Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5220-5783

Bc. Juraj Vincúr

Identifikácia tém zdrojového kódu

Diplomová práca

Študijný program: Softvérové inžinierstvo
Študijný odbor: 9.2.5 Softvérové inžinierstvo
Miesto vypracovania: Ústav informatiky a softvérového inžinierstva, FIIT STU Bratislava
Vedúci práce: Ing. Ivan Polášek, PhD.

máj 2015

STU FIIT
SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Zadanie diplomovej práce

Meno študenta: **Bc. Juraj Vincúr**

Študijný program: Softvérové inžinierstvo

Študijný odbor: Softvérové inžinierstvo

Názov práce: **Identifikácia tém zdrojového kódu**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:

Všeobecný cieľ:

Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.

Špecifický cieľ:

Vytvorte riešenie zodpovedajúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.

Riaďte sa pokynmi Vášho vedúceho.

Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti, alebo o priebežné výsledky Vášho riešenia, alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnite zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav informatiky a softvérového inžinierstva FIIT STU v Bratislave

Vedúci práce: **Ing. Ivan Polášek, PhD.**

Termíny odovzdania:

podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu

Predmety odovzdania:

V každom predmete dokument podľa pokynov na www.fiit.stuba.sk v časti:
home > Informácie o > štúdiu > organizácia štúdia > diplomový projekt.

V Bratislave dňa 17. 2. 2014

SLOVENSKÁ TECHNICKÁ UNIVERZITA * Fakulta informatiky a informačných technológií * BRATISLAVA

prof. Ing. Pavol Návrat, PhD.
riaditeľ Ústavu informatiky a softvérového inžinierstva

STU FIIT
SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY
A INFORMAČNÝCH TECHNOLÓGIÍ

Návrh zadania diplomovej práce

Finálna verzia do diplomovej práce [1]

Študent:

Meno, priezvisko, tituly:	Juraj Vincúr, Bc.
Študijný program:	Softvérové inžinierstvo
Kontakt:	jur0.bg@gmail.com

Výskumník:

Meno, priezvisko, tituly:	Ivan Polášek, Ing. PhD.

Projekt:

Názov:	Identifikácia tém zdrojového kódu
Názov v angličtine:	Source code topics identifying
Miesto vypracovania:	Ústav informatiky a softvérového inžinierstva, FIIT STU, Bratislava
Oblasť problematiky:	získavanie informácií

Text návrhu zadania[2]

Pri vývoji softvéru je proces znovupoužitia existujúceho zdrojového kódu dôležitý a napomáha k rýchlejšej tvorbe nových programov. Aby programátor vedel aplikovať vhodný existujúci fragment kódu, musí ho najskôr lokalizovať a porozumieť mu. Pre čo najrýchlejšiu orientáciu v zdrojovom kóde napomáha jeho samotná štruktúra, ale pre vytvorenie si celkového obrazu o programe je vhodné zobrazenie jednotlivých častí implementácie do logických celkov, t.j. tém, na základe ich významu. Efektivita tohto zobrazenia závisí od kvality výsledkov a času potrebného na ich získanie.

Analyzujte súčasný stav v problematike automatizovaného zhlukovania dokumentov do jednotlivých tematických celkov. Zamerajte sa na riešenia umožňujúce dodatočné pridávanie dát do modelu. Porovnajte existujúce prístupy najmä z pohľadu výpočtovej efektivity a kvality výsledkov. Preskúmajte možnosti predspracovania zdrojového kódu a vhodnej reprezentácie výsledkov.

Navrhnite a implementujte novú, výpočtovo efektívnu metódu zhlukovania častí softvéru, založenú na vhodnej kombinácii existujúcich riešení s algoritmom inšpirovaným hmyzími spoločenstvami a s využitím konceptu aspektovo orientovaného prístupu.

Overte kvalitu riešenia problému experimentom a dosiahnuté výsledky overte komparatívnou analýzou s mapou obsahu (zoznamom tém) získanou pomocou už známych metód zhlukovania.

---

[1] Vytlačiť obojstranne na jeden list papiera

[2] 150-200 slov (1200-1700 znakov), ktoré opisujú výskumný problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

Literatúra[3]

- VAIJAYANTHI, Priya, NATARAJAN, AM, MURUGADOSS, Raja. Ants for Document Clustering. In: International Journal of Computer Science Issues. 2012, vol. 9, no. 2, s. 493 – 499.
- YANG, Yan, KAMEL, Mohamed S. An aggregated clustering approach using multi-ant colonies algorithms. In: Pattern Recognition. 2006, vol. 39, no. 7, s. 1278 – 1289.

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval(a) Bc. Juraj Vincúr, konzultoval(a) a osvojil(a) si ho Ing. Ivan Polášek, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 3.2.2014

Podpis študenta

Podpis výskumníka

Vyjadrenie garanta predmetov Diplomový projekt I, II, III

Návrh zadania schválený: áno / ~~nie~~[4]

Dňa: 14. 2. 2014

Podpis garanta predmetov

[3] 2 vedecké zdroje, každý v samostatnej rubrike a s údajmi zodpovedajúcimi bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

[4] Nehodiace sa prečiarknite

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo

Autor: Juraj Vincúr, Bc.
Diplomový projekt: Identifikácia tém zdrojového kódu
Vedenie diplomového projektu: Ivan Polášek, Ing. PhD.
December, 2014

Dokument je zameraný na analýzu súčasného stavu zhlukovacích algoritmov inšpirovaných hmyzími spoločenstvami. Sústredíme sa najmä na algoritmus mravčej kolónie. Poskytujeme prehľad modelov a princípov najzaujímavejších implementácií.

Venujeme sa všetkým etapám procesu zhlukovania od spracovania dokumentov až po určovanie jednotlivých zhlukov. Viac priestoru venujeme analýze metód redukcie priestoru, ktoré sú hlavným faktorom v otázke zvýšenia efektivity algoritmov, či už z pohľadu výpočtových alebo pamäťových nárokov.

V snahe dosiahnuť čo najlepšie výsledky v čo najkratšom čase, smerujeme poznatky analýzy k návrhu online zhlukovacieho algoritmu, ktorý je schopný aktualizovať už existujúci model bez jeho prepočítavania.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software engineering

Author: Juraj Vincúr, Bc.
Diploma Thesis: Source code topics identifying
Supervisor: Ivan Polášek, Ing. PhD.
2014, December

In this thesis we analyze common clustering algorithms inspired by swarm intelligence. We focus on Ant Colony Optimization, providing an overview of most interesting implementations.

During the analysis, we consider each step of clustering process, devoting extra space to dimensionality reduction methods that we consider as major factor in terms of performance optimization.

In order to achieve the best performance, we bias our efforts towards proposition of online clustering method, capable of updating existing model by new observations without recalculation.

Čestné prehlásenie

Prehlasujem, že som diplomovú prácu vypracoval samostatne s využitím uvedených zdrojov literatúry.

V Bratislave, dňa 14. mája 2015 ……..

Juraj Vincúr

Obsah

1 Úvod

Pri vývoji softvéru je proces znovupoužitia existujúceho zdrojového kódu dôležitý a napomáha k rýchlejšej tvorbe nových programov. Aby programátor vedel aplikovať vhodný existujúci fragment kódu, musí ho najskôr lokalizovať a porozumieť mu. Nestačí však chápať iba vybrané časti kódu. Naopak, určitá miera porozumenia celkovej štruktúry je nevyhnutná, aby si programátor uvedomoval, aký dopad bude jeho zmena predstavovať pre softvérový systém ako celok. Pre čo najrýchlejšiu orientáciu v zdrojovom kóde napomáha jeho samotná štruktúra, ale pre vytvorenie si celkového obrazu o programe je vhodné zobrazenie jednotlivých častí implementácie do logických celkov, t.j. tém, na základe ich významu.

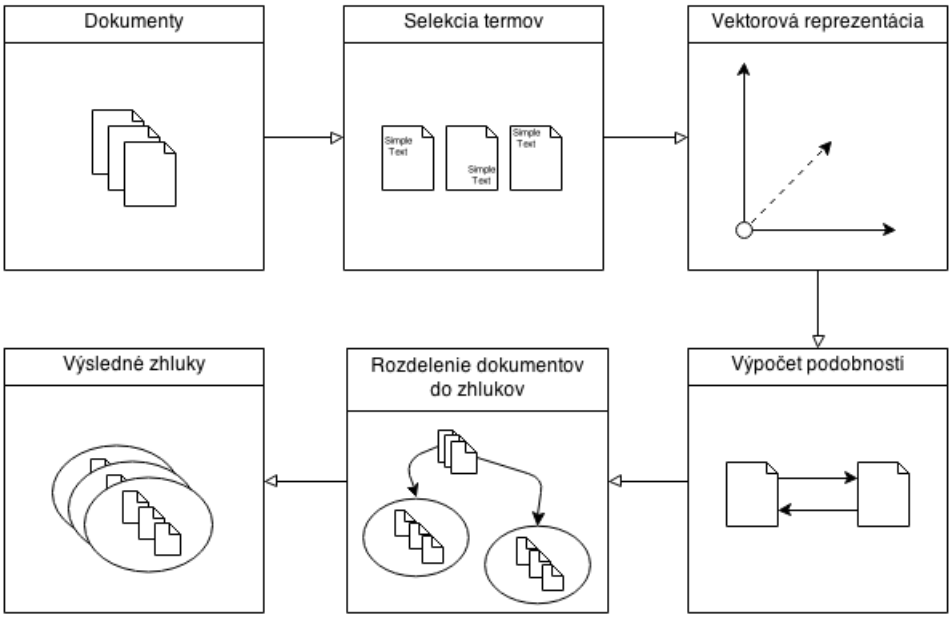
Organizáciu týchto častí zdrojového kódu do tém možno docieliť pomocou *zhlukovania*. Zhlukovanie možno definovať ako proces klasifikácie jednotlivých pozorovaní do skupín (zhlukov) podľa ich vzájomnej podobnosti. Dvojice pozorovaní v rámci jedného zhluku majú byť podobné vo vysokej miere, zatiaľ čo dvojice pozorovaní rôznych zhlukov majú dosahovať hodnoty podobností výrazne nižšie. Pozorovanie môže v doméne zdrojových kódov reprezentovať rôzne typy komponentov (napr. triedy, funkcie, metódy) v závislosti od zvolenej granularity. V prípade dekompozície objektovo-orientovaných systémov, považujeme za najvhodnejšiu granularitu na úrovni tried, zatiaľ čo v kontexte lokalizácie určitej funkcionality uprednostňujeme granularitu na úrovni funkcií a metód.

Vzhľadom na periodické zmeny softvérových systémov (pridávanie funkcionality, odstraňovanie chýb, refaktoring, ...) považujeme tieto množiny dát za dynamické. V týchto prípadoch je z hľadiska výpočtovej zložitosti vhodnejšie aplikovať inkrementálne prístupy, ktoré sú schopné modifikovať už získané analýzy bez opätovného spracovania predošlých pozorovaní. Výpočtová zložitosť predstavuje kritický aspekt dekompozície, nakoľko rozhodovanie na základe neaktuálnej abstrakcie softvérového systému môže viesť k degradácii jeho štruktúry.

V tejto práci sa venujeme doméne zdrojových kódov, jej vlastnostiam a možným aplikáciám inkrementálnych prístupov vo fáze predspracovania, vytvárania zhlukov, ale aj vizualizácie. Taktiež sme navrhli a implementovali nový prístup zhlukovania, založený na správaní mravčej kolónie, zameraný na získavanie prirodzených zhlukov.

2 Zhlukovanie

Základná definícia zhlukovania (z angl. clustering) podľa Brian Everittet al. [1] znie nasledovne: Nech je daný určitý počet objektov, ktoré sú opísané množinou numerických hodnôt. Zhlukovanie spočíva v navrhnutí schémy pre zoskupovanie objektov do určitého počtu tried, kde objekty patriace jednej triede sú podobné na základe určitej miery a rozdielne voči objektom druhej triedy.



Obrázok 1 - zhlukovanie.

Celkový proces zhlukovania, vrátane fázy predspracovania, možno rozdeliť do týchto krokov (viď.Obrázok 1):

1. Extrakcia dokumentov zo vstupných dát
2. Selekcia termov z dokumentov
3. Vytvorenie vektorovej reprezentácie
4. Určenie vzájomnej podobnosti dokumentov
5. Rozdelenie dokumentov do zhlukov
6. Určenie výsledných zhlukov

2.1. Selekcia termov

Vymedzenie základných pojmov:

- *Token* – sekvencia po sebe idúcich znakov v určitom dokumente.
- *Typ* - súbor všetkých tokenov pozostávajúcich z rovnakej sekvencie znakov.
- *Term* - normalizovaný typ.

Príklad a porovnanie jednotlivých základných pojmov ilustruje Tabuľka 1.

Text	I tried to speak Spanish, and my friend tried to speak English.
Tokeny	{ I, tried, to, speak, Spanish, and, my, friend, tried, to, speak, English }
Typy	{ I, tried, to, speak, Spanish, and, my, friend, English }
Termy	{ try, speak, spanish, friend, english }

Tabuľka 1- príklad vzťahov text, token, typ, term

Proces selekcie termov v prirodzenom jazyku pozostáva najčastejšie z troch hlavných operácií v tomto poradí: *tokenizácia, odstránenie stop slov a normalizácia tokenov* (viď. Obrázok 2 vľavo).

2.1.1. Tokenizácia

Rozdelenie sekvencie znakov na jednotlivé časti, tokeny. Identifikátory v zdrojovom kóde často pozostávajú z viacerých slov, ktoré opisujú akcie a entity danej časti implementácie[2]. Spolu s komentármi tvoria najlepší indikátor sémantického obsahu. Na základe konvencií, bývajú tieto slovné spojenia najčastejšie označené jedným z nasledujúcich príznakov:

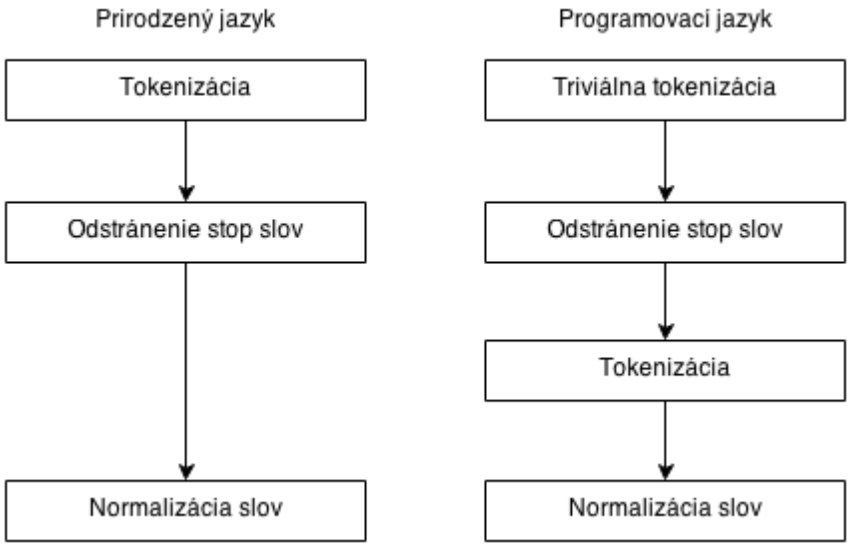
- *camelcase* – označenie spojenia slov striedaním malého a veľkého písma (prvéDruhéSlovo)
- *deliaci znak* – oddelenie jednotlivých slov deliacim znakom (prvé-druhé, prvé_druhé)

Avšak nezanedbateľné množstvo identifikátorov (v priemere 15%) [2], nezodpovedá týmto konvenciám a ich tokenizácia vyžaduje náročnejšie spracovanie. Robustné riešenie tohto problému bolo prezentované v práci [2].

2.1.2. Stop slová

Dokumenty môžu obsahovať častý výskyt slov nereprezentujúcich ich sémantický obsah, čoho dôsledkom je vyššia pravdepodobnosť nesprávneho vyhodnotenia ich podobnosti. V prípade zdrojových kódov ide napr. o kľúčové slová daného programovacieho jazyka. Za

účelom ich eliminácie sa do procesu získavania termov zahŕňa filtrovanie takzvaných stop slov. Ide o zoznam vopred definovaných slov, ktorých výskyt sa ignoruje.



Obrázok 2 - Proces selekcie termov.

Používanie stop slov v niektorých prípadoch vedie k odstráneniu sémanticky dôležitých tokenov, tiež známe ako problém stop slov (z angl. stop wordsproblem) [3]. Tomuto správaniu možno v doméne programovacích jazykov jednoducho predísť nasledovným pozmenením operácií procesu selekcie termov (viď. Obrázok 2 vpravo):

1. *Triviálna tokenizácia* – rozdelenie na tokeny len z lexikálneho hľadiska daného jazyka (prázdne znaky, interpunkcia, ...).
2. *Odstránenie stop slov* – v tomto prípade teda skutočne odstraňujeme len tokeny, ktoré reprezentujú slová vymedzené pre daný jazyk.
3. *Tokenizácia* – následná tokenizácia zložených identifikátorov.
4. *Normalizácia tokenov.*

2.1.3. Normalizácia slov

Cieľom normalizácie tokenov je redukcia rôznych slovných tvarov na základnú spoločnú reprezentáciu. Existujú dva hlavné prístupy:

- *Stemovanie* – heuristické odstraňovanie sufixov a prefixov, najčastejšie na základe sady pravidiel definovaných pre daný jazyk.

- *Lematizácia* – získavanie slovného základu, lemy, zo všetkých odvodených slov, pomocou slovníka daného jazyka.

V prípade stemovania dochádza k dvom hlavným problémom [4]:

- *Podstemovanie* (z angl. understemming) - slová odvodené od rovnakého základu nie sú tomuto základu spätne priradené. Príkladom v anglickom jazyku pre Porterov algoritmus sú slová dentist a dentistry.
- *Prestemovanie* (z angl. overstemming) - slová odvodené od rôznych základov sú priradené tomu istému základu. Príkladom pre tento prípad sú slová illegal a illegible, ktoré Porterov algoritmus upraví na základ illeg.

Lematizácia tieto problémy eliminuje prítomnosťou slovníka slov daného jazyka, ktorý každému tvaru priraďuje korektný slovný základ. Ide o presnejší, no zároveň časovo aj pamäťovo náročnejší proces.

Problém pri lematizácií nastáva pri homonymách. Príkladom je anglické slovo saw, ktoré môže byť použité ako sloveso píliť alebo ako minulý čas od slovesa see, vidieť.

2.2. Model vektorového priestoru

Model vektorového priestoru (z angl. Vector Space Model, ďalej VSM)je jedna z často používaných techník získavania informácií (z angl. Information Retrieval), reprezentujúca dokumenty prostredníctvom vektorov. Pre dokument d_i je t-rozmerný vektor v danom priestore definovaný nasledovne:

$$\overrightarrow{d_i} = (w_{i1}, w_{i2}, \dots, w_{ij}, \dots, w_{it})$$

kde w_{ij} predstavuje váhu termut$_j$ v dokumente d_i. Maticová reprezentácia kolekcie takýchto vektorov sa nazýva *term-dokument matica* (viď.Tabuľka 2).

Podobnosť dokumentov v tomto priestore je daná kosínusom uhla, ktorý ich vektory zvierajú, známe tiež ako kosínusová podobnosť (z angl. Cosine Similarity).

Dok / Term	1	2	3	4
1	w_{11}	w_{21}	w_{31}	w_{41}
2	w_{12}	w_{22}	w_{32}	w_{42}
3	w_{13}	w_{23}	w_{33}	w_{43}
4	w_{14}	w_{24}	w_{34}	w_{44}

Tabuľka 2 - príklad term-dokument matice s vyznačenou váhou pre term 2 v dokumente 3

2.2.1. Reprezentácia dokument term matice

Náhodne vybraný projekt JHotDraw verzie 7.0.6 obsahuje 355 deklarácií typov a 1335 unikátnych termov. Dokument term matica pre danú kolekciu obsahuje 17 664 nenulových prvkov, čo predstavuje hustotu matice približne len 3.73%. Podobné hodnoty hustôt možno bežne pozorovať v praxi. Tieto typy matíc sú označované ako riedke matice. Ich vhodnou reprezentáciou možno znížiť pamäťové aj výpočtové nároky.

Vybrané reprezentácie riedkych matíc:

- *LIL formát* (z angl. List of Lists format) - Riadok je reprezentovaný dvoma listami. Jeden uchováva zoradené indexy jednotlivých stĺpcov a druhý samotné hodnoty nenulových prvkov. Tieto listy sú uložene v ďalšom liste, ktorý reprezentuje celý korpus. Táto reprezentácia sa najčastejšie používa, pokiaľ chceme inkrementálne pridávať riadky. Umožňuje efektívne pristupovanie k riadkom, no nie k stĺpcom. Zároveň je pomalá pri aritmetických výpočtoch.
- *DOK formát* (z angl. Dictionary of Keys format) - Korpus je reprezentovaný slovníkom s kľúčmi (riadok, stĺpec). Vhodná reprezentácia, pokiaľ chceme meniť prvky, riadky, stĺpce alebo k nim často pristupovať. Aritmetické výpočty nad týmto formátom sú tiež pomalé.
- *COO formát* (z angl. Coordinate format) - Formát reprezentuje korpus tromi rovnako dlhými poľami. Prvé pole obsahuje indexy riadkov, druhé indexy stĺpcov a tretie hodnoty nenulových prvkov. Spárovaním prvkov prvých dvoch polí získame x, y

koordináty v reprezentovanej matici pre prislúchajúci nenulový prvok. Neumožňuje prístup k prvkom, riadkom, stĺpcom a ani aritmetické operácie. Ako jediný formát umožňuje duplicitné prvky, ktoré sú pri konverzií do iných formátov sčítané.

- *CSR formát* (Compressed Sparse Row format) - Reprezentácia pomocou troch polí. Jedno uchováva hodnoty nenulových prvkov, druhé indexy nenulových prvkov v rámci jedného riadka a tretie určuje rozsahy prvkov z druhého poľa podľa toho, ktorým riadkom prislúchajú.

```
dáta =    [1,2,3,4,5,6]
indexy =  [0,2,2,0,1,2]
rozsahy = [0,2,3,6]
```

  Na základe poľa rozsahy vieme, že prvky poľa od indexu 0 až po 2 prislúchajú prvému riadku, prvky od 2 až po 3 prislúchajú druhému riadku a prvky 3 až 6 tretiemu.

```
matica =  [1,0,2]
          [0,0,3]
          [4,5,6]
```

  Táto reprezentácia umožňuje rýchle výpočty a pristupovanie k riadkom. Nie je však vhodná, pokiaľ chceme maticu meniť.

- *CSC formát* (compressed sparse column format) - Rovnaký prístup ako csr, ale stĺpcovo orientovaný. Pole indexov určuje, v ktorom riadku sa nachádza daný prvok a nie v ktorom stĺpci. Pole rozsahov určuje, ktoré prvky prislúchajú ktorým stĺpcom, nie riadkom.

```
dáta =    [1,4,5,2,3,6]
indexy =  [0,2,2,0,1,2]
rozsahy = [0,2,3,6]
matica =  [1,0,2]
          [0,0,3]
          [4,5,6]
```

2.2.2. Zhodnotenie

Identifikovali sme viacero možných formátov na reprezentáciu dokument term matice. Každý z nich prináša svoje výhody a nevýhody. Ani jeden z formátov nie je vhodný na aritmetické výpočty a efektívne vykonávanie zmien v štruktúre súčasne, preto je vhodné použiť kombináciu dvoch. Dok formát ako základnú reprezentáciu používanú pri vytváraní matice, jej ukladaní a načítavaní. Pred vykonávaním samotných výpočtov konvertujeme

túto maticu do csr alebo csc formátu. Konverzia medzi jednotlivými formátmi je vykonaná v lineárnom čase, ktorý je zanedbateľný.

2.3. Váhovanie termov

Váhovanie je proces priradenia numerickej hodnoty, váhy, každému kľúčovému slovu, termu, získaného z korpusu. V doméne zdrojových kódov možno rozlíšiť dva hlavné prístupy:

- váhovanie založené na vnútornej štruktúre
- váhovanie založené na frekvenciách výskytu termov

2.3.1. Frekvenčné váhovanie

Najčastejšie používaný prístup adaptovaný z domény textových dokumentov. Ako z názvu vyplýva, váha termu je určená počtom jeho výskytov v dokumente (ďalej $f_{t,d}$). Pokiaľ túto hodnotu predelíme dĺžkou dokumentu, získame normalizovaný variant váhy s označením TF (z angl. Term Frequency). TF spadá do kategórie lokálnych váh, pretože na jej výpočet sú nevyhnutné len údaje o jednom dokumente. V prípade, že používame informácie o kolekcií dokumentov, ide o váhu globálnu. Príkladom je IDF (z angl. Inverse Document Frequency). Hodnota základnej formy IDF pre daný term je odvodená od celkového počtu dokumentov N a od počtu jeho výskytov v kolekcií D (ďalej fd_t). Kombináciou TF a IDF získavame váhovanie TF-IDF. Ak sa kolekcia D v čase mení, pridávaním, odstraňovaním alebo upravovaním jednotlivých dokumentov, mení sa aj IDF a celý TF-IDF model je nutné prepočítať. Jedným z riešení tohto problému je TF-ICF (z angl. Term Frequency – Inverse Corpus Frequency), kde výpočet globálnych váh je realizovaný na základe inej, rozsiahlejšej a statickej kolekcie podobných dokumentov (rovnaká doména). Ďalší, v princípe veľmi podobný prístup prezentovaný v práci [29], využíva pri výpočte globálnej váhy hodnoty špecifickosti a všeobecnosti daného termu, ktoré boli určené na základe lexikálnej ontológie pre anglický jazyk.

Odlišný spôsob váhovania termov poskytuje pravdepodobnostný model s názvom Okapi BM25. Ide o jednu z najlepších vážiacich funkcií. V rámci výpočtu sú zavedené parametre k_1 a b, ktoré slúžia na škálovanie počtu výskytu kľúčových slov a dĺžky dokumentu, v

ktorom sa nachádzajú. Experimenty ukázali, že hodnoty týchto parametrov $b = 0.75$ a $k_1 \in \langle 1.2, 2 \rangle$ dosahujú najlepšie výsledky.

Označenie	Vzorec	Myšlienka
TF	$TF_{t,d} = f_{t,d}/len(d)$	Dlhé dokumenty majú vyššiu pravdepodobnosť výskytu daného termu.
IDF	$IDF_{t,D} = log\frac{N}{fd_t}$	Termy, ktoré sa nachádzajú vo veľkom počte dokumentov, nie sú natoľko informatívne.
TF-IDF	$TFIDF_{t,d} = TF_{t,d} \times IDF_{t,D}$	Kombinácia oboch myšlienok.
BM25	$BM25_{t,d} = \frac{(k_1+1)f_{t,d}}{f_{t,d} + k_1(1 - b + b(\frac{len(d)}{avglen(D)})} \times IDF_{t,D}$	Zmena lokálnej váhy, prihliada sa aj na priemernú dĺžku dokumentu v kolekcií.

2.3.2. Štrukturálne váhovanie

Zdrojový kód predstavuje istú formu komunikácie, kde jednu stranu komunikačného kanála predstavuje vývojár a druhou môže byť stroj, používateľ alebo iný vývojár. V prípade stroja je komunikácia formálna a záleží len na štruktúre a syntaxi. V ostatných prípadoch vývojár nie je viazaný striktnými pravidlami a využíva bohatú slovnú zásobu ako nositeľa sémantickej informácie. Táto sémantická informácia je obsiahnutá v identifikátoroch a komentároch.

2.3.2.1. Komentáre

Fowler v knihe [30] označuje komentáre ako potenciálne pachy, pretože sú často nevhodne používane ako "dezodoranty", ktoré slúžia na "zakrytie" neprehľadného kódu. Zároveň tvrdí, že komentáre je vhodné použiť, keď:

- nevieme, čo robiť
- nie sme si istý tým, čo robíme
- chceme vysvetliť, prečo sme niečo spravili

Vo všetkých vymenovaných prípadoch ide o druh informácie, ktorá pomáha budúcim programátorom dostať sa do obrazu. Toto však nemusí byť druh informácie, ktorý by určoval koncept danej triedy. Naopak sa javí, že "zapáchajúce" použitia komentárov sú pre nás prínosnejšie. Vysvetľujú čo by daný úsek implementácie mal robiť.

V práci [32] autor poskytuje krátke zhrnutie prác a argumentov, ktoré sú pre aj proti používaniu komentárov pri indexovaní zdrojových kódov. Rozdielnosť jednotlivých názorov nás vedie k hypotéze, že najlepším riešením bude používať len komentáre určitého typu alebo priradiť rôzne váhy rôznym typom.

Komentáre v jazyku Java delíme na tri typy (viď.Tabuľka 3):

- *Javadoc* - Javadoc predstavuje štandard písania komentárov pre triedy a metódy. Zároveň umožňuje automatické generovanie dokumentácie, a teda ide aj o externú formu komunikácie. Javadoc komentár má poskytovať všetky informácie, ktoré o entite musíme vedieť na to, aby sme ju vedeli správne použiť. Na základe týchto poznatkov predpokladáme, že tento typ je najčistejším zdrojom dát. Pomocou značiek (z angl. tags) môžeme taktiež dodatočne filtrovať nežiadúce polia ako napríklad autor, verzia alebo parametre.
- *Riadkové* - interná forma, predstavuje najviac zašumené dáta. Vývojári v nich často odôvodňujú svoje zásahy do zdrojového kódu, čo často nijakým spôsobom neurčuje koncept danej entity.
- *Blokové* - taktiež interná forma, predpokladáme nižšie zašumenie. Veľmi časté využitie je zakomentovanie predchádzajúcej verzie výpočtu alebo časti algoritmu.

