Specifikace ročníkového projektu (PRG033)

# VYHLEDÁVÁNÍ TÉMĚŘ IDENTICKÝCH DOKUMENTŮ V ROZSÁHLÝCH KOLEKCÍCH

**Autor:** Daniel Benčík ( eubie@seznam.cz )
**Vedoucí RP:** Mgr. Pavel Pecina

## MOTIVACE

V posledních letech díky rozmachu Internetu a vývoji v oblasti uchovávání dat vznikly rozsáhlé kolekce dokumentů – vezměme pro příklad český internet, jehož obsah je odborníky odhadován na padesát milionů dokumentů. Jako následek řady faktorů (např. redundance / mirroring, spam, plagiátorství) mohou takovéto kolekce obsahovat některé dokumenty více než právě jednou – za kopii dokumentu v tomto případě považujeme buď naprostou shodu originálu a kopie či podobnost natolik velikou, že oba dokumenty mohou být považovány za stejné. Rozšíření téměř identických kopií dokumentů je často nepředvídatelné a potenciálně s sebou přináší problémy týkající se zvýšených nároků na uchovávání dat, snížení výkonu search enginů a kvality jejich odpovědí apod. Velké koncentrace (téměř) identických dokumentů by též mohly ovlivnit statistiky týkající se kolekcí, následkem čehož by mohly být nepříznivě ovlivněny výsledky některých aplikací založených na strojovém učení, které byly na takové kolekci trénované.

## 1. ZADÁNÍ

Řešení úlohy spočívá v efektivní implementaci algoritmů pro hledání velmi podobných dokumentů v kolekcích čítajících až miliony dokumentů.

## 2. ÚVOD DO PROBLEMATIKY VYHLEDÁVÁNÍ TÉMĚŘ IDENTICKÝCH DOKUMENTŮ

S neustále rostoucími velikostmi jednotlivých kolekcí se čas na jejich zpracování stával neúnosným a proto se na místo tradičních metod, které počítaly přímou podobnost každých dvou dokumentů kolekce, např. pomocí kosinové podobnosti, musely najít efektivnější algoritmy, většinou založené na hashování. Naneštěstí, signatury dokumentů vytvářené takovými algoritmy (zmiňme např. MD5) se ukazují jako nedostačující, neboť nejsou schopny zachytit pouze malou změnu dokumentu a jsou tedy pro hledání téměř identických dokumentů zcela nevhodné.

Řešení, které by splňovalo naše očekávání a požadavky, by mělo kombinovat přínosy obou výše zmíněných metod: přesnost porovnání každých dvou dokumentů pomocí kosinové míry, které má zřejmě, díky nutnosti naplnit celou matici podobnosti kolekce, složitost $O(n^2)$ a rychlost metod založených na hashování. Zároveň by naše řešení nemělo trpět výše zmíněnou slabinou klasických hashovacích algoritmů.

## 2.1. LOCALITY SENSITIVE HASHING

Nejprve dobře definujme pojem "schopnost zaznamenat i malou změnu dokumentu". Jestliže hashovací funkce dokumentu *A* přiřadí signaturu $S_A$, pak pokud změníme dokument *A* jen málo, měla by signatura upraveného dokumentu být velmi podobná signatuře $S_A$. Můžeme tuto vlastnost vyjádřit též ekvivalencí

$$dokument_A \approx dokument_B \Leftrightarrow sig(A) \approx sig(B) \quad (V1)$$

($\approx$ *...je podobno, podobnost signatur se v našem případě určuje Hammingovou podobností)*

Takovou vlastnost splňují hashování označovaná jako Locality Sensitive Hashing, neboli LSH ([1]).

Algoritmus, který bude naše řešení při převodu kolekce dokumentů na její signaturový model používat, je následující.

