COMP4321 (Spring 2019)

Final Report

Group 13

Name: CHAN Nok Hin **Student ID: 20349103** **E-mail: nhchanaa@connect.ust.hk**

Name: CHOI Chin Cheng **Student ID: 20350413** **E-mail: ccchoiac@connect.ust.hk**

Name: KUNG Wai Tat **Student ID: 20240527** **E-mail: wtkung@connect.ust.hk**

Url: http://18.205.161.64/search/

Overview

The system mainly consists of the spider, indexer and web server. We used Python 3.7 in windows to implement the system. The spider is responsible for crawling the web page recursively. The indexer will process the page crawled and store useful information in the database. For web server, Django framework is used. The frontend will process the query inputted from the user. The backend will read the database and retrieve relevant pages based on the similarity between the query and the page. Then, these pages will be presented to the user. Details of individual parts will be discussed later.

File structures

We used sqlitedict 1.6.0 in python as database because it is a lightweight and simple database with a python dictionary-like interface, which is suitable for storing and accessing key-value pairs. It will store the database in (*.sqlite).

The script "spider.py" consist of the spider and indexer. The databases (*.sqlite) created by the indexer are shown below. All of them are stored in key-value pairs. As restricted by sqlite, the keys are String and values can be any pickle-able objects.

Type	Name
String → Int	url2pageID
String → String	pageID2Url
String → List	pageID2Meta
String → Int	word2wordID
String → String	wordID2word
String → Dictionary	forwardIndex
String → List of List	invertedIndex
String → Int	title2TitleID
String → String	titleID2Title
String → List	forwardIndexTitle

String → List of List	invertedIndexTitle

- url2pageID:
  It maps an url to a unique integer page id. This page id will represent that web page in all other database.
  (url → page_id)

- pageID2Url:
  It maps a unique integer page id to an url. It is the inverse of url2pageID.
  (page_id → url)

- pageID2Meta:
  It maps a page id to a list. The list contains all meta information of a web page, namely page title (String), last modified date (String), size (Int) and children (List). The children are represented by its page id. The method of obtaining the parent-child relationship will be discussed later.
  (page_id → [title, date, size, [child1, child2, …]])

- word2wordID:
  It maps a word or phrase (bigram) to a unique integer word id. This word id will represent that word or phrase in all other database.
  (word → word_id)

- wordID2word:
  It maps a unique integer word id to a word or phrase (bigram). It is the inverse of word2wordID.
  (word_id → word)

- forwardIndex:
  It maps a page id to a dictionary that maps word id to frequency. It stores all cleaned unigram and bigram, and their corresponding frequency of a web page.
  (page_id → { w_id1: freq1, w_id2: freq2 , … })

- invertedIndex:
  It maps a word id to a list of list containing page id, frequency and tf-idf. The page id and frequency are stored during indexing, while the ti-idf is computed by running "tfidf.py", so it is stored as 0 during indexing.
  (word_id → [[page_id1, freq1, tfidf_1], ...])

- title2TitleID:
  It maps unigram and bigram title token to a unique integer id. Similar to word2wordID, this id will represent the title token in all other database.
  (title_token → title_id)

- titleID2Title:

It maps a unique integer id to a unigram and bigram title token. It is the inverse of title2TitleID.
(title_id → titile_token)

- forwardIndexTitle:
  It maps a page id to a list containing the title tokens. Like forwardIndex, these title tokens are cleaned unigram and bigram of a page's title. But we are not interested in the frequency of title tokens.
  (page_id → [title_token_1, title_token_2, ...])

- invertedIndexTitle:
  It maps a title token id to a list of list containing the page id and tf-idf. Similar to the invertedIndex, the tf-idf is default set to 0 and being computed by running "tfidf.py".
  (title_token → [[page_id1, tfidf_1], [page_id2, tfidf_2], ...])

Some database is not created during indexing, they are created by running other scripts, which are shown below.

