FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Vyhledávání informací
Komplexní IR systém

KIV/IR

Pavel Třeštík
A22N0137P

19. května 2023

1 Úvod a analýza

Cílem práce je implementovat systém na vyhledávání informací. Komplexní v této práci znamená to, že program umožňuje předzpracovat dokumenty, za indexovat množinu dokumentů a nad nimi implementovat alespoň 2 vyhledávací modely. Zdroj dat byl zvolen na prvním cvičení tohoto předmětu. Tato práce bere data získaná z `https://en.wikipedia.org`. Crawler (nebo česky pavouk) získávající data z webu není součástí této práce, ale byl odevzdán na prvním cvičení. Je také poskytnuto rozhraní pro semestrální práci, které by se mělo dodržet.

První částí práce je preprocessing. Je požadována jak `tokenizace`, tak `stemming` či `lemming`. Tokenizace je přímočará a jedná se pouze o rozdělení textu. Další zpracování slouží k převedení slov do základní formy, aby se dala snáze indexovat stejná slova, která se mohou lišit například pouze rodovou koncovkou.

Druhá část je samotné indexování. Zadání specifikuje, že se musí jednat o `invertovaný index`, tedy index, ve kterým se ukládá seznam ID dokumentů pro každý `term` (zpracované slovo) jako klíč. Zadání požaduje pouze `in-memory` index, ale za bonusové body je možné implementovat i ukládání indexu do souboru.

Třetí část je vyhledávání dotazů nad vytvořeným indexem. Zadání požaduje 2 typy dotazování a těmi jsou `kosinová podobnost` a `logické vyhledávání`. Vrácené výsledky jsou řazené podle relevance, která musí nějak vycházet z použitého typu dotazování.

Poslední částí je vhodně popsat implementaci a uživatelskou příručku v dokumentaci práce.

2 Uživatelská příručka

Výsledkem práce je CLI (Command Line Interface) aplikace. Tedy uživatel program ovládá z příkazové řádky. Ovládání aplikace je realizováno pomocí příkazů (dále command) a často „podpříkazů“ (dále subcommand) s parametry. Asi není potřeba podrobně popisovat úplně všechny příkazy, ale jen ty nejdůležitější a nejrelevantnější. Všechny dostupné příkazy, jejich způsob použití a vysvětlení jejich funkčnosti může být viděn na Obrázku 1.

Každý command a subcommand má dlouhou a krátkou formu. Na příklad příkaz **print page 8** může by zkrácen na **p p 8**. Příslušné zkratky jsou také

na Obrázku 1.

Program umožňuje uživateli vytvořit více indexů bez nutnosti restartu programu. Je také možné nastavit způsob indexování dat pro každý index, před jeho vytvořením. Všechny příkazy a některé dotazy jsou ale vykonávány pouze nad zrovna zvoleným indexem. Zvolený index je samozřejmě možné změnit příslušným příkazem.

```
>>> help
Following commands are available:
    help | h - this menu
    exit | e - exits the program
    print | p [subcommand] - print current page of query results or specify subcommand to print other information
        page | p [pageNumber] - prints page with [pageNumber] or current page
        indexes | i - prints all currently available indexes
        doc | d {docNumber} - prints a document from current page with {docNumber}
    index | i {subcommand} - manipulates indexes. Requires subcommand
        new | n {path1} [path2] [...] - creates a new index with documents read from '.json' files in target directory or the file {path} points to
        add | a {path1} [path2] [...] - adds new document/s to current index (use print i to list indexes). Reads '.json' file or all files from directory
        use | u {indexNumber} - sets index with {indexNumber} as current index. Use print indexes to list all available indexes
        delete | d {indexNumber} [deleteFile: true/false] - deletes index with {indexNumber} from memory and if its a file index and [deleteFile] is true, also delete index file
        save | s {indexNumber} {path} [indexName] - saves index with {indexNumber} to {path} (must be a dir) with filename [indexName] - setting the index name
        load | l {path} - loads an index from {path} file
        name | m {name} - set {name} of current index (this name is used as default when saving index to a file)
    query | q [subcommand] {text} - executes a query on current index. Requires query {text}. Use [subcommand] to specify query type
        cos | c {text} - uses cosine similarity and tf-idf weights to select results
        bool | b {text} - uses boolean operations to select results
    set | s {subcommand} - allows to manipulate settings for indexing. Only applies to newly created index. Same settings are automatically used to process queries. Requires {subcommand} to specify what is being set
        stemming | s {true/false} - apply stemming to tokens (true is default)
        stopwords | sw {true/false} - use stopwords or allow all tokens (false is default)
        indexing | i {number = {1, 2, 3}} - specify what is being indexed. {number} is an enum
            1 - only uses the text from a document (default)
            2 - only uses the title from a document
            3 - uses 'title text' combination
```