*V..... slovník - množina termů získaných že všech dokumentů*
*n..... mohutnost slovníku*
*k..... požadovaná délka výstupních signatur (v bitech)*

Vstupem algoritmu je seznam dokumentů kolekce, seznam BOWů (Bag Of Words) dokumentů kolekce a slovník *V* velikosti *n.*

*Krok 1:* Náhodně zvolme *k* vektorů délky *n* z N(0,1) rozdělení - $r_1,...,r_k$

*Krok 2:* Normalizujme vektory $r_1,...,r_k$ na délku jedna

*Krok 3:* Pro každý dokument *d* vytvořme odpovídající vektor *v* délky *n*

```
for (int i = 0; i < n; i++){
    if (V[i] is_in BOW_d)
        v[i] = tf_idf_V[i];
    else
        v[i] = 0;
};//for
```

*Krok 4:* Pro každý dokument $d_i$ spočteme $LSH(d_i)$ následujícím způsobem

```
for (int j = 0; j < k; j++){
```

$$LSH(d_i)[j] = 1 \Leftrightarrow v_i.r_j >= 0$$

$$LSH(d_i)[j] = 0 \Leftrightarrow v_i.r_j < 0$$

```
};//for
```

...kde *a.b* znamená skalární součin vektorů *a* a *b*.

## 2.2. PLEB ALGORITMUS

Nutnou podmínkou, kterou naše řešení musí splňovat je řádové snížení časové složitosti, v opačném případě jeho konstrukce pozbývá významu. Pokud porovnáváme, stejně jako to dělají dosud nejrozšířenější algoritmy, každé dvě dvojice dokumentů, řešíme úlohu mnohem složitější, než je ta, kterou řešit máme – namísto hledání téměř identických dokumentů řešíme úlohu "Výpočet podobností každých dvou dokumentů kolekce". Pokud je naším záměrem odhalit pouze velmi blízké duplicity, je velmi pravděpodobné, že drtivá většina dat získaných tradičním způsobem nám bude k ničemu ( pokud se dva dokumenty odlišují velkou měrou, není jejich nalezení naším záměrem, ale jejich kosinovou vzdálenost přesto počítáme).

Řešení problému časové náročnosti výpočtu tím, že odstraníme výpočet podobností, které nám pro řešení naší úlohy nepřinesou relevantní informace, je uvedeno v [1] pod názvem PLEB – Point Location in Equal Balls. Obecnou podmínkou funkce algoritmu je existence hashovací funkce splňující *(V1)* z článku *2.1*

*Krok 1:* převeď každý dokument na jemu příslušející signaturu
*Krok 2:* následuje L iterací samotné výkonné části algoritmu:
*Iter – begin:*
*Iteracni_Krok1:* Lexikograficky setřiď seznam signatur
*Iteracni_Krok2:* Projdi seznam a najdi téměř identické dokumenty
*Iteracni_Krok3:* Zvol libovolnou permutaci s právě jedním cyklem délky dva a aplikuj ji na všechny signatury
*Iter – end*

Komentář k *Iteracni_Krok1*: Pro každou signaturu ze setříděného seznamu zřejmě platí, že v jejím okolí jsou signatury jí podobné a díky *(V1)* platí, že i odpovídající dokumenty jsou si podobné – pro každou signaturu seznamu je tedy potřeba prohlížet v seznamu výše a níže položené signatury do doby, dokud je jejich podobnost větší než požadovaná prahová hodnota. Průchodem celého seznamu signatur můžeme pro všechny dokumenty najít signatury podobné a tím dostat část požadovaného výstupu, odtud plyne *Iteracni_Krok2*.

Komentář k *Iteracni_Krok2*: Mějme nyní dvě libovolné signatury:

$$sig(A) : 01010101011100$$
$$sig(B) : 01110101011100$$

Obě signatury se liší v jediném bitu, vycházeje tedy z *(V1)* se budou i dokumenty *A* a *B* lišit málo. V seznamu signatur ale obě signatury leží díky lexikografickému uspořádání daleko od sebe a tato dvojice tedy v *Iteracni_Krok2* nebude odhalena a zařazena do výstupu programu. K nápravě slouží poslední krok iterace. Náhodně zvolíme permutaci s právě jedním cyklem délky dvě a tuto aplikujeme na všechny signatury. Při vhodné volbě signatury (v našem případě cyklus {3,14}) bude

$$sig'(A) = perm[sig(A)] : 01010101011100$$
$$sig'(B) = perm[sig(B)] : 01010101011101$$