Type	Name
String → Float	docNorm
String → Float	titleNorm
String → List of Int	pageID2Parent

- docNorm:
  It maps a page id to its content norm which is a float. This norm can be computed by running "tfidf.py", which utilizes mainly the invertedIndex.
  (page_id → content_norm)

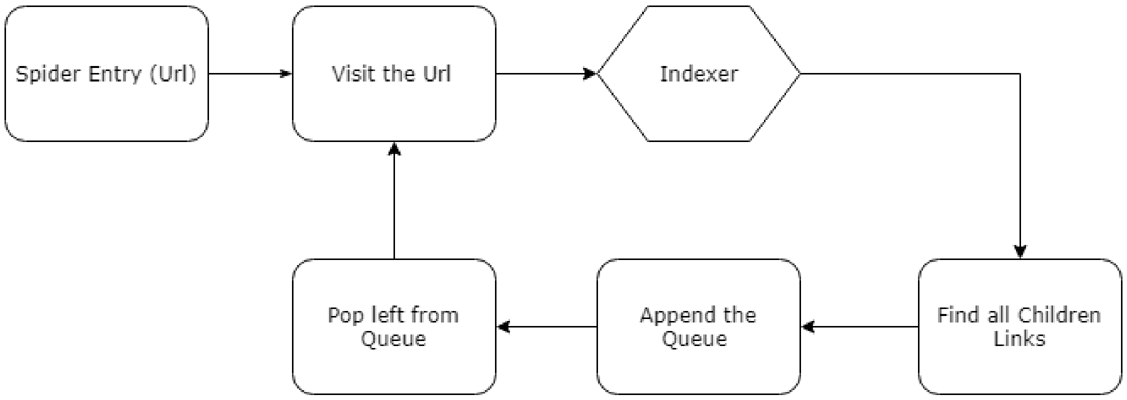
- titleNorm:
  It maps a page id to its title norn which is a float. This norm can be computed by running "tfidf.py", which utilizes mainly the invertedIndexTitle.
  (page_id → title_norm)

- pageID2Parent
  It maps a page id to its list of parent id. It is computed in a separate script "findParent.py" so as to lower the workload of the indexer.
  (page_id → [parent_1, parent_2 , …])

The tf-idf value in the invertedIndex and invertedIndexTitle are also modified by running "tfidf.py".

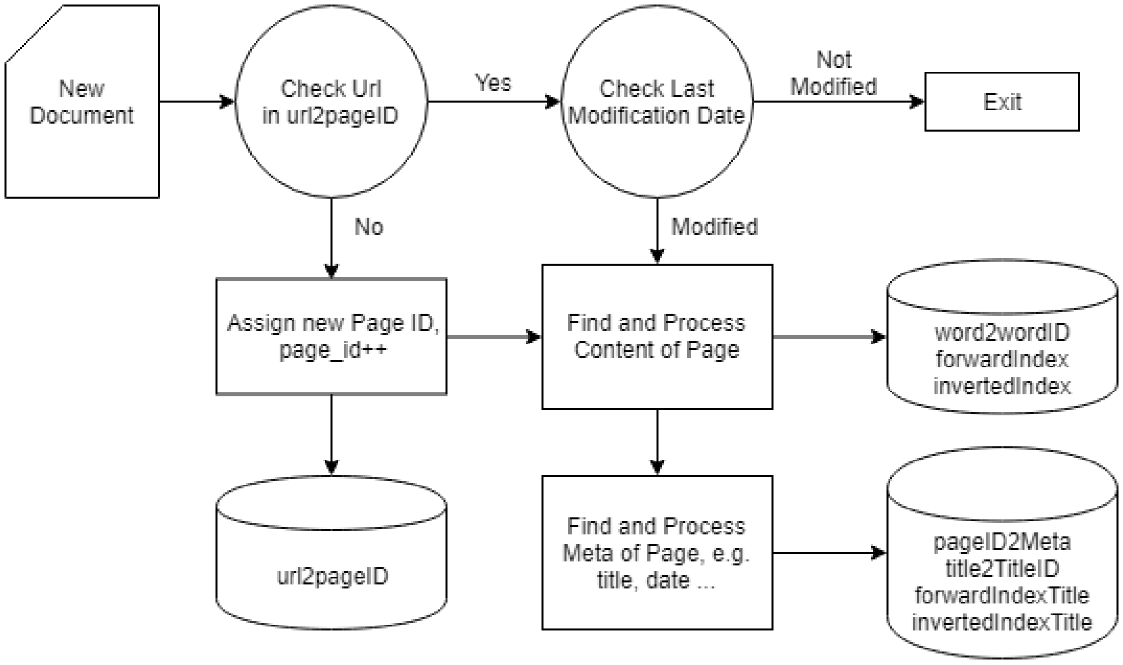
<u>System Details</u>

Spider:



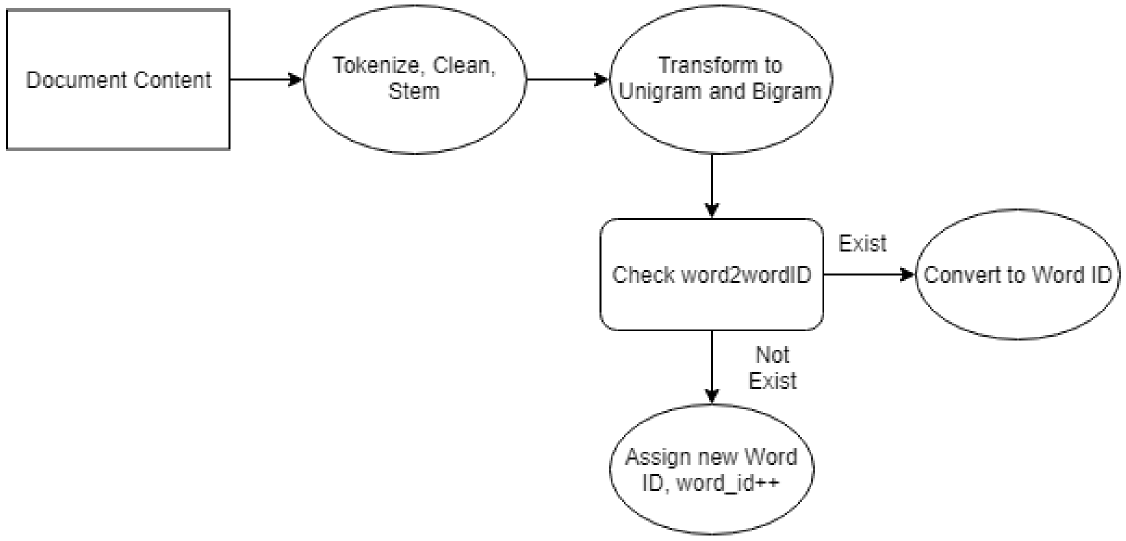
A BFS spider.

Indexer:


New Document
Check Url in url2pageID
Yes
Check Last Modification Date
Not Modified
Exit
No
Modified
Assign new Page ID, page_id++
Find and Process Content of Page
word2wordID forwardIndex invertedIndex
url2pageID
Find and Process Meta of Page, e.g. title, date ...
pageID2Meta title2TitleID forwardIndexTitle invertedIndexTitle

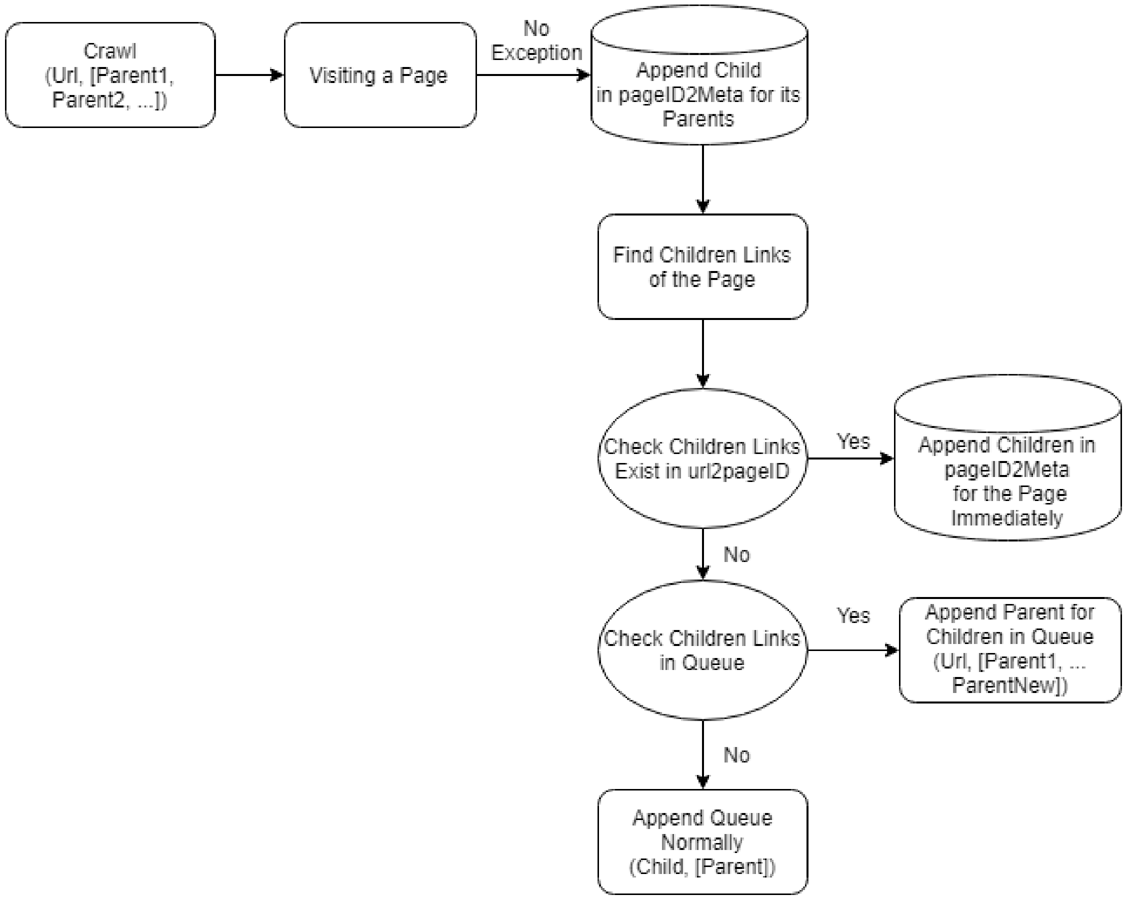
The flow of visiting a new page. If the page is not visited, index and visit the page. If the page is visited but it is modified, visit the page and update the database. Otherwise, exit.

