dočasný index a následně nad těmito daty provede standardní FCA analýzu, jaká byla popsána v předchozích částech dokumentu.

Výsledkem budou opět data ve formátu JSON, která budou obsahovat výsledek FCA analýzy. Nalezneme tam specializace, generalizace a podobné dotazy a některá další data o kontextu a konceptuálním svazu. Dále tam bude seznam dokumentů, který bude seřazený dle relevance. Výstup je prakticky stejný, jako je grafický výstup vyhledávače, pouze máme výsledky v surové JSON podobě.

Tohoto můžeme využít na vylepšení výsledků běžného vyhledávače, například Googlu. Napíšeme webovou aplikaci, na které bude textové pole pro zadání dotazu. Uživatel zadá dotaz, aplikace tento dotaz přepošle API Googlu a získá výsledky pro zadaný dotaz. Následně převede výsledky z Googlu do takového formátu, kterému rozumí vyhledávač.

My od Google získáme seznam několika odkazů, které odpovídají zadanému dotazu. Pro každý odkaz máme k dispozici tři zásadní údaje: URL, titulek a popisek. Z těchto tří údajů vytvoříme sadu dokumentů takto: každý odkaz bude tvořit jeden samostatný dokument. Takže pokud nám Google vrátí na dotaz 50 výsledků, budeme vytvářet 50 dokumentů. URL dokumentu bude stejné jako URL, které vrátil Google; stejně tak titulek. Obsahem dokumentu pak bude popisek odkazu. Popisek dokumentu může být buď stejný jako jeho obsah nebo bude prázdný.

Takto poskládaná data, zároveň s původním dotazem, který jsme kladli Googlu, pošleme vyhledávači. On z nich seskládá dočasný index a vrátí nám návrhy, které vygenerovala FCA analýza. Tyto návrhy pak můžeme předložit uživateli. Tím dostaneme aplikaci, která už je téměř stejná jako SearchSleuth.

Jednou z cest, kam může směřovat tento vyhledávač, je rozšíření API a napsání dalších aplikací, které toto API využívají. Jednoduchou úpravou algoritmů ve vyhledávači můžeme například vytvořit API, které bere na vstupu seznam dokumentů a na výstupu vrací kategorizovaný seznam dokumentů – ke každému dokumentu přiřadí jednu, případně i více, kategorii. To se může hodit v případě, kdy máme velké množství dokumentů, které chceme setřídit do nějakých kategorií. Podobných způsobů využití lze jistě vymyslet více.

## 5.2. Struktura aplikace a používané programy

Jádro vyhledávače je psáno v Pythonu 3, nainstalovaná verze pak běží na Pythonu 3.2. Zdrojové soubory jsou rozděleny do balíčků, které obsahují moduly. Mezi nejdůležitější balíčky patří:

`preprocess:` Obsahuje moduly, které jsou zodpovědné za budování nového indexu, úpravu existujícího indexu, odstranění HTML značek z HTML stránek a podobně.

`retrieval:` Tento balíček je zodpovědný za dolování informací z indexu, parsování dotazu nebo hodnocení dokumentů.

`fca:` Definuje základní struktury pro práci s FCA, jako například kontext nebo koncept.

`fca-extension:` Po získání dat z indexu se moduly z tohoto balíčku snaží nalézt správné návrhy na vylepšení dotazu.

Veškerou komunikaci s externími aplikacemi pak obstarává soubor `/src/search`, který volá funkce z modulu `/src/other/interface.py`. Ten obsahuje funkce zodpovědné za komunikaci se všemi ostatními balíčky a moduly.

Aplikace používá dva stemmery, jeden pro český jazyk a jeden pro anglický. Pro český jazyk je využit kód, jehož autorem je Ljiljana Dolamic, jako anglický stemmer je použit program implementující Porter2 algoritmus, který napsal

Matt Chaput. Pro převod PDF do prostého textu aplikace používá program Xpdf `http://www.foolabs.com/xpdf/`. Na převod ODT do prostého textu není použita žádná externí aplikace, pouze `unzip` a obecný XML parser.

