

Shakespeare Search Engine

Final Version

徐丰河 3170102881

王 晖 3170203638

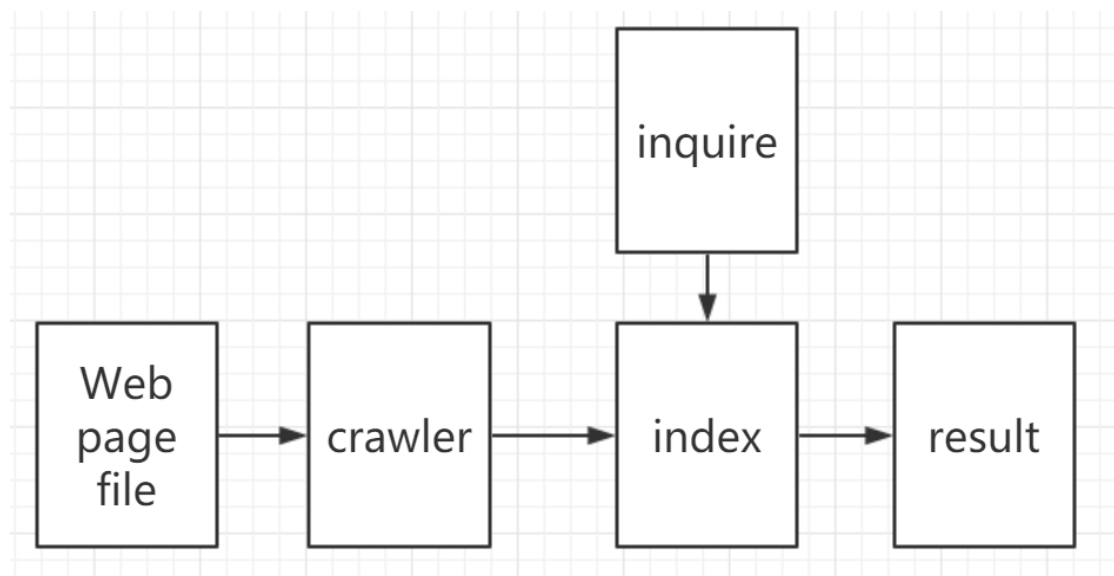
Date: 2019-03-25

Chapter 1: Introduction

Problem description, purpose of this report, and (if any) background of the data structures and the algorithms. ***Be concise.***

Our project is written to index the website <http://shakespeare.mit.edu/> , to provide services to the public for better user experiences and to explore more about the technologies applied behind search engines.

Following is an overview of the project:



Chapter 2: Data Structure / Algorithm Specification

Description (pseudo-code preferred) of all the algorithms involved for solving the problem, including specifications of main data structures.

Or, if you are to introduce a new data structure and its related operations, do it in this chapter.

2.0 Data Structure

2.0.0 B Tree with LRU for Indexing

```

class BTree:
    def __init__(self, degree, node_index=0, root_file_path='data/root'):
        # node_index: the node file index, should be unique, stored in a file
        self.node_index = node_index # should read from file, otherwise = 0, a new tree

        self.root = Node(True, self.__generate_node_index()) # root: a leaf node
        self.degree = degree # if degree = 2, 2-3-4 tree

        # DISK-WRITE(self.root)
        self.root.disk_write()
        self.root_path = root_file_path
        self.lru_list = []
        self.lru_size = 30

```

```

class Node:
    def __init__(self, is_leaf=True, file_index=-1):
        self.file_index = file_index
        self.is_leaf = is_leaf
        self.n_keys = 0
        self.keys = []
        self.child = []
        self.child_index = []

```

The B Tree we use is a Python implementation of the B Tree presented on Introduction to Algorithms, 3rd Edition[1]

B-TREE-CREATE(T)

```

1   $x = \text{ALLOCATE-NODE}()$ 
2   $x.\text{leaf} = \text{TRUE}$ 
3   $x.n = 0$ 
4   $\text{DISK-WRITE}(x)$ 
5   $T.\text{root} = x$ 