Content Processing and Keyword Indexing:



There are several procedures to process the page content. First, tokenize the content. Then, lower the tokens, remove punctuation, remove stopwords and do stemming using Porter algorithm. The cleaned tokens are transformed into unigram and bigram. After that, they will be indexed for further processing.

Finding Parent-children Relationship:


Crawl (Url, [Parent1, Parent2, ...])
Visiting a Page
No Exception
Append Child in pageID2Meta for its Parents
Find Children Links of the Page
Check Children Links Exist in url2pageID
Yes
Append Children in pageID2Meta for the Page Immediately
No
Check Children Links in Queue
Yes
Append Parent for Children in Queue (Url, [Parent1, ... ParentNew])
No
Append Queue Normally (Child, [Parent])

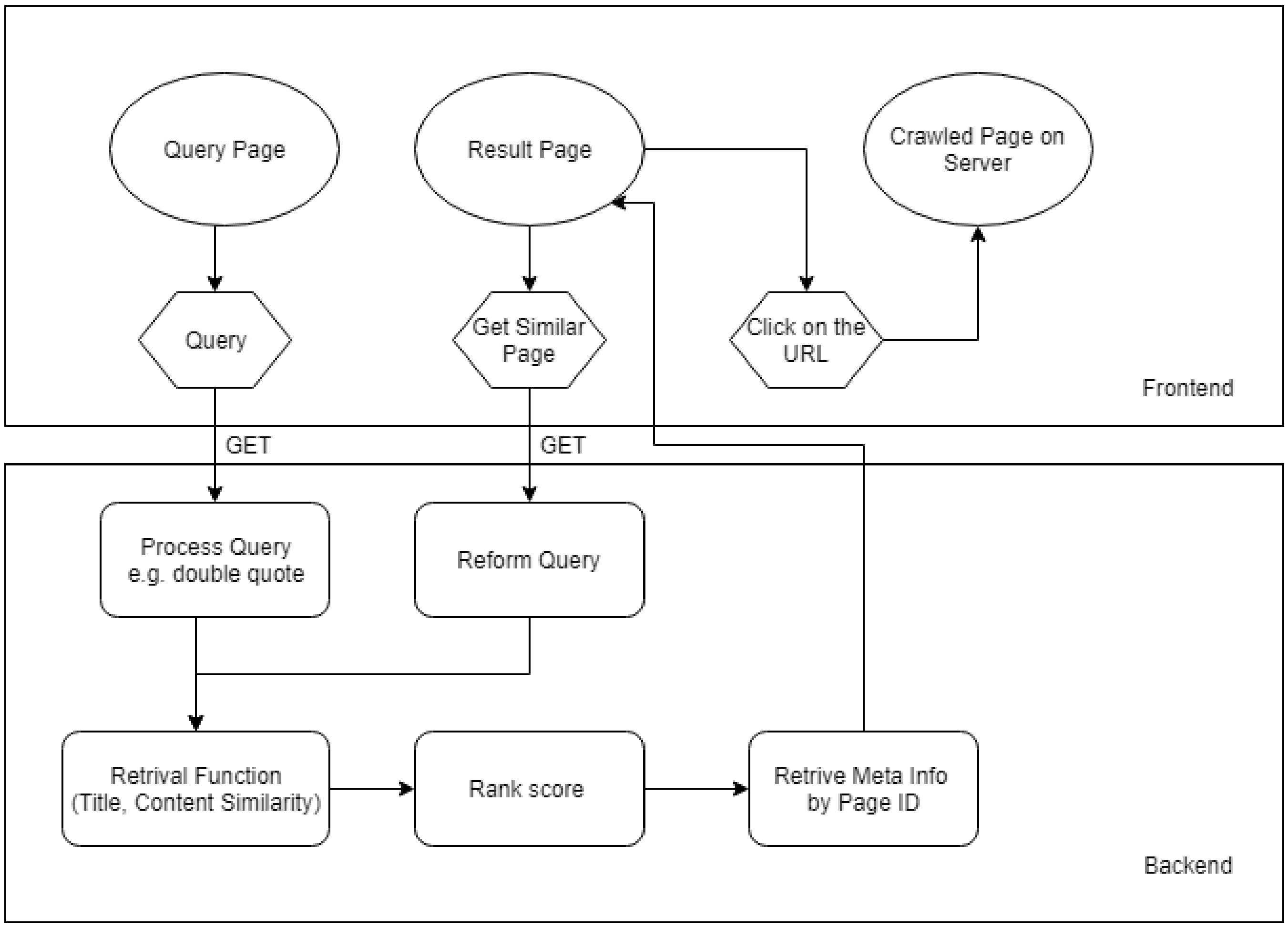
First, we want to store children page id for a parent. Second, we want to ensure the children is accessible, i.e. no exception during crawling and indexing, before assigning to the parent.