Obrázek 1: Výpis příkazu help

2.1 Command: print | p

Command print je možné použít bez parametrů, ale v tomto případě se pak jedná o zkratku pro subcommand **page**, který vypíše poslední zobrazenou stránku. Pro zobrazení více specifických informací je třeba použít příslušný subcommand. Těmi jsou **page**, **indexes**, **doc**.

2.1.1 Subcommand: page | p

Jako parametr je nutné poskytnout číslo stránky. Při poskytnutí validního čísla stránky bude tato stránka vypsána. Obsahem stránky je seznam výsledků (resp. podmnožiny výsledků), které byly získány posledním dotazem. Jednotlivé výsledky jsou číslovány. Pro významné použití příkazu proto musí být nejdříve zadán nějaký dotaz. Při zadání dotazu je automaticky vypsána první stránka výsledků.

```
>>> p p 1
====== Last query results (page 1/1) ======
1. ex5 (score: 0.6613115)
2. ex4 (score: 0.20086059)
3. ex6 (score: 0.16441788)
4. ex7 (score: 0.0124726035)
```

Obrázek 2: Příkaz 'print page 1' nad dotazem ze cvičení (q tropical fish sea)

2.1.2 Subcommand: indexes | i

Tento subcommand nebere žádné parametry - pokud jsou nějaké poskytnuty, tak jsou ignorovány. Subcommand vypíše očíslovaný seznam všech indexů co jsou zrovna v paměti a tedy mohou být použity. Index, který je zrovna vybrán a nad kterým se provádí dotazy je označen jako např. `1 (current). Index...` - tedy obsahuje **current**.

2.1.3 Subcommand: doc | d

Tento subcommand požaduje jeden parametr, kterým je číslo dokumentu. Subcommand vypíše název a obsah zvoleného dokumentu. Číslo dokumentu odpovídá číslu vypsanému na stránce příkazem **print page X**.

2.2 Command: index | i

Command index vyžaduje subcommand. Dostupné subcommandy jsou: **new**, **add**, **use**, **delete**, **save**, **load**, **name**.

2.2.1 Subcommand: new | n, add | a

Subcommand **new** vytváří nový index. Subcommand **new** přidává dokumenty do zrovna `zvoleného` indexu. Oba dva tyto subcommandy přijímají stejné parametry, kterými jsou cesty do adresářů nebo souborů, ze kterých se čtou dokumenty. Alespoň jedna cesta je požadována, ale je možné zadat libovolný počet cest, ze kterých se dokumenty načítají.

Dokumenty jsou načítány z JSON souborů. Cesta předána subcommandu může být adresář, v tom případě jsou přečteny všechny soubory, které mají příponu `.json`. Nebo předaná cesta vede na jeden konkrétní `.json` soubor, který je načten samostatně.

Při načítaní dokumentů je nutné, aby měl každý dokument unikátní ID. Pokud dokument má duplicitní ID, a dokument s tímto ID je již načten, tak nově čtený dokument je přeskočen. ID dokumentů jsou řetězce a v datech pro tuto práci jsou jako ID použity části URL odkazů na Wikipedia článek.