/** * Creates a parse exception for when an entity could * not be resolved. * * @param name The name of the entity. * * </dl><dl><dt>Preconditions:</dt><dd> * <code>name != null</code> * <code>name.length() > 0</code> * </dd></dl> */	**// Java exception handling suxx** this.parseCharArray(input, offset, end, /*startingLineNr*/ 1);

Tabuľka 3 - komentáre.

2.3.2.2. Identifikátory

V abstraktnom syntaktickom strome sa identifikátory vyskytujú v týchto hlavných kontextoch:

- deklarácia typu
- deklarácia premennej
- deklarácia metódy
- volanie metódy

Z dôvodu prehľadnosti a zrozumiteľnosti kódu sa vývojár snaží určiť meno každej z týchto entít tak, aby čo najpresnejšie určovalo jej koncept, a teda samotný názov považujeme za najreprezentatívnejšieho nositeľa sémantickej informácie. Jeho frekvencia výskytu v kóde však býva relatívne malá. Aj na základe práce [31] predpokladáme, že váhy termov obsiahnutých v týchto názvoch by sme mali zvyšovať. Zároveň usudzujeme, že koeficient zvýšenia váh pre jednotlivé názvy entít by mal byť rôzny v závislosti od toho, v akom kontexte sa nachádzajú. Napríklad deklarovaná metóda určuje správanie triedy priamo (vyšší koeficient), zatiaľ čo volaná metóda určuje správanie len čiastočne (nižší koeficient).

2.3.3. Zhodnotenie

Identifikátory a komentáre patria k hlavným nositeľom sémantickej informácie v zdrojových kódoch. Výskumná práca[31] preukázala, že identifikátory, ktoré sú použité ako názvy deklarovaných metód majú zo sémantického hľadiska vyššiu váhu. Zároveň preukázala, že zvyšovanie váhy pri volaných metódach zvyšuje aj šum v dátach. V našej práci budeme experimentovať s ďalšími dvoma kontextami, v ktorých sa identifikátory vyskytujú a budeme sa snažiť určiť najvhodnejšiu kombináciu jednotlivých škál.

V [32] autor tvrdí, že komentáre obsahujú cenné informácie, ktoré by nemali byť ignorované. Zároveň však táto problematika vyvolala vlnu diskusie a nejednoznačné závery. Našou hypotézou v tomto prípade je, že pravda leží uprostred a filtrovanie len určitých typov komentárov zvýši kvalitu našich výsledkov.

2.4. Podobnosť dokumentov

Dokumenty kolekcie sú najčastejšie reprezentované ako body alebo ako vektory v n-rozmernom priestore. V prípade bodov dokážeme určiť podobnosť na základe ich

vzájomnej vzdialenosti, ktorú vieme vyrátať rôznymi metrikami, ako sú napríklad Minkowského L1 a L2 metriky, známe tiež ako Manhattanská a Euklidova vzdialenosť. Čím väčšia je táto vzdialenosť, tým menšia je podobnosť dokumentov. V prípade vektorov určujeme podobnosť ako kosínus uhla, ktorý tieto dva vektory zvierajú. Čím vyššie hodnoty nadobúda, tým väčšia je podobnosť.

2.4.1. **Manhattanská vzdialenosť**

Metrika, ktorá vracia sumu absolútnych hodnôt rozdielov každej dimenzie dvoch bodov v n-rozmernom priestore (viď. Obrázok 3). Majme body $U = [x_1, x_2, \dots, x_n]$ a $V = [y_1, y_2, \dots, y_n]$. Manhattanova vzdialenosť, MD, týchto dvoch bodov je určená nasledovne:

$$MD(U,V) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|$$

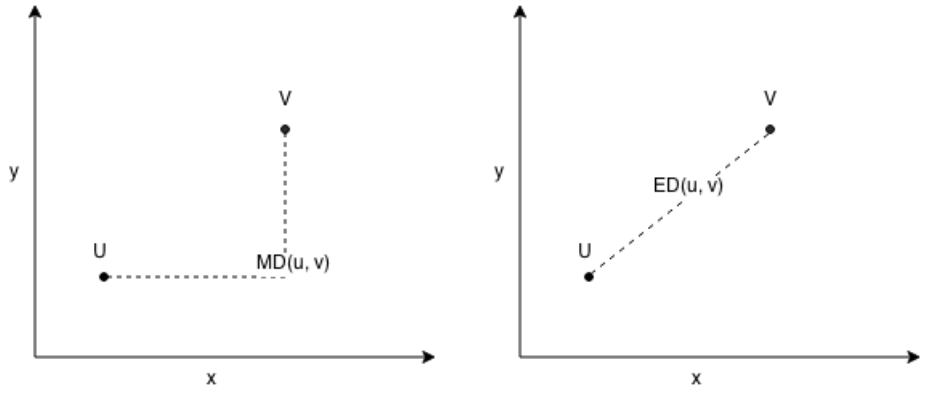
$$MD(U,V) = \sum_{i=1}^{n} |x_i - y_i|$$

2.4.2. Euklidovská vzdialenosť

Znázornená v dvojdimenzionálnom priestore na obrázku (viď. Obrázok 3). Pre tú istú dvojicu bodov U, V je určená vzťahom:

$$ED(U,V) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}$$

$$ED(U,V) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

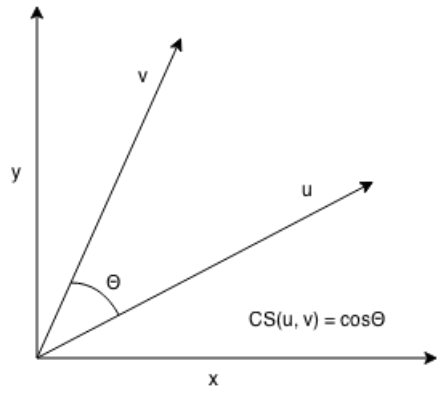


Obrázok 3 - Manhatanská (vľavo) a Euklidova (vpravo) vzdialenosť.

2.4.3. Kosínusová podobnosť

Podobnosť dvoch vektorov $u = (x_1, x_2, \dots, x_n)$ a $v = (y_1, y_2, \dots, y_n)$ reprezentujúcich dokumenty vo vektorovom priestore je daná rovnicou:

$$CS(u, v) = \frac{\sum_{i=1}^{n} x_i \, y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$



Obrázok 4 - Kosínusová podobnosť.

2.4.4. Zhodnotenie

Pri porovnávaní textových dokumentov sa najčastejšie používa kosínusová podobnosť, pretože v danej doméne dosahuje najlepšie výsledky. Táto miera nie je metrikou, avšak tento fakt nás nijako neobmedzuje.

2.5. Obohacovanie vektorov

Krátke dokumenty môžu byť sémanticky podobné aj v prípade, že im zodpovedajúce inštancie vektorového priestoru neobsahujú spoločné prvky. Tento fakt je spôsobený tým, že ten istý koncept môže byť vyjadrený rôznymi slovnými spojeniami. Obohacovanie vektorov sa snaží riešiť tento problém pridávaním synoným, hyperným, hyponým alebo n-gramov. V práci [33] identifikovali tri základné varianty obohacovania:

- *Syntaktické* - pridávanie slov vyskytujúcich sa v identifikovaných n-gramoch, ktoré najlepšie vystihujú daný koncept.

- *Sémantické synonymické* - pre podstatné mená určíme na základe ontológie daného jazyka množiny synoným a pridáme ich do vektora. Tento spôsob pridáva do dát veľa nežiadúceho šumu.
- *Sémantické hypernomické* - term vo vektore je nahradený prvými n hypernymami. Nie natoľko zovšeobecňujúce ako synonymický prístup.

Varianty uvedené vyššie vznikli v doméne textových dokumentov. Druhý a tretí variant spadá do kategórie *ontologického obohacovania*. V doméne zdrojových kódov však koncept nie je určený len v rámci jedného dokumentu. Napríklad triedy môžu správanie dediť od nadtried alebo ho pozmeňovať. Volané metódy v rámci tried môžu byť deklarované v iných triedach. Z týchto dôvodov bolo zavedené aj *štrukturálne obohacovanie*.

2.5.1. Štrukturálne obohacovanie

Vykonávané na základe vzťahov, získaných z AST pomocou väzieb (z angl. bindings). V jazyku Java ide o rozhranie, ktoré reprezentuje entitu (balík, typ, metóda, anotácia...). Súbor týchto rozhraní poskytuje štrukturálnu reprezentáciu programu z pohľadu kompilátora. Vzťahy, ktoré možno identifikovať:

- *Dedenie* - pre každú triedu vieme určiť nadtriedu, od ktorej dedí. To nám umožňuje pridať termy extrahované z nadtriedy do vektora reprezentujúceho danú triedu s určitou váhou. Myšlienkou tohto typu obohatenia je, že nadtrieda dopĺňa správanie (prostredníctvom deklarovaných metód a atribútov), a teda priamo aj koncept triedy. Z toho taktiež usudzujeme, že pri obohacovaní netreba škálovať hodnoty vektora nadtriedy.
- *Implementácia rozhrania* - Väzby nám poskytujú informácie o tom, ktoré rozhrania daná trieda implementuje. Potenciál tohto poznatku vidíme len v názve rozhrania, ktoré však už je obsiahnuté v deklarácii triedy (implements) a v komentároch daného rozhrania.
- *Vnorené triedy* - pri extrakcií sú súčasťou tela triedy a zároveň sú extrahované aj ako samostatný dokument, a teda nevidíme zmysel v ďalšom obohacovaní. Za úvahu by stál len prípad, v ktorom chceme upravovať váhy (pravdepodobne zvyšovať) jednotlivých kľúčových pojmov získaných z vnorenej triedy.

Opačný smer obohacovania vedie k nadmernej eliminácií nulových prvkov (väčšinou vzťah 1:N) za cenu pridávania nežiadúceho šumu, čo má nepriaznivý vplyv na zložitosť výpočtu a kvalitu výsledkov.

2.5.2. Ontologické obohacovanie

Obohacovanie založené na vzťahoch získaných zo slovníka. Môže ísť o slovník pre určitú doménu alebo aj všeobecný slovník. Doména v tomto prípade môže predstavovať napríklad projekt. Takto špecifický slovník nám poskytuje podstatne relevantnejšie dáta, ale je zároveň náročné ho vytvoriť a udržiavať. Z tohto dôvodu sa prikláňame ku všeobecným lexikálnym databázam, ako je napríklad Wordnet. Ide o ontológiu popisujúcu slovnú zásobu anglického jazyka, čo nám v doméne zdrojových kódov len vyhovuje. Každé slovo nadobúda viacero významov. Význam je reprezentovaný množinou synoným (z angl. synset), ktoré definujú koncept. Jednotlivé množiny synoným sú navzájom prepojené nasledujúcimi sémantickými vzťahmi:

- *hyponymá* - špecifickejšia forma konceptu daného slova (pes je hyponymum od zviera)
- *hypernymá* - opak hyponým, abstrakcia konceptu (zelenina je hypernymum od mrkva)
- *antonymá* - slová opačného významu (škaredý je antonymum pre pekný)
- *meronymá* - sémantický vzťah agregácie (kocka je meronymum pre stavebnica)
- *holonymá* - opačný smer vzťahu ako pri meronymách (okno je holonymum pre sklo)

Práca [33] poukazuje na neúčinnosť sémantického synonymického obohacovania vektorov. Pridávanie synoným termom s vysokou váhou v konečnom dôsledku znižuje ich váhu globálnu a pridávanie synoným termom s nízkou váhou zvyšuje mieru zašumenia. Zároveň výsledky tejto práce preukazujú mierne zlepšenie výsledkov pri použití hyperným.

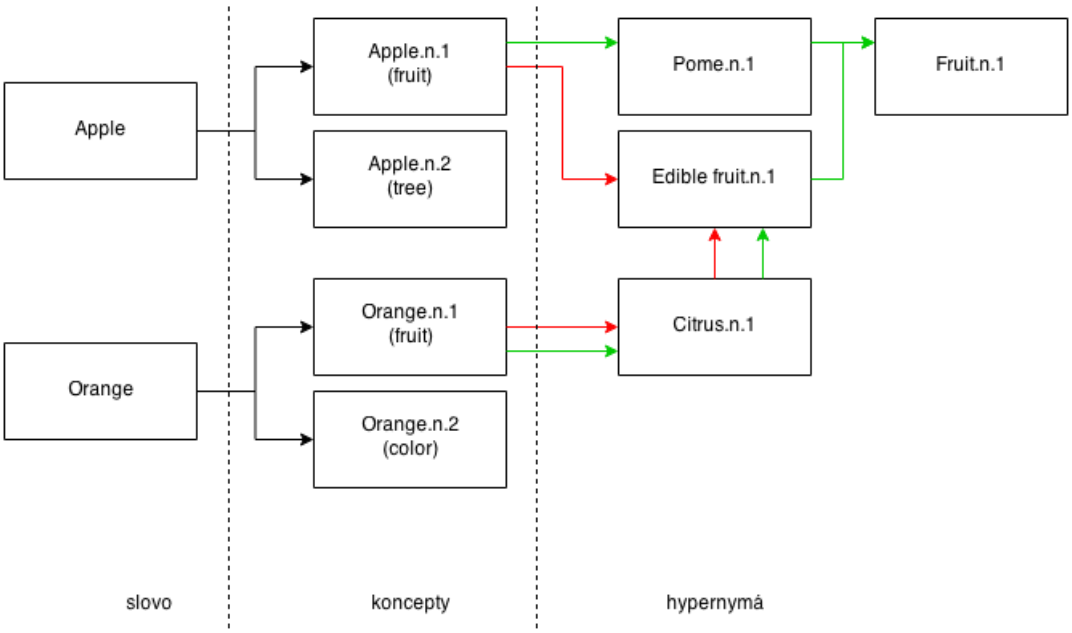
Uhlár v projekte [34] vykonáva obohacovanie v dvoch fázach a len pre podstatné mená:

1. Pridanie synoným z vybranej množiny synoným (označuje ako koncept) k podstatným menám.
2. Pridanie množiny hyperným pre každý koncept identifikovaný v kroku 1.

Opis samotnej metódy je neúplný, nejasný, vety sú nedokončené a identifikovali sme viacero nedostatkov:

- Zvyšuje zašumenie obohacovaním vektorov o synonymá.
- Daný prístup aplikuje vždy len na tvar podstatných mien, aj keď ten nemusí byť bežne používaný. Ideálnym príkladom je sloveso get. Jeho najčastejšie používaný význam v tvare podstatného mena je potomok, no tento význam sa používa zriedka aj v hovorenej reči, nie to ešte v zdrojových kódoch. Pridaním synoným a hyperným k danému konceptu len znásobuje chybu.
- Za najrelevantnejší koncept a hypernymum považuje vždy prvý prvok v poradí. Toto priradenie je síce najpravdepodobnejšie, ale často chybné. Výšku tejto pravdepodobnosti nezohľadňuje.
- Zvýšený počet dimenzií nie je adekvátny prínosom.

Dôvodom vzniku daných nedostatkov je podľa nás až prílišné zjednodušenie problematiky, ktorú si priblížime nasledovným príkladom (viď.Obrázok 5). Anglické slová apple a orange majú viacero významov. Naším cieľom je priradiť k obom najnižšie spoločné hypernymum, edible fruit. Najprv musíme správne určiť koncept daného slova. Ku každému konceptu môže byť priradených viacero hyperným. Dôsledkom nesprávneho zvolenia konceptu, hypernyma alebo jeho úrovne dochádza k zlyhaniu, a teda aj k zbytočnému zvýšeniu počtu dimenzií daného priestoru.



Obrázok 5 - Problematika synonymity.

2.5.3. Zhodnotenie

V práci [34] najlepšie zlepšenie výsledkov ako sme predpokladali dosahuje obohacovanie na základe dedenia. Ostatné vzťahy, ktoré tu boli identifikované považujeme za neužitočné. V prípade agregácie sám autor konštatuje mizivé výsledky. Čiastočné triedy sa v jazyku Java nevyskytujú, no aj tak ich považujeme za veľmi príbuzné k vzťahu vnorenia tried, ktorý je už vyriešený spôsobom extrakcie z AST. Samotné štrukturálne obohacovanie nerieši problém synoným. Z ontologických prístupov v práci [33] dosiahlo najlepšie výsledky hypernomické obohacovanie.

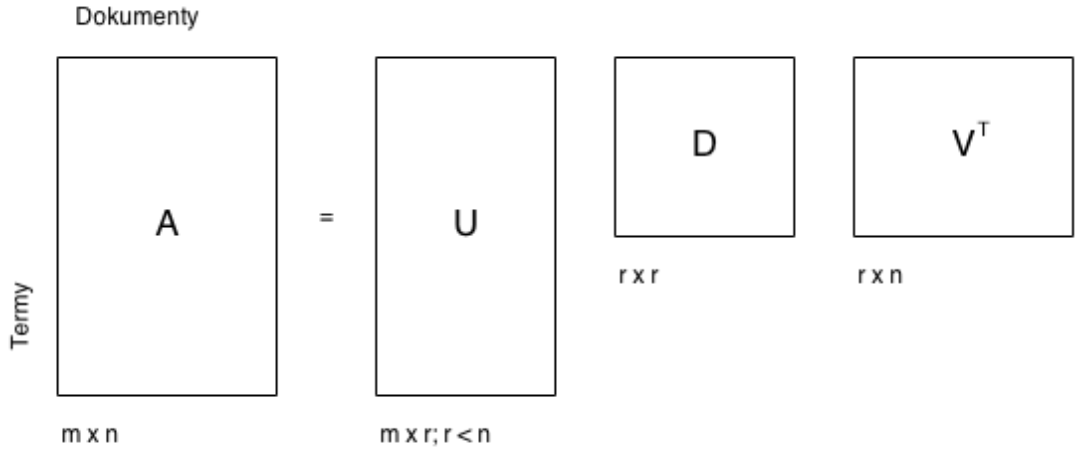
Uhlár vo svojom projekte používa oba druhy ontologického obohacovania. Na základe identifikovaných problémov a nedostatkov spôsobených jednoduchým ponímaním danej problematiky sme sa rozhodli navrhnúť a implementovať vlastný algoritmus, ktorý bude prihliadať na pravdepodobnosti jednotlivých priradení, či už slovného druhu alebo konceptu.

2.6. Redukcia priestoru

Jedným z najkritickejších problémov zhlukovania dokumentov je vysoká dimenzionalita ich obsahu. Tento problém je tiež známy pod názvom kliatba dimenzionality (z angl. curse of dimensionality). Možným riešením je proces projekcie dokumentov do menej rozmerného priestoru, nazývaný redukcia priestoru (z angl. dimensionality reduction).

2.6.1. Singulárny rozklad

Singulárny rozklad (z angl. Singular Value Decomposition, ďalej SVD) je rozklad matice A na dve ortogonálne matice U, V a diagonálnu maticu D (viď.Obrázok 6).



Obrázok 6 - SVD.

Nenulové prvky matice D predstavujú singulárne čísla matice A, a teda predstavujú odmocniny vlastných hodnôt (z angl. eigenvalues) matice A^TA alebo AA^T. Stĺpce matice U, ľavé singulárne vektory, získame horizontálnym spojením vlastných vektorov (z angl. eigenvectors) matice AA^T. Stĺpce matice V, pravé singulárne vektory, získame horizontálnym spojením vlastných vektorov matice A^TA.

Nakoľko vlastné vektory symetrickej matice sú ortogonálne a lineárne nezávislé, môžu byť použité ako bázické vektory, a teda ako reprezentácia multidimenzionálneho priestoru.

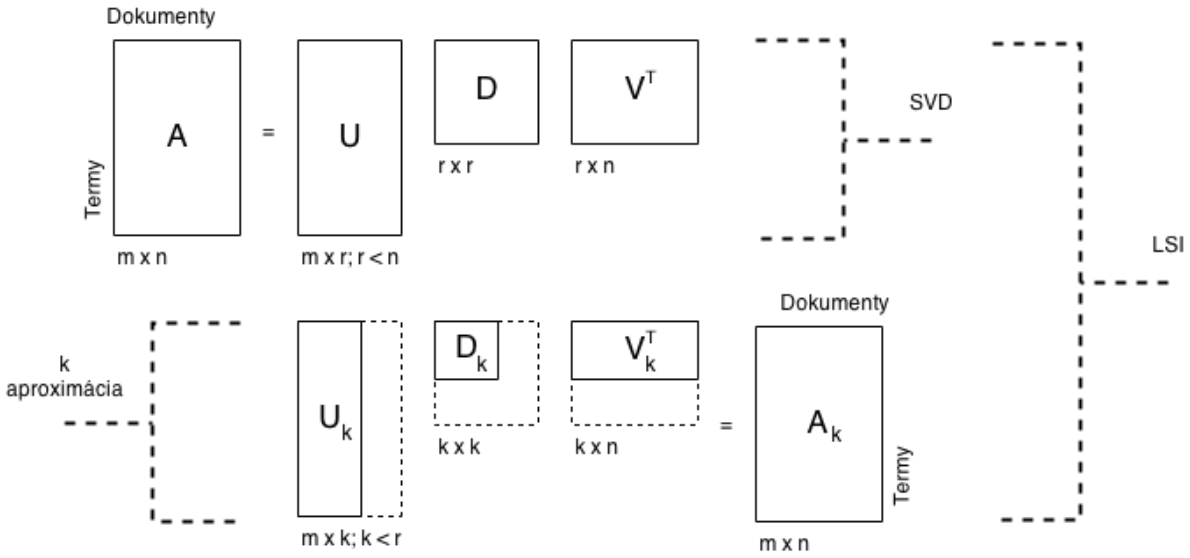
Vzťah medzi vlastným vektorom v a vlastnou hodnotou λ, ľubovoľnej symetrickej matice M, možno určiť rovnicou $Mv = \lambda v$. Vlastnú hodnotu ľubovoľnej symetrickej matice M rozmeru n, možno vypočítať na základe vzťahu $\det(\lambda I_n - M) = 0$; kde I_n predstavuje jednotkovú maticu veľkosti n.

Takýto rozklad nám umožňuje aproximovať pôvodnú maticu, a teda mapovať stĺpce a riadky pôvodnej matice do k-dimenzionálneho priestoru. Túto k-aproximáciu pôvodnej matice získame tak, že berieme do úvahy len k stĺpcov matice U a k riadkov matice V (viď.Obrázok 7), známe tiež ako *redukovaná SVD* (z angl. truncated SVD).

2.6.2. Latentné sémantické indexovanie

Latentné sémantické indexovanie (z angl. Latent Semantic Indexing, ďalej LSI) je redukčná metóda, ktorá mapuje objekty vysoko-rozmerného priestoru na ich nízko-rozmernú reprezentáciu. Tento proces možno opísať nasledujúcimi krokmi (viď. Obrázok 7):

1. Zostrojenie term-dokument matice A.
2. Dekompozícia matice A pomocou SVD.
3. Získanie k-aproximácie matice A prostredníctvom redukovanej SVD. Výsledkom je rozmerovo ekvivalentná matica A_k.



Obrázok 7 - LSI.

2.6.2.1. Aktualizácia LSI modelu

Metódy aktualizácie LSI modelu možno rozdeliť do týchto hlavných kategórií:

- *Prepočítanie modelu* (z angl. recompute)
- *Pridávanie do modelu* (z angl. folding-in) – mapovanie nových dokumentov do existujúceho LSI modelu bez jeho prepočítania, čo ma za následok stratu ortogonality a skreslenie podobností dokumentov.
- *Aktualizácia modelu* (z angl. updating) – kompromis medzi pridávaním do modelu a jeho prepočítavaním. Na základe zmien LSI modelu, ktoré vznikli pri mapovaní nových dokumentov do existujúceho LSI priestoru je aproximovaný nový, aktuálny

model. V práci [6] autor porovnáva vybrané algoritmy (viď.Tabuľka 4). Ďalšie porovnanie možno nájsť v novších prácach [7] a [8].

- *Skladanie modelu* (z angl. folding-up) – kombinácia pridávania do modelu a aktualizácie modelu, pričom aktualizácia prebieha len po pridaní určitého počtu dokumentov. Existuje však aj prístup, kedy je čas aktualizácie určený na základe aktuálneho stavu modelu (napr. strata ortogonality) nazývaný tiež adaptívne skladanie modelu (z angl. adaptive folding-up) [7].

Algoritmus	Distribuovateľný	Inkrementálny		Štruktúra matice	Sledovanie subpriestoru[1]
		Dokumenty	Slová		
Zha& Simon [11]	Nie	Áno	Áno	hustá	Áno
Levy &Lindenbaum [10]	Nie	Áno	Nie	hustá	Áno
Brand [9]	Nie	Áno	Áno	hustá	Nie
Řehůřek [6]	Áno	Áno	Nie	riedka	Áno

Tabuľka 4 - porovnanie vybraných ONLINE algoritmov [6]

2.6.3. Analýza hlavných komponentov

Analýza hlavných komponentov (z angl. Principal Concept Analysis, ďalej PCA) je technika využívajúca lineárnu transformáciu na vytvorenie zjednodušenej dátovej reprezentácie zachovávajúcej charakteristiky vstupnej množiny dát.

Majme vstupnú maticu X, centrovanú podľa priemeru (z angl. meancentered), pozostávajúcu z d dimenzií a n pozorovaní (z angl. observations). Jej transformáciu do k-dimenzionálneho priestoru možno zapísať nasledovne: $Y_{kxn} = E_{dxk}^{T} X_{dxn}$; kde E predstavuje projekčnú maticu rozmerov d x k, pozostávajúcu z hlavných komponentov matice X a Y jej reprezentáciu v k-dimenzionálnom priestore s rozmermi k x n. Hlavné komponenty matice X získame pomocou jej singulárneho rozkladu, ako k vlastných vektorov s najvyššími vlastnými hodnotami ľavých singulárnych vektorov (matica U - viď. Singulárny rozklad).

[1] Sledovanie subpriestoru (z angl. subspace tracking) umožňuje znižovanie relevancie starých pozorovaní, zabúdanie.

2.6.4. Diskrétna kosínusová transformácia

Diskrétna kosínusová transformácia (z angl. Discrete Cosine Transformation, ďalej DCT) transformuje sekvenciu vstupných vzoriek na množinu koeficientov frekvenčnej domény. DCT reprezentuje signál, množinu čísel, ako sumu kosínusových funkcií s rôznymi frekvenciami.

Majme množinu A pozostávajúcu z n hodnôt. $A = \{a_0, a_2, a_3, \ldots, a_{n-1}\}$

Koeficienty jednorozmernej diskrétnej kosínusovej transformácie, w_k získame nasledovne:

$$w_k = c_k \sqrt{\frac{2}{n} \sum_{t=0}^{n-1} a_t cos\left[\frac{\pi}{n}\left(t + \frac{1}{2}\right)k\right]}; k = 0,1,\ldots,n-1$$

$$c_k = \begin{cases} \frac{1}{\sqrt{2}}, & k = 0 \\ 1, & k > 0 \end{cases}$$

Nulová frekvencia w_0, tiež známa pod akronymom DC (z angl. DirectCurrent) je teda určená vzťahom:

$$w_0 = \frac{a_0 + a_1 + \cdots + a_{n-1}}{\sqrt{n}}$$

A	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
W_A	12.73	-6.44	0.0	-0.67	0.0	-0.2	0.0	-0.05
Odignorujme koeficienty pre ktoré platí $\|\boldsymbol{w_k}\| < 1$								
W_A`	12.73	-6.44	0.0	0.0	0.0	0.0	0.0	0.0
Aplikujme inverznú DCT na W_A', a teda aproximujme pôvodné hodnoty								
A_{aprox}	1.34	1.82	2.71	3.87	5.13	6.29	7.18	7.66

Tabuľka 5 - príklad výpočtu koeficientov DCT[2]

[2]Hodnoty v tabuľke sú zaokrúhlené na dve desatinné miesta, pri výpočte sa používali nezaokrúhlené hodnoty.

2.6.5. Náhodná projekcia

Náhodná projekcia (z angl. Random Projection, ďalej RP) je metóda, ktorá využíva náhodnú maticu na mapovanie objektov do nízko-rozmerného priestoru. Táto projekcia je založená na vete Johnson-Lindenstrauss [13], ďalej JL, podľa ktorej môžeme premietnuť množinu bodov do menej rozmerného, náhodne vybraného vektorového subpriestoru a zároveň približne zachovať Euklidovu vzdialenosť medzi nimi.

Predpokladajme potrebu redukcie množiny dát d-dimenzionálneho priestoru na ich k-dimenzionálnu reprezentáciu, kde počet inštancií je n. Túto redukciu možno zapísať vzťahom $Y_{kxn} = R_{kxd}X_{dxn}$, kde R predstavuje náhodnú maticu rozmerov k x d a X originálnu maticu rozmerov d x n. Existuje niekoľko algoritmov pre výpočet R podľa JL. Podľa práce [28], môžeme získať elementy náhodnej matice R nasledovne:

$$r_{i,j} = \begin{cases} +\sqrt{3}\ s\ pravdepodobnosťou\ P_r = \dfrac{1}{6} \\ 0\ s\ pravdepodobnosťou\ P_r = \dfrac{2}{3} \\ -\sqrt{3}\ s\ pravdepodobnosťou\ P_r = \dfrac{1}{6} \end{cases}$$

2.6.6. Náhodné indexovanie

Náhodne indexovanie (z angl. Random Indexing) [37] je technika založená na náhodnej projekcií, ktorá reprezentuje dáta prostredníctvom konštantne dlhých vektorov, vektorov kontextu (z angl. context vector). Tieto vektory sú určené sumou všetkých vektorov indexu (z angl. index vector), ktoré určujú jeho kontext. Definícia kontextu môže byť rôzna. Veľmi jednoduchým príkladom pre dokumenty v prirodzenom jazyku je určovanie kontextu slova na základe n-gramov, kedy kontext môžu predstavovať všetky slová, ktoré sa v texte vyskytujú bezprostredne pred alebo za daným slovom (pozorujeme 3-gram). Index vektor je ternárny, riedky vektor, ktorého položky nadobúdajú hodnoty 0, -1, alebo 1. Vytvára sa náhodným generovaním, ktoré zohľadňuje tieto parametre:

- Dĺžka vektora – konštanta, ktorá definuje v konečnom dôsledku počet dimenzií redukovaného priestoru.
- Počet nenulových kladných aj záporných prvkov – konštanta, ktorá určuje koľko položiek vektora nadobúda hodnotu -1 a zároveň koľko položiek nadobúda hodnotu 1.