However, the children page id is not known until it is being visited. Therefore, we keep track of the parent page id for the children. Specifically, when invoking the crawl function, the parameter will have the children url and a list of parent id. If a child has no exception, append the child page id to the parents in pageID2Meta.

When finding children links of a page, there are 3 situations.

1. The children are indexed before, which also means it has no exception. So, directly assign the children page id to the parent
2. The children are not indexed but in the queue, which means it may have exception. Append the parent id to the list of parents of the children inside the queue (child_url, [parent_1, …, parent_new]). These parents will be assigned the child if the child page is visited successfully.
3. The children are neither indexed or in the queue. Then, append the queue normally (child_url, [parent]).

Web-Server:



When the user input the query, it will be split in the backend. For double quote query, it will be split as phrase, and the unigrams of the phrase will be added in the query. For example, the query (database “computer science”) will become [‘database’, ‘computer science’, ‘computer’, ‘science’]. Then, it will be cleaned by removing stopwords, stemming and so on.

After that, the query will be passed to the retrieval function and compute the cosine similarity between the query and documents. It will calculate the title and content similarity separately, and the title similarity will be weighted heavier. Calculating the inner product will be the main computational overhead as the ti-idf of each document word and the document norm are already pre-computed and stored in the invertedIndex and docNorm database. The situation is similar for title similarity.

The result will be ranked and the meta information of the pages will be retrieved from the database. Finally, it sends data back to the frontend and presents the result.

If the url of the page is clicked, it will link to the page in the server.

Algorithms

Stemming:

Porter’s algorithm is used in the stemming process for consistency reason. It applies the same set of rules (5 steps to remove suffixes) to all words. The stem of a word can be expected so the stem is consistent when there are changes to the corpus. For other algorithms like successor variety and entropy method, the collection of words determines the resulting stem. It highlights the consonant-vowel components structure of a word, which is frequently used in the rules. Since our goal is to index an educational website, we assume that most of the contents are written in formal languages.

Cosine Similarity:

Cosine similarity is a ranking function which can be used in search engines. The essence of cosine similarity is transferring the document and query into vectors and comparing the cosine angle between these two vectors. The value will be ranged from 0 to 1. The higher the value, the higher the similarity between the page and the query. We will make full use of cosine similarity for every page in our database and sort these values in descending order. We will also map the page into corresponding url which will be shown in the search engine result.

To speed up computation during query time, the tf-idf for each document word and the document norm are pre-computed by running “tfidf.py”.