Načítaný dokument musí mít následující strukturu:

```
{
  "docId1": {
    "title": "example 1"
    "timestampCrawled": "2023-05-06T16:23:39.686+02:00"
    "content: "long text 1"
  },
  "docId2": {
    "title": "example 2"
    "timestampCrawled": "2023-05-06T16:24:39.686+02:00"
    "content: "long text 2"
  },
  ...
}
```

Listing 1: Struktura JSON souboru s indexovatelnými dokumenty

2.2.2 Subcommand: save | s

Subcommand **save** uloží index vybrán jedním z parametrů do souboru. Jsou požadovány 2 parametry a 1 dobrovolný. Požadované parametry jsou: číslo indexu, který bude uložen a adresářová cesta, kam bude index uložen. Dobrovolným parametrem je řetězec, který je aplikován jako jméno indexu.

Druhý parametr musí být adresář. Index je do tohoto adresáře uložen buď jako jméno indexu nebo pod časovou známkou v momentu uložení.

2.2.3 Subcommand: load | l

Subcommand **load** načte index ze souboru, který je načten z parametru. Požadován je jeden parametr, kterým je cesta k indexovému souboru. Index je

po načtení nastaven jako `zvolený`. Při načítání indexu jsou do cache načteny i dokumenty indexu, protože není implementovaný „lazy loading“.

2.2.4 Subcommand: delete | d

Subcommand **delete** smaže index, který je vybrán parametrem. Je požadován 1 parametr, kterým je číslo indexu, který má být smazán. Navíc je možné předat i 1 nepovinný parametr, kterým je logická hodnota (true/ false), která určí, zda-li má být smazán i soubor indexu. Za předpokladu, že mazaný index je uložen i v souboru. To znamená, že je možné smazat index pouze z paměti, ale stále zachovat uložený soubor. Index není automaticky ukládán při smazání, takže pokud je index modifikován a uživatel si ho přeje smazat pouze z paměti, tak je třeba ho nejprve ručně uložit.

2.3 Command: query | q

Command **query** nevyžaduje specifikaci subcommand. Pokud subcommand není specifikovaný, tak se automaticky předpokládá subcommand **cos**. Dostupnými subcommandy jsou: **cos**, **bool**.

2.3.1 Subcommand: cos | c

Subcommand **cos** vyžaduje text jako parametr. Text je poté zpracován jako dotaz. Dotaz je vyhodnocen kosinovou podobností využitím TF-IDF vah. Dotazem je tedy pouze text a každé slovo, které se v indexu nachází přispívá k relevanci výsledku.

```
>>> q tropical sea fish
====== Last query results (page 1/1) ======
1. ex5 (score: 0.6613115)
2. ex4 (score: 0.20086059)
3. ex6 (score: 0.16441788)
4. ex7 (score: 0.0124726035)
```

Obrázek 3: Cosine similarity dotaz: tropical fish sea)

2.3.2 Subcommand: bool | b

Subcommand **bool** vyžaduje text jako parametr. Text je poté zpracován jako logický výraz. V dotazu je možné použít operátory `AND`, `OR`, `NOT`. Pokud mezi slovy není specifikován operátor, tak se automaticky předpokládá `AND`. Logické operátory není možné řetězit - tedy pokud bude použit například dotaz `slovo1 AND OR NOT slovo2`, tak se dotaz zpracuje jako `slovo1 NOT slovo2`, protože se dotaz vykonává z leva a platí pouze poslední operátor.

Je také možné použít závorky pro určení priority části dotazu. Zde ovšem uživatel může narazit na chybu implementace, protože navzdory prioritě jsou výsledky částí dotazů spojovány v určitém pořadí a to může být chybné o proti prioritě. Tato chyba je následkem chybné implementace. Na obrázku 4 lze vidět správně vykonaný dotaz a očekávané výsledky. Na obrázku 5 je velmi podobný dotaz, který by měl dávat stejné výsledky, ale kvůli chybě ve spojování priorit vrací špatné výsledky.

```
>>> q b ((tropical OR sea) OR live OR country) czechia
====== Last query results (page 1/1) ======
1. ex8 (score: 1.0)
2. ex7 (score: 1.0)
```