Webové rozhraní je napsáno v PHP 5. HTML kód je validní dle současného návrhu HTML 5.

## 5.3. Dokumentace

Dokumentace k jednotlivým částem programu se nachází na webu spolu se zdrojovými kódy aplikací. K distribuci zdrojového kódu byl použit web `www.github.com`, přesné adresy jednotlivých projektů:

Zdrojové kódy jádra vyhledávače se nachází na adrese

- `https://github.com/havrlant/fca-search`

Dokumentace, popis všech možností nastavení a popis API jádra aplikace se nachází na wiki stránce

- `https://github.com/havrlant/fca-search/wiki/dokumentace`

Zdrojové kódy webového rozhraní se nachází na adrese

- `https://github.com/havrlant/fca-seach-web`

Dokumentace, popis všech možností nastavení a popis API webového rozhraní se nachází na wiki stránce

- `https://github.com/havrlant/fca-seach-web/wiki/dokumentace`

Zdrojového kódy ukázky práce s API se nachází na adrese:

- `https://github.com/havrlant/fca-seach-api-test`

Zdrojové kódy tohoto textu se nachází na adrese:

- `https://github.com/havrlant/fca-search-text`

## 5.4. Podobné vyhledávače

V historii již existovaly vyhledávače, které fungovaly velice podobně jako CLaSeek. Jedná se o vyhledávače „CREDO“, „FooCA“ a „SearchSleuth“. V současné době CREDO a SearchSleuth nefungují vůbec a k použití FooCA je nutné zažádat o registraci tvůrce tohoto vyhledávače.

**CREDO** Zkratka pochází z „Conceptual REorganization of DOcuments“. Jeho autory jsou C. Carpineto a G. Romano. CREDO spolupracovalo s Googlem. Uživatel vložil do vyhledávače dotaz, CREDO tento dotaz nejprve poslalo pomocí SOAP API do Googlu, nechalo si vrátit prvních 100 výsledků a poté nad těmito daty provedlo FCA analýzu. CREDO počítalo pouze dolní sousedy a to do druhé úrovně. Začínalo se u největšího konceptu. Smyslem bylo, abych z těch sto výsledků, které dostal na začátku, postupně ukazoval uživateli ty odkazy, které odpovídají nějakému konceptu.

CREDO poté zobrazilo uživateli seznam odkazů spolu se stromovou strukturou návrhů na změnu dotazu. Tyto návrhy ale nebyly interaktivní, pouze po kliknutí na návrh se z té stovky vrácených dotazů vyfiltrovaly ty, které odpovídaly novému dotazu. Žádný nový dotaz do Google neproběhl. Detailnější informace jsou v článku [9].

**FooCA** Název vznikl ze spojení FCA a Google. Autorem je Bjoern Koester. Vyhledávač funguje na adrese `http://fooca.webstrategy.de`, ale vyžaduje registraci. FooCA, podobně jako CREDO, spolupracuje s webovými vyhledávači, konkrétně s Googlem a Yahoo. Uživatel vloží dotaz do FooCA, ten přepošle dotaz beze změny do Google API nebo do Yahoo API a dále pracuje s navrácenými výsledky. Po skončení FCA analýzy zobrazí uživateli výsledek ve formě tabulky – kontextu – nebo ve formě diagramu znázorňující konceptuální svaz.

Detailnější informace jsou v článku [13].

**SearchSleuth** je nejvíce podobný vyhledávači popsanému v této práci. Jeho autory jsou J. Ducrou a P. Eklund. SearchSleuth spolupracoval s Yahoo a jeho API. Po obdržení výsledků od vyhledávače sestaví kontext, nalezne koncept dotazu a znovu vyhledá ve vyhledávači obsah intentů horních sousedů, čímž SearchSleuth získá více výsledků a může sestavit rozšířený kontext. CLaSeek toto provádí pomocí dodatečného OR dotazu. SearchSleuth také do kontextu nepřidává klíčová slova z dotazu. Jinak je FCA část obou vyhledávačů velice podobná.