For tf-idf, we extract the word frequency of each document that containing a word from the invertedIndex database. Then, the tf-idf is calculated by the following formula:

$$tf_{A,doc} = \frac{Number\ of\ word\ A\ occurs\ in\ the\ document}{Max\ term\ frequency\ of\ the\ document}\ , idf_{A,doc} = log\frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ word\ A}$$

Here, the base of log is not important. idf value for each term will be affected by the same multiplier so the rank will not be affected if different log base is used. tf_{max} is obtained from forward index.

tf-idf is calculated by multiplying tf and idf, and then it is put back to the invertedIndex.

$$Word\ A \rightarrow [\ [\ docID, tf, tfidf_A\ ]\ , \ldots]$$

For document norm, we will loop through the keywords for each document in the forwardIndex and utilize the tf-idf just computed. The norm is calculated by the following formula:

$$norm_{doc} = \sqrt{(tfidf_A^2 + tfidf_B^2 + \cdots)}$$

It is stored in docNorm database (page_id → doc_norm).

At query time, we count the number of occurrences of each word and put it as vector

$$query =< query_{word1}, query_{word2}, \ldots , query_{word\ n} >$$

Afterwards, we match the query word in the invertedIndex and accumulate the inner product for the matched document. Then, we can calculate the cosine similarity between a document and query by the following formula

$$cosine\ similarity_{doc \leftrightarrow query}\ = \frac{\sum_{word \in doc} tfidf_{word} * query_{word}}{norm_{doc} * norm_{query}}$$

Where the numerator is the dot product between the tf-idf vector of the document and the query vector of the document, and the denominator is the multiple of tf-idf norm and query norm. The inner product is computed at query time, the document norm is extracted from docNorm, and query norm is easy to compute.

We calculate the cosine similarity for the title and query in a similar manner, tf-idf and title norm are pre-computed in invertedIndexTitle and titleNorm database

We have 2 kinds of cosine similarity for a query, one is title and the other is content. We would like to combine these 2 similarities into one and give a higher weight for the title similarity such that when the query matches the title, higher similarity will be given to the page which show on the top of the results.

$$Combine\ cosine\ similarity = \frac{Title\ cosine\ similarity * 3 + Content\ cosine\ similarity}{4}$$

The combined cosine similarity is then sorted by descending order and the corresponding features of the page is extracted from the pageID2Meta databases, such as page title, url, scrapped time, parent's link, etc.

<u>Installation procedure</u>

1. The system can be run under Python 3.7 for windows. Make sure Python 3.7 and pip are installed.
2. Install packages

```
>> python -m pip install --user virtualenv
```

3. To activate the virtualenv

```
>> >> pip install bs4 requests nltk sqlitedict Django
```

4. To run the web server

```
>> cd mysite
>> python manage.py runserver
```
Go to http://localhost:8000/search/

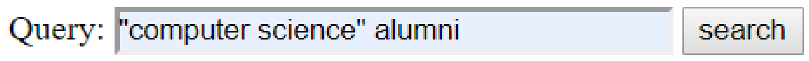
Highlight of features

Get Similar Pages:

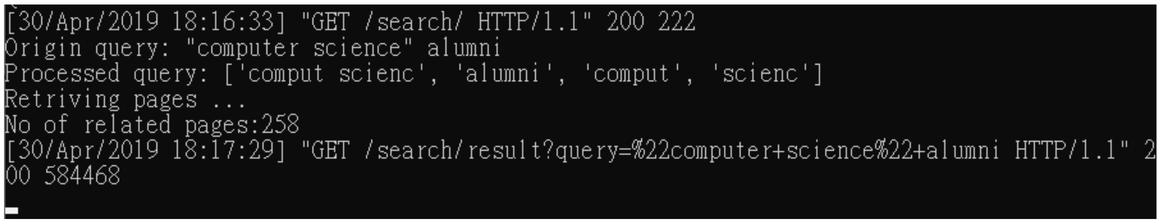
The button "Get Similar Pages" will extract the top 5 most frequent keywords, reformulate the query and search through these keywords.