Obrázek 4: Příklad booleovského dotazu, kde jsou priority správně

```
>>> q b czechia ((tropical OR sea) OR live OR country)
====== Last query results (page 1/1) ======
1. ex5 (score: 1.0)
2. ex8 (score: 1.0)
3. ex7 (score: 1.0)
```

Obrázek 5: Příklad booleovského dotazu, kde jsou priority špatně

2.4 Command: set | s

Command **set** vyžaduje subcommand. Subcommandy jsou: **stemming**, **stopwords** a **indexing**. S využitím subcommandů je tímto možné nastavit

některé parametry indexování dokumentů pro nově vytvořený index. Tyto parametry jsou také aplikovány při zpracování dotazů.

2.4.1 Subcommand: indexing | i

Subcommand **indexing** vyžaduje jeden číselný parametr. Toto číslo musí být v rozmezí $< 1, 3 >$ a nastavuje následující vlastnost:

- 1 - indexován je pouze text dokumentu (content)
- 2 - indexován je pouze název dokumentu (title)
- 3 - indexován je název spojen s textem dokumentu

3 Implementace

V této části jsou popsány implementace některých částí práce a případné problémy s jejich řešením.

3.1 Preprocessing

Preprocessing (nebo česky předzpracování) je v programu použit na několika místech. Těmi je indexování a parsování dotazů. Aby bylo zajištěno, že dotaz je zpracován stejným způsobem jako indexovaná data, tak je preprocessing volán z třídy **Preprocessor**, která je singleton.

3.1.1 Tokenizace

Tokenizace v této práci má pokročilejší implementaci. To znamená, že dělení textu na slova není realizováno pouze dělením bílými znaky, ale porovnáním proti regexu, který umožňuje vyextrahovat url, cifry, datumy, apod. Tokenizer je navíc rozšířen o možnost vyhazování stop slov (stopwords). Protože data pochází z anglické wikipedie, tak s prací je poskytnut použitý seznam anglických stopwords.

Tokenizace je implementována ve třídě **AdvancedTokenizer** a implementuje poskytnuté rozhraní **Tokenizer**. Vstupní data jsou vždy tokenizována, ale je možné vybrat zda-li mají být použita stopwords - toto nastavení se nachází ve třídě **IndexSettings**.

3.1.2 Stemming

Pro další zpracování dat byl vybrán stemming pro jeho jednoduchost aplikování na slovo. Nevýhodou stemmingu je, že nekontroluje sémantický význam slova, takže mohou vznikat nesmyslné výsledky, které jsou poté v indexu. Další nevýhodou je, že stemming/ lemming je specifický dle jazyka, což je relativně velkou nevýhodou v této práci, protože data pochází z Wikipedie. Ačkoliv jsou data získána z anglické verze Wikipedie, tak se velmi často objevují jednotlivá slova či názvy v jiných jazycích, než je angličtina.

Byl použit anglický stemmer třetí strany. Tento stemmer se nachází v balíčku **preprocessing.RovoMe.EnglishStemmer** a je možné ho získat z GitHubu `https://github.com/RovoMe/PorterStemmer`. Stemmer byl rozšířen pouze o implementaci poskytnutého rozhraní **Stemmer**, což je požadováno zadáním. Zda-li se má stemming aplikovat, je možné nastavit v **IndexSettings**.

3.2 Index

Zadání požaduje, aby byl index implementován jako invertovaný index. Indexem tedy je mapa termů (zpracované slovo), kde klíčem je term a hodnotou klíče je seznam dokumentů, ve kterých se tento term nachází. V základní podobě indexu jsou v seznamu ukládána pouze ID dokumentů, ale v této práci jsou ukládány objekty **IndexEntry**, což je pomocná třída, která obsahuje referenci na dokument (instance pomocné třídy **DocumentRef**), frekvenci výskytů slova v dokumentu a TF-IDF váhu, kterou term má v daném dokumentu. Tyto údaje jsou použity pro vyhledávání kosinovou podobností a frekvence výskytu je použita k určení relevance dokumentu v booleovských dotazech. **DocumentRef** obsahuje pouze ID dokumentu a celkovou sumu čtverců dokumentu, která je potom odmocněna (normována) v kosinové podobnosti.