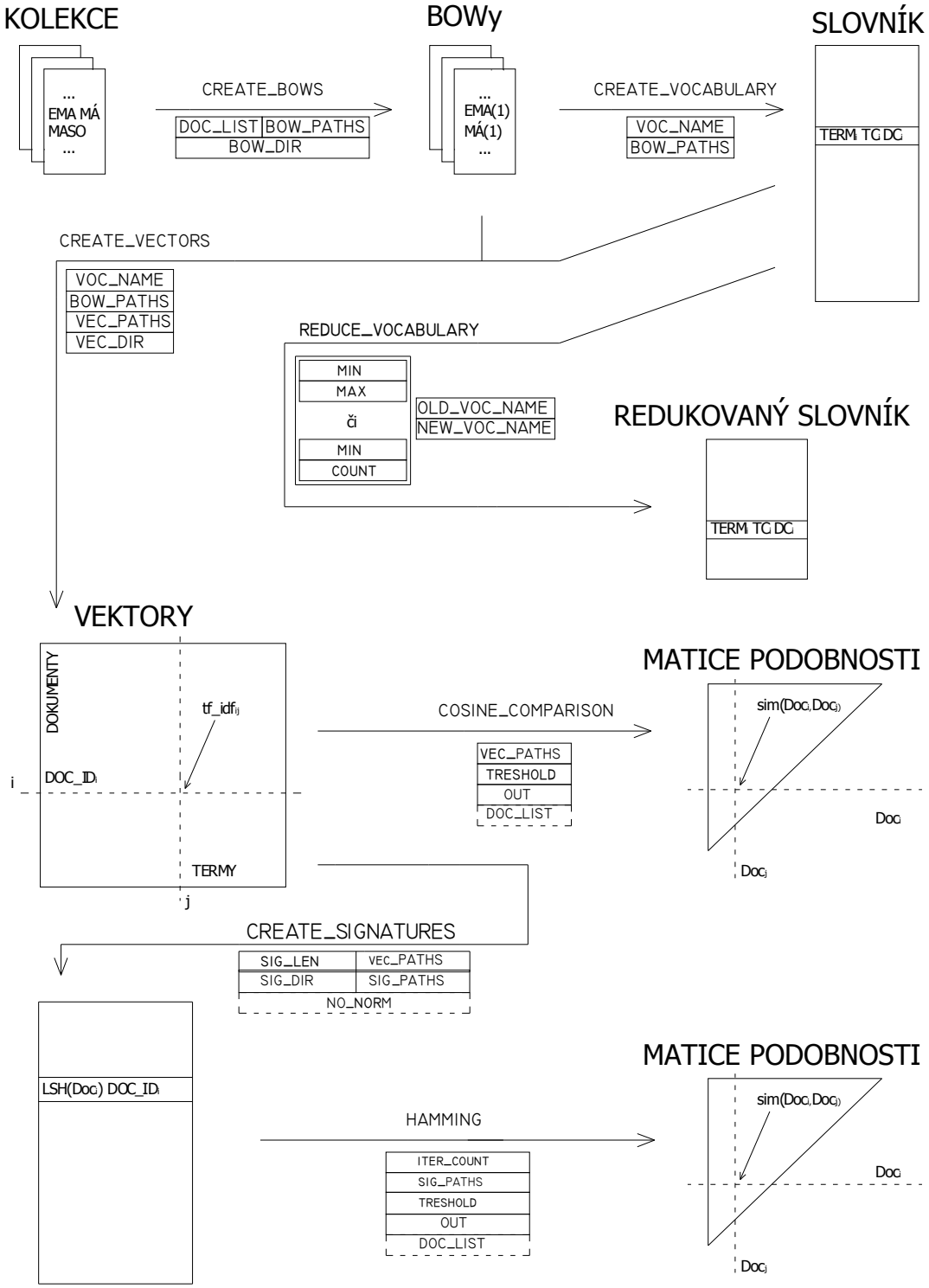
Podobnost nově vzniklých signatur je veliká a tedy se dvojice *(A, B)* při další iteraci v *Iteracni_Krok2* dostane do výstupu programu, což chceme.