Testing

Query page:

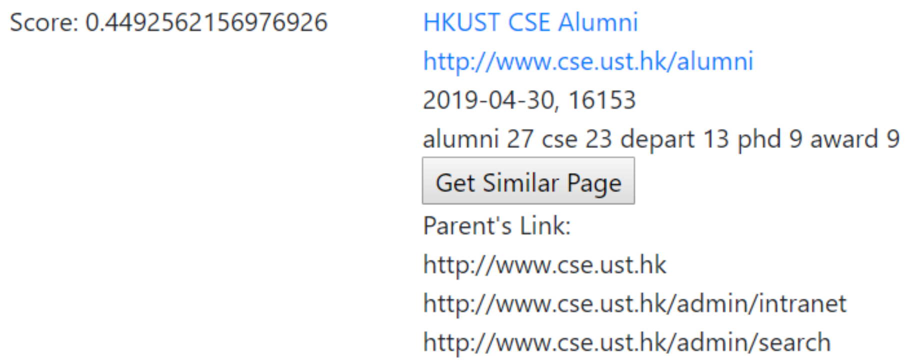


Process the query:



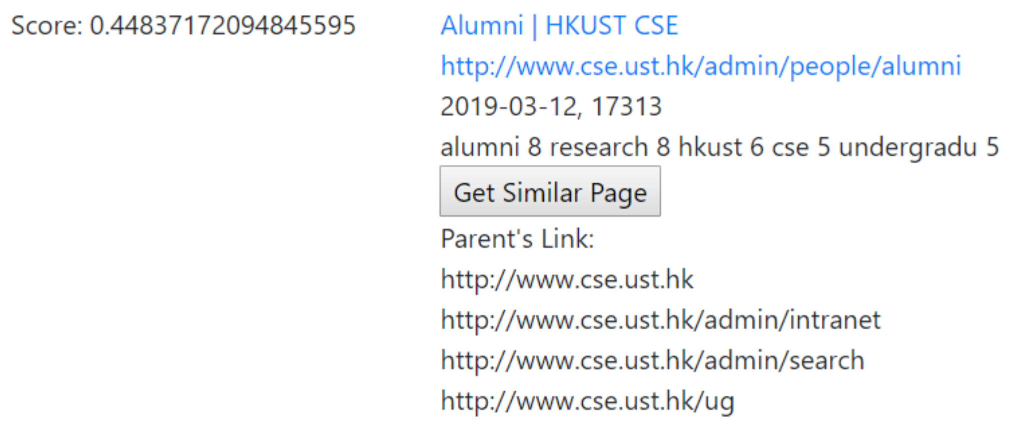
```
[30/Apr/2019 18:16:33] "GET /search/ HTTP/1.1" 200 222
Origin query: "computer science" alumni
Processed query: ['comput scienc', 'alumni', 'comput', 'scienc']
Retriving pages ...
No of related pages:258
[30/Apr/2019 18:17:29] "GET /search/result?query=%22computer+science%22+alumni HTTP/1.1" 2
00 584468
```

The first two results:



Score: 0.4492562156976926
HKUST CSE Alumni
http://www.cse.ust.hk/alumni
2019-04-30, 16153
alumni 27 cse 23 depart 13 phd 9 award 9

Parent's Link:
http://www.cse.ust.hk
http://www.cse.ust.hk/admin/intranet
http://www.cse.ust.hk/admin/search

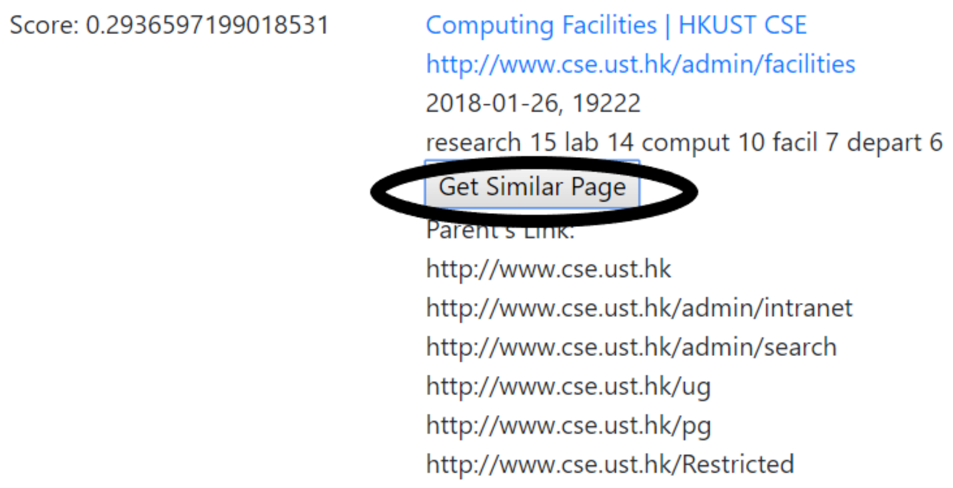


Score: 0.44837172094845595

Alumni | HKUST CSE
http://www.cse.ust.hk/admin/people/alumni
2019-03-12, 17313
alumni 8 research 8 hkust 6 cse 5 undergradu 5