Počet kladných a záporných hodnôt je rovnaký, aby pravdepodobnosť výskytu kladnej a zápornej hodnoty v každej položke bola rovnaká. To má pri sčítaní vektorov za následok vzájomne rušenie šumu spôsobeného náhodným generovaním.

Tabuľka 6 - príklad vytvárania vektora kontextu.

Výskyt slova „prichádza“ v korpuse	**Vektory indexu slov predstavujúcich kontext**		**Výpočet vektora kontextu slova „prichádza“**
... auto prichádza pomaly ...	auto	1 0 0 0 0 -1	1 0 0 0 0 -1
	pomaly	1 -1 0 0 0 0	1 -1 0 0 0 0
... vlak prichádza neskoro ...	vlak	-1 0 1 0 0 0	-1 0 1 0 0 0
	neskoro	0 0 -1 0 0 1	0 0 -1 0 0 1
... autobus prichádza načas ...	autobus	0 0 0 -1 1 0	0 0 0 -1 1 0
	načas	0 1 0 -1 0 0	0 1 0 -1 0 0
			1 0 0 -2 1 0

Príklad určovania kontextu slova „prichádza“ ilustruje Tabuľka 6. V prvom stĺpci možno vidieť všetky výskyty slova „prichádza“ v korpuse. Ako kontext tohto príkladu uvažujeme prvé predchádzajúce a nasledujúce slovo. V druhom stĺpci tabuľky sú zobrazené náhodne vygenerované vektory indexu pre slová určujúce kontext. V poslednom stĺpci je realizovaný výpočet výsledného vektora kontextu slova „prichádza“, ktorý je určený sumou všetkých index vektorov zo stĺpca č. 2.

2.6.7. Zhodnotenie

Porovnanie jednotlivých metód z hľadiska priemernej presnosti znázorňuje Tabuľka 7. LSI dosahuje najlepšie výsledky, avšak je výpočtovo najnáročnejšia. PCA využíva pre výpočet projekčnej matice taktiež SVD, no vlastné vektory kovariančnej matice možno získať bez počítania A^TA, a teda je táto transformácia výpočtovo menej náročná. RP zachováva euklidovú vzdialenosť, nie kosínusovú, a teda je vhodnejšia pre spracovanie obrazových dát. Pri vysoko rozmernom priestore sa kosínusová vzdialenosť blíži Euklidovej, a teda RP dosahuje výsledky podobné PCA pri oveľa menšej zložitosti [15]. V doméne redukcie priestoru sa však snažíme získať čo najmenší počet dimenzií.

Možným riešením problému zložitosti LSI je použitie online algoritmov, ktoré umožňujú aktualizácie modelu dodatočným pridávaním nových dokumentov v podstatne nižšom čase.

Tabuľka 7 - priemerná presnosť vybraných metód [13][14]

	Priemerná presnosť (z angl. precision) pre k = 200		
Dátový set	**PCA**	**RM**	**LSI**
CRANFIELD	0.176	0.101	0.382
MEDLINE	0.239	0.139	nevhodné k
	Priemerná presnosť pre k = 300		
CRANFIELD	0.186	0.111	0.351
MEDLINE	0.253	0.174	nevhodné k

Nami dodatočne identifikovaná technika náhodného indexovania bola doposiaľ overená sériou experimentov. Krátke zhrnutie niektorých z nich možno nájsť v práci [37]. Presnosť tejto metódy po aplikovaní lematizácie dosiahla hodnotu 72%. LSI dosiahlo za rovnakých podmienok najlepšiu hodnotu 64.4%. Ďalšou výhodou RI je jeho prirodzená inkrementalita pri pridávaní nových pozorovaní.

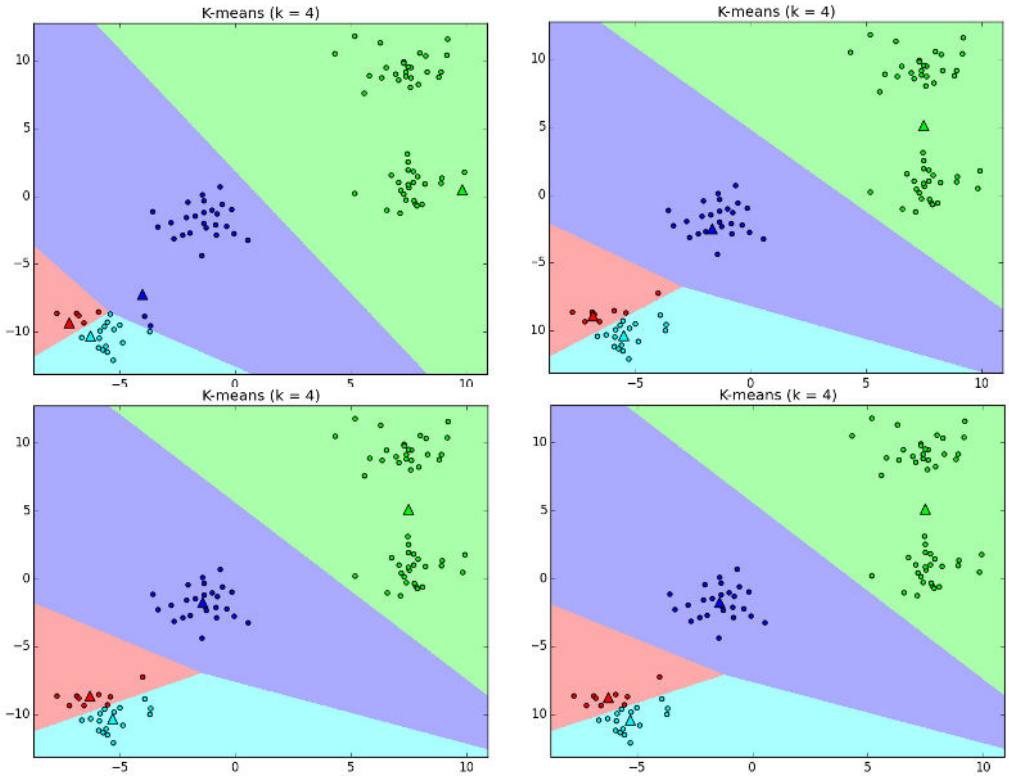
3 Zhlukovacie algoritmy

3.1. Nehierarchické metódy

Nehierarchická zhlukovacia analýza vytvára daný počet skupín objektov, použitím iteratívneho algoritmu, ktorý optimalizuje dané kritérium. Často prezentovaným algoritmom danej skupiny je algoritmus k-centier (z angl. K-means, ďalej K-means).

K-means

Jeden z najjednoduchších algoritmov bez učiteľa, ktorý rieši problém zhlukovania objektov. K v názve predstavuje počet cieľových zhlukov a je zároveň povinným parametrom danej metódy.



Obrázok 8 - Ukážky zhlukov medzi fázami.

Segmentácia daných objektov do tried sa vykonáva nasledovne:

1. Zvolí sa náhodne k objektov (pozorovaní) z množiny, ktoré predstavujú počiatočné centrá.
2. Každý objekt je následne priradený najbližšiemu z centier (vznikne k zhlukov).

3. Hodnota každého z centier je aktualizovaná na hodnotu centroidu zhluku, ktorý reprezentuje.
4. Pokiaľ sa zmenila hodnota aspoň jedného z centier, pokračuj v bode 2.

Algoritmus k-means zaručuje konvergenciu do lokálneho optima, nie však globálneho (viď.Obrázok 8).

3.2. Hierarchické metódy

Myšlienkou hierarchickej zhlukovacej analýzy (z angl. Hierarchical Clustering Analysis, ďalej HCA) je vytvorenie binárneho stromu dát, ktorý spája podobné skupiny objektov. Existujú dva hlavné typy algoritmov:

- Aglomeratívne - prístup zdola-nahor. Na začiatku zhlukovania každý dokument predstavuje jeden zhluk, triedu. V nasledujúcich fázach spájame najpodobnejšie zhluky, až pokiaľ nie sú všetky objekty v jednom zhluku. Každá úroveň výsledného stromu predstavuje rozdelenie objektov. Existuje množstvo kritérií na určovanie podobnosti zhlukov, ako príklad použijeme algoritmus SLCA (z angl. Single-linkage Clustering Algorithm).
- Divízne - prístup zhora-nadol. Všetky objekty predstavujú jeden veľký zhluk. Objekty zhlukov sú následne rozdeľované do dvoch tried na základe určitého kritéria, až pokiaľ zhluky neobsahujú len jeden objekt. Príkladom tejto skupiny je algoritmus divíznych k-centier (z angl. bisecting K-means, ďalej BK-means).

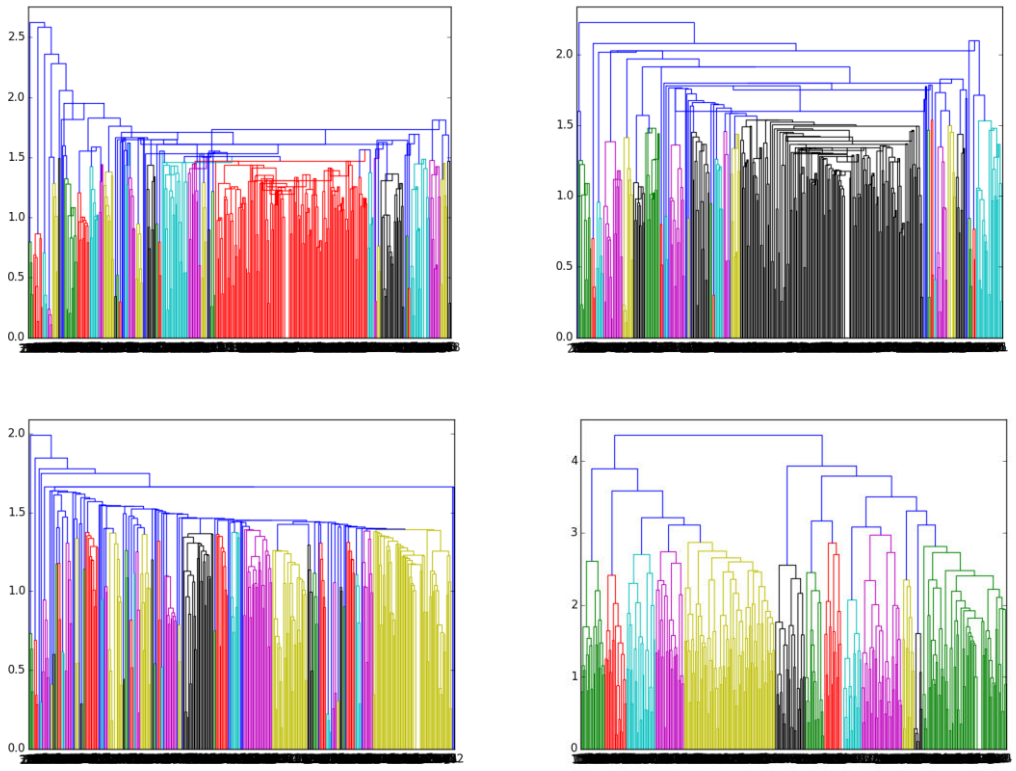
3.2.1. SLCA

Aglomeratívny prístup, ktorý ako mieru podobnosti dvoch zhlukov používa minimálnu vzdialenosť medzi ich objektmi. Samotný algoritmus pozostáva z týchto krokov:

1. Každý objekt je priradený práve jednému zhluku.
2. Zo všetkých zhlukov sú vybrané dva najbližšie na základe miery vzdialenosti:
$$d(c_1, c_2) = \min_d d(o_i, o_j); o_i \in c_1 \wedge o_j \in c_2$$
3. Vybrané zhluky sú spojené do jedného výsledného zhluku.
4. Ak počet aktuálnych zhlukov je väčší ako jedna, algoritmus pokračuje v bode 2.

Existuje množstvo variácií daného algoritmu, väčšina z nich sa líši len v miere vzdialenosti, ktorú používajú. Príklady:

- CLCA (z angl. Complete-Linkage Clustering Algorithm) - používa maximálnu vzdialenosť.
- UPGMA (Unweighted Pair Group Method with Arithmetic Mean) - vzdialenosť určuje priemer vzdialeností medzi objektmi zhlukov.
- UPGMC (z angl. Unweighted Pair-Group Method using Centroid) - používa vzdialenosť medzi centroidmi jednotlivých zhlukov.



Obrázok 9 - Porovnanie UPGMM, UPGMC, SLCA, CLCA.

Na základe porovnania výsledkov daných algoritmov, ktoré možno vidieť na obrázku (viď.Obrázok 9) usudzujeme, že zhluky pri CLCA sú najvyváženejšie. V ostatných prípadoch dochádza k identifikácií veľkého počtu malých zhlukov a jedného veľké zhluku.

3.2.2. Bk-means

Divízna metóda HCA, ktorá pri delení zhlukov využíva algoritmus k-means. Proces zhlukovania je vykonaný v týchto krokoch:

1. Vytvorenie zhluku, do ktorého patria všetky objekty.
2. Výber zhluku, ktorý sa má rozdeliť.
3. Aplikovanie algoritmu k-means s k=2 na vybraný zhluk n-krát (vznikne n možných rozdelení).
4. Výber a aplikácia najlepšieho z n rozdelení (rozdelenie zhluku na dva nové).
5. Pokračovanie v kroku 2., pokiaľ sa nedosiahne stanovený počet zhlukov alebo iné stanovené kritérium.

Kritický je v tomto prípade výber zhluku, ktorý sa má rozdeliť. Jedným z možným kritérií je výber zhluku s najmenšou kvalitou alebo toho najväčšieho.

3.3. Hybridné metódy

Využívajú prvky hierarchických aj nehierarchických prístupov. Jednou z najefektívnejších a zároveň inkrementálnych metód je algoritmus s názvom BIRCH.

3.3.1. BIRCH

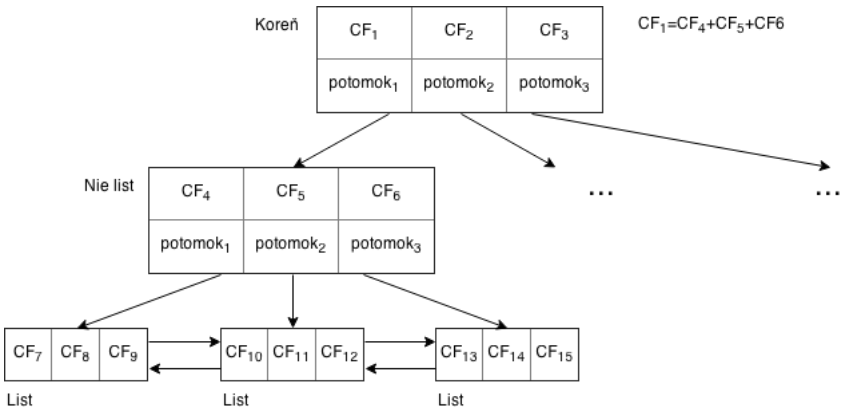
BIRCH (z angl. Balanced Iterative Reducing and Clustering using Hierarchies) [35] je inkrementálna metóda zhlukovania, vhodná pre rozsiahle databázy, pôvodne navrhnutá pre obrazové dáta. Zhluk je reprezentovaný pomocou zhlukovacej črty (z angl. Clustering feature, ďalej CF) ako trojica $CF = (N, \overrightarrow{LS}, SS)$, kde N predstavuje počet objektov v zhluku, $\overrightarrow{LS}$ lineárnu sumu vektorov objektov zhluku $\overrightarrow{LS} = \sum_{i=1}^{N} \overrightarrow{X_i}$ a SS sumu štvorcov vektorov objektov zhluku $SS = \sum_{i=1}^{N} \overrightarrow{X_i}^2$.

Majme skupinu troch objektov reprezentovanú vektormi $(3,4)$, $(2,6)$, $(4,5)$. Tejto skupine zodpovedá črta $CF = (3, (9,15), (29,77))$.

Črty sú aditívne, a teda platí $CF_1 + CF_2 = (N_1 + N_2, \overrightarrow{LS_1} + \overrightarrow{LS_2}, SS_1 + SS_2)$.

Každá črta je umiestnená v CF strome (z angl. CF tree). CF strom je výškovo vyvážený strom s dvomi parametrami. Faktor vetvenia B, prah T, ktorý určuje maximálnu hodnotu

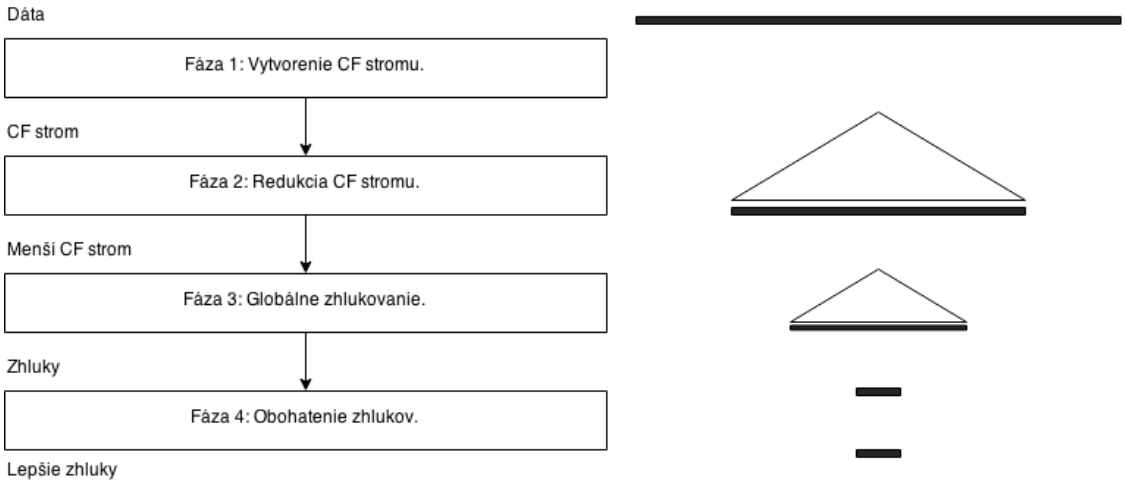
priemeru alebo polomeru zhluku a maximálny počet prvkov L uložených v liste. Každý, nie listový prvok, obsahuje najviac B prvkov v podobe $[CF_i, potomok_i]$, kde $potomok_i$ predstavuje ukazovateľ na i-tého potomka. Každý list obsahuje L prvkov v podobe $[CF_i]$, ukazovateľa na predchádzajúci list a ukazovateľa na nasledujúci list (viď.Obrázok 10).



Obrázok 10 - CF strom.

Zhlukovanie prebieha v štyroch fázach (viď. Obrázok 11):

1. *Prvé skenovanie dát a vytvorenie CF stromu* - Strom je vytváraný inkrementálne, postupným pridávaním objektov. Pri pridávaní prechádzame strom od koreňa k listom. V každom uzle sa rozhodujeme na základe vzdialenosti medzi objektom a črtami. Vyberáme vždy potomka, ktorého vzdialenosť je najmenšia. Týmto spôsobom získame list L_i, ktorý je najbližší k danému objektu. Pokiaľ L_i môže prijať ďalší prvok, je tento prvok vložený a hodnoty predkov sú aktualizované. Pokiaľ L_i nemôže prijať ďalší prvok, je nutné vykonať rozdelenie, a teda pridať rodičovi nový list. Podrobnejšie informácie možno nájsť v práci [35].
2. *Vytvorenie menšieho CF stromu* - Menší strom je zostrojený kombináciou pôvodného stromu a nového stromu, ktorý vznikol zvýšením prahovej hodnoty.
3. *Globálne zhlukovanie* - Využitie existujúcich zhlukovacích algoritmov na určenie výsledných zhlukov. Tie pracujú nad črtami a nie nad pôvodnými dátami.
4. *Obohatenie zhlukov* - Uzly sú premiestnené do tých zhlukov, ktoré sú na základe výsledkov z tretej fázy najbližšie. Následne sú zhluky prepočítané. Táto fáza nie je povinná, no umožňuje odstránenie šumu a zlepšenie kvality výsledkov.



Obrázok 11 - Fázy BIRCH.

3.4. Zhodnotenie

Divízne algoritmy sa pri rozdeľovaní zhlukov rozhodujú na základe globálneho pohľadu. Tento pohľad získavajú napríklad prostredníctvom iného zhlukovacieho algoritmu, čoho dôsledkom je podstatne vyššia výpočtová náročnosť. Pokiaľ si pevne stanovíme maximálnu úroveň, do ktorej divízne zhlukovanie vykonávame, môžu tieto algoritmy dosahovať lepšie výsledky ako aglomeratívne prístupy v približne rovnakom čase. Výhodu aglomeratívnych prístupov však vidíme v bližšej analógií k mravčím algoritmom. Konkrétne v reprezentácií, pokiaľ uvažujeme o jednotlivých mravcoch ako objektoch, ktoré zhlukujeme. V oboch prípadoch je vhodná práca s kovariančnou maticou.

BIRCH zaviedol veľmi efektívnu reprezentáciu dát. Práca [36] v doméne textových dokumentov však poukazuje na kvalitu výsledkov nie výrazne lepšiu ako algoritmus k-means. Algoritmus BIRCH je citlivý na poradie, v akom sú objekty pridávané a taktiež na parametre, aké mu zadáme.

4 Inteligencia roja

Inteligencia roju (z angl. Swarm Inteligence, ďalej SI) predstavuje oblasť informačných technológií, ktorá analyzuje a navrhuje efektívne výpočtové metódy určené na riešenie zložitých problémov, inšpirované správaním skutočných hmyzích spoločenstiev a rojov.

4.1. Algoritmus roja častíc

Algoritmus roja častíc (z angl. Particle Swarm Optimization, ďalej PSO) je robustná stochastická optimalizačná technika, založená na správaní rojov. Využíva skupinu agentov, častíc, tvoriacich roj pohybujúcich sa častíc v skúmanom priestore, hľadajúci najlepšie možné riešenie.

Každá častica predstavuje bod v n-rozmernom priestore, ktorej pohyb je určený rovnako na základe jej vlastných skúseností, ako aj na skúsenostiach susedných častíc. Každá častica si pamätá najlepšie riešenie, ktoré bola schopná dosiahnuť P_{BEST} a najlepšie riešenie zo všetkých susedných častíc G_{BEST}.

Princíp PSO spočíva v usmerňovaní pohybu častíc ku lokáciám, kde dosiahli najlepšie riešenie P_{BEST} a zároveň aj smerom, kde susedné častice dosiahli lepšie riešenie G_{BEST}, pričom váha P_{BEST} a G_{BEST} je pri ich kombinácií určená náhodne.

4.2. Algoritmus mravčej kolónie

Algoritmus mravčej kolónie (z angl. Ant Colony Optimization, ďalej ACO) je metaheuristický prístup riešenia zložitých optimalizačných problémov. Pôvodným zdrojom jeho inšpirácie je správanie mravcov pri hľadaní najkratšej cesty medzi mraveniskom a potravou. Tie zanechávajú pri pohybe feromónovú stopu, ktorá slúži ako komunikačné médium. V prípade, že na ňu narazia ostatné mravce, prestanú sa zväčša pohybovať náhodne a nasledujú túto stopu. Tá sa však časom stráca. Kratšie cesty majú tendenciu väčšej frekvencie prechodu, a preto sa stávajú atraktívnejšie. Dlhšie cesty časom úplne vyprchajú, a teda sa zachovajú len najlepšie riešenia.

V súčasnosti existuje niekoľko základných modelov ACO. V kontexte zhlukovania dát prezentovali v práci Deneubourget al. [16], ako prvý, metódu inšpirovanú triedením lárv podľa veľkosti (z angl. broodsorting), známu tiež ako základný model (z angl. BasicModel,

ďalej BM). Inicializácia BM spočíva v náhodnom rozmiestnení dát a mravcov do dvojrozmernej mriežky. Každý mravec sa po tejto mriežke pohybuje náhodne, s určitou pravdepodobnosťou zdvihnutia objektu, na ktorý narazí alebo pustenia objektu, ktorý nesie. Táto pravdepodobnosť je ovplyvnená dátami v jeho okolí, susedstve, a to tak, že pravdepodobnosť pustenia je zvýšená, ak je obklopený podobnými dátami a naopak pravdepodobnosť zdvihnutia je zvýšená, ak je obklopený rozdielnymi dátami.

Lumer a Faieta [17] zovšeobecnili základný model tak, aby bol schopný zhlukovať vektorovo reprezentované dáta. Pri úpravách pravdepodobností sa používa priemerná podobnosť objektu, ktorý mravec nesie ku všetkým objektom v jeho okolí. V tejto práci taktiež prezentovali koncept krátkodobej pamäte, pričom každý z mravcov si pamätá určitý počet ostatných pustení dát a pri zdvihnutí nového objektu je jeho pohyb upravený tak, aby smeroval k miestu predchádzajúceho spustenia podobných dát. Tento model je tiež známy pod akronymom LM alebo SACA.

Gutowitz v práci [18] vychádzajúc taktiež zo základného modelu, diskriminuje oblasti, ktoré sú úplne prázdne alebo naopak plné a to tak, že mravce v týchto oblastiach nedvíhajú ani nepúšťajú objekty, a teda sa vyhýbajú aktivitám, ktoré priamo nevedú k vytváraniu zhlukov.

V. Ramos, F. Muge and P. Pina [19] identifikovali problém modelu LM, ktorý vytvára veľký počet malých zhlukov a navrhli nasledujúce riešenie. Mravce sa nepohybujú úplne náhodne, ale na základe feromónových stôp, čím sa vyhýbajú preskúmavaniu prázdnych oblastí, čo vedie k zvýšeniu efektivity algoritmu [20].

Handl and Meyer [21] rozšírili koncept krátkodobej pamäte mravcov a umožnili im skoky do oblastí, v ktorých už podobný objekt pustili. Podobný prístup prezentovali v práci [22], no tu však poskytli mravcom schopnosť teleportovať sa po pustení objektu k ďalšiemu izolovanému.

H. Azzag, N. Monmarche, M. Slimane and G. Venturini [23] prezentovali nový zhlukovací algoritmus, AntTree, ktorý používa stromovú reprezentáciu, kde mravec predstavuje dáta. Tento návrh bol následne rozšírený v práci Marcelo Errecalde [24] o možnosť preradenia mravcov iným zhlukom a kombináciami identifikovaných zhlukov. Hlavným prínosom

tohto algoritmu, nazvaného AAT (z angl. Adaptive AntTree), bola schopnosť odpojenia mravca a jeho presun na vhodnejšiu pozíciu bez závažného zvýšenia zložitosti algoritmu.

Xiaohua Xu and Ling Chen Chen Y. [25] vytvorili model spiacich mravcov (z angl. Ant Sleeping Model, ďalej ASM), v ktorom sú dáta taktiež reprezentované mravcami, avšak v dvojrozmernej mriežke. Mravce nadobúdajú dva stavy, aktívni a spiaci. V prípade, že sa nachádza v nevhodnom prostredí (je obklopený rozdielnymi dátami) má väčšiu pravdepodobnosť zobudiť sa a zotrvať v aktívnom stave, hľadajúc si lepšie miesto pre spánok. Pokiaľ toto miesto lokalizuje, zvýši sa jeho pravdepodobnosť upadnutia do spánku.

Yan Yang a Mohamed S. Kamel [26] vyvinuli metódu paralelných mravčích kolónií, v ktorej sú zhluky získané viacerými nezávislými kolóniami rôzneho typu a parametrov (rýchlosť, pravdepodobnosti akcií) kombinované centrálnym agentom, kráľovnou. Tento prístup výsledkami prekonal metódy využívajúce len jednu kolóniu.

Pedram [27] navrhol novú stratégiu zhlukovania skupinami mravcov, kde každá skupina zberá dáta patriace práve jednému zhluku, ktorý reprezentuje. Počet skupín mravcov je teda ekvivalentný požadovanému počtu výsledných zhlukov. Tento model dosahuje lepšie výsledky ako LM.