Index také obsahuje některé dodatečné informace, kterými jsou:

- mapování ID dokumentů na reference dokumentů - aby se předešlo duplikaci referencí na stejný dokument a snazší přístup
- absolutní cesty k dokumentům - aby bylo možné načíst indexované dokumenty při načítání indexu ze souboru
- počet dokumentů v indexu

- nastavení indexu - zda-li aplikovat stopwords, stemming a způsob indexování textu

3.3 Vyhledávání

Implementovány jsou 2 způsoby vyhledávání. Těmi jsou **cosine similarity** a **boolean search**.

3.3.1 Cosine similarity

Cosine similarity (česky kosinová podobnost) je způsob porovnávání podobnosti dokumentů využitím TF-IDF vah. Tato podobnost je počítána vzorcem viz Obrázek 6 (obr. z wiki). V této práci je implementována ve třídě **CosineSimilarity**.

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Obrázek 6: Vzorec kosinové podobnosti

Pro každý term dotazy je získán příslušný seznam IndexEntry objektů. Problémem je, že z indexu dostaneme pouze seznam dokumentů pro term. My ale potřebujeme spočítat váhu pro každý relevantní dokument, ne pro term. Proto je třeba seskupit relevantní dokumenty podle jejich referencí (**DocumentRef**, teoreticky by stačilo jako ID, ale v indexu jsou použity tyto objekty). Zároveň ale potřebujeme znát, které termy jsou v dokumentu relevantní a tím nám vznikne konstrukce mapa map seznamů. Nejedná se o nejhezčí konstrukci, ale je pak snadné tímto spočítat kosinovou podobnost.

Ta je spočtena dle vzorce 6 . Zde ušetříme čas procesoru tím, že již máme spočteny sumy vah čtverců pro každý dokument. To je spočítáno při indexaci. Také nám stačí spočítat sumu čtverců dotazu pouze jedinkrát pro dotaz. Pro každý relevantní dokument tedy musíme spočítat jen sumu násobků v čitateli.

Nakonec stačí vytvořit seznam výsledků, který je seřazen podle spočtené podobnosti.

3.3.2 Boolean search

V boolean search/ query (česky logický výraz) je pro každý term získán seznam dokumentů a nad seznamy dokumentů se poté provádí logické akce `AND`, `OR` a `NOT`. Výraz je možné rozdělit na menší části, které jsou vykonávány prioritně pomocí závorek. Implementace zpracování těchto dotazů je v této práci implementována manuálně - tedy bez použití knihoven třetích stran (např. Lucene).

Dotaz je vyhodnocován zleva. Prozatím je ignorována priorita závorkami. Protože výraz je vyhodnocován z leva, tak jednotlivé operace fungují následovně. Výsledek prvního termu je vložen do konečného seznamu výsledků dotazu. Poté jsou čteny další tokeny. Pokud je token logický operátor, tak se pouze nastaví proměnná operátoru a čte se další token. Pokud je další token term, tak se získá jeho seznam seznam výsledků. Poté jsou seznamy spojeny operací, která je uložena v proměnné operátoru. Máme tedy 2 seznamy. „První“ (levý, první term) a „druhý“ (pravý, druhý term). Operátory fungují následovně:

- AND - výsledkem AND operace je pouze první seznam, ze kterého jsou odstraněny záznamy, které nejsou v obou seznamech. Záznamy, které jsou v obou seznamech jsou sloučené dohromady.
- NOT - výsledkem NOT operace je pouze odstranění všech záznamů druhého seznamu z prvního. Jeden z důvodů proč se vykonává pouze poslední logický operátor (pokud jich je zřetězeno více za sebou) je to, aby nevznikl operátor **OR NOT**, který by byl velice náročný a pravděpodobně by obsahoval velice velkou část celého indexu.
- OR - výsledkem OR operace je první seznam sloučen s druhým. Klíče, které existují pouze v druhém seznamu jsou přidány do prvního a hodnoty klíčů existující i v prvním seznamu jsou sloučeny.