Parent's Link:
http://www.cse.ust.hk
http://www.cse.ust.hk/admin/intranet
http://www.cse.ust.hk/admin/search
http://www.cse.ust.hk/ug

Get similar pages:

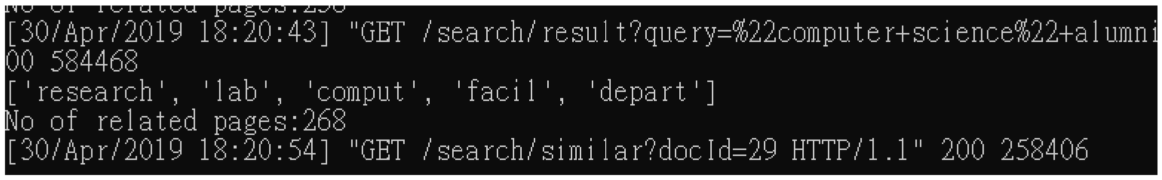


Score: 0.2936597199018531

Computing Facilities | HKUST CSE
http://www.cse.ust.hk/admin/facilities
2018-01-26, 19222
research 15 lab 14 comput 10 facil 7 depart 6

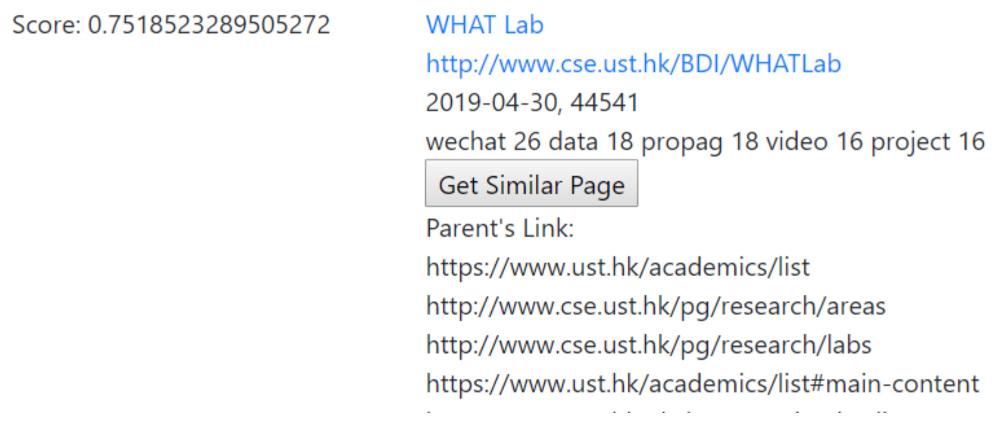
Parent's Link:
http://www.cse.ust.hk
http://www.cse.ust.hk/admin/intranet
http://www.cse.ust.hk/admin/search
http://www.cse.ust.hk/ug
http://www.cse.ust.hk/pg
http://www.cse.ust.hk/Restricted

Reformulated query:



```
[30/Apr/2019 18:20:43] "GET /search/result?query=%22computer+science%22+alumni
00 584468
['research', 'lab', 'comput', 'facil', 'depart']
No of related pages:268
[30/Apr/2019 18:20:54] "GET /search/similar?docId=29 HTTP/1.1" 200 258406
```

First result of similar page:



Score: 0.7518523289505272

WHAT Lab
http://www.cse.ust.hk/BDI/WHATLab
2019-04-30, 44541
wechat 26 data 18 propag 18 video 16 project 16

Parent's Link:
https://www.ust.hk/academics/list
http://www.cse.ust.hk/pg/research/areas
http://www.cse.ust.hk/pg/research/labs
https://www.ust.hk/academics/list#main-content

Conclusion

Strengths:

- Relatively fast retrieval because the tf-idf and document norm are pre-computed, and the meta information is stored as key-value pair in the database
- High relevance if the query matches the title of a page

Weakness:

- Word or phrase indexing only allow up to bigram due to storage limitation
- Title matching dominant the similarity score. Page content is less important
- The retrieval time will be slow down if more pages are crawled

Things to improve:

- Multithreading in spider and indexer to keep track of the crawling time and speed up crawling
- Separate the spider and indexer. Better OOP structure
- Better database design for keyword indexing, e.g. separate unigram and bigram storage