Detailnější informace jsou v článku [11] a [10]. Podobný systém byl také použit například pro analýzu CHM souborů, proprietárního formátu Microsoftu pro nápovědu. Více informací v [17].

# Závěr

Výsledkem této práce je prototyp webového vyhledávače CLaSeek, který primárně pracuje nad statickou sadou dokumentů. Umí prohledávat několik typů dokumentů, automaticky z dokumentů získávat název a popisek, zvládá booleovské dotazy, kontroluje překlepy a výsledné dokumenty vrací seřazené dle relevance.

Po navrácení výsledných dokumentů provádí jejich analýzu a prostřednictvím formální konceptuální analýzy nabízí tři skupiny návrhů na úpravu dotazu: konkrétnější dotazy, obecnější dotazy a podobné dotazy. CLaSeek tak pomáhá uživateli upravovat dotaz tak dlouho, dokud není spokojen s výsledky vyhledávání.

CLaSeek dále poskytuje veřejné webové API, díky kterému je možné vyhledávači zasílat vlastní data. Vyhledávač pak sestaví dočasný index a provede nad zaslanými daty stejnou analýzu, jako v případě statického indexu.

Kvalita výsledků je přímo úměrná kvalitě zaindexovaných dokumentů. Pokud do vyhledávače vložíme obsahově bohaté dokumenty, jsou návrhy většinou dobré. Celá analýza je poměrně rychlá, CLaSeek pro daný dotaz vrátí všechny výsledky obvykle za několik sekund. Všechny kódy jsou volně přístupné pod BSD licencí a vyhledávač tak lze případně nainstalovat na vlastní server.

Další výzkum může proběhnout v části, která zpracovává samotné dokumenty. Například vylepšit algoritmy, které hledají atributy dokumentů, zdokonalit stemmer, zjišťovat synonyma a podobně. V části hledající návrhy je možné zavést fuzzy FCA, kdy do formálního kontextu neuložíme pouze informace o tom, zda dokument dané slovo obsahuje, ale i informaci o tom, jak moc je slovo pro dokument důležité. Je možné také rozšířit API, které CLaSeek nabízí a nad tímto API postavit další externí služby.

# Conclusions

CLaSeek is a prototype of a web search engine that works with static set of documents. It can process several types of documents, extracts a title and a description of a document, handles Boolean queries, checks typing errors and the result set of documents is ordered by relevance.

After retrieving the result set of documents, formal concept analysis is performed. We try to find three types of query suggestions: generalisations, specialisations and categorisation. CLaSeek helps user to refine query until he is satisfied with the result.

CLaSeek provides a public web API. It can be used to post your own data. The search engine builds up a temporary index and then formal concept analysis is performed to obtain query suggestions.

The quality of results is determined by the quality of the documents. If we put into the search engine documents with rich and good content we usually obtain good results. The analysis is rather quick. CLaSeek returns results for a single query in a few seconds. The search engine is open source software under the BSD licence so everyone can use it on their own server.

Further research can be done in the following areas: document processing like improve attributes searching, improve stemmer, search for synonyms, etc. We can use fuzzy FCA instead of crisp FCA. We can use information about word importance to build a new fuzzy formal context. We can extend API and create some new applications that use it.

# Reference


[1] Module difflib :: Class SequenceMatcher. 2008, [Online].
URL http://epydoc.sourceforge.net/stdlib/difflib.sequencematcher-class.html

[2] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN 0-201-10088-6.

[3] Baeza-Yates, R. A.; Ribeiro-Neto, B.: *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN 020139829X.