## 3. IMPLEMENTACE

### 3.1. DEKOMPOZICE ÚLOHY

Na implementační úrovni bude úkol řešen balíkem programů, z nichž každý se bude starat o jinou část fáze hledání duplicit. Vstupně výstupní závislosti jednotlivých programů ukazuje diagram níže. Každá šipka představuje právě jeden program, jehož povinné parametry jsou orámovány plnou čarou, zatímco čárkovanou čarou jsou parametry volitelné.

Všechny programy balíku v případě volání s parametrem "-h" či bez parametrů podají informaci o způsobu volání a významu svých jednotlivých parametrů.



## 3.2. POPIS JEDNOTLIVÝCH KOMPONENT BALÍKU

### 3.2.1. CREATE_BOWS (DOC2BOW)

**Vstup:**

a) **DOC_LIST -** soubor obsahující jména souborů kolekce
b) **BOW_PATHS** – soubor k uložení cest k jednotlivým BOW souborům
c) **BOW_DIR –** adresář k uložení objemných BOW souborů, soubory v něm uložené budou jména mít bag_of_words1.txt, bag_of_words2.txt..

**Poznámka**: BOW soubory jsou soubory obsahující samotné Bag_of_words jednotlivých dokumentů kolekce. Z důvodu jejich velikosti budou tato data muset být rozdělena do více souborů. Počet Bag_of_words, které budou zapsány do jednoho BOW souboru by neměl přesáhnout 3500 (vycházeje z maximální průměrné velikosti 0.5MB Bag_of_words jednoho dokumentu a FS schopného uložit maximálně soubor velikosti 2GB – např. Ext2).

**Funkce:**

Program pro každý dokument kolekce dokumentů vytvoří BOW.

**Formát výstupu:**

BOW soubory budou mít následující formát:
*"$jmeno_prvniho_souboru:mohutnost_bow_prvniho_souboru\n*
*term1:pocet_vyskytu_termu1\n*
*term2:....*
*posledni_term:pocet_vyskytu_posledniho_termu\n*
*$jmeno_druheho_souboru:..."*

BOW_PATHS soubor bude mít následující formát:
"VELIKOST_KOLEKCE *velikost_kolekce\n*
*cesta/jmeno_prvniho_BOW*
*cesta/jmeno_druheho_BOW*
*...."*

### 3.2.2. CREATE_VOCABULARY(DOC2VOC, BOW2VOC)

**Vstup:**

a) **BOW_PATHS –** soubor s cestami k jednotlivým BOW souborům kolekce
b) **VOC_NAME –** jméno souboru pro uložení slovníku

**Funkce:**

Program přečte všechny záznamy o jednotlivých dokumentech z kolekce, sestaví množinu všech termů vyskytujících se v kolekci a z dat v BOW souborech sestaví statistiky výskytu jednotlivých termů v rámci kolekce.

**Formát výstupu:**

*“* VELIKOST_SLOVNIKU *Velikost_Slovniku\n*
*$Term_1$\t$TC_1$\t$DC_1$\n*
*$Term_2$\t$TC_2$\t$DC_2$*\n
*.....”*

kde *TC* (Term Count) je celkový počet výskytů termu v rámci kolekce
*DC* (Document Count) je celkový počet dokumentů obsahujících term

**Poznámka k implementaci:** Implementace programu bude využívat asociativního kontejneru typu map pro dosažení složitosti $O(n \log n + m)$ namísto $O(n^2 + m)$ při použití sekvenčního kontejneru typu vector (m je celkový počet záznamů o termech v BOW souborech, n je velikost vytvořeného slovníku)

### 3.2.3. REDUCE_VOCABULARY (VOC2VOC, LEX2LEX, REDUCELEX)

**Vstup:**

**a)** **OLD_VOC_NAME –** jméno souboru obsahující starý slovník
**b)** **NEW_VOC_NAME –** jméno souboru pro uložení nového slovníku
**c)** Jedna z následujících možností:
- **MIN_TF, MAX_TF –** hranice TF pro výběr slovníku
- **MIN_TF, COUNT** – spodní hranice TF a požadovaná velikost nového slovníku
- **MIN_DF, MAX_DF –** hranice DF pro výběr slovníku
- **MIN_DF, COUNT** – spodní hranice DF a požadovaná velikost nového slovníku
- **MIN_TFIDF, MAX_TFIDF –** hranice TF_IDF pro výběr nového slovníku
- **MIN_TFIDF, COUNT** – spodní hranice TF_IDF a požadovaná velikost nového slovníku