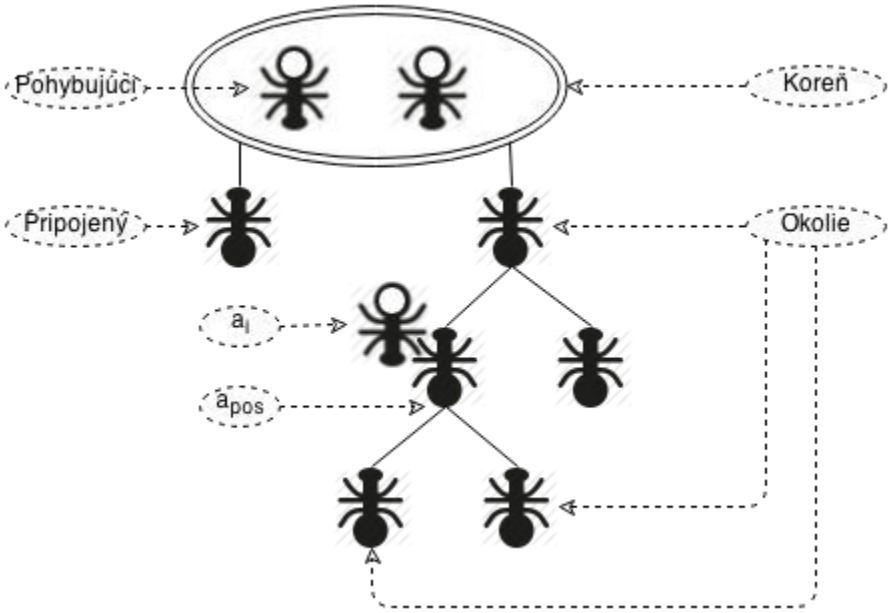
4.2.1. Zhodnotenie

Hlavná výhoda PSO spočíva v jeho rýchlej konvergencií k optimálnemu riešeniu, ktorá je porovnateľná s mnohými globálne optimalizačnými algoritmami ako napr. genetické algoritmy. Hlavným problémom je mapovanie problému na PSO častice, ktoré často nie je triviálne.

V rámci ACO sme uvažovali o dvoch hlavných reprezentáciách, mriežke a strome. Problém mriežky spočíva v zmene jej veľkosti a v identifikovaní hraníc zhlukov. Z toho dôvodu uprednostňujeme stromovú reprezentáciu, ktorá je ľahko škálovateľná, prispôsobivá a môže byť dokonca aj samo vyvažujúca. Zhluky v tomto prípade tiež možno jednoducho identifikovať (viď.Obrázok 13).

4.2.2. AntTree

Algoritmus, v ktorom dáta môžu mať ľubovoľnú reprezentáciu pokiaľ z nej dokážeme získať podobnosť dvoch vzoriek, a teda existuje funkcia $Sim(i,j)$, ktorá dvojici (d_i, d_j); $i \in <1, N>, j \in <1, N>$ priradí hodnotu z intervalu $<0,1>$, kde 0 znamená úplne rozdielné dáta a 1 úplne zhodné. N predstavuje počet vzoriek.



Obrázok 12 - Znázornenie vybraných pojmov algoritmu AntTree.

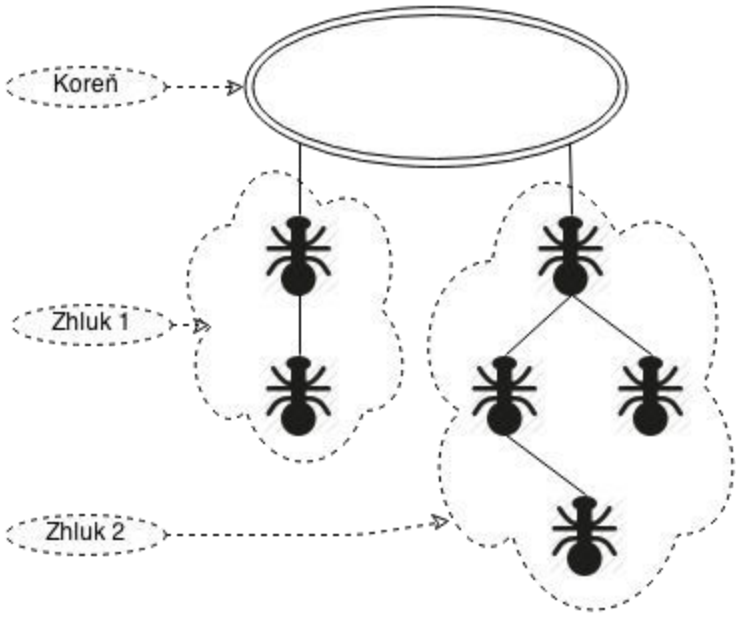
Hlavné vlastnosti algoritmu:

- Každý mravec predstavuje uzol, ktorý má byť zaradený do stromovej štruktúry, čiže dáta, ktoré majú byť priradené do zhluku.
- Koreň stromu, virtuálny uzol, nepredstavuje dáta, ale počiatočnú pozíciu, na ktorú sa mravce pri inicializácií pripájajú a následne na mravce pripojené k nemu, k jeho potomkom, až kým nie sú zaradené do štruktúry.
- „Pohyb“ po strome je určený $Sim(i,j)$ a lokálnym okolím mravca.

Pre každého mravca a_i; $i \in <1, N>$ platí:

- Výstupné spojenie uzla a_i je spojenie, po ktorom mravec a_i môže postupovať k ďalšiemu mravcovi.

- Vstupné spojenie umožňuje postup iného mravca k mravcovi a_i.
- Dátová vzorka d_i je reprezentovaná mravcom a_i.
- Hranica podobnosti $T_{Sim}(a_i)$ a hranica rozdielnosti $T_{Dsim}(a_i)$ sú lokálne aktualizované mravcom a_i.



Obrázok 13 - Ilustrácia zhlukov.

Hlavný princíp algoritmu:

Nech a_{pos} predstavuje mravca, na ktorom sa mravec a_i práve nachádza. Hranicu rozdielnosti získame vzťahom $T_{Dsim}(a_{pos}) = Min[Sim(a_j, a_k)]$; kde a_j, a_k sú deti a_{pos}. Nech a^+ predstavuje najpodobnejšieho mravca z detí a_{pos} ku mravcovi a_i. V prípade, že je mravec $a_i$na ťahu, porovná $Sim(a_i, a^+)$ s $T_{Dsim}(a_{pos})$. Ak $Sim(a_i, a^+) < T_{Dsim}(a_{pos})$, mravec a_i nie je dostatočne podobný s a^+ a bude pripojený k a_{pos} ,čím dôjde k vytvoreniu nového zhluku. V opačnom prípade sa mravec a_i posunie k a^+ ,aby našiel svoje umiestnenie v tomto podstrome.

Podrobnosti algoritmu sú prezentované v práci [23].

5 Dolovanie aspektov

Veľký vplyv na udržiavateľnosť a pochopiteľnosť softvérových systémov má aj ich dekompozícia do menších celkov. Dôsledkom tohto rozdelenia je oddelenie jednotlivých záležitostí. Niektoré vybrané pretínajúce záležitosti nie je možné oddeliť len s využitím OOP prostriedkov. Tento problém je tiež známy ako tyrania dominantnej dekompozície (z angl. tyranny of the dominant decomposition) [43]. Symptómami pretínajúcich záležitostí sú zapletenie (z angl. tangling) a roztrúsenie (z angl. scattering) kódu. Aspektovo orientované programovanie umožňuje modularizáciu týchto záležitostí, ich transformáciou do aspektov. Existujúce OOP systémy, ktoré chceme refaktorovať, sú však často rozsiahle, a preto proces identifikácie potenciálnych aspektových kandidátov v kóde je časovo náročný a vyžaduje určitú úroveň automatizácie. Existuje množstvo techník dolovania aspektov, medzi najznámejšie patria: Fan In analýza, analýza na základe identifikátorov (z angl. identifier analysis) a dynamická analýza.

Tabuľka 8 - Vzor kódu a prislúchajúce Fan In hodnoty.

```
interface A {
   public void m();
}
class B implements A {
   public void m() {};
}
class C1 extends B {
   public void m() {};
}
class C2 extends B {
   public void m() { super.m();};
}
class D {
   void f1(A a) { a.m(); }
   void f2(B b) { b.m(); }
   void f3(C1 c) { c.m(); }
}
```

Metóda	Potenciálny volajúci	Fan In hodnota
A.m	*D.f1, D.f2, D.f3*	*3*
B.m	*D.f1, D.f2, D.f3, C2.m*	*4*
C1.m	*D.f1, D.f2, D.f3*	*3*
C2.m	*D.f1, D.f2*	*2*

5.1. Fan In analýza

Fan In analýza je zameraná hlavne na pretínajúce záležitosti prejavujúce sa roztrúsením kódu. Táto metóda vytvára ohodnotenú množinu pozostávajúcu zo všetkých metód softvérového systému. Ohodnotenie je dané hodnotou Fan In, ktorú možno pre konkrétnu metódu určiť ako celkový počet jej možných volaní z rozdielnych tiel metód. Jej výsledkom je podmnožina len vysoko ohodnotených metód, avšak bez akejkoľvek

informácie o ich príslušnosti k pretínajúcim záležitostiam. Táto príslušnosť musí byť určená manuálne. Výsledky teda poskytujú len odporúčania, kde by sme mali pretínajúce záležitosti hľadať. Veľkosť výslednej množiny záleží od prahu, ktorý určuje minimálne ohodnotenie. Určenie optimálnej hodnoty prahu je špecifické pre každú skúmanú aplikáciu a veľmi priaznivo ovplyvňuje celkovú účinnosť. Príklad volaní a k nim prislúchajúcich hodnôt Fan In ilustruje Tabuľka 8 [38].

5.2. Analýza založená na identifikátoroch

Analýza založená na identifikátoroch vytvára skupiny kandidátov (programových entít) s podobnými názvami. Táto technika aplikuje algoritmus formálnej konceptovej analýzy (z angl. Formal Concept Analysis, ďalej FCA) na názvy všetkých metód tried skúmanej implementácie. Výsledkom FCA sú tzv. koncepty, skupiny elementov, ktoré zdieľajú v názve spoločné slová. Príkladom je koncept "read", ktorý zahrňuje metódy ako napr. "readChar" alebo "readDouble". Veľkosť konceptu je určená počtom priradených entít a slúži ako prahová hodnota analýzy. Je taktiež ako pri Fan In analýze špecifická pre každú skúmanú aplikáciu. Jej príliš vysoké hodnoty spôsobujú, že niektoré záležitosti nemusia byť vôbec identifikované a nízke hodnoty zas rapídne zvyšujú mieru úsilia potrebného pri manuálnej časti analýzy. Príklad zoskupenia získaného algoritmom FCA ilustruje Tabuľka 9.

Tabuľka 9 - Koncept "figure, request" a koncept "figure, request, update"

	figure	request	drawing	remove	update	dependend
AbstractFigure.removeDependendFigure	x			x		x
CompositeFigure.figureRequestRemove	x	x		x		
CompositeFigure.figureRequestUpdate	x	x			x	
DecoratorFigure.figureRequestRemove	x	x		x		
DecoratorFigure.figureRequestUpdate	x	x			x	

5.3. Dynamická analýza

Dynamická analýza vychádza z údajov zozbieraných počas vykonávania vybraných prípadov použitia softvérových systémov. Výsledkom vykonania každého z týchto scenárov je trasa vykonávania. Vstupom algoritmu FCA sú scenáre, ktoré predstavujú objekty a volané metódy, ktoré predstavujú atribúty. Pri analýze komplexných systémov je vysoká pravdepodobnosť identifikácie veľkého počtu konceptov. Dynamická analýza rieši

tento problém taktiež rôznymi stratégiami dodatočného filtrovania. Medzi základné patrí filtrovanie tých konceptov, ktoré obsahujú iba jeden objekt (viď. Tabuľka 10, koncept reprezentovaný šedými bunkami). Najzaujímavejšie sú tie koncepty, ktoré sú tvorené skupinou atribútov spoločnou pre viacero scenárov (viď. Tabuľka 10, čiarkovaná oblasť).

Tabuľka 10 - Dynamická analýza, príklad dvoch konceptov (šedá - roztrúsenie a čiarkovaná - zapletenie).

	A.m1()	A.m2()	B.m3()	C.m4()
Scenár 1	x		x	x
Scenár 2	x			
Scenár 3	x	x		
Scenár 4	x	x		

5.4. Dolovanie aspektov a zhlukovanie

Existuje niekoľko prác, ktoré pri identifikácii pretínajúcich záležitostí využívajú zhlukovanie, avšak všetky len na úrovni metód. Výhodou týchto prístupov je, že ich výsledkom sú sady metód príbuzných istej pretínajúcej záležitosti. Práca [40] definuje vlastnú mieru podobnosti pri vytváraní zhlukov metód a jednotlivé zoskupenia reprezentujúce pretínajúce záležitosti zoraďuje vo výsledkoch podľa relevancie. Zároveň sa tento prístup snaží aj o identifikáciu bodov prierezu. V práci [41] autor definuje vlastnú Fan In analýzu pre zhluky metód, ktorú používa na ich zoradenie vo výsledkoch. Autor v [42] porovnáva dva rôzne typy zhlukovania metód. Zhluky zoraďuje na základe ich podobnosti voči nulovému vektoru. Najmenej podobné sú vo výsledkoch zobrazené ako prvé. Nie sme si vedomí žiadnej existujúcej práce, ktorá využíva zhluky na úrovni tried.

5.5. Zhodnotenie

Fan In analýza je zameraná len na tie pretínajúce záležitosti, ktoré sa prejavujú roztrúsením kódu. Jej efektivita je závislá od prahovej hodnoty. Aj pri určení optimálneho prahu môže dochádzať k filtrovaniu relevantných kandidátov. Analýza založená na identifikátoroch skúma oba symptómy, avšak jej hlavnou nevýhodou je, že sa spolieha na dodržiavanie konvencií pri pomenúvaní metód. Dynamická analýza sa dá aplikovať len na spustiteľný kód a skúma len definované scenáre. Je teda problém pokryť celý systém.

Identifikovali sme niekoľko prístupov, ktoré dokazujú, že zhlukovanie môže pozitívne ovplyvniť kvalitu výsledkov jednotlivých analýz. Nenašli sme však žiadnu metódu, ktorá by využívala zhluky na úrovni tried.

6 Opis problémovej domény

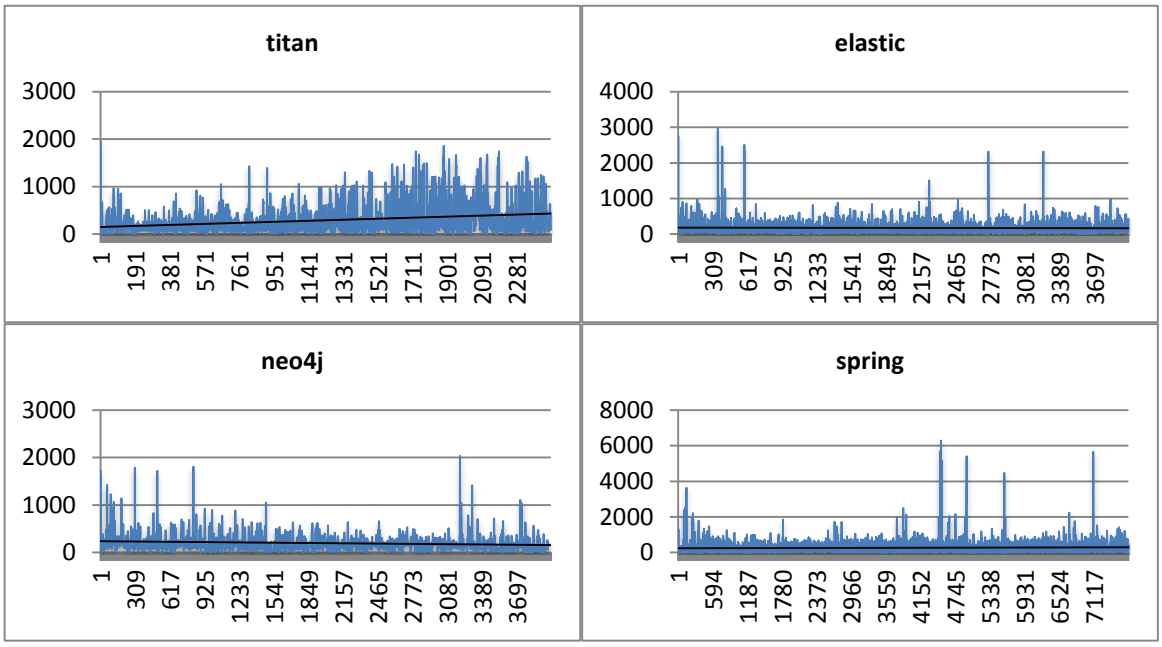
Väčšina techník dolovania dát pristupuje k dátam ako ku textovým dokumentom. Z tohto dôvodu je doména zdrojových kódov relatívne slabo pokrytá. Predpokladáme, že preddefinovaná štruktúra kódu a spôsob akým vzniká môžu odhaliť jeho zaujímavé vlastnosti, ktoré by sme mohli využiť počas návrhu a implementácie nášho riešenia.

Tabuľka 11 - Priemerný čas medzi odovzdaniami (z angl. commit).

	Titan	Elastic	Neo4j	Spring	celkovo
priemer	7.21	3.91	2.17	5.97	4.82

Na základe zozbieraných dát (viď. Tabuľka 11) môžeme tvrdiť, že zdrojové kódy predstavujú dynamické množiny dát. Zmeny sú väčšinou aplikované v malých dávkach. Predpokladáme, že počet dimenzií priestoru zodpovedajúceho dokumentom jednej dávky je relatívne malý, pretože tieto dokumenty sa väčšinou podieľajú na implementácií spoločného cieľa alebo série cieľov. Pre potvrdenie našej hypotézy sme vykonali experiment, v ktorom simulujeme vývoj softvérového systému postupnou aplikáciou jednotlivých odovzdaní. Výsledky experimentu zobrazuje Obrázok 14. Zhrnutie možno nájsť v tabuľke nižšie (viď.Tabuľka 12).

Obrázok 14 - Závislosť počtu dimenzií subpriesotu v čase.



Ako môžeme vidieť priemerná hodnota počtu dimenzií v subpriesotre sa pohybuje od 172.07 po 297.14, čo je relatívne malý počet vzhľadom na celkový počet dimenzií úplného priestoru (viď. Tabuľka 13).

Tabuľka 12 - Priemerný počet dimenzií prírastku.

	Neo4j	**Elastic**	**Spring**	**Titan**	**celkovo**
medián	146	132	200	196	199
priemer	201.92	172.07	270.07	297.14	235.30

Tabuľka 13 - Charakteristika vzorky projektov.

	Neo4j	**Elastic**	**Spring**	**Titan**
riadky	5753	5593	9833	1238
stĺpce	5257	5297	6562	3226
nenulové	225114	249107	414638	63009
hustota	0.74%	0.84%	0.64%	1.58%
termov	39.13	44.54	42.17	50.90

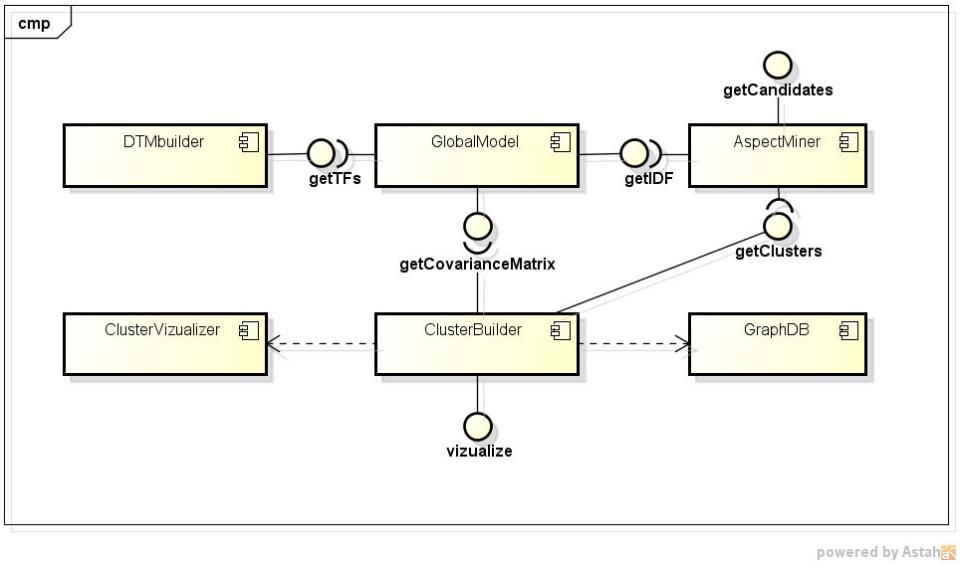
V charakteristike projektov (viď. Tabuľka 13) možno pozorovať najnižšiu hustotu 0.64% a najvyššiu možnú 1.58%. Zo skúseností vieme, že tieto hodnoty sú porovnateľné s hustotami dokument-term matíc v doméne dokumentov prirodzeného jazyka.

7 Návrh

Celkový pohľad na navrhnutý algoritmus možno vidieť na obrázku (viď. Obrázok 15). Zodpovednosti jednotlivých komponentov sú nasledovné:

- *DTMbuilder* - vytvorenie matice kľúčových pojmov.
- *GlobalModel* - úprava lokálnych váh (TF) na základe váh globálnych (TF-IDF).
- *AspectMiner* - získanie aspektových kandidátov.
- *ClusterBuilder* - vytvorenie zhlukov a ich vizualizácia.
- *ClusterVizualizer* - knižnica potrebná na vizualizáciu zhlukov.
- *GraphDB* - externá knižnica slúžiaca na prácu s grafovou databázou.

Obrázok 15 - Diagram komponentov, celok.



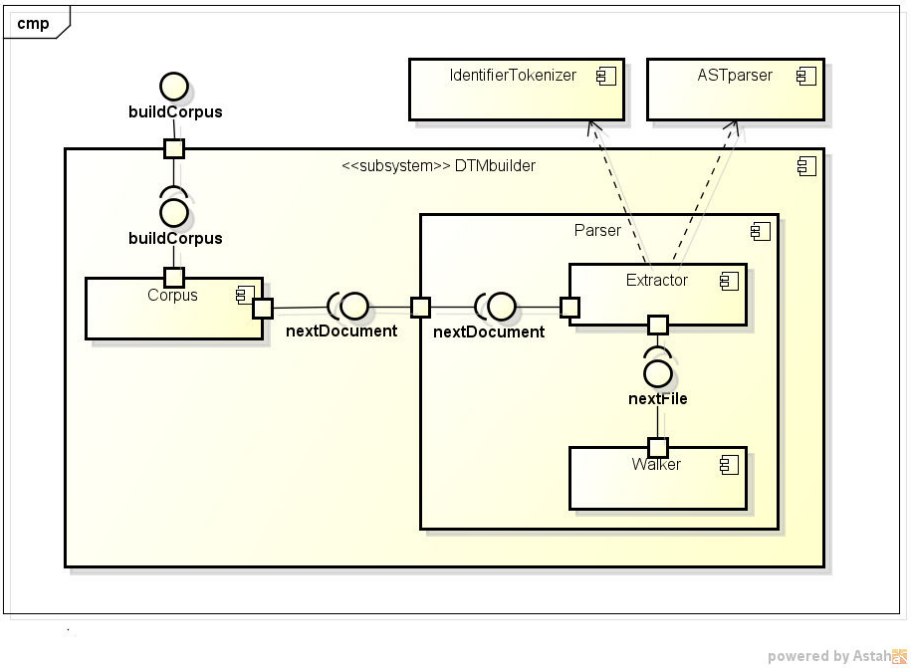
7.1. Vytvorenie dokument term matice

Navrhnutý modul, ktorý zodpovedá za vytvorenie dokument-term matice pozostáva z týchto komponentov (viď.Obrázok 16):

- *Walker* - komponent, ktorý umožňuje iteráciu nad jednotlivými zdrojovými súbormi. Môže ísť o rôzne typy súborov, získaných buď prechádzaním štruktúry priečinkov alebo súbormi špecifikovanými v jednotlivých projektoch. Projektové vlastnosti je v tomto prípade vhodné skúmať externými knižnicami.

- *Extractor* - časť implementácie, ktorej hlavnou úlohou je extrakcia dokumentov danej granularity zo získaných zdrojových súborov. Tá býva najčastejšie vykonávaná pomocou AST stromu, a preto daný komponent zodpovedá aj za tokenizáciu a štrukturálne váhovanie termov.
- *Parser* - Kompozícia komponentov Walker a Extractor. Tento modul sme zaviedli, aby bolo možné jednoducho vymeniť celú časť zodpovednú za extrakciu dokumentov.

Obrázok 16 - Diagram komponentov, DTMbuilder.



7.2. Váhovanie

Nami navrhovaný algoritmus vykonáva štrukturálne aj frekvenčné váhovanie. Štrukturálne plánujeme vykonávať na základe poznatkov získaných z AST dvoma spôsobmi:

- Na základe AST tokenizácie - termy získané z komentárov, reťazcov a JavaDoc komentárov majú rôzne škálované váhy.
- Na základe AST uzlov - termy získané z názvov deklarácii typov a metód majú zvýšené váhy.

V rámci frekvenčného váhovania aplikujeme algoritmus TF-IDF.

7.3. Reprezentácia korpusu

V našej práci navrhujeme dva rôzne prístupy reprezentácie korpusu. Prvá využíva nízku hustotu dokument-term matice a druhá je založená na technike náhodného indexovania.

V oboch prípadoch rozlišujeme primárnu a sekundárnu reprezentáciu korpusu. Primárna predstavuje dokument-term maticu s frekvenčnými váhami bez normalizácie. Tieto váhy sú konštantné v čase, a tak nie je potrebné ich získavať znovu, pokiaľ sa dokument nezmenil. Primárna reprezentácia slúži len na účely serializácie. Za účelom získania podobnosti dokumentov musíme najskôr vypočítať sekundárnu reprezentáciu (model), ktorú získame aplikovaním IDF vektora na normalizovanú maticu primárnej reprezentácie. Podobnosť je definovaná ako normalizovaný skalárny súčin (kosínusová podobnosť). Použitie TF-IDF váh považujeme za dôležité, nakoľko potlačujú pravdepodobnosť nesprávnej identifikácie zhlukov, spôsobenú pretínajúcimi záležitosťami. Problém TF-IDF váh pri inkrementálnych prístupoch však spočíva v tom, že sa ich hodnoty menia pridávaním, mazaním, ale aj upravovaním jednotlivých dokumentov. Aby sme tento problém eliminovali alebo dokonca využili v náš prospech, uchovávame podobnosť dokumentov v matici kovariancie. Predpokladáme, že ak sú dva dokumenty v určitej, nie príliš skorej etape vývoja podobné, mali by byť podobné aj v neskorších etapách. Dokonca sa domnievame, že nami určené hodnoty podobnosti sú presnejšie, nakoľko zohľadňujú aspekt času. Ak dva dokumenty boli vysoko podobné predtým ako sa výrazne zmenili hodnoty IDF (napr. pridaním modulu), predpokladáme, že by mali byť približne rovnako podobné aj po zmene korpusu, dokonca aj v tom prípade ak by prepočítané TF-IDF tvrdilo opak.

7.3.1. Prístup založený na hustote

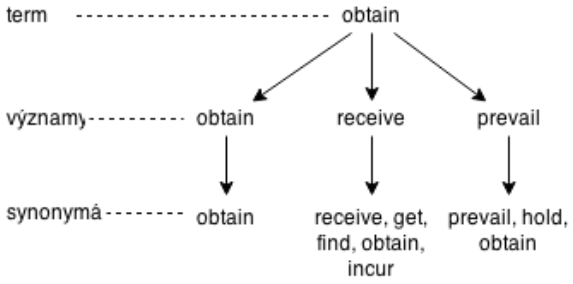
Tento prístup je založený na prezentovanom predpoklade nízkej dimenzionality subpriestoru inkrementov. Keď chceme pridať do matice kovariancie nový dokument, musíme vypočítať jeho podobnosť voči všetkým dokumentom, ktoré sa už v matici nachádzajú. Avšak, nakoľko počet dimenzií týchto prírastkov sa javí byť v priemere konštantný, iba jedna dimenzia tohto subpriestoru narastá a to dimenzia dokumentov. Táto dimenzia sa však tiež dá filtrovať, nakoľko je vysoká pravdepodobnosť, že relatívne nízko rozmerný vektor reprezentujúci triedu (v priemere 44 unikátnych termov), extrahovaný z vysoko rozmerného priestoru (v priemere 5085 unikátnych termov) nemá žiadne spoločné

alebo len veľmi málo spoločných termov so zostrojeným subpriestorom (v priemere 235 termov).

7.3.2. Prístup s náhodným indexovaním

Druhý prístup využíva techniku náhodného indexovania. Táto technika je vo svojej podstate inkrementálna, avšak v doméne zdrojových kódov sme narazili na niekoľko problémov. Nie sme si vedomí žiadnej práce, ktorá by túto techniku aplikovala v doméne zdrojových kódov. Definície kontextov prebrané z dokumentov prirodzeného jazyka by pravdepodobne dosahovali slabé výsledky, nakoľko sú dokumenty týchto domén lexikálne aj syntakticky rozdielne. Dokumenty extrahované zo zdrojových súborov sú malé, obsahujú len obmedzenú slovnú zásobu, ktorá je organizovaná do stanovených štruktúr. Prihliadajúc na tieto skutočnosti sme sa rozhodli aplikovať nový, netradičný prístup definície kontextu. Namiesto určovania kontextu na základe rozloženia slov v dokumentoch, určujeme kontext na základe vzťahov medzi slovami definovanými ontológiou. V našej implementácií navrhujeme používať ontológiu anglického jazyka Wordnet, ktorá obsahuje viac ako 117 000 množín synoným. Ako príklad uvádzame výpočet vektora kontextu anglického slova „obtain“ (viď. Obrázok 17).

Obrázok 17 - Výpočet vektora kontextu slova "obtain".