```

B-TREE-SEARCH(x, k)

```

1   $i = 1$ 
2  while  $i \leq x.n$  and  $k > x.\text{key}_i$ 
3       $i = i + 1$ 
4  if  $i \leq x.n$  and  $k == x.\text{key}_i$ 
5      return  $(x, i)$ 
6  elseif  $x.\text{leaf}$ 
7      return NIL
8  else  $\text{DISK-READ}(x.c_i)$ 
9      return B-TREE-SEARCH( $x.c_i, k$ )

```

B-TREE-INSERT(T, k)

```
1   $r = T.root$ 
2  if  $r.n == 2t - 1$ 
3       $s = \text{ALLOCATE-NODE}()$ 
4       $T.root = s$ 
5       $s.leaf = \text{FALSE}$ 
6       $s.n = 0$ 
7       $s.c_1 = r$ 
8      B-TREE-SPLIT-CHILD( $s, 1$ )
9      B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-TREE-INSERT-NONFULL( $r, k$ )
```

B-TREE-INSERT-NONFULL(x, k)

```
1   $i = x.n$ 
2  if  $x.leaf$ 
3      while  $i \geq 1$  and  $k < x.key_i$ 
4           $x.key_{i+1} = x.key_i$ 
5           $i = i - 1$ 
6       $x.key_{i+1} = k$ 
7       $x.n = x.n + 1$ 
8      DISK-WRITE( $x$ )
9  else while  $i \geq 1$  and  $k < x.key_i$ 
10       $i = i - 1$ 
11       $i = i + 1$ 
12      DISK-READ( $x.c_i$ )
13      if  $x.c_i.n == 2t - 1$ 
14          B-TREE-SPLIT-CHILD( $x, i$ )
15          if  $k > x.key_i$ 
16               $i = i + 1$ 
17      B-TREE-INSERT-NONFULL( $x.c_i, k$ )
```

B-TREE-SPLIT-CHILD(x, i)

```
1   $z = \text{ALLOCATE-NODE}()$ 
2   $y = x.c_i$ 
3   $z.\text{leaf} = y.\text{leaf}$ 
4   $z.n = t - 1$ 
5  for  $j = 1$  to  $t - 1$ 
6       $z.\text{key}_j = y.\text{key}_{j+t}$ 
7  if not  $y.\text{leaf}$ 
8      for  $j = 1$  to  $t$ 
9           $z.c_j = y.c_{j+t}$ 
10  $y.n = t - 1$ 
11 for  $j = x.n + 1$  downto  $i + 1$ 
12      $x.c_{j+1} = x.c_j$ 
13  $x.c_{i+1} = z$ 
14 for  $j = x.n$  downto  $i$ 
15      $x.\text{key}_{j+1} = x.\text{key}_j$ 
16  $x.\text{key}_i = y.\text{key}_t$ 
17  $x.n = x.n + 1$ 
18 DISK-WRITE( $y$ )
19 DISK-WRITE( $z$ )
20 DISK-WRITE( $x$ )
```

2.0.1 Term – Inverted Index

The inverted index is implemented by hashing (using dictionary in Python). Since Terms(Inverted Indices) are linked to the nodes of the index, i.e., B Tree, they hold a good property of both B Tree and Hash Table.

```
class Term:
    def __init__(self, term=None):
        self.term = term
        self.times = 0
        self.occure = dict()

    def jsonify(self):
        d = dict()
        d['term'] = self.term
        d['times'] = self.times
        d['occure'] = self.occure
        return d

    def unjsonfy(self, d):
        self.term = d['term']
        self.times = d['times']
        self.occure = d['occure']
        return self

    def get_links(self):
        bdict = BDict()
        base_uri = 'http://shakespeare.mit.edu/'
        return [(base_uri + bdict.search(1, key), len(self.occure[key])) for key in self.occure.keys()]
```

The word counting requirement (1 point) is implemented inside a member function of the Term class. When a stemmed word finds its corresponding Term instance, it is inserted, and the times field of the instance is incremented by 1.

```
def insert(self, doc, ith):
    if doc in self.occurences:
        self.occurences[doc].append(ith)
    else:
        self.occurences[doc] = [ith]
    self.times += 1
```