**Funkce:**

Program přefiltruje obsah souboru OLD_VOC_NAME dle zadaných kriterií a výsledek uloží do NEW_VOC_NAME

**Formát výstupu:**

*“* VELIKOST_SLOVNIKU *Velikost_Slovniku\n*
*$Term_1$\t$TC_1$\t$DC_1$\n*
*$Term_2$\t$TC_2$\t$DC_2$*\n
*.....”*

kde *TC* (Term Count) je celkový počet výskytů termu v rámci kolekce
*DC* (Document Count) je celkový počet dokumentů obsahujících term

### 3.2.4. CREATE_VECTORS (DOC2VEC, BOW2VEC)

**Vstup:**

a) **BOW_PATHS –** jméno souboru se jmény BOW souborů
b) **VOC_NAME –** jméno souboru obsahující slovník
c) **VEC_DIR –** jméno adresáře pro uložení VEC souborů
d) **VEC_PATHS** – jméno souboru pro uložení cest k VEC souborům

**Poznámka**: VEC soubory jsou soubory obsahující samotné vektory jednotlivých dokumentů kolekce. Z důvodu jejich velikosti budou tato data muset být rozdělena do více souborů. Počet vektorů, které budou zapsány do jednoho VEC souboru by neměl přesáhnout 3500, aby bylo zaručeno, že ani při slovníku o mohutnosti řádově 100 000 slov nebude na žádném FileSystemu problém uložit jakýkoliv VEC soubor (konstanta 3500 byla počítána vycházeje z FS schopného uložit maximálně 2GB soubor)

**Funkce:**

Program vytvoří z dat uložených v BOW souborech a dat slovníku vektory délky rovnající se mohutnosti slovníku dle postupu uvedeného v *2.1., Krok 3*.

**Formát výstupu:**

soubor VEC_PATHS bude mít následující formát:
**“**VELIKOST_KOLEKCE *cislo\n*
VELIKOST_SLOVNIKU *cislo\n*
*cesta_VEC_souboru1\n*
*cesta_VEC_souboru2\n*
*....”*

VEC soubory budou mít následující formát:
**“***soubor1 vektor_souboru1\n*
*soubor2 vektor_souboru2\n*
*....”*

**Poznámka k ukládání vektorů:**

Jelikož jsou vektory ve VEC souborech řídké, budou uloženy pomocí seznamu dvojic pozice:hodnota a do tohoto seznamu budou zahrnuty pouze takové dvojice, ve kterých je hodnota odlišná od nuly

### 3.2.5. COSINE_COMPARISON (COSINE)

***Vstup:***

a) **VEC_PATHS** – jméno souboru obsahující cesty k VEC souborům
b) **DOC_LIST** – jméno souboru se jmény souborů - dokumentů kolekce, jejichž podobnost vůči všem ostatním bude počítána
c) **TRESHOLD** – prahová hodnota přijímaných dvojic dokumentů
d) **OUT** – jméno souboru k uložení výsledků

***Funkce:***

Program v závislosti na vstupních parametrů:

1/ parametr a) - vypočte matici podobnosti pro všechny dokumenty kolekce
2/ parametry a) a b) - vypočte řádky matice podobnosti pouze pro dokumenty, jejichž jména (cesty) jsou uvedeny v DOC_LIST
3/ všechny parametry – oproti 2/ se do výstupu dostanou pouze ty dvojice, které svou podobností převyšují prahovou hodnotu
4/ parametry a) a c) - oproti 1/ se do výstupu dostanou pouze ty dvojice, které svou podobností převyšují prahovou hodnotu

***Formát výstupu:*** do výstupu se v obou případech dostanou pouze ty dvojice, jejichž podobnost přesahuje hodnotu TRESHOLD

V případě, chybějícího parametru DOC_LIST, bude výstup následujícího formátu:

*"dokument1_id:dokument2_id:podobnost$_{1,2}$:dokument3_id:podobnost$_{1,3}$...\n*
*dokument2_id:dokument3_id:podobnost$_{2,3}$:dokument4_id:podobnost$_{2,4}$...*
*..." (horní trojúhelník matice podobnosti)*