Slovo „obtain“ sa nachádza v ontológii v troch rôznych významoch: „obtain“, „receive“, „prevail“. Každému z týchto významov je priradená množina slov, ktoré tento význam zdieľajú. Kontext vektor slova „obtain“ teda určíme ako sumu všetkých index vektorov týchto slov, všetkých možných významov (obtain, receive, get, find, obtain, incur, prevail, hold, obtain). Zaujímavosťou tohto prístupu je, že jeho prirodzená forma výskytu „obtain“ má v kontexte väčšinou najvyššiu početnosť, a teda má vo vektore kontextu najvyššiu váhu. Nato, aby sme získali potrebné vektory indexu, stačí nám pre každé slovo v ontológií určiť

prislúchajúci náhodne vygenerovaný vektor konštantnej dĺžky. Keďže navrhujeme používať robustný tokenizér a lematizáciu je len malá pravdepodobnosť, že narazíme na term, ktorý sa v ontológii nebude nachádzať. Ak sa však takéto niečo stane, jednoducho ho pridáme do ontológie a pridelíme mu nový náhodne vygenerovaný vektor.

7.4. Zhlukovanie

Inšpiráciu nášho algoritmu sme tak ako algoritmus AntTree čerpali zo správania mravcov, ktoré v rôznych situáciách vytvárajú rôzne štruktúry. Zaujímavý pre nás je však príklad ohnivých mravcov, ktoré sú počas záplav schopné spoločným prepojením vytvoriť štruktúru podobnú člnu. Vďaka tejto štruktúre, v ktorej sú schopné vydržať týždne, dokážu prežiť. V našom prístupe predpokladáme, že každý jeden z mravcov dokáže vnímať len tie mravce, ktoré sa nachádzajú v jeho dosahu (napr. v určitej maximálnej vzdialenosti). Taktiež predpokladáme, že pred spájaním boli mravce rozmiestnené v priestore, takže tým ako sa každý mravec prichytí na najbližších susedov v jeho dosahu, formujú sa najprv zhluky najbližších mravcov a až následne sa tieto zhluky spoja do jedného veľkého plavidla. Správanie tohto druhu mravcov je často prirovnávané k správaniu tekutín. Pozorovanie, ktoré opisujeme možno prirovnať k šíreniu olejových škvŕn po vodnej hladine. Navrhujeme pracovať len so zjednodušeným modelom tejto štruktúry, kde sa každý mravec môže pripojiť len na jedného suseda (reálne na siedmich). Počet zachránených mravcov potom závisí od veľkosti plavidla (počtu prepojených mravcov). Aby sme tento počet maximalizovali navrhujeme zakázať tie spojenia, ktoré by spôsobili cykly. Týmto spôsobom získame kostru grafu, ktorá je blízka minimálnej alebo je minimálna. V našom prístupe každý mravec reprezentuje práve jeden dokument.

7.4.1. Vytváranie kostry

Zjednodušený model štruktúry založený na kostre sme zvolili z viacerých dôvodov. Kostru je jednoduché udržiavať, analyzovať a vizualizovať. Zhlukovacie algoritmy založené na kostre sú schopné identifikovať aj tie zhluky, ktorých hranice sú nepravidelné (nie sférické) [39]. Náš algoritmus vytvára kostru nasledovne:

1. Každý mravec, agent, má obmedzený dosah k najbližších susedov (podľa kosínusovej podobnosti získanej z matice kovariancie).
2. Každý agent chytí najbližšieho suseda v dosahu, s ktorým nevytvorí v grafe cyklus.

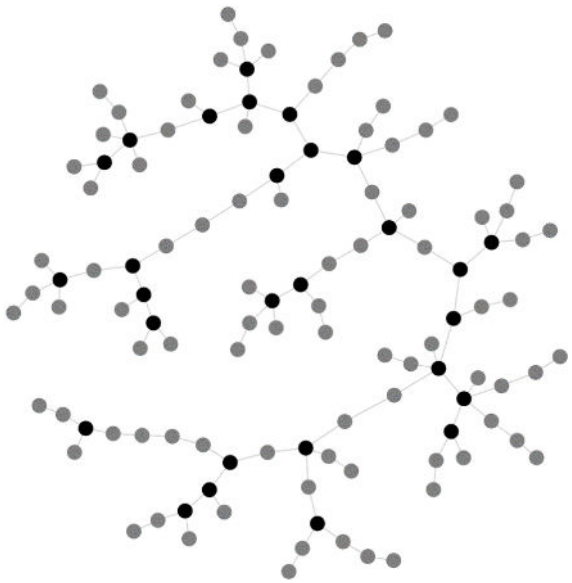
3. Po prejdení všetkých agentov dostaneme výslednú kostru.

Tento spôsob vytvárania kostry bol uprednostnený voči iným algoritmom na hľadanie minimálnych kostier z dôvodu, že využíva konkurenciu, ktorá má za následok tendenciu vytvárať relatívne rovnako veľké zhluky, no nie za tú cenu, aby vznikali zhluky neprirodzené. Ďalším dôvodom je, že takto vytvorená kostra je prirodzene inkrementálna. Takto vytvorenú kostru dokážeme tiež získať za cenu nižších výpočtových nárokov a predpokladáme, že použitie kostry, ktorá je len blízka minimálnej nemá pozorovateľný dopad na kvalitu výsledných zhlukov.

7.4.2. Identifikácia bodov zhlukovania

Vo vytvorenej kostre označíme tie uzly, ktoré majú viac ako 2 prepojenia (viď. Obrázok 18). Tieto uzly budeme nazývať body zhlukovania a predpokladáme, že predstavujú dobrú aproximáciu medoidu pre skupinu tvorenú samotným bodom zhlukovania a uzlov naň pripojených. Treba podotknúť, že táto skupina je tvorená navzájom podobnými dokumentmi, nakoľko jej uzly sú susedné v kostre, ktorá je blízka minimálnej. Uzly, ktoré majú menej ako dve prepojenia považujeme za listy alebo za takzvané spojky, ktoré len spájajú potenciálne zhluky a preto ich z hľadiska lokalizácie zhlukov považujeme za nerelevantné.

Obrázok 18 - Kostra a body zhlukovania (čierna).

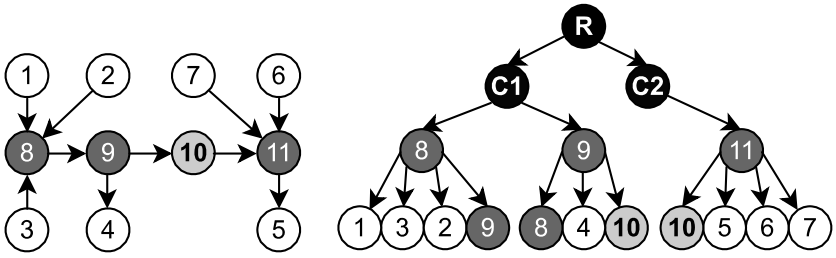


7.4.3. Formovanie zhlukov

Označenie bodov zhlukovania v kostre odhalilo vysokú pravdepodobnosť výskytu skupiniek navzájom prepojených bodov zhlukovania (viď. Obrázok 18). Tieto zoskupenia považujeme za kandidátov zhlukov. Na identifikáciu reálnych zhlukov z kandidátov možno zvoliť rozličné stratégie. Nami navrhovaná stratégia pozostáva z týchto krokov:

1. Filtrovanie tých bodov zhlukovania, ktoré patria zoskupeniu pozostávajúcemu z nie viac ako x bodov zhlukovania. Parameter x v tomto prípade určuje úroveň abstrakcie, a teda nepriamo určuje aj výsledný počet zhlukov. Hodnota tohto parametra by mala byť zvolená na základe celkového počtu dokumentov.
2. Zo zostávajúcich bodov zhlukovania sa inicializuje prehľadávanie do šírky:
    a. Ak prehľadávanie narazí na uzol, ktorý už je priradený, prehľadávanie z tohto bodu nepokračuje. Ak sa však prehľadávanie nachádza v prvej fáze, je tento uzol priradený aj tomuto bodu zhlukovania.
    b. Ak prehľadávanie narazí na uzol, ktorý priradený nie je, priradí ho bodu zhlukovania, z ktorého bolo toto prehľadávanie inicializované. Treba si uvedomiť, že priradenie uzlov v prvej fáze prehľadávania iba jednému bodu zhlukovania by zničilo našu reprezentáciu medoidu (mohlo by sa stať, že bod zhlukovania by nemal priradené žiadne uzly). Tento postup môže viesť k priradeniu jedného uzla viacerým zhlukom (viď. Obrázok 19, uzol 10). Tomuto správaniu sa možno vyhnúť použitím inej stratégie, ktorá zohľadňuje počet spojok medzi bodmi zhlukovania.
3. Zostrojenie podstromu z kostry, ktorý obsahuje len body zhlukovania. Pre každý komponent v tomto podstrome je každý uzol komponentu (bod zhlukovania) priradený zodpovedajúcemu zhluku. Zhluk je označený termom, ktorý sa najčastejšie vyskytuje v jeho dokumentoch.
4. Každý uzol reprezentujúci zhluk je napojený na koreň. Výsledná stromová štruktúra je znázornená na obrázku (viď. Obrázok 19). Tmavo šedé uzly na obrázku reprezentujú body zhlukovania, svetlo šedé spojky a čierne zhluky a koreň.

Obrázok 19 - Kostra (vľavo) a zodpovedajúca stromová hierarchia (vpravo).



7.4.4. Aktualizácia štruktúry

V našej práci sme navrhli dve hlavné stratégie v udržiavaní kostry a stromovej štruktúry. V prvej, model vektorového priestoru a matica kovariancie sú vytvárané inkrementálne a kostra spolu so stromovou hierarchiou sú vždy prepočítané, nakoľko vyžadujú len jednu iteráciu cez dokumenty. Táto stratégia bude pravdepodobne dosahovať presnejšie výsledky, avšak výpočtovo bude náročnejšia.

Druhá stratégia vytvára kostru a stromovú hierarchiu taktiež inkrementálne. Identifikované zhluky v tomto prípade budú stabilnejšie, no pravdepodobne menej presné.

Ktorú stratégiu je vhodné zvoliť závisí od celkového počtu dokumentov. Tieto stratégie možno taktiež kombinovať, čoho výsledkom by bola technika známa tiež pod názvom vkladanie (z angl. folding-in).

Aktualizácia kostry je triviálna záležitosť. Pokiaľ chceme pridať pozorovanie, stačí ho jednoducho v kostre pripojiť k najbližšiemu uzlu (platí rovnaké pravidlo s cyklami). Odstránenie uzlu z kostry spôsobí, že k uzlom, ktoré boli naň pripojené sa správame ako k novým pozorovaniam.

Aktualizácia stromovej hierarchie je zložitejší proces, avšak môže byť docielená aplikovaním týchto pravidiel:

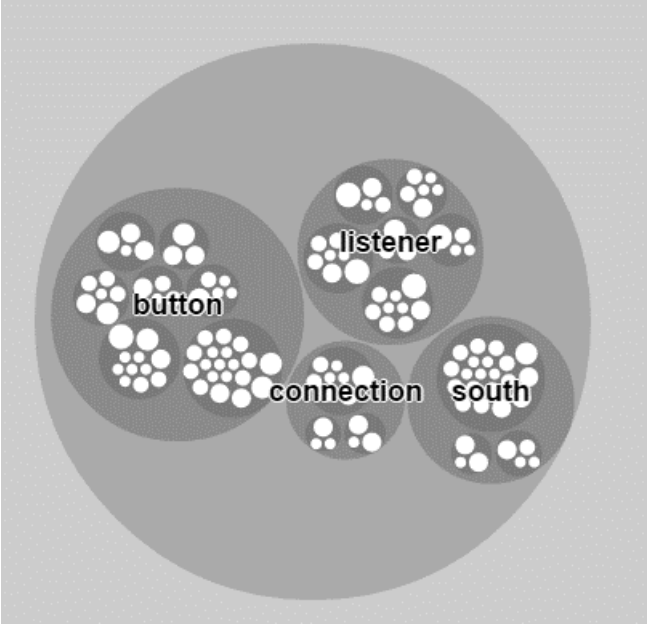
- Ak sa vytvorí nový bod zhlukovania v kostre, musíme inicializovať prehľadávanie do šírky z tohto uzla. Vytvorí sa zoznam dosiahnutých bodov zhlukovania. Priradenie uzlov k týmto bodom zhlukovania musí byť prehodnotené.
- Ak nové pozorovania sú priradené k existujúcim bodom zhlukovania, netreba vykonať žiadne ďalšie kroky.

- Ak nové pozorovanie je pripojené k listu, je toto pozorovanie priradené k rovnakému bodu zhlukovania.
- Ak odstránime bod zhlukovania z kostry, všetky jeho deti musia byť znovu priradené.
- Ak odstránime list, netreba vykonať žiadne ďalšie kroky.
- Ak odstránime spojku, musíme prehodnotiť iba uzly, ktoré sa nachádzajú medzi týmto uzlom a ostatnými bodmi zhlukovania.

7.5. Vizualizácia

V rámci prototypu sme vyskúšali dve rôzne vizualizácie, dendrogram (viď.Obrázok 9) a graf (viď.Obrázok 18). Na základe získaných skúseností s jednotlivými reprezentáciami sme sa rozhodli vytvoriť vizualizáciu zhlukov prostredníctvom webovej stránky s využitím identifikovanej hierarchie (viď. Obrázok 20).

Obrázok 20 - Príklad hierarchickej vizualizácie.



7.6. Dolovanie aspektov

Nami navrhovaný prístup je založený na zhlukoch a ich rozložení v balíkoch. Princíp tejto metódy možno opísať v nasledujúcich krokoch:

1. **Vytvorenie zhlukov.** Kovariančná matica predstavuje vstup pre zhlukovací algoritmus. V našom prípade používame metódu aglomeratívneho zhlukovania CLCA (z angl.

complete linkage clustering analysis). Ako kritérium pri vytváraní zhlukov používame maximálnu vzdialenosť v rámci zhlukov, ktorá v tomto prípade bola určená manuálne (3.35 pre projekt JHotDraw). Vlastný zhlukovací algoritmus sme nepoužili z toho dôvodu, aby sme zamedzili potencionálnemu znehodnoteniu výsledkov experimentu. V prípade, že by sa v implementácii nášho zhlukovacieho algoritmu vyskytovala chyba, museli by sme celý experiment opakovať.

2. **Identifikácia pretínajúcich tém.** Ako pretínajúcu tému označujeme ten zhluk, ktorý sa rozkladá cez viac ako 4 balíky. Tento parameter musí byť tiež manuálne určený.
3. **Ohodnotenie termov.** Termy v rámci každej pretínajúcej témy sú ohodnotené podľa toho, v koľkých balíkoch sa vyskytujú. Tie termy, ktoré majú vyššie ohodnotenie ako 2 budeme označovať pretínajúce termy.
4. **Ohodnotenie metód.** Na základe pretínajúcich termov, je ohodnotená každá metóda pretínajúcej témy nasledovne:

$$Rm = \sum_{i}^{N_c} Rt_i \cdot \frac{N_c}{N} \qquad (1)$$

kde N označuje počet termov, z ktorých sa skladá názov metódy, N_c počet obsiahnutých pretínajúcich termov a Rt_i ohodnotenie i-teho pretínajúceho termu v danej téme. Zlomok pretínajúcich termov bol zavedený za účelom diskriminácie metód zložených z veľkého počtu termov, ktoré prirodzene majú vyššiu pravdepodobnosť výskytu pretínajúceho termu.

5. **Filtrovanie výsledkov.** V rámci ohodnocovania, podobne ako Fan In analýza, ignorujeme tie metódy, ktoré začínajú slovami "get", "set" alebo "to". Výsledkom metódy je zoznam pretínajúcich tém a k nim prislúchajúcich ohodnotených metód, ktorých hodnota je vyššia ako hodnota prahu (nami určená 2) .

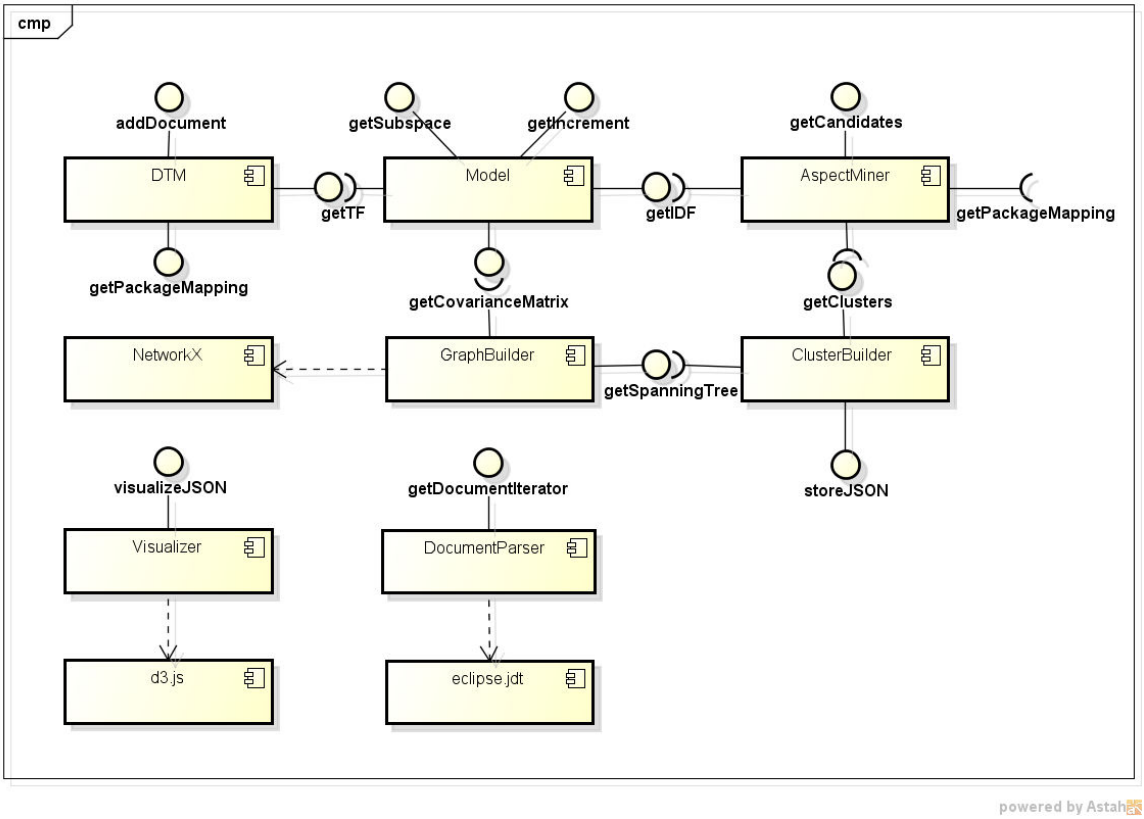
8 Implementácia

Realizácia nášho projektu pozostáva zo série skriptov v jazyku Python (jadro), série zdrojových súborov v jazyku Java (predspracovanie dokumentov) a webovej stránky pozostávajúcej z html a javascript kódu (vizualizácia). Počas implementácie sme sa snažili zamerať najmä ma modularitu výsledného riešenia, ktorá nám počas experimentovania umožní jednoducho zamieňať moduly reprezentujúce rôzne navrhované stratégie.

8.1. Architektúra

Celkový obraz štruktúry našej implementácie zobrazuje diagram komponentov (viď. Obrázok 21).

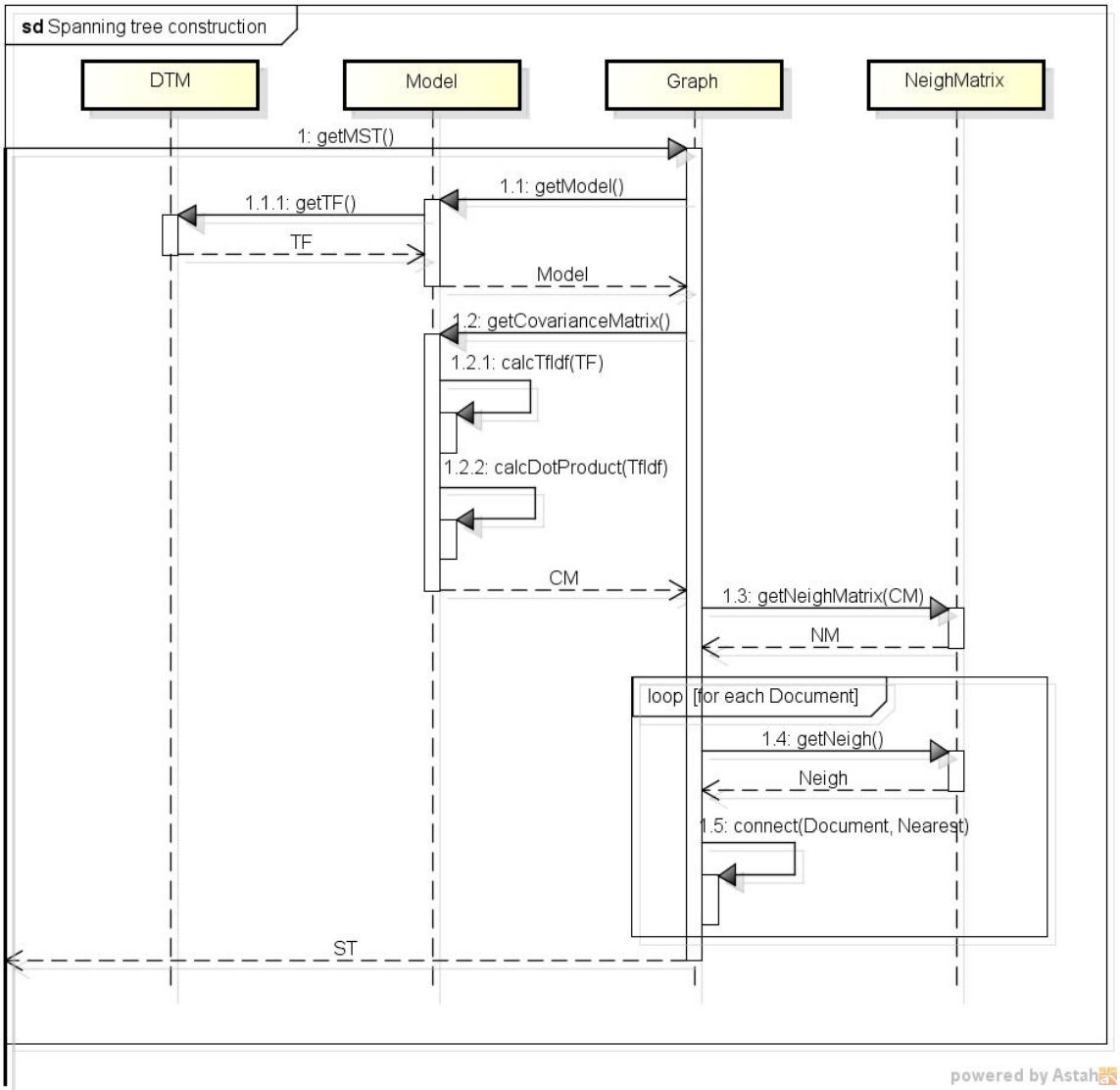
Obrázok 21 - Diagram komponentov prezentovaného riešenia.



Hlavný komponent, Model, uchováva maticu nenormalizovaných TF váh získaných z DTM. Na základe týchto váh vytvára v prípade potreby maticu aktuálnych TF-IDF váh, z ktorej dokáže určiť subpriestor (z angl. subspace), vypočítať celú maticu kovariancie alebo získať len jej prírastok.

Komponent GraphBuilder sa stará o vytváranie kostry, ktorá predstavuje abstrakciu prepojenia mravcov. Postupnosť jednotlivých interakcií pri jej vytváraní je znázornená v diagrame sekvencií (viď. Obrázok 22). Kostru vytvára na základe matice kovariancie, ktorú získava od komponentu Model. Na prácu s grafmi využíva knižnicu NetworkX.

Obrázok 22 - Diagram sekvencií procesu vytvárania kostry.

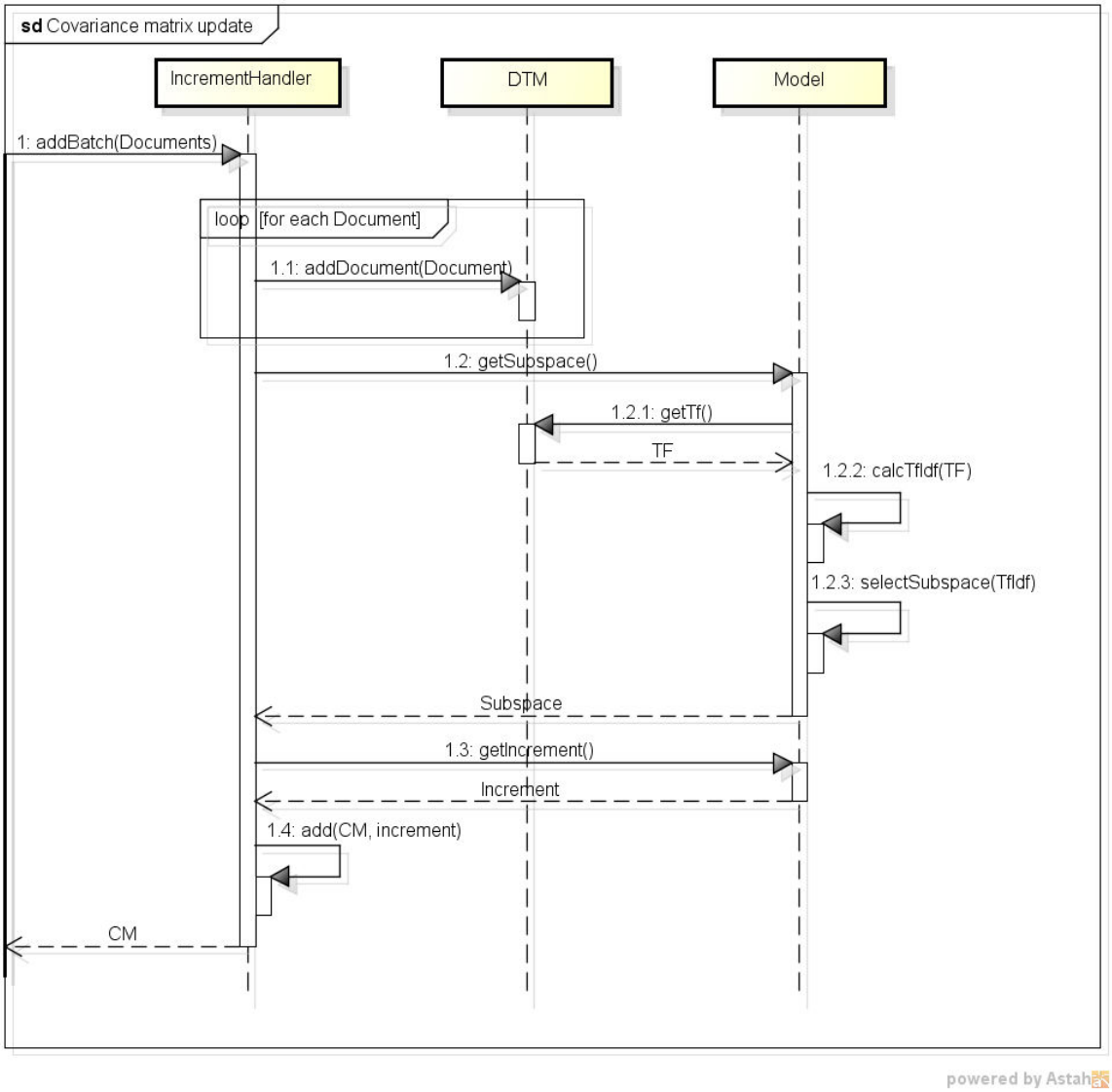


Komponent ClusterBuilder vytvára na základe kostry poskytnutej komponentom GraphBuilder jednotlivé zhluky. Tie sú reprezentované pomocou hierarchického stromu, ktorý sa následne ukladá vo formáte JSON na disk.

Komponent Visualizer slúži na vykreslenie identifikovaných zhlukov, načítaných zo súboru JSON. Na vykreslenie dát využíva knižnicu d3.js.

Komponent DocumentParser, implementovaný v jazyku Java, vytvára iterátor nad dokumentmi extrahovanými zo zdrojových súborov. Na extrakciu využíva poznatky získané analýzou abstraktného syntaktického stromu, ktorý nám poskytla knižnica Eclipse JDT. Komunikácia medzi komponentmi v jazyku Java a v jazyku Python je zabezpečená pomocou knižnice Py4J.

Obrázok 23 - Diagram sekvencií, aktualizácia matice kovariancie.



Využitím opísaných komponentov možno jednoducho implementovať aktualizácie kovariančnej matice v prípade pridania nových dokumentov. Tento proces je podrobne opísaný v diagrame sekvencií (viď. Obrázok 23).

AspectMiner komponent predstavuje druhú časť nášho projektu zameranú na dolovanie aspektov. Pri analýze využíva vektor IDF, získaný od komponentu Model, zhluky identifikované komponentom ClusterBuilder a príslušnosť jednotlivých dokumentov (tried) k balíkom. Výsledkom analýzy je zoznam ohodnotených aspektových kandidátov.

8.2. Predspracovanie dokumentov

Vo fáze predspracovania dokumentov využívame nástroje sady Eclipse JDT. Dokumenty extrahujeme pomocou triedy ASTVisitor. Pri prechádzaní uzlov AST berieme do úvahy len tie, ktoré sú typu TypeDeclaration. Každý takýto uzol predstavuje jeden dokument (triedu). Na základe väzieb identifikovaných pomocou ITypeBinding vieme pre každú triedu určiť nadtriedu pokiaľ existuje. To nám umožní neskôr obohatiť vektor tejto triedy o kľúčové pojmy nadtriedy. V našej implementácii obohacujeme s váhou 1.0.