Nyní je přidáno zpracování závorek. Závorky v podstatě dělí dotaz na více malých dotazů a výsledky těchto dotazů by mělo být možné sloučit stejným způsobem jako seznamy pro term. V implementaci ale vzniká problém ten, že výsledkem termu je seznam záznamů v indexu. Výsledkem dotazu či jeho části ale je mapa, kde klíčem je dokument ID a hodnotou je seznam záznamů v indexu pro tento dokument. Ačkoliv se tedy aplikují stejné logické operátory a provádí se stejná operace, tak je ale třeba oddělené implementace (nejde tedy použít stejnou metodu).

Celkově pro vyhodnocování logického výrazu by bylo možné použít strom. V této části už je ale dotaz zpracováván z leva a ne stromově. Je tedy nutné využít jiný způsob vyhodnocení. Dotaz či jeho „pod-dotaz“ může mít technicky pouze formu `prefix (infix) suffix`. Například rekurzí je poměrně snadné vyhodnotit prefix a infix a ty spojit do sebe. Celkově pak ale vzniká problém, jak a kam se vlastně připojí suffix - jak se dá zjistit, kde suffix začíná a končí?

Tento problém byl nakonec řešen tak, že jsou nejdříve připraveny „pod-dotazy“ dle závorek a těm je přidělena priorita podle zanoření - priorita = hloubka zanoření. Ty jsou poté zpracovány a podle priority spojovány. Tímto řešením ovšem vznikly další nepředvídané problémy. Například dotaz `(sub query 1) OR (sub query 2)` je těžké spojit, protože ačkoliv je správně rozdělen do 3 „pod-dotazů“, tak 2 „pod-dotaz“ je pouze operátor OR, s čímž nebylo počítáno a mergování bylo prováděno vždy použitím operátoru na konci „pod-dotazu“ (výchozí AND). Tento problém, byl vyřešen tím, že byla zavedena pomocná privátní třída **ProcessedClause**, která obsahuje operátor spojení na začátku i na konci „pod-dotazu“ a lze tedy určit, jestli je „pod-dotaz“ pouze spojem mezi 2mi druhými „pod-dotazy“.

Před odevzdáním ale byl objeven nový závažný problém, který by bylo velice složité řešit s využitím tohoto způsobu zpracování priorit. Tímto problémem je, že do zásobníku se odkládají mezivýsledky, dokud se nedorazí do „pod-dotazu“ s nejvyšší prioritou. V moment kdy se do tohoto „pod-dotazu“ dorazí, tak se všechny výsledky ze zásobníku slučují do jednoho až do vyprázdnění zásobníku. To je ovšem problém, protože se takhle nespojí případné suffixy dotazu. V odevzdané verzi práce je tento problém bohužel stále přítomný a nelze správně vykonat například dotaz 5.

4 Výsledky

Práce implementuje funkčnosti viz Tabulka 1.

Tokenizace	Booleovský model (AND, OR, NOT)
Preprocessing - stopslova	Priorita závorkami
Preprocessing - stemmer	Dokumentace
In-memory invertovaný index	File-based index
TF-IDF model	Pozdější doindexování dat
Kosinová podobnost	Vlastní implementace vyhledávání dotazů
Vyhledávání vrací top X výsledků dle relevance	