V případě, že bude parametr DOC_LIST zadán, bude formát výsledku totožný jako v předchozím případě, bude ale na rozdíl od předchozího případu obsahovat pouze řádky příslušející dokumentům obsaženým v DOC_LIST

### 3.2.6. CREATE_SIGNATURES (VEC2SIG)

**Vstup:**

**a) VEC_PATHS -** jméno souboru obsahující vektory dokumentů kolekce
**b) SIG_LEN –** celočíselná požadovaná délka signatur v bitech
**c) NO_NORM –** boolovská informace určující, zda se budou kromě náhodných vektorů $r_1,...,r_k$ normalizovat i vektory dokumentů kolekce, jako jediný volitelný parametr programů balíku má implicitní hodnotu a to TRUE
**d) SIG_DIR** – jméno adresáře k uložení SIG souborů
**e) SIG_PATHS –** jméno souboru k uložení cest k SIG souborům

**Poznámka:** SIG soubory jsou soubory obsahující dvojice signatura - jméno souboru. V zájmu úspory místa na disku a následné úspory času při čtení souboru signatur by bylo vhodné ukládat signatury nikoliv jako bitové řetězce, ale jako řetězce šestnáctkových cifer (každá šestnáctková cifra by odpovídala čtveřici bitů, které tvoří binární zápis šestnáctkové cifry). Z toho vyplývá podmínka, aby požadované délky signatur byly dělitelné čtyřmi

**Funkce:**

Program vytvoří LSH signatury zadané délky pro všechny dokumenty kolekce

**Formát výstupu:**

SIG soubory budou mít následující formát:

*" signatura1:jmeno_souboru1_se_signaturou1:jmeno_souboru2_se_signaturou1..\n*
*signatura2:jmeno_souboru1_se_signaturou2:jmeno_souboru2_se_signaturou2..\n*
*...."*

*(je totiž možné, že jedna signatura bude ve výsledku odpovídat více než jednomu dokumentu)*

SIG_PATHS bude mít následující formát:

*"VELIKOST_KOLEKCE cislo\n*
*DELKA_SIGNATUR_V_BITECH cislo\n*
*cesta_SIG_souboru1\n*
*...."*

### 3.2.7. HAMMING

**Vstup:**

a) **SIG_PATHS –** jméno souboru obsahujícího cesty k SIG souborům
b) **ITER_COUNT –** počet iterací, kterými má program projít
c) **TRESHOLD -** prahová hodnota přijímaných dvojic dokumentů
d) **DOC_LIST –** jméno souboru se jmény souborů - dokumentů kolekce, jejichž podobnost vůči všem ostatním bude počítána
e) **OUT** – jméno souboru k uložení výsledků

**Funkce:**

Pomocí algoritmu PLEB iterovaného ITER_COUNT-krát program najde dokumenty s podobností větší než TRESHOLD. Pokud bude zadán i parametr DOC_LIST, bude program počítat pouze podobnosti souborů obsažených v DOC_LIST se všemi soubory kolekce.

***Formát výstupu:*** do výstupu se v obou případech dostanou pouze ty dvojice, jejichž podobnost přesahuje hodnotu TRESHOLD

V případě, chybějícího parametru DOC_LIST, bude výstup následujícího formátu:

*"dokument1_id:dokument2_id:podobnost$_{1,2}$:dokument3_id:podobnost$_{13}$...\n*
*dokument2_id:dokument3_id:podobnost$_{2,3}$:dokument4_id:podobnost$_{2,4}$...*
*..." (horní trojúhelník matice podobnosti)*

V případě, že bude parametr DOC_LIST zadán, bude formát výsledku totožný jako v předchozím případě, bude ale na rozdíl od předchozího případu obsahovat pouze řádky příslušející dokumentům obsaženým v DOC_LIST

## 4. ODKAZY NA LITERATURU

[1] Ravichandran, D., Pantel,P., Hovy, E.: Randomized Algorithms and NLP: Using Locality Sensitive Hash Functions for High Speed Noun Clustering, *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics, strany 622 – 629, 2005*