Pomocou triedy IScanner tokenizujeme extrahované dokumenty. V tejto fáze môžeme rôznym typom tokenov priraďovať rôzne váhy. Naša implementácia využíva túto skutočnosť len v prípade Javadoc komentárov. Tým sme priradili váhu 0.5. Jeden z typov tokenov reprezentuje kľúčové slová vyhradené pre jazyk Java. Tokeny tohto typu pri spracovaní úplne ignorujeme.

Druhý typ váh tokenov aplikujeme na základe AST. Ak sa tokeny dokumentu nachádzajú v názve deklarácie typu prideľujeme im váhu 4.0. Ak sa tieto tokeny nachádzajú v názve deklarácie metódy prideľujeme im váhu 3.0.

Na každý extrahovaný token aplikujeme tokenizér s názvom IdentifierNameTokenizer [2]. Tento tokenizér dokáže extrahovať termy z identifikátorov aj v prípade, že ich názov nezodpovedá konvenciám (napr. CamelCase). Následne z týchto termov filtrujeme tie, ktoré sa nachádzajú v zozname anglických stop slov a taktiež slová „get“ a „set“. Taktiež filtrujeme termy kratšie ako 3 znaky alebo termy pozostávajúce len z čísel. Z každého tokenu nám môže vzniknúť viacero termov. Tie sú následne normalizované pomocou lematizácie. Lokálna váha každého termu je určená vynásobením počtu výskytov tokenu z ktorého pochádza s AST váhou a IScanner váhou tohto tokenu. Globálna váha termu je teda následne určená spočítaním lokálnych váh termu v danom dokumente. Výsledkom

tejto fázy predspracovania je slovník pozostávajúci z kľúčov (termov) a k nim prislúchajúcich hodnôt (váh).

8.3. Reprezentácia korpusu

Obe matice, primárna aj sekundárna, sú riedke (0.64-1.58%). Ich ukladanie v klasických formátoch by viedlo k zahlteniu pamäte už pri veľmi malom počte spracovaných dokumentov. Aby sme zamedzili tomuto správaniu, reprezentujeme obe matice vhodnými formátmi. Primárnu maticu reprezentujeme pomocou DOK formátu, nakoľko zasahujeme do jej štruktúry a zároveň na ňu neaplikujeme žiadne výpočty. Za ukladanie a načítanie primárnej matice zodpovedá štandardný modul cPickle. Sekundárna matica je reprezentovaná CSR formátom, pretože na ňu často aplikujeme výpočet skalárneho súčinu. Obe reprezentácie sú súčasťou knižnice scipy, modulu scipy.sparse. Potenciálny problém, identifikovaný počas implementácie, predstavuje hustota kovariančnej matice, ktorá dosahuje podstatne vyššie hodnoty ako hustota DTM, a teda aj podstatne vyššie pamäťové nároky. Nakoľko však náš prístup vychádza z kostry zostrojenej spájaním vysoko podobných dvojíc, nemusíme si pamätať všetky hodnoty. Napríklad odfiltrovaním hodnôt menších ako 0.1 sme znížili hustotu matice zo 72% až na 10%. Ďalšou možnosťou je pamätať si v každom riadku kovariančnej matice len konštantný počet najvyšších hodnôt.

Implementáciu algoritmu náhodného indexovania používame našu vlastnú. Akceptuje dva vstupné parametre: dĺžku vektora a počet kladných aj záporných položiek. Predvolenú hodnotu dĺžky vektora sme nastavili na 1024. Počet kladných aj záporných položiek bol nastavený na 4.

8.4. Zhlukovanie

Nami implementovaný algoritmus zhlukovania prijíma iba jeden parameter, úroveň abstrakcie. Na reprezentáciu kostry sme použili triedu Graph knižnice Netwrokx. V prípade stromovej štruktúry sme potrebovali orientovaný graf, a teda sme použili triedu DiGraph. Algoritmus prehľadávania do šírky sme museli implementovať vlastný, nakoľko implementácia Networkx nepodporuje prehľadávanie inicializované z viacerých bodov súčasne. Pri vytváraní kostry sme nastavili veľkosť okolia na 7 najbližších mravcov. Z knižnice Networkx sme použili aj metódu na vytváranie podgrafu a metódu na export grafovej štruktúry do súboru JSON.

8.5. Vizualizácia

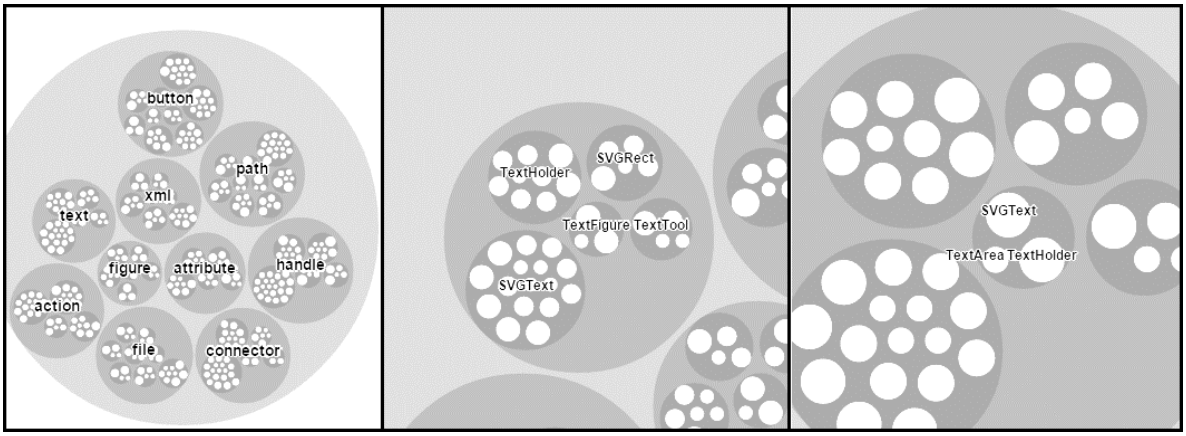
Za účelom vizualizácie sme implementovali html stránku s použitím Javascript knižnice d3.js. Vstup vizualizácie predstavuje JSON súbor s uloženou stromovou hierarchiou. Pre zobrazenie zhlukov sme modifikovali d3.js šablónu s názvom BubbleLayout. Ako náhľad jednotlivých dokumentov sme poskytli možnosť prezerania príslušného kódu priamo v prehliadači.

9 Overenie

Nakoľko v našej práci kladieme dôraz na identifikáciu prirodzených zhlukov, externé techniky validácie by pravdepodobne boli najvhodnejšie pre vyhodnotenie kvality identifikovaných zhlukov. Nakoľko však neexistuje vhodná, vyhodnotená množina dát, aplikujeme metódu relevantných úsudkov (z angl. relevance judgments).

Pri aplikovaní nášho prístupu na malý projekt JHotDraw, s abstrakciou x=2, získame celkovo 10 zhlukov (viď. Obrázok 24). Taktiež možno pozorovať, že získane zhluky sú skutočne približne rovnako veľké. Pri porovnávaní získaných zhlukov s organizáciou tried do balíkov sme narazili na 3 zhody. Balík „figures“ je v rámci JHotDraw definovaný ako sada figúr a ich nástrojov. Náš zhlukovací algoritmus veľmi presne identifikoval zhluk „figure“ a dokonca od neho aj oddelil dodatočný zhluk s názvom „handle“. Ďalší balík, „application“, definovaný ako nástroje spojené s užívateľským prostredím, bol taktiež veľmi presne identifikovaný našim algoritmom pod názvom „action“. Ostatné zhluky nezodpovedajú balíkom tak priamo, ale tento fakt nepredstavuje negatívne pozorovanie. Práve naopak, pozorujeme veľmi dobre formované zhluky. Príkladom je zhluk „file“, ktorý pozostáva z tried zodpovedných za načítavanie, ukladanie, otváranie, zapisovanie a zatváranie projektových súborov. Ďalším veľmi vhodným príkladom je zhluk „xml“, ktorý pozostáva z tried zodpovedných za validáciu, spracovanie, čítanie a vytváranie xml dokumentov.

Obrázok 24- Kruhové rozloženie (JHotDraw).

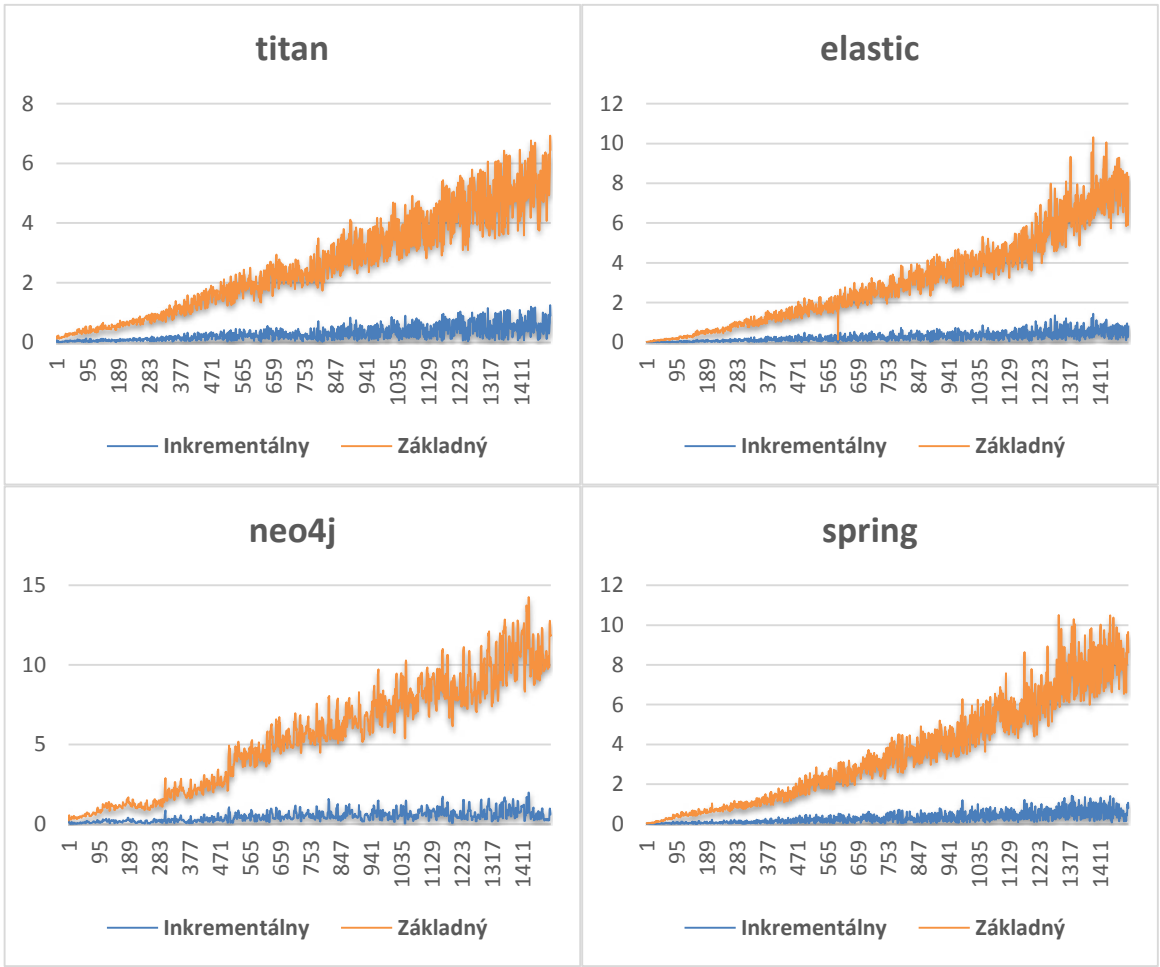


Kvalitu bodov zhlukovania ako reprezentantov hodnotíme tiež pozitívne. Dobrým príkladom je bod zhlukovania „MoveHandle“, ktorý reprezentuje dokumenty:

„MoveToFrontAction“, „MoveAction“, „MoveToBackAction“, „WestHandle“ a „HandleListener". Podobné správania možno pozorovať pri všetkých projektoch.

Na základe vyhodnotenia časovej zložitosti (viď. Obrázok 25) môžeme usúdiť, že nami navrhnutý inkrementálny prístup vytvára matice kovariancie v podstatne nižšom čase. Taktiež môžeme pozorovať lineárne škálovanie s pomalým rastom pre inkrementálny prístup a pravdepodobne mocninové škálovanie s rýchlym rastom pre základný prístup.

Obrázok 25 - Porovnanie časov inkrementálneho a základného prístupu pri výpočte matice kovariancie.



Zaujímavé pozorovanie zišlo z porovnania reprezentácií korpusu (viď. Tabuľka 14). Vidíme, že hoci technika náhodného indexovania podstatne redukovala celkový priestor, subpriestor s ktorým pracujeme pri jednotlivých inkrementoch má paradoxne dimenzionalitu v priemere vyššiu. Je možné, že sme dĺžku vektorov stanovili zbytočne

vysokú, no zároveň nepredpokladáme, že by 200 dimenzií bolo pre túto techniku postačujúcich.

Tabuľka 14 - Priemerný počet dimenzií prírastku, RI a prístup založenú na hustote.

	Neo4j	Elastic	Spring	Titan
hustota	201.92	172.07	270.07	297.14
RI	741.43	777.68	691.98	803.31

Pre potvrdenie vzájomnej podobnosti dokumentov inkrementu sme vypočítali priemernú hodnotu v matici kovariancie (viď. Tabuľka 15). Diagonálne hodnoty v matici sme predtým vynulovali, no i napriek tomu môžeme pozorovať vysoké hodnoty podobnosti.

Tabuľka 15 - Priemerná hodnota v matici kovariancie (diagonála nastavená na 0).

	Titan	Elastic	Neo4j	Spring	celkovo
priemer	0.35	0.48	0.34	0.39	0.39

Pre overenie úspešnosti aproximácie medoidov sme určili sumu vzdialeností priradených uzlov od pravého medoidu a nami odhadovaného medoidu. Chybu sme určili ako rozdiel týchto dvoch súm. Z dát v tabuľke (viď. Tabuľka 16) vidíme, že naša aproximácia medoidu zlyhá v len zanedbateľnom množstve prípadov a aj to len s nízkou priemernou chybou.

Obrázok 26 - Vyhodnotenie aproximácie medoidu.

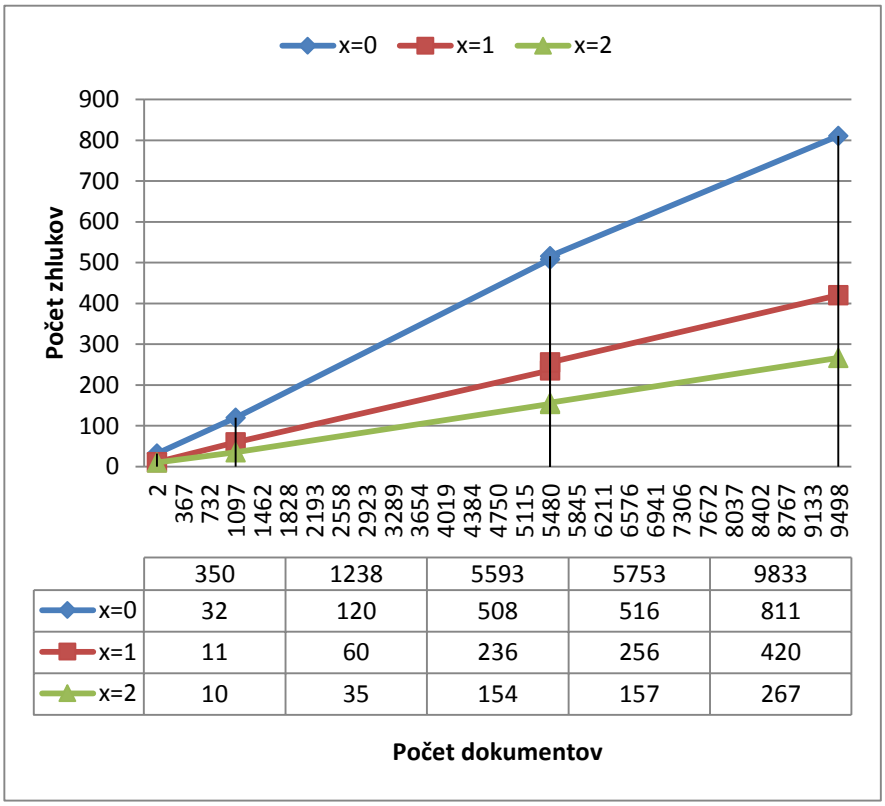
	nesprávne	celkovo	chyba	priemerná chyba
JHotDraw	4	86	0.07	0.02
Titan	8	310	0.22	0.03
Elastic	73	1336	3.96	0.05
Neo4j	68	1396	3.34	0.05
Spring	156	2349	4.58	0.03

Tabuľka 16 znázorňuje výsledky overenia správnosti nami definovaného kontextu. Môžeme vidieť, že medzi hodnotami kovariančnej matice získanej klasickým spôsobom a hodnotami matice kovariancie získanej pomocou RI existuje silný vzťah. Druhé pozorovanie, ktoré nás v tejto domnienke utvrdzuje je minimálny vplyv na výsledné zhluky v prípade, že tieto stratégie zameníme.

Tabuľka 16 - Hodnota korelácie medzi kovariančnými maticami.

	JHotDraw	Titan	Elastic
korelácia	0.655202	0.660299	0.647714

Počet identifikovaných zhlukov sa javí byť lineárne závislý od celkového počtu dokumentov. Teoreticky by mohlo ísť aj o logaritmické škálovanie, avšak takéto závery by vyžadovali štatisticky významnejšiu vzorku dát.



	350	1238	5593	5753	9833
x=0	32	120	508	516	811
x=1	11	60	236	256	420
x=2	10	35	154	157	267

10 Zhodnotenie

V tejto práci prezentujeme novú, inkrementálnu techniku vizualizácie softvérových systémov s využitím zhlukovania, založenej na správaní kolónie ohnivých mravcov. Implementované riešenie je len mierne citlivé na zmenu jediného parametra, ktorý poskytuje a má tendenciu vytvárať približne rovnako veľké zhluky a dokonca aj v prípade, že zhluky nie sú sférické.

Navrhli sme rôzne stratégie v krokoch predspracovania, zhlukovania, ale aj vizualizácie. Identifikovali sme niektoré zaujímavé vlastnosti zdrojových kódov a zdrojových domén.

V evaluácií sme podporili naše domnienky, predpoklady a skutočnosť, že náš prístup je skutočne schopný identifikovať prirodzené zhluky, ktoré považujeme za najdôležitejšie v zmysle vizualizácie softvérových systémov.

Sme si vedomí mnohých potenciálnych vylepšení nášho prístupu. V oblasti optimalizácie by sme radi analyzovali možnosti aproximácie k najbližších hodnôt v lineárnom čase. Taktiež by nás zaujímalo ako by aplikácia metódy df-cut, mapovania synoným na jednu spoločnú dimenziu a filtrovanie dimenzie dokumentov ovplyvnilo konečnú hustotu matice a tým pádom aj časovú náročnosť.

Počas experimentov nás napadla zaujímavá otázka: Je možné používať dáta získané zo systému na kontrolu verzie (ako napr. Git) pri formovaní zhlukov? Nakoľko sme dokázali, že dokumenty ovplyvnené jedným odovzdaním sú si naozaj podobné, predpokladáme, že takáto technika by mohla byť skutočne efektívna.

11 Zoznam použitej literatúry

[1] Brian S. Everitt, Sabine Landau, and Morven Leese. 2009. Cluster Analysis (4th ed.). Wiley Publishing.

[2] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In Proceedings of the 25th European conference on Object-oriented programming (ECOOP'11), Mira Mezini (Ed.). Springer-Verlag, Berlin, Heidelberg, 130-154.

[3] Eduard Dragut, Fang Fang, Prasad Sistla, Clement Yu, and Weiyi Meng. 2009. Stop word and related problems in web interface integration. Proc. VLDB Endow. 2, 1 (August 2009), 349-360.

[4] Jiaul H. Paik, Swapan K. Parui, Dipasree Pal, and Stephen E. Robertson. 2013. Effective and Robust Query-Based Stemming. ACM Trans. Inf. Syst. 31, 4, Article 18 (November 2013), 29 pages.

[5] Huang A., Similarity measures for text document clustering. (2008), 49-56.

[6] Radim Řehůřek. 2011. Subspace tracking for latent semantic analysis. In Proceedings of the 33rd European conference on Advances in information retrieval (ECIR'11), Paul Clough, Colum Foley, Cathal Gurrin, Hyowon Lee, and Gareth J. F. Jones (Eds.). Springer-Verlag, Berlin, Heidelberg, 289-300.

[7] J. E. Mason and R. J. Spiteri, A new adaptive folding-up algorithm for information retrieval, 2007

[8] Baker, Christopher G., Kyle A. Gallivan, and Paul Van Dooren. "Low rank incremental methods for computing dominant singular subspaces." Linear Algebra and its Applications 436.8 (2012): 2866-2888.

[9] Brand, M.,Fast low-rank modifications of the thin singular value decomposition. Linear Algebra and its Applications 415(1), 20–30 (2006)

[10] Levy, A., Lindenbaum, M.: Sequential Karhunen–Loeve basis extraction and its application to images. IEEE Transactions on Image processing 9(8), 1371 (2000)

[11] Zha, H., Simon, H.: On updating problems in Latent Semantic Indexing. SIAM Journal on Scientific Computing 21, 782 (1999)

[12] Sanjoy Dasgupta and Anupam Gupta. 2003. An elementary proof of a theorem of Johnson and Lindenstrauss. Random Struct. Algorithms 22, 1 (January 2003), 60-65.

[13] Sophia Sakellaridi, Haw-ren Fang, and Yousef Saad. 2008. Graph-Based Multilevel Dimensionality Reduction with Applications to Eigenfaces and Latent Semantic Indexing. In Proceedings of the 2008 Seventh International Conference on Machine Learning and Applications (ICMLA '08). IEEE Computer Society, Washington, DC, USA, 194-200.

[14] Vinay, Vishwa, et al. "A comparison of dimensionality reduction techniques for text retrieval." Machine Learning and Applications, 2005. Proceedings. Fourth International Conference on. IEEE, 2005.

[15] Sampath Deegalla and Henrik Bostrom. 2006. Reducing High-Dimensional Data by Principal Component Analysis. Random Projection for Nearest Neighbor Classification. In Proceedings of the 5th International Conference on Machine Learning and Applications (ICMLA '06). IEEE Computer Society, Washington, DC, USA, 245-250.

[16] J.-L. Deneubourg, S. Gross, N. Franks, A. Sendova-Franks, C. Detrain and L. Chretien, " The dynamics of collective sorting: Robot-like ants and ant-like robots", In Proceedings of the First International Conference on Simulation of Adaptive Behavior: From Animals to Animats, Cambridge, MA, MIT Press, 1991, pp. 356-363.

[17] E.D. Lumer and B. Faieta, "Diversity and adaptation in populations of clustering ants", Cambridge: MIT Press, In D. Cliff, P. Husbands, J.-A. Meyer, & S.W. Wilson (Eds.), From animals to animats: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, 1994, pp. 501-508.

[18] H. Gutowitz, "Complexity-seeking ants", In Proc. Of the Third European Conference on Artificial Life, 1993.

[19] V. Ramos, F. Muge and P.Pina, "Self-Organized Data and Image Retrievals Consequence of Inter-Dynamic Synergistic Relationships in Artificial Ant Colonies", In J. Ruiz-del-Solar, A. Abrahan and M. Koppen Eds., Soft-Computing Systems – Design, Management and Applications, Frontiers in Artificial Intelligence and Applications: IOS Press, Amsterdam, v. 87, 2002, pp. 500-509.

[20] V. Ramos and Ajith Abraham, "Swarms on continuous data", In CEC'03 Congress on Evolutionary Computation, IEEE Press, 2003, pp. 1370-1375.

[21] J. Handl, J and B. Meyer, “Improved Ant-Based Clustering and Sorting in a Document Retrieval Interface”, Proceedings of the 7th International Conference on Parallel Problem Solving from Nature, LNCS 2439, 2002.

[22] J. Handl, J. Knowles and M. Dorigo, “Strategies for the increased robustness of ant-based clustering, Lecture Notes in Computer Science, Vol. 2977, 2004, pp. 90-104.

[23] H. Azzag, N. Monmarche, M. Slimane and G. Venturini, “AntTree: a new model for clustering with artificial ants”, Evolutionary Computation, CEC’03, Vol. 4, 2003, 2642-2647.

[24] Diego Alejandro Ingaramo, Guillermo Leguizamon and Marcelo Errecalde, “Adaptive clustering with artificial ants”, Journal of Computer Science and Technology, Science Press, Vol. 5, No. 4, 2005.

[25] Xiaohua Xu and Ling Chen Chen Y., “A4C: anadaptive artificial ants clustering algorithm”, Proceedings of the 2004 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology, 2004, CIBCB’04, 2004, pp. 268-275.

[26] Yan Yang and Mohamed S. Kamel, “Anaggregatedclusteringapproachusingmulti-antcoloniesalgorithms”, Pattern Recognition, Vol. 39, Issue 7, July 2006, pp. 1278-1289.

[27] Zahra Sadeghi, Mohammad Teshnehlab and Mir Mohsen Pedram, “K-Ants Clustering – A New Strategy Based on Ant Clustering”, ICS 2008, Tamkang, Vol. 2, 12 Feb. 2009, pp. 329-334.

[28] Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. J. Comput. Syst. Sci. 66, 4 (June 2003), 671-687.

[29] SAKRE, Mohammed M, KOUTA, Mohammed M, ALLAM, Ali M N. WEIGHTING QUERY TERMS USING WORDNET ONTOLOGY. . 2009, vol. 9, no. 4, s. 349 – 358.

[30] FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code. . 2002

[31] BASSETT, B, KRAFT, NA. Structural information based term weighting in text retrieval for feature location. In: *Program Comprehension (ICPC), 2013 …* [online]. 2013. [cit. 11.12.2014].

[32] MAHMOUD, Anas, NIU, Nan. Source code indexing for automated tracing. In: *Proceeding of the 6th international workshop on Traceability in emerging forms of software engineering - TEFSE '11* [online]. 2011, s. 3.

[33] STEIN, Benno, MEYER, Sven, POTTHAST, Martin. Syntax versus Semantics : Analysis of Enriched Vector Space Models. . 2006, no. August, s. 47 – 52.

[34] STU, Fiit. Fakulta informatiky a informačných technológií Vyhľadávanie využiteľných znalostí v rámci softvérového projektu a ich grafická reprezentácia Diplomová práca. . 2012

[35] ZHANG, Tian, RAMAKRISHNAN, R, LIVNY, M. BIRCH: an efficient data clustering method for very large databases. In: *ACM SIGMOD Record* [online]. 1996, vol. 1. [cit. 11.12.2013], s. 103 – 114.

[36] KARPOV, Ilya, FEDERAL, Alexandr Goroslavskiy. Application of BIRCH to text clustering. , s. 102 – 105.

[37] Sahlgren, Magnus. 2005. An introduction to random indexing. In Methods and applications of semantic indexing workshop at the 7th international conference on terminology and knowledge engineering, TKE (Vol. 5).

[38] Marin, M.; van Deursen, A.; Moonen, L., "Identifying aspects using fan-in analysis," Reverse Engineering, 2004. Proceedings. 11th Working Conference on , vol., no., pp.132,141, 8-12 Nov. 2004

[39] C. T. Zahn. 1971. Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters. IEEE Trans. Comput. 20, 1 (January 1971), 68-86.

[40] FILLUS, EK, VERGILIO, SR. A Clustering Based Approach for Aspect Mining and Pointcut Identification. In: 6th Latin American Workshop on Aspect-Oriented Software Development [online]. 2012.

[41] ZHANG, Danfeng, GUO, Yao, CHEN, Xiangqun. Automated Aspect Recommendation through Clustering-Based Fan-in Analysis. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering [online]. 2008. s. 278 – 287.

[42] MOLDOVAN, GS, SERBAN, G. Aspect mining using a vector-space model based clustering approach. In: Proceedings of Linking Aspect Technology and evolution [online]. 2006.

[43] TARR, P. et al. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). 1999

Príloha A Obsah CD

CD obsahuje:

- elektronickú verziu práce vo formáte pdf
- elektronické verzie článkov vo formáte pdf
- priečinok src obsahujúci zdrojové kódy
- priečinok bin obsahujúci jar export implementácie v jazyku Java (serverovú časť)

Príloha B Technická dokumentácia

B.1 Minimálne požiadavky

Zoznam podporných prostriedkov a knižníc:

- Python 2.7.8 +
    - Numpy
    - Networkx
    - Scipy
    - Scikit
    - Py2Neo
    - Py4j
    - Nltk
    - Matplotlib
- Java 1.8.0 +
    - org.eclipse.core.contenttype_3.4.200.v20140207-1251.jar
    - org.eclipse.core.jobs_3.6.0.v20140424-0053.jar
    - org.eclipse.core.resources_3.9.0.v20140514-1307.jar
    - org.eclipse.core.runtime_3.10.0.v20140318-2214.jar
    - org.eclipse.equinox.common_3.6.200.v20130402-1505.jar
    - org.eclipse.equinox.preferences_3.5.200.v20140224-1527.jar
    - org.eclipse.jdt.core_3.10.0.v20140604-1726.jar
    - org.eclipse.osgi_3.10.0.v20140606-1445.jar
    - commons-io-2.4.jar
    - intt.jar
    - py4j0.8.2.1.jar
    - jsoup-1.8.1.jar

Príloha C Článok vytvorený v rámci predmetu AOVS

Dolovanie aspektov s využitím zhlukov tried

Juraj VINCÚR

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xvincurj@stuba.sk

Abstrakt. Dolovanie aspektov je proces identifikácie pretínajúcich záležitostí v existujúcich systémoch. Motiváciou v tomto prípade je snaha refaktorovať existujúce objektovo orientované systémy na ich aspektové formy, ktoré sú menej náročné na vývoj a údržbu.