Tabulka 1: Implementované funkčnosti

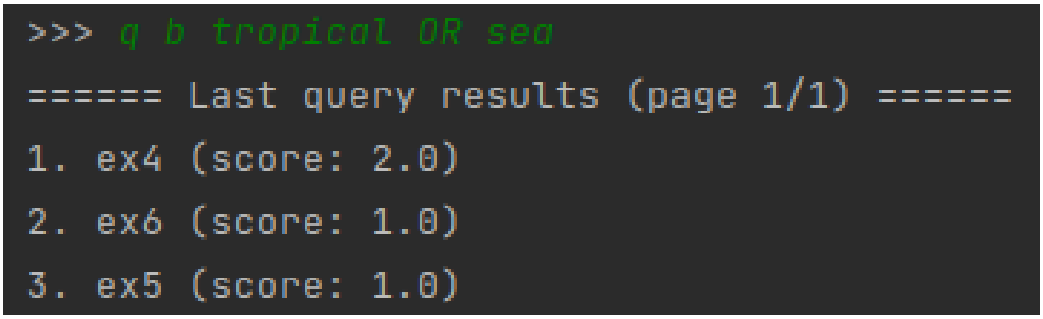
Výsledky evaluace lze dělit na podle kosinové podobnosti a booleovského vyhledávání. Kosinová podobnost s využitím TF-IDF vah funguje zcela správně. Výsledky byly ověřeny indexací dokumentů ze cvičení a použitím shodných dotazů jako na cvičení. Příklad jednoho takovéhoto dotazu můžeme vidět na Obrázku 3.

Booleovské vyhledávání nebylo úplně proti čemu ověřit, ale na druhou stranu dává logický smysl. Tedy pokud nad daty ze cvičení (Obrázek 7) pustíme například dotaz **q b tropical OR sea**, tak očekáváme 3 dokumenty (Obrázek 8). V datech můžeme snadno dohledat, které dokumenty obsahují slovo `tropical` nebo `sea`. Několika podobnými dotazy můžeme snadno ověřit funkčnost booleovského vyhledávání.

Pokud ovšem začneme používat závorky pro určení priority tak můžeme narazit na implementační chybu, kdy se dotaz nevyhodnotí správně. Toto můžeme vidět na Obrázcích 4 a 5. Bohužel tato chyba bylo objevena až těsně před odevzdáním a nebylo možné ji opravit.

```
{
  "ex4": {
    "title": "ex4",
    "timestampCrawled": "",
    "content": "tropical fish include fish found in tropical enviroments"
  },
  "ex5": {
    "title": "ex5",
    "timestampCrawled": "",
    "content": "fish live in a sea"
  },
  "ex6": {
    "title": "ex6",
    "timestampCrawled": "",
    "content": "tropical fish are popular aquarium fish"
  },
  "ex7": {
    "title": "ex7",
    "timestampCrawled": "",
    "content": "fish also live in Czechia"
  },
  "ex8": {
    "title": "ex8",
    "timestampCrawled": "",
    "content": "Czechia is a country"
  }
}
```


Obrázek 7: Data ze cvičení



Obrázek 8: Výsledek prvního dotazu

Vytvoření indexu (přibližně z 10MB dat)	30-40s
Vytvoření indexu (přibližně z 170MB dat)	64% po 2h - zrušeno
Kosinový dotaz nad tímto indexem („tropical fish sea“) - 121 výsledků	3ms
Kosinový dotaz nad tímto indexem („magnum 357“) - 6333 výsledků	35ms
Booleovský dotaz nad tímto indexem („tropical fish sea“) - 5 výsledků	5ms
Booleovský dotaz nad tímto indexem („tropical OR fish OR sea“) - 121 výsledků	1ms
Booleovský dotaz nad tímto indexem („magnum 357“) - 10 výsledků	1ms
Booleovský dotaz nad tímto indexem („magnum OR 357“) - 6333 výsledků	37ms

Tabulka 2: Vlastní statistiky pro evaluaci systému

5 Závěr

Tato práce demonstruje jednoduchý vyhledávací systém. Systém umožňuje za indexovat dokumenty a poté nad indexem provádět dotazy. Indexy je do

určité míry možné ovlivnit nastavením parametrů indexu. Indexy mohou být ukládány a načítány ze souborů.

Práci je určitě možné vylepšit řadou funkcí a optimalizací. Na některých místech by bylo možné ušetřit operační paměť lepším návrhem základní struktury indexu. Další dobré rozšíření by byl „lazy loading“ dokumentů, který momentálně není implementován. Vhodnou optimalizací by bylo například zaměření se na snížení času indexace, protože přibližně 10MB dat se indexuje zhruba 30 vteřin.