[4] Bělohlávek, R.: Introduction to Formal Concept Analysis. 2008.
URL http://phoenix.inf.upol.cz/esf/ucebni/formal.pdf

[5] Bělohlávek, R.; Outrata, J.: Improving web search with FCA.
URL http://phoenix.inf.upol.cz/~outrata/download/texts/fcawebsearch-slides.pd

[6] Bělohlávek, R.; Outrata, J.: Relational Data Analysis 1. 2008.
URL http://phoenix.inf.upol.cz/esf/ucebni/rda_fca_i.pdf

[7] Bělohlávek, R.; Outrata, J.: Relational Data Analysis 2. 2008.
URL http://phoenix.inf.upol.cz/esf/ucebni/rda_fca_ii.pdf

[8] Bělohlávek, R.; Outrata, J.: Relational Data Analysis 3. 2008.
URL http://phoenix.inf.upol.cz/esf/ucebni/rda_fca_iii.pdf

[9] Carpineto, C.; Romano, G.: Exploiting the Potential of Concept Lattices for Information Retrieval with CREDO. *j-jucs*, aug 2004: s. 985–1013.
URL http://www.jucs.org/jucs_10_8/exploiting_the_potential_of

[10] Dau, F.; Ducrou, J.; Eklund, P.: Concept Similarity and Related Categories in Information Retrieval using Formal Concept Analysis. In *International Journal of General Systems*, Taylor & Francis Group, 2011.

[11] Ducrou, J.; Eklund, P.: SearchSleuth: The Conceptual Neighbourhood of an Web Query. In *CLA'07*, 2007.

[12] Ganter, B.; Wille, R.: *Formal Concept Analysis: Mathematical Foundations*. Berlin/Heidelberg: Springer, 1999.

[13] Koester, B.: Conceptual Knowledge Retrieval with FooCA: Improving Web Search Engine Results with Contexts and Concept Hierarchies. *Advances in Data Mining*, 2006: s. 176–190.
URL http://dx.doi.org/10.1007/11790853_14

[14] Lindig, C.: Fast Concept Analysis. In *Working with Conceptual Structures - Contributions to ICCS 2000*, Shaker Verlag, August 2000, s. 152–161.

[15] Manning, C. D.; Raghavan, P.; Schtze, H.: *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008, ISBN 0521865719, 9780521865715.

[16] Wille, R.: Restructuring lattice theory: an approach based on hierarchies of concepts. In *Ordered sets*, editace I. Rival, Dordrecht–Boston: Reidel, 1982, s. 445–470.

[17] Wormuth, B.; Eklund, P.: Restructuring Help Systems using Formal Concept Analysis. In *3rd Int. Conference on Formal Concept Analysis*, LNAI3403, Springer Verlag, 2005, s. 129–144.

# A. Obsah přiloženého CD

Popis struktury přiloženého CD:

**bin/** Obsahuje dva archivy – `fca-search.zip` a `fca-search-web.zip`. V prvním jsou soubory nutné ke spuštění jádra vyhledávače. Aplikace je napsaná v Pythonu, což je interpretovaný jazyk, takže k jejímu spuštění je potřeba interpret Pythonu. archiv neobsahuje žádný spustitelný binární soubor. Jak nainstalovat a používat aplikaci je popsáno v dokumentaci, viz 5.3.

Druhý archiv, `fca-search-web.zip`, obsahuje soubory zajišťující běh webového rozhraní. Aplikace je psána v PHP. Jak nainstalovat a používat aplikaci je opět popsáno v dokumentaci.

**doc/** Adresář obsahuje text diplomové práce ve formátu PDF a soubory nutné k jejímu sestavení, tj. zdrojový soubor ve formátu LaTeX a několik obrázků.

**src/** Adresář obsahuje zdrojové kódy aplikace. Protože aplikace nemá žádné spustitelné soubory, nachází se zde stejné archivy jako v adresáří `bin/`.

**readme.txt** V tomto souboru je stručný popis toho, kde je současná verze nainstalována, jak ji používat a odkazy na dokumentaci.