V tejto práci prezentujeme nový prístup, ktorý využíva zhluky tried hierarchického zhlukovania a ich rozloženie cez balíky. Za indikátor pretínajúcich záležitostí považujeme zvýšený výskyt balíkov, v ktorých boli triedy daného zhluku deklarované. Na overenie našej metódy využívame komparatívnu analýzu z ďalšími tromi metódami aplikovanými na GUI rámec JHotDraw implementovaný v jazyku Java.

1 Úvod

Aj v dobre dekomponovaných systémoch sa vyskytujú pretínajúce záležitosti, ktoré pomocou OOP prostriedkov nemožno eliminovať. Tento problém je tiež známy ako tyrania dominantnej dekompozície (z angl. tyranny of the dominant decomposition) [1]. Symptómami pretínajúcich záležitostí sú zapletenie (z angl. tangling) a roztrúsenie (z angl. scattering) kódu, ktorých dôsledkom je zvýšenie úsilia potrebného pri vývoji alebo údržbe systému.

Aspektovo orientované programovanie umožňuje modularizáciu týchto záležitostí ich transformáciou do aspektov. Existujúce OOP systémy, ktoré chceme refaktorovať sú však často rozsiahle, a preto proces identifikácie potenciálnych aspektových kandidátov v kóde je časovo náročný a vyžaduje určitú úroveň automatizácie. Existuje množstvo techník dolovania aspektov, medzi najznámejšie patria Fan In analýza, analýza na základe identifikátorov (z angl. identifier analysis) a dynamická analýza.

V našej práci navrhneme nový prístup, ktorý bude využívať dáta potrebné pri procese zhlukovania zdrojových kódov. Technika zhlukovania sa využíva na získanie informatívneho zobrazenia, ktoré umožňuje programátorovi vytvoriť si celkový obraz o skúmanom programe a často býva súčasťou podporných nástrojov. Naším cieľom je dokázať a poukázať na možnosť využitia teda aj tak potrebných dát a zároveň prekonať niektoré obmedzenia spomínaných existujúcich techník.

2 Existujúce techniky

V tejto práci vykonáme komparatívnu analýzu navrhovanej metódy s troma existujúcimi technikami dolovania aspektov: Fan In analýza, analýza založená na identifikátoroch a dynamická analýza.

2.1. Fan In

Výsledkom Fan In analýzy je jedna množina vysoko ohodnotených metód bez akejkoľvek informácie o ich príslušnosti k pretínajúcim záležitostiam. Táto príslušnosť musí byť určená manuálne počas refaktorizácie systému. Dôležitosť, a teda aj poradie každej z metód v množine určuje hodnota Fan In. Tá je určená počtom rozdielnych tiel metód, ktoré môžu vyvolať danú metódu. Veľkosť výslednej množiny záleží od prahu, ktorý určuje minimálne ohodnotenie. Určenie optimálnej hodnoty prahu je špecifické pre každú skúmanú aplikáciu a veľmi priaznivo ovplyvňuje celkovú účinnosť. Príklad volaní a k nim prislúchajúcich hodnôt ilustruje Tabuľka 1.

Tabuľka 1. Vzor kódu a prislúchajúce Fan In hodnoty.

```
interface A {
   public void m();
}
class B implements A {
   public void m() {};
}
class C1 extends B {
   public void m() {};
}
class C2 extends B {
   public void m() { super.m();};
}
class D {
   void f1(A a) { a.m(); }
   void f2(B b) { b.m(); }
   void f3(C1 c) { c.m(); }
}
```

Metóda	Potenciálny volajúci	Fan In hodnota
A.m	*D.f1, D.f2, D.f3*	*3*
B.m	*D.f1, D.f2, D.f3, C2.m*	*4*
C1.m	*D.f1, D.f2, D.f3*	*3*
C2.m	*D.f1, D.f2*	*2*

2.2. Analýza založená na identifikátoroch

Kandidáti sú identifikovaní vytváraním skupín programových entít s podobnými názvami. Táto technika aplikuje algoritmus formálnej konceptovej analýzy (z angl. formal concept analysis, ďalej FCA) na všetky metódy a triedy skúmanej implementácie. Výsledkom FCA sú tzv. koncepty, skupiny elementov, ktoré zdieľajú v názve spoločné slová. Príkladom je koncept "read", ktorý zahrňuje metódy ako napr. "readChar" alebo "readDouble". Veľkosť konceptu je určená počtom priradených elementov a slúži ako prahová hodnota analýzy. Je taktiež ako pri Fan In analýze špecifická pre každú skúmanú aplikáciu. Jej príliš vysoké hodnoty spôsobujú, že niektoré záležitosti nemusia byť vôbec identifikované a nízke hodnoty zas rapídne zvyšujú mieru úsilia potrebného pri manuálnej časti analýzy. Príklad zoskupenia získaného algoritmom FCA ilustruje Tabuľka 2.

Tabuľka 2. Koncept "figure" + "request".

	figure	request	drawing	remove	update
drawingRequestUpdate		x	x		x
figureRequestRemove	x	x		x	
figureRequestUpdate	x	x			x
figureRequestRemove	x	x		x	
figureRequestUpdate	x	x			x

2.3. Dynamická analýza

Dynamická analýza vychádza z údajov zozbieraných počas vykonávania vybraných prípadov použitia. Výsledkom vykonania každého z týchto scenárov je trasa vykonávania. Scenáre predstavujú vlastnosti a volané metódy elementy pre algoritmus FCA. Pretínajúca záležitosť je identifikovaná v dvoch prípadoch

(viď. Tabuľka 3): ak metódy v rámci jedného scenára patria rôznym triedam (roztrúsenie kódu) alebo ak jedna trieda definuje metódy pre viacero scenárov (zapletenie kódu).

Tabuľka 3. Dynamická analýza, roztrúsenie (šedá) a zapletenie (čiarkovaná).

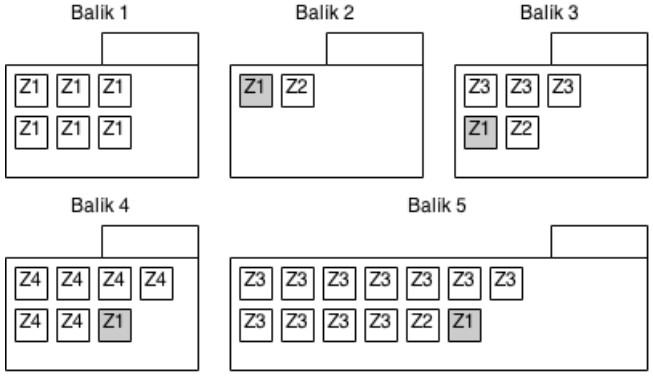
	A.m1()	A.m2()	B.m3()	C.m4()
Scenár 1	x		x	x
Scenár 2	x			
Scenár 3	x	x		
Scenár 4	x	x		

3 Zhlukovanie a užitočné dáta

Zhlukovanie spočíva v navrhnutí schémy pre zoskupovanie objektov do určitého počtu tried, kde objekty patriace jednej triede sú podobné na základe určitej miery a rozdielne voči objektom druhej triedy.

Objekty zhlukovania v kontexte zdrojových kódov môžu predstavovať rôzne časti implementácie na základe určitej granularity (triedy, metódy, ...). Sú reprezentované najčastejšie pomocou vektorov a ich vzájomná vzdialenosť je určená napr. kosínusom uhla, ktoré tieto vektory zvierajú (kosínusová vzdialenosť). Hodnoty jednotlivých dimenzií vektorov predstavujú váhy extrahovaných termov a sú určené na základe zvoleného váhovania. Rozlišujeme dve hlavné zložky váh: lokálne - určujú váhu termu v rámci dokumentu a globálne, ktoré určujú váhu termu vzhľadom na celú kolekciu objektov. Cieľom globálnej váhy je diskriminovať tie kľúčové slová, ktoré sú spoločné pre veľké množstvo objektov, a teda nie sú špecifické pre danú entitu. Čím je ich výskyt častejší, tým sú hodnoty globálnej váhy nižšie a naopak.

Na základe uvedených skutočností predpokladáme možnosť využitia globálnej váhy pri dolovaní aspektov. Druhý potenciálne užitočný zdroj dát vidíme v získaných zhlukoch a v ich rozložení cez jednotlivé balíky (viď. Obrázok 1). V tomto smere nás utvrdzujú aj poznatky a pozorovania z práce [2].



Obrázok 1.Mapa rozloženia balíkov a zhlukov (tém).

4 Naša metóda

Nami navrhovaný prístup je založený na zhlukoch a ich rozložení v balíkoch. Princíp tejto metódy možno opísať v nasledujúcich krokoch:

1. **Predspracovanie zdrojových kódov.** Použitím nástrojov sady Eclipse JDT extrahujeme a následne tokenizujeme telá deklarácií typov. Zároveň vytvárame mapovanie, ktoré pre každý extrahovaný typ uchováva prislúchajúci balík.
2. **Vytvorenie dokument-term matice.** V tejto fáze pomocou INTT tokenizéra [3] rozkladáme zložené tokeny z prvého kroku na termy. Termy sú následne normalizované lematizáciou

založenou na ontológii WORDNET. Frekvencia výskytu jednotlivých termov v objektoch (dokumentoch) je uložená do dokument-term matice, avšak termy, ktoré patria do zoznamu stop slov sú ignorované.

3. **Vytvorenie kovariančnej matice.** Kovariančná matica je symetrická matica určujúca podobnosť ľubovoľných dvoch objektov kolekcie. Výpočet jej prvkov je realizovaný prostredníctvom modelu TF-IDF nad všetkými vektormi dokument-term matice, ktorých vzájomná vzdialenosť je určená na základe kosínusovej vzdialenosti.
4. **Zhlukovanie.** Kovariančná matica predstavuje vstup pre zhlukovací algoritmus. V našom prípade používame metódu aglomeratívneho zhlukovania CLCA (z angl. complete linkage clustering analysis). Ako kritérium pri vytváraní zhlukov používame maximálnu vzdialenosť v rámci zhlukov, ktorá v tomto prípade bola určená manuálne (3.35).
5. **Identifikácia pretínajúcich tém.** Ako pretínajúcu tému označujeme ten zhluk, ktorý sa rozkladá cez viac ako 4 balíky. Tento parameter musí byť tiež manuálne určený.
6. **Ohodnotenie termov.** Termy v rámci každej pretínajúcej témy sú ohodnotené podľa toho, v koľkých balíkoch sa vyskytujú. Tie termy, ktoré majú vyššie ohodnotenie ako 2 budeme označovať pretínajúce termy.
7. **Ohodnotenie metód.** Na základe pretínajúcich termov, je ohodnotená každá metóda pretínajúcej témy nasledovne:

$$Rm = \sum_{i}^{N_c} Rt_i \cdot \frac{N_c}{N} \quad (1)$$

kde N označuje počet termov, z ktorých sa skladá názov metódy, N_c počet obsiahnutých pretínajúcich termov a Rt_i ohodnotenie i-teho pretínajúceho termu v danej téme. Zlomok pretínajúcich termov bol zavedený za účelom diskriminácie metód zložených z veľkého počtu termov, ktoré prirodzene majú vyššiu pravdepodobnosť výskytu pretínajúceho termu.

8. **Filtrovanie výsledkov.** V rámci ohodnocovania, podobne ako Fan In analýza, ignorujeme tie metódy, ktoré začínajú slovami "get", "set" alebo "to". Výsledkom metódy je zoznam pretínajúcich tém a k nim prislúchajúcich ohodnotených metód, ktorých hodnota je vyššia ako hodnota prahu (nami určená 2.0) .

5 Evaluácia

Našu metódu sme aplikovali na projekt JHotDraw, verzia 7.0.6. Ide o GUI rámec implementovaný v jazyku Java, ktorý obsahuje 3752 deklarácií metód. Identifikovali sme 4 pretínajúce témy a celkovo 346 kandidátov. Pri vyhodnocovaní sa sústredíme najmä na pokrytie jednotlivých záležitostí, ktoré boli opakovane identifikované v prácach [4][5][6] (viď. Tabuľka 4). Tieto práce využívali a porovnávali tri metódy: Fan In analýzu, analýzu založenú na identifikátoroch a dynamickú analýzu.

Pretínajúce záležitosti identifikované v spomínaných prácach zobrazuje Tabuľka 4. Stĺpce tabuľky určujú, ktorý prístup dokázal identifikovať metódy patriace daným záležitostiam. Vidíme, že náš prístup dokázal identifikovať aj tie záležitosti, ktoré Fan In analýza alebo analýza založená na identifikátoroch nedokázali.

Tabuľka 4. Vybrané záležitosti a porovnanie metód.

Záležitosť	Fan In	Identifikátory	Dynamická analýza	Náš prístup
Observer	+	+	+	+
Úpravy späť, dopredu	+	+	+	+
Persistencia	+	+	+	+
Uistenie podmienok	+	-	-	+
Prenos do popredia / pozadia	-	-	+	-
Správa rutín	-	+	+	+
Premiestňovanie figúr	-	+	+	+

Fan In analýza zlyhala pri záležitosti správa rutín, pretože ide o sadu podobných volaní a nie jednotlivca, ktoré sú často volané. Fan In vo všeobecnosti identifikuje len tie metódy, ktoré sú roztrúsené vo veľkej miere. Analýza založená na identifikátoroch síce dokázala danú záležitosť lokalizovať, no neidentifikovala metódy ako north, west, south, east, pretože nemajú podobné názvy. Dynamická analýza tieto volania identifikovala, avšak jej všeobecnou nevýhodou je, že identifikuje len tie záležitosti, ktoré spadajú do prípadu použitia a sú vykonané. Náš prístup prekonáva problémy Fan In analýzy a problém identifikátorov kombináciou prvkov oboch týchto prístupov.

Náš prístup dokázal identifikovať aj záležitosť prenos do popredia / pozadia, avšak pri vypnutí filtrovania podľa prahovej hodnoty, kedy počet výsledných metód bol dvojnásobný. Rozšírený zoznam nami identifikovaných záležitostí zobrazuje Tabuľka 5. Zaujímavá je nesprávne identifikovaná záležitosť metódy kolekcií. Tieto metódy sú vo Fan In analýze dodatočne filtrované spolu s metódami set*, get* čo nás len utvrdzuje v domnienke príbuznosti týchto prístupov.

Ďalším zaujímavým pozorovaním sú dve rovnaké záležitosti identifikované v rôznych pretínajúcich témach. Nástroje v téme č.1 predstavujú sadu metód spojených s udalosťami, no nástroje v téme č.2 s rôznymi konkrétnymi nástrojmi daného rámca. Príslušnosť k jednotlivým témam teda môže vhodne dotvárať obraz o pretínajúcich záležitostiach. Niektoré metódy (napr. draw) boli rozšírené cez všetky témy. V tomto prípade sa jedná o globálne rozšírené záležitosti a zobrazenie do tém nám uľahčuje ich identifikáciu.

Tabuľka 5. Identifikované záležitosti.

Záležitosť	Počet	Príklady
Observer	16	figureadded, figurechanged, keyreleased, keypressed, keytyped, removenotify, addnotify
Composite	17	addenum, addnode, removenode, removesheetlistener, removeactionlistener
Úpravy späť, dopredu	3	undo, redo, nonundoableedit
Persistencia	13	read, write, save, load, close, readchar, writestorable, readobject
Uistenie podmienok	7	iseditable, isnullvalueallowed, canconnect, isvisible
Správa rutín	15	north, east, south, west, northeast, trianglerotationhandler, createhandles, handleconnect, handledisconnect
Premiestňovanie figúr	3	moveby, moveaction, moveto
Nástroje - téma 1	4	abstracttool, tooldone, toolstarted, toolevent
Nástroje - téma 3	5	texttool, textareatool, selectiontool, pathtool, creationtool
Zoom	1	zoomaction
Vytváranie skupín	4	cangroup, groupfigures, groupaction, selectgroup
Inicializácia	5	init, initproject, initcomponents, initapplication, initactions
Draw	7	draw, drawtext, drawstroke, drawfill, drawfigure
Predvolené hodnoty	4	defaultdrawingeditor, defaultappletapplication, defaultdomfactory, defaultdrawing
Metódy kolekcií	7	add, remove, hashcode, sort, size, length, isempty

Pokiaľ sme pri ohodnocovaní termov používali aj ich globálne váhy, došlo k prílišnému a nesprávnemu zvýhodneniu metód pozostávajúcich z často používaných slov ako napr. string. Tento prístup len zvyšoval citlivosť algoritmu na definovanú množinu stop slov.

6 Diskusia

Nami navrhnutá metóda je triviálna a slúži len na overenie myšlienky, no i tak si vieme predstaviť mnoho odlišných variácií. Jedna z možností je v rámci tém brať do úvahy len spoločné metódy balíkov a nie termy. Dôsledkom by malo byť zníženie šumu, avšak potenciálne by mohlo dôjsť ku strate dát a k zvýšenej citlivosti na kvalitu zhlukov.

Využívanie IDF vektoru bolo v našej metóde deaktivované, avšak to vôbec neznamená, že je táto informácia nepoužiteľná. Naopak predpokladáme, že by bolo vhodné implementovať novú metódu, ktorá by využívala primárne hodnoty globálnych váh. Zároveň predpokladáme, že výsledky takejto metódy by boli veľmi podobné Fan In analýze.

Určitú mieru zamyslenia si vyžaduje filtrovanie metód začínajúcich na set a get. V tomto prípade by bolo možno vhodnejšie len pridať tieto slová do zoznamu stop slov, a teda ich pri ohodnocovaní ignorovať. Nie je dokázané, že tieto metódy nemôžu byť vhodným počiatočným bodom pri dolovaní aspektov.

Ďalšiu možnosť vidíme vo vytvorení nástroja, ktorý by umožňoval dynamicky meniť kritérium hierarchického zhlukovania. Používateľ by si tak mohol meniť úroveň zhlukov podľa vlastného uváženia. Zároveň rôzne úrovne môžu odhaliť rôzne pretínajúce záležitosti a pretínajúce témy sa pri menších zhlukoch môžu stať špecifickejšie.

7 Súvisiace práce

Existuje niekoľko prác, ktoré pri identifikácii pretínajúcich záležitostí využívajú zhlukovanie, avšak všetky len na úrovni metód. Výhodou týchto prístupov je, že ich výsledkom sú sady metód príbuzných istej pretínajúcej záležitosti. My sa však zameriavame na využitie dát už používaného zhlukovania a to býva najčastejšie na úrovni tried, nakoľko toto zobrazenie je vhodnejšie pri vytváraní pohľadu na projekt (podporné nástroje). Naopak zhlukovanie na úrovni tried je vhodnejšie pri vyhľadávaniach istej funkcionality (vyhľadávače).

Práca [7] definuje vlastnú mieru podobnosti pri vytváraní zhlukov metód a jednotlivé zoskupenia reprezentujúce pretínajúce záležitosti zoraďuje vo výsledkoch podľa relevancie. Zároveň sa tento prístup snaží aj o identifikáciu bodov prierezu.

V práci [4] autor definuje vlastnú Fan In analýzu pre zhluky metód, ktorú používa na ich zoradenie vo výsledkoch.

Autor v [8] porovnáva dva rôzne typy zhlukovania metód. Zhluky zoraďuje na základe ich podobnosti voči nulovému vektoru. Najmenej podobné sú vo výsledkoch zobrazené ako prvé. Nie sme si vedomý žiadnej existujúcej práce, ktorá využíva zhluky na úrovni tried.

8 Záver

Prezentovali sme nový prístup dolovania aspektov založený na zhlukoch tried a ich rozložení v rôznych balíkoch. Snahou tohto prístupu je identifikovať tie metódy, ktoré sa skladajú z kľúčových slov pretínajúcich tém spoločných pre všetky balíky.

Na základe porovnania s troma základnými technikami v uvedených štúdiách sme preukázali, že naša metóda z hľadiska pokrytia spolu s dynamickou analýzou dosahovala najlepšie výsledky. Výsledky zodpovedajú zjednoteniu pokrytia Fan In analýzy a analýzy založenej na identifikátoroch.

Povaha výsledkov je podobná výsledkom analýzy založenej na identifikátoroch, a preto usudzujeme vhodnosť dodatočnej aplikácie algoritmu FCA, ktorý by momentálne manuálny proces priradenia metód záležitostiam značne automatizoval.

Ďalšiu možnosť rozvoja algoritmu vidíme v použití náhodného indexovania (z angl. random indexing), ktorá umožňuje určovať podobnosť medzi kľúčovými slovami, metódami a aj triedami.

Taktiež by sme chceli preskúmať možnosti automatického nastavenia parametrov algoritmu alebo vytvorenie nástroja, ktorý by ich zmenu umožňoval dynamicky.

Zoznam použitej literatúry

[1] TARR, P. et al. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). 1999

[2] KUHN, Adrian, DUCASSE, S, GÍRBA, T. Semantic clustering: Identifying topics in source code. In: Information and Software Technology [online]. 2007, no. June., s. 1 – 30.

[3] BUTLER, S. Intt: Identifier name tokenisation tool. http://oro.open.ac.uk/28352/, March 2011.

[4] ZHANG, Danfeng, GUO, Yao, CHEN, Xiangqun. Automated Aspect Recommendation through Clustering-Based Fan-in Analysis. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering [online]. 2008., s. 278 – 287.

[5] CECCATO, M, MARIN, M, MENS, K. Applying and combining three different aspect mining techniques. In: Software Quality Journal [online]. 2006.

[6] BARBOSA, FS. Comparing Three Aspect Mining Techniques. In: Simpósio Doutoral em Engenharia Informática [online]. 2008., s. 1 – 12.

[7] FILLUS, EK, VERGILIO, SR. A Clustering Based Approach for Aspect Mining and Pointcut Identification. In: 6th Latin American Workshop on Aspect-Oriented Software Development [online]. 2012.

[8] MOLDOVAN, GS, SERBAN, G. Aspect mining using a vector-space model based clustering approach. In: Proceedings of Linking Aspect Technology and evolution [online]. 2006.

Príloha D Návrh článku

Článok navrhnutý pre jednu z konferencií IFIP 2015, IEEE INES 2015, ACM Graz 2015 sa nachádza na nasledujúcich stranách. Kvalita tohto návrhu z hľadiska formátovania a grafického prevedenia je nižšia ako u originálu, preto odporúčame prezerať verziu priloženú na CD nosiči.

Software visualization using clustering inspired by ant colony

Juraj Vincúr
Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 2
SK-84216 Bratislava 4
juraj.vincur@gmail.com

Ivan Polášek
Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 2
SK-84216 Bratislava 4
polasek@fiit.stuba.sk

1 ABSTRACT
In this paper, we propose an incremental method for extraction, identification and visualisation of topics in software project based on behavior of ant colony. Main motivation of our project is to create visualization based on natural clusters which provide best hindsight of software system structure for development participants.

Categories and Subject Descriptors
H.3.3 **[Information Storage and Retrieval]**: Information Search and Retrieval—Clustering; I.5.3 **[Pattern Recognition]**: Clustering; I.5.4; **[Pattern Recognition]**: Applications—Text processing;

General Terms
Algorithms, Measurement, Performance, Experimentation

Keywords
Clustering, Ant colony, Ontology, Visualization

2 INTRODUCTION
Understanding the structure of large and complex software systems is difficult task that demands huge effort. A certain level of understanding by software engineers is necessary when they perform daily development and maintenance tasks. Decomposing software system into groups of source code components with shared or similar logic allows them to see its overall picture which helps them realize how their modifications affect overall system structure.

Such a decomposition may be achieved by applying cluster analysis. Clustering is unsupervised classification of observations into groups (clusters) based on their similarity. Observations assigned to one cluster should be highly similar while observations from two different clusters should yield low similarity. Single observation may refer to various type of source code components (e.g. classes, functions, methods) based on chosen granularity. In object-oriented systems in terms of system decomposition granularity of classes should be most suitable while in feature location context granularity of methods and functions will be preferred.

Since software systems changes over time (e.g. by adding features, bug fixing or refactoring) we consider them as dynamic data sets. We measured average time of 5 hours between commits in sample of randomly chosen open-source projects (see **Table 6**). For dynamic data sets incremental approaches yield better computation performance since these methods alter existing clusters without reclassifying past data. Computation performance is crucial because decisions made on non-up-to-date visualization of software system may be wrong and lead to degradation of its structure.

Table 6. Average time between commits in hours

	Titan	Elastic	Neo4j	Spring	Total
avg	7.21	3.91	2.17	5.97	4.82

Several incremental approaches were developed. In this paper, we will focus on 3 types of cluster analysis techniques since our method is partially related to them.

First type of methods are members of hierarchical cluster analysis which aim to build hierarchy of clusters by using generally two strategies. Agglomerative, also known as bottom-up approach, which in initial phase considers each observation as cluster. Phase after phase pair of clusters are merged until only one huge cluster remains or other defined criterion is met. Divisive, top-down approach, treats data in opposite direction. First, all data points are in one huge cluster, which is recursively split until only one data point remain in each cluster or criterion is met. An efficient and scalable representative of agglomerative methods is BIRCH (Balanced Iterative Reducing and Clustering) [2] which requires two input parameter: the distance threshold and the tree branching factor. With respect to these parameters the BIRCH algorithm build CF-tree consisted of summary information (clustering features) about candidate clusters. Then, instead of using full data

Conference'10, Month 1–2, 2010, City, State, Country.

set, another agglomerative clustering algorithm is applied on these features to refine clusters. This approach is especially suitable for very large data sets that cannot be loaded into memory. However, since each node in CF-tree can contain only limited number of data items obtained clusters may not correspond to natural clusters. BIRCH also requires significant effort to tune its parameters [1]. Non spherical clusters trouble BIRCH as well since it uses diameter or radius to control boundaries of clusters. This method is also highly order dependent.

Second type of algorithms are based on minimum or maximum spanning tree. Multiple non incremental approaches are described in [3]. This type of methods may be considered as subset of hierarchical clustering. Analogy between them is well described in paper [4], where MST implementation of single-linkage clustering is presented. We are not aware of any incremental approach based on spanning tree even though their base implementations are relatively poor in performance (with respect to BIRCH).

Third group consists of ant colony algorithms. These algorithms, members of swarm intelligence family, are inspired by multiple spices and behavior models of ants. A brief survey may be found in [5].

This paper is organized as follows. In section 2 we present results of few experiments related to source code domain. Section 3 describes our approach. In section 4 results are discussed, followed by conclusion and future plans.

3 KNOW YOUR DOMAIN

Most data mining techniques treat data as textual documents therefore domain of source code is poorly covered. We assume that predefined structure of source code and way how it is produced may help us reveal some interesting features of these data sets.

Table 7. Properties of DTM of chosen projects

	Neo4j	Elastic	Spring	Titan
row	5753	5593	9833	1238
col	5257	5297	6562	3226
nnz	225114	249107	414638	63009
density	0.74%	0.84%	0.64%	1.58%
avg	39.13	44.54	42.17	50.90

As we already state, based on **Table 6**, these data sets should be considered dynamic. Changes are usually applied in incremental manner in small batches. Dimensionality of space representing only documents changed by applying single batch should be relatively low since these documents are usually related (e.g. classes affected by single commit, see **Table 8**). To support our hypothesis we conduct an experiment in which we simulate development of few open source projects handled by Git version control system. Description of chosen projects may be found in **Table 7** which contains these properties: number of rows (classes), columns (unique terms), non-zero items, density of matrix and average number of unique terms per document (class). In initial phase we select in chronological order only those commits that affect source code files. Then checking out commit after commit we count in each phase unique terms extracted from affected source files. Results of experiments are shown in **Figure 2**. Data are summarized in **Table 9**. As we can see, average values of dimensionality of such a subspace lie between 172.07 and 297.14, which is relatively small compared to total number of dimensions (see **Table 9**). Moreover, it doesn't seem to be strongly related to project size. Titan, the smallest member of our sample with 1238 classes reach largest average dimensionality 270.07. Trend lines in **Figure 2** even suggest that this dimensionality may be constant (in average).

Table 8. Average value in covariance matrix of increment (diagonal values set to 0)

	Titan	Elastic	Neo4j	Spring	Total
avg	0.35	0.48	0.34	0.39	0.39

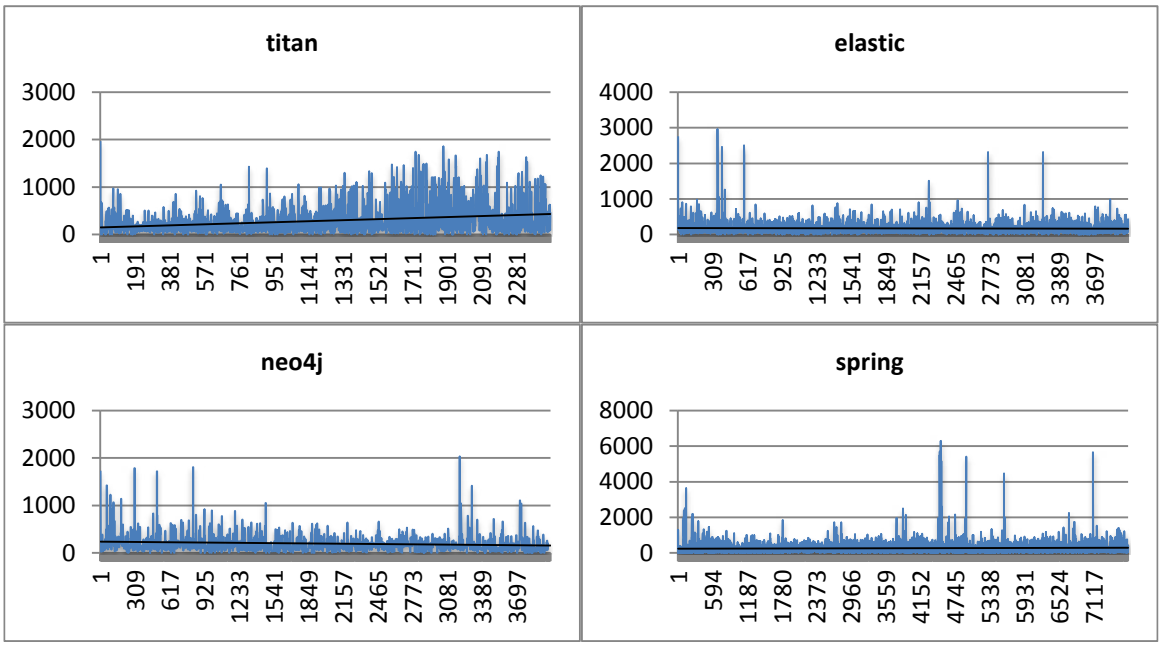
Another aspect we were interested in is density of document-term matrix (DTM) in source code domain. Sample projects yield minimum density of 0.64% and maximum 1.58% (see **Table 7**) which is comparable to domain of textual documents.

Both observations are considered in design and implementation of our clustering approach to avoid curse of dimensionality.

Table 9. Average and median of dimensions per commit

	Neo4j	Elastic	Spring	Titan	Total
med	146	132	200	196	199
avg	201.92	172.07	270.07	297.14	235.30

Figure 2. Number of dimensions per commit subspace (with trend lines)



4 OUR APPROACH

We present new clustering approach inspired by behavior of fire ant colony during floods. Fire ants have unique ability to gather together and form rafts on the surface of water. This formation in which they can retain for weeks, allows them to survive. We assume that each ant has limited vision range in which he can sense other ants. We also assume that during gathering ants are dispersed in space so by grabbing nearest ants (from vision range) first clusters of ants are formed and then these clusters are merged to one big raft (hierarchy). Moving principles of these ants are often compared to fluids. The behavior we described is similar to oil spreading on the surface of water. In this paper, we work with simplified model which consider only one possible way of grabbing another ant (instead of 7) by its jaw. Number of survivors then depends on number of connected ants (raft size). To maximize it, we add restriction to suppress cycle connections between ants, thus we get spanning tree representation, which is close to (sometimes exactly) minimum.

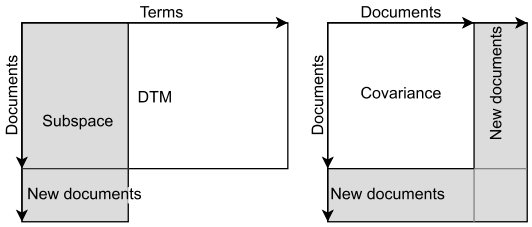
Main steps of our approach will be discussed in following order:

1. Preprocessing (building corpus representation).
2. Spanning tree construction.
3. Identification of clusterpoints (points of interest).
4. Assigning documents to points of interest.
5. Merging points of interest to clusters.
6. Visualization of cluster hierarchy.

4.1.Corpus representation

Quality of clusters is highly dependent on precision of similarity measure obtained in preprocessing step. In our work we implemented two different approaches of corpus representation. The first exploits low density of document-term matrix and second is based on dimensionality reduction technique called random indexing. Both representations use cosine similarity, which is defined as normalized dot product.

Figure 3. Adding increment to covariance matrix



4.1.1. Density approach

Primary corpus representation is constructed as document-term matrix with term frequency weighting. No normalization vector or global scale is applied because these may change in time by editing document corpus. However, vector of document lengths and average document length are updated for further usage. Document lengths are used for calculation of IDF vector and average document length for pivot normalization.

TF weights are used for serialization purposes only. In order to calculate document similarity, TF-IDF model has to be computed by applying IDF vector to normalized primary matrix. We consider global weight in TF-IDF essential due to its tendency to suppress false cluster identification caused by crosscutting concerns. For example, using raw TF weights, term "map" widely spread across classes in source code of distributed database may cause identification of oversized cluster that will contain all these classes. Problem with TF-IDF

model in terms of incremental approaches is that weights change by adding, removing or editing documents. To overcome this behavior or even employ it we compute covariance matrix. We assume that documents similar at some particular not too early point of project development should be relatively equally similar at later point of development as well. Moreover, we assume that earlier point similarity is more precise because it takes in account aspect of time. If two documents were highly similar before some words become widely spread (by adding some module or by merging branches) they are probably similar even if new TF-IDF weights say otherwise. To add new documents to covariance matrix their similarity to all documents has to be calculated (see **Figure 3**). However, since the subspace for these documents trends to be constant in number of dimensions (term dimension of DTM), only one dimension of matrix is growing (document dimension of DTM). Moreover, document dimension may be filtered as well because there is high probability that relatively low dimensional vector (rounded average of 44 dimensions for our sample) extracted from high-dimensional space (rounded average 5085) representing single class has no common dimensions with constructed subspace (rounded average 235).

Since both matrices, TF weights and TF-IDF model are sparse, we address memory efficiency problem by storing them using sparse matrix schema. More specifically, TF weights are stored using dictionary of keys format, which assigns to each row, column tuple corresponding value. This structure is efficient for constructing sparse matrices incrementally, for row and column slicing, but not efficient for arithmetic operations. On the other hand, TF-IDF model is represented by compressed sparse row matrix format. This format consists of indices (array of column indices), data (array of nonzero values) and pointer array whose items points to row starts in indices and data. It is an efficient representation for arithmetic operations and dot product, but very slow representation for column slicing or changing structure.

4.1.2. Dimensionality reduction approach

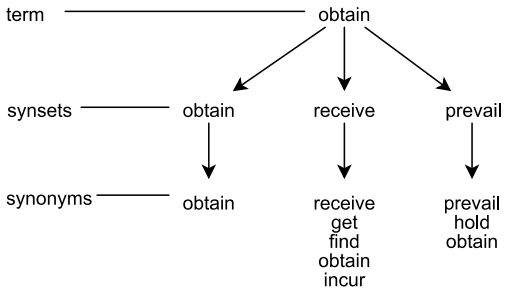
Most approaches try to solve dimensionality problem by applying dimensionality reduction techniques. Uhlár and Polášek [6] applied Latent Semantic Indexing (LSI) in clustering algorithm inspired by bee behavior. Authors observed that this powerful technique comes with great resource price. Creating reduced matrix for small project (180 documents) took 2 minutes. As a result of this performance issue clustering of 3207 classes took them more than 18 hours. Using more efficient and incremental algorithm of LSI like one presented by Řehůřek [7] would reduce this time significantly. However, presented algorithm is only incremental in documents, thus accuracy may degrade very quickly by adding documents containing new terms and reduced space needs to be recalculated. Algorithms incremental in both documents and terms are also discussed in paper [7], but these algorithms still require update step which cost certain amount of computation time.

To avoid these difficulties, our second approach of corpus representation use incremental dimensionality reduction technique, called Random Indexing (RI) [8]. This method, based on Random Projection, represents terms by context vectors. Context vector of term is calculated by summing index vectors of terms defining its context (e.g. sum of index vectors of all terms that occur immediately before or after this term in corpus). Index vectors are ternary (consist of -1, 0, 1) sparse vectors of constant length generated randomly for each term of corpus. Sum of context vectors of terms occurring in document multiplied by number of occurrences, give us context vector of this document which corresponds to single line of reduced DTM with TF weights. After obtaining full DTM representation in reduced space, we may compute TF-IDF model exactly same way as in density approach.

We are not aware of any application of RI in source code domain. Common context definitions applied to textual documents will probably performance poorly since source code documents are lexically and syntactically different. They consist of limited vocabulary organized to formally defined structures. These documents are small, in average there are only 44 distinct terms per class.

With these limitations in mind, we propose nontraditional context definition. Instead of obtaining context from distribution of terms in source code we obtain it from word relations defined in ontology. We use Wordnet [9], lexical database of English, which consist of more than 117 000 synsets. Synset is defined as a set of synonyms that share a common meaning.

Figure 4. Ontology hierarchy



As an example see **Figure 4**. Word "obtain" has 3 defined meanings (represented by 3 synsets). Names of these synsets are: "obtain", "receive" and "prevail". Each of them consists of words with shared common meaning. In our approach we would create context vector of term "obtain" by summing index vectors of all these words. Analogically, context vector of any term in corpus can be calculated. To

obtain index vectors, randomly generated vector of defined length is assigned to each word of dictionary (this step has to be done only once). Since we use lemmatization and robust tokenizer [10] capable of tokenizing identifier names with high accuracy, we were able to map almost every term to ontology. To add non defined word to dictionary, only single assignment of new vector is required. If any synset changes, only context vectors of related terms need to be updated.

4.2.Spanning tree construction

Simplified model based on spanning tree structure has been chosen due to several benefits. It is easy to maintain, analyze and visualize. Moreover, spanning tree clustering methods are capable of identifying clusters with irregular boundaries (e.g. not spherical) [11]. Our algorithm inspired by swarm intelligence creates spanning tree in following steps. Each agent representing document, has limited vision of k nearest nodes (value of 7 was sufficient for each of our experiments) based on cosine similarity, obtained from covariance matrix (storing full covariance matrix isn't necessary). Each agent grabs the nearest one from vision range that will not produce cycle in our graph, thus only one pass over documents is necessary to produce full spanning tree. This approach has been chosen over other algorithms with concurrency (like Borůvka) due to time complexity, its natural incrementality and fact that using nearly minimum spanning tree has no significant effect on quality of resulted clusters. Note that representing each document as single agent brings certain level of rivalry (concurrency) that leads to tendency of equally sized clusters, but not for price of having non-natural clusters.

4.3.Clusterpoints

In created spanning tree, we select nodes that have more than 2 links. We call these points clusterpoints and they are good approximation of medoid for group consisted of selected node and nodes adjacent to him. Note that this group represents group of mutually similar documents since it was obtained from spanning tree close to minimum. **Table 10** presents measured number of incorrectly assigned medoids, total number of identified clusterpoints, total error (sum of differences between total dissimilarity of real medoid and approximated medoid) and average error. As we can see only few medoids are approximated incorrectly and even these bad approximations yield low average error. Selecting nodes with degree higher than 2 filter out those that are connecting only two components (cluster candidates). We consider these nodes unimportant in terms of locating clusters and we will call them connectors.

Table 10. Average and total error of medoid approximation

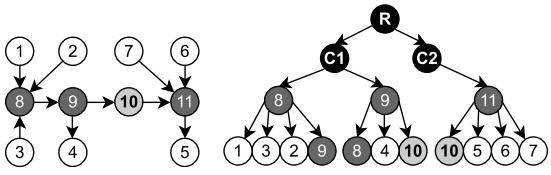
	missed	total	error	avg error
JHotDraw	4	86	0.07	0.02
Titan	8	310	0.22	0.03
Elastic	73	1336	3.96	0.05
Neo4j	68	1396	3.34	0.05
Spring	156	2349	4.58	0.03

4.4.Forming clusters

Selecting clusterpoints in spanning tree revealed high probability of forming groups of adjacent clusterpoints. These groups refer to cluster candidates. Different strategies may be applied to obtain real clusters. The one we present, might be described in these steps:

1. Filter out those clusterpoints that belong to group consisted of not more than x adjacent clusterpoints. Parameter x defines level of abstraction, thus number of resulted clusters depends on its value which should be set with respect to total number of documents.
2. From each remaining clusterpoint run breadth-first search (BFS). If traversed node is already assigned, do not continue the search from this node. Otherwise, assign the node to clusterpoint. Note that by assigning nodes traversed in the first phase of BFS to only one clustrepoint will destroy our medoid representation. That's why an exemption is granted and these nodes can be assigned to multiple clusterpoints (as copy). As we can see on **Figure 5** this may result in assigning one document to multiple clusters (document with id 10). To avoid this rare phenomenon, strategy considering number of connectors between clusterpoints should be applied or simply assign this connector to clusterpoint, which is more similar.
3. Create sub-tree from spanning tree by selecting only clusterpoints. For each component in the sub-tree, assign each node (clusterpoint) of this component to corresponding cluster. Cluster names are obtained by trivial labeling which extract most common word from set of assigned documents.
4. Connect each cluster to root node. Resulted hierarchy tree data structure is on **Figure 5**.

Figure 5. Spanning tree (left) and corresponding hierarchy (right). Black nodes - root and identified clusters; dark gray - clusterpoints; gray - connectors



4.5.Updating structure

Two main strategies has to be considered in terms of maintaining spanning tree and cluster hierarchy structure. In first, building of vector space model and

covariance matrix is managed incrementally and construction of spanning tree and cluster hierarchy is recalculated each time since it requires only single pass over documents. This strategy will probably yield more accurate results, but cause certain level of overhead.

Second strategy maintains spanning tree and cluster hierarchy incrementally as well. Formed clusters will be more stable, but probably less accurate.

Which strategy should be chosen depends on total number of documents. These strategies may be combined as well. Resulted technique will refer to something known as folding-in.

4.5.1. Spanning tree

Addition of new observations to spanning tree can be maintained easily by connecting them with nearest nodes, considering cycle restriction (same as base). Removal of node from spanning tree causes that nodes previously connected to it have to be treated as new ones.

4.5.2. Clusters

Maintaining cluster hierarchy is slightly more difficult. It can be done by applying following rules:

- If new clusterpoint is formed in spanning tree, BFS has to be initialized from this node. List of reached clusterpoints is constructed. Node assignments to these clusterpoints have to be reconsidered.
- If new document is assigned to existing clusterpoint no further steps are necessary.
- If new document is connected to leaf node, it is assigned to same clusterpoint and cluster.
- If clusterpoint is removed, its children in cluster hierarchy tree has to be reassigned.
- If leaf is removed, no further steps are necessary.
- If connector is removed, only nodes that lie between this node and other clusterpoints are reconsidered.

4.6.Data visualization

Most companies develop large software projects using version control system. Source code is physically located on one or multiple servers what allows us to create web based visualization tools. Main benefits of this approach are that users are able to see results everywhere, immediately without any installation required. There are many visualization JavaScript libraries suitable for this purpose. Most of them require only json dump in specified format.

In our work, we choose d3.js which provides robust documentation and large database of examples. From these examples we adapt two different techniques to visualize our cluster hierarchy. First, we apply trivial graph representation (see

Figure 6) which suffers from non-intuitive hierarchy, too complex for large projects, although it provides fair amount of information for debugging of our tool. Second, circle packing layout is applied (see **Figure 7**). This approach is much more intuitive, providing good level of 3 level abstraction that result in significant reduction of space required for visualization. **Figure 7** shows how circle packings are zoomed after clicking "text" cluster and "TextFigure" clusterpoint.

Figure 6 and **Figure 7** are equivalent representations of sample project consisted of 350 classes with total number of 10 identified clusters. Comparing those two figures we can clearly see that first representation can't even fit space sufficient for visualizing whole project by circle packing layout. As preview of documents, source code browsing feature is added to both approaches.

Figure 6. Trivial graph representation with source code browsing (JHotDraw)

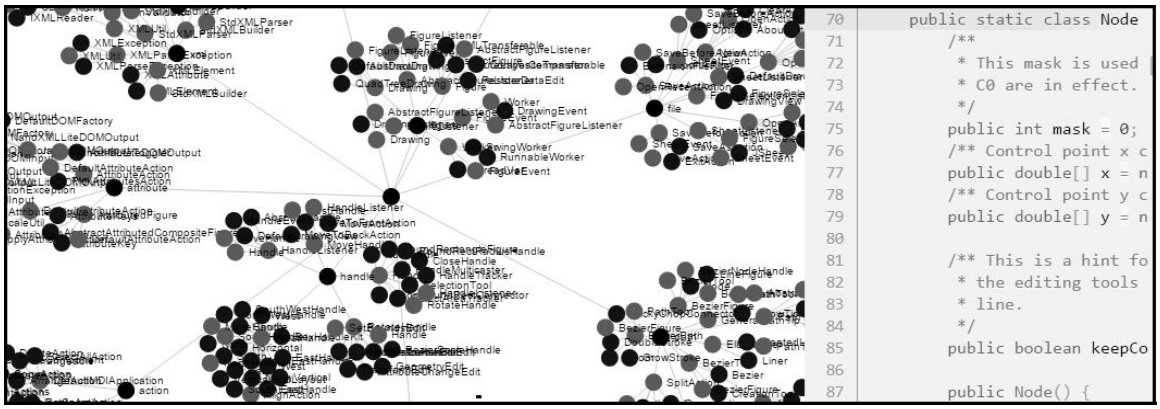
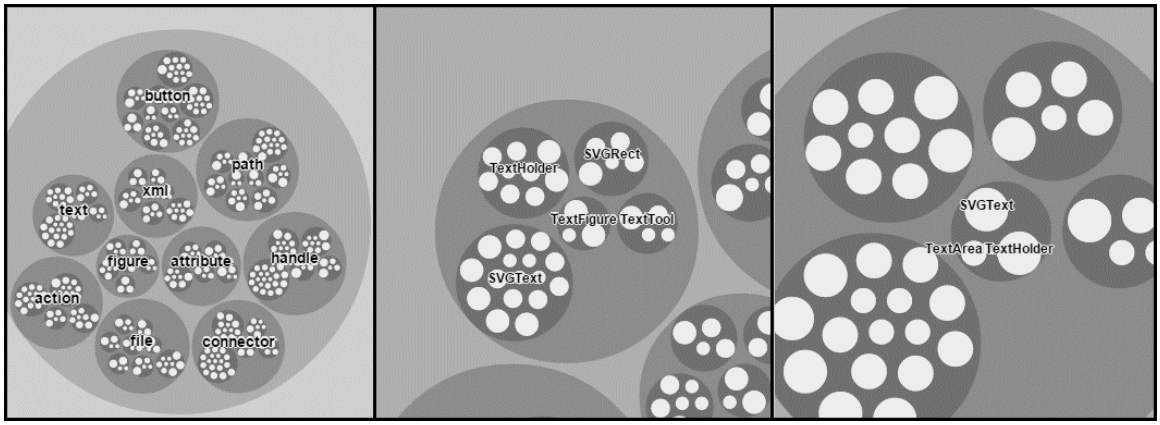


Figure 7. Circle packing layout (JHotDraw)



5 RESULTS AND EVALUATION

Discussing our results we will mainly focus on quality of clusters from view of software engineer trying to understand the structure of existing software system. To evaluate this aspect of our technique we will be interested in these questions:

- How natural are identified clusters?
- Is the number of clusters reasonable according to project size?
- Was trade-off between obtaining natural and relatively equally sized clusters reached?
- Are clusterpoints reasonable representations of group of documents?

To answer them, and external validity techniques will be most suitable, but since we are not aware of any evaluated data set from source code domain relevance judgments will be applied.

Taking relatively small project as JHotDraw yield with abstraction level x=2 10 clusters. As we can see in **Figure 7** (left side of picture) these clusters are relatively equally sized. Considering distribution of classes over packages we got 3 matches with our clusters. Package figures of JHotDraw (defined as kit of figures together with their tools and handles) refers to cluster with very precisely labeled cluster "figure". Moreover, this package was split and it refers to cluster "handle" as well. Another package application (defined as user interface related classes) was precisely identified as cluster "action". Other clusters are not corresponding to packages, but this is not a negative observation. We actually observe very well formed clusters. As example we identify cluster "file" which is related to opening, saving, loading, reading or closing project files. Another good example is cluster "xml" which refers to classes responsible for xml validation, parsing, reading, writing, building and other utilities. We judge use of clusterpoints as representative of group of documents very positively. A good example is MoveHandle class which represents classes MoveToFrontAction, MoveAction, MoveToBackAction, WestHandle and HandleListener. Similar behavior is observed in all sample projects.

What we didn't expect is rare occurrence of clusters with exactly same label. However, this phenomenon is caused by trivial labeling that does not consider IDF of terms correctly. Terms which are widely spread, but not considered important by clustering algorithm are set as labels because number of their occurrences is high.

Another thing which was quite confusing was frequent occurrence of term "Map" in project Titan (distributed database) across clusters. After small analysis we realized that term "map" occurs in 345 classes from total number of 1238 and so it is related to cross-cutting topic. IDF even in this example works correctly.

During browsing source code of documents assigned to same cluster, we realize that sometimes their vocabulary isn't similar. This is caused by fact that we use vector enrichment in case of generalization as proposed in [6]. Such an observation may be confusing for user as well, thus some abstraction of this relation should be added to visualization.

Our context definition for RI achieve good results since only minimal changes in clusters occur (e.g. one cluster disappear since documents were differently distributed) and number of dimensions were lowered significantly.

Table 11. Average number of dimensions per document (RI)

	Neo4j	Elastic	Spring	Titan
avg	741.43	777.68	691.98	803.31

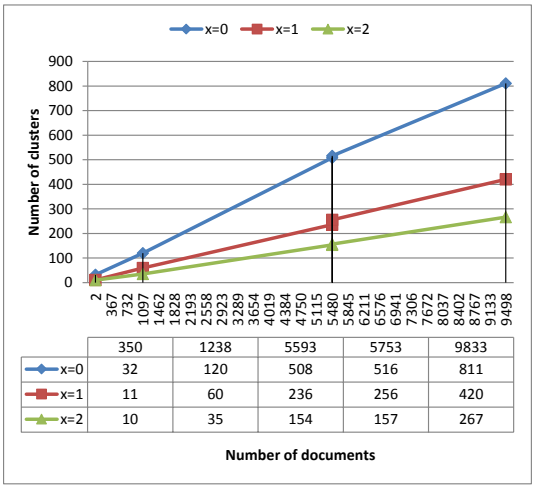
Interesting observation has been made in terms of dimensionality reduction. Since in our density approach we reach average dimensionality of 235.3, applying RI will actually enlarge number of dimensions of increment (see **Table 11**). This is caused by fact that dimensionality reduction techniques tend to lower sparsity significantly. Length of index vector was

probably set too high (1024), but question here is, would be 200 dimensions or less sufficient for RI?

We also observer that even though cosine is not metric, in low dimensional space triangular inequality is preserved with higher probability.

Size of clusters are considered rational with respect to total number of documents. According to **Figure 8** number of clusters may scale linearly or even logarithmic but to support these assumptions statistically more significant number of samples has to be analyzed.

Figure 8. Number of identified clusters



	350	1238	5593	5753	9833
x=0	32	120	508	516	811
x=1	11	60	236	256	420
x=2	10	35	154	157	267

In, results of performance evaluation of computing covariance matrix are presented. Incremental approach is compared with calculation of full covariance matrix each time, to see how both of them scale by number of applied commits. As we can see, time required by base approach grows significantly faster (estimated average complexity $O(n^2)$) than time required by incremental (estimated average complexity $O(n)$). Moreover, incremental approach will scale even better by applying document filtering. Since our current implementation of adding increments to covariance matrix scales linearly, there's yet no point of managing spanning tree and cluster hierarchy incrementally (we can recalculate them by single pass over documents; first strategy).

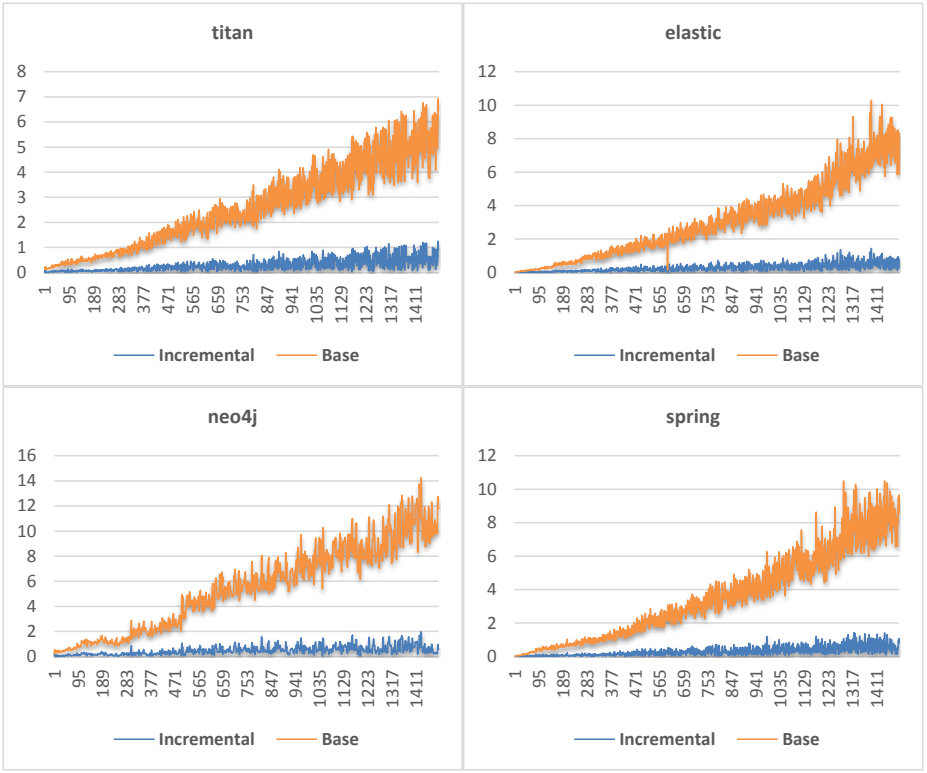
6 CONCLUSION

In this paper we proposed new incremental technique for software system visualization, inspired by behavior of fire ant colony, slightly sensible to optional tuning of parameters, which aims to identify natural relatively equally sized clusters even with irregular boundaries.

Different strategies in steps of preprocessing, clustering and even visualizing data has been proposed. Some features of documents of source code domain has been observed, analyzed and employed.

We support our assumptions by evaluation that our technique is capable of obtaining natural clusters that are crucial in terms of software system visualization.

Figure 9. Comparison of computational time between base and incremental approach of computing covariance matrix



7 FUTURE WORK

There are many ways of enhancing proposed method. In terms of performance we would like to analyze possibilities of approximating precisely k nearest documents with linear complexity. We would be also interested in impact of df-cut [12] and mapping synonym terms to single dimension on density of DTM, and thus on tuning performance of our density approach. We would also like to evaluate performance gain by adding document dimension filtering feature. Experiments to determine optimal number of dimensions for reduced space of RI and to evaluate our context definition should be also conducted. Other context definitions using POS tagger should be analyzed. To boost quality of clusters different merging strategies should be analyzed as well.

During experiments, interesting question comes to mind: Can we use data from version control systems to enhance clusters? We assume that such a technique may be really effective since we prove that classes affected by single commit are strongly related (similar).

8 REFERENCES

[1] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. 2012. StreamKM++: A clustering algorithm for data streams. J. Exp. Algorithmics 17, Article 2.4 (May 2012), 1.2 pages. DOI=10.1145/2133803.2184450 http://doi.acm.org/10.1145/2133803.2184450

[2] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. SIGMOD Rec. 25, 2 (June 1996), 103-114. DOI=10.1145/235968.233324 http://doi.acm.org/10.1145/235968.233324

[3] Oleksandr Grygorash, Yan Zhou, and Zach Jorgensen. 2006. Minimum Spanning Tree Based Clustering Algorithms. In Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '06). IEEE Computer Society, Washington, DC, USA, 73-81. DOI=10.1109/ICTAI.2006.83 http://dx.doi.org/10.1109/ICTAI.2006.83

[4] John C Gower and GJS Ross. Minimum spanning trees and single linkage cluster analysis. Applied statistics, pages 54–64, 1969.

[5] Jafar, O. a. M. and Sivakumar, R. Ant-based Clustering Algorithms: A Brief Survey. International Journal of Computer Theory and Engineering 2, 5 (2010), 787–796.

[6] Uhlar, M.; Polasek, I., "Extracting, identifiyng and visualisation of the content in software projects," Nature and Biologically Inspired Computing (NaBIC), 2012 Fourth World Congress on , vol., no., pp.72,78, 5-9 Nov. 2012 DOI=10.1109/NaBIC.2012.6402242

[7] Radim Řehůřek. 2011. Subspace tracking for latent semantic analysis. In Proceedings of the 33rd European conference on Advances in information retrieval (ECIR'11), Paul Clough, Colum Foley, Cathal Gurrin, Hyowon Lee, and Gareth J. F. Jones (Eds.). Springer-Verlag, Berlin, Heidelberg, 289-300.

[8] Sahlgren, Magnus. 2005. An introduction to random indexing. In Methods and applications of semantic indexing workshop at the 7th international conference on terminology and knowledge engineering, TKE (Vol. 5).

[9] George A. Miller. 1995. WordNet: a lexical database for English. Commun. ACM 38, 11 (November 1995), 39-41. DOI=10.1145/219717.219748 http://doi.acm.org/10.1145/219717.219748

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In Proceedings of the 25th European conference on Object-oriented programming (ECOOP'11), Mira Mezini (Ed.). Springer-Verlag, Berlin, Heidelberg, 130-154.

[11] C. T. Zahn. 1971. Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters. IEEE Trans. Comput. 20, 1 (January 1971), 68-86. DOI=10.1109/T-C.1971.223083 http://dx.doi.org/10.1109/T-C.1971.223083

[12] Kevin Dela Rosa and Jeffrey Ellen. 2009. Text Classification Methodologies Applied to Micro-Text in Military Chat. In Proceedings of the 2009 International Conference on Machine Learning and Applications (ICMLA '09). IEEE Computer Society, Washington, DC, USA, 710-714. DOI=10.1109/ICMLA.2009.49 http://dx.doi.org/10.1109/ICMLA.2009.49