2.0.2 Bidirectional Dictionary

BDict class contains 2 dictionaries, and is intended for reduce the space consumed to uniquely identify a document.

```
class BDict:
    def __init__(self, file_name='data/link_label'):
        # if the file is not found
        self.file_name = file_name
        if not pathlib.Path(file_name).is_file():
            self.dict1 = dict()
            self.dict2 = dict()
            self.link_index = 0
        else:
            # if is found, read from the file
            self.disk_read()
```

We have an index generator, a new link is assigned with a index to identify and thus avoiding a page being crawled and index several times.

```
def __generate_link_index(self):
    self.link_index += 1
    return self.link_index - 1

def insert_new_link(self, link):
    link_index = self.__generate_link_index()
    self.__insert(link_index, link)
    return link_index

def is_page_crawled(self, key):
    if type(key) == int: # bug: this does not work
        return key in self.dict1
    if type(key) == str: # works fine
        return key in self.dict2
    return False
```

2.2 Algorithms

2.2.0 Term Filter

The term filter is applied when analyzing pages and querying terms to make sure that everything crawled and queried are sanitized.

For simplicity, omitting the pseudo code, we will just provide a straight forward example.

The process for filtering a term:

Term: *'Romeo and Juliet love each other.'*

Tokenized: [*'Romeo', 'and', 'Juliet', 'love', 'each', 'other', '.'*]

Punctuations Filtered: [*'Romeo', 'and', 'Juliet', 'love', 'each', 'other'*]

Stopwords Filtered: [*'romeo', 'juliet', 'love'*]

After Stemming: [*'romeo', 'juliet', 'lov'*]

As you can see, these are the words that make sense and thus will be fed to our search engine to index or query. (query processing: 2 points)

2.2.1 Web Page Crawler

After observing the structure of the website, we find that there are roughly 3 levels, so we just use a 3-level for-loop here.

Pseudo-code:

- Crawl the 0th-level page

 - Get all relative links on 0th-level page

 - Crawl all links to get 1st-level pages and analyze

 - Get all relative links on 1st-level pages

 - Crawl all 2nd-level links and analyze

- end

In the process, all full.html are ignored and made a list exclusively. Because full.html is a union of all chapters of a play, so analyzing these pages will roughly double the time of analyzing all the pages. And to avoid missing these full pages, inside the query function, we return full.html so long as a sub element of full.html is found.

2.2.2 LRU – Least Recently Used Page Replacement Algorithm

```

def __insert_lru(self, node):
    self.lru_list.append(node)
    if len(self.lru_list) > self.lru_size:
        self.lru_list[0].disk_write()
        self.lru_list.pop(0)

def __is_in_lru(self, node):
    if node == self.root:
        return True
    return node in self.lru_list

def __lru_high_priority(self, node):
    self.lru_list.append(self.lru_list.pop(self.lru_list.index(node)))

```

This algorithm is implemented using linked list. It is used in the BTree class in order to make sure that the mostly recently used nodes are inside the buffer for faster indexing and query.

2.2.3 Buffer-Disk Storage Mapping

Python supports JSON fairly well. All the data are firstly converted to a hash table (dictionary in Python), then stored in JSON(Javascript Object Notation) on disk, which is friendly for web servers.

```

# convert the node to json style to write on the disk
def __jsonfy(self):
    d = dict()
    d['file_index'] = self.file_index
    d['is_leaf'] = self.is_leaf
    d['n_keys'] = self.n_keys
    # should be maintained, or read from the children, type(child)
    # d['keys'] = self.keys[0:self.n_keys]
    d['keys'] = list(map(lambda key: key.jsonfy(), self.keys[0:self.n_keys]))
    d['child_index'] = self.child_index
    # print('d: ', d)
    return d

# get the node data from json document
def __unjsonfy(self, d):
    self.file_index = d['file_index']
    self.is_leaf = d['is_leaf']
    self.n_keys = d['n_keys']
    # self.keys = d['keys'].unjsonfy()
    # print(d['keys'])
    self.keys = list(map(lambda key: Term().unjsonfy(key), d['keys']))
    self.child_index = d['child_index']
    # bug fixed: should copy by value, not by reference
    self.child = self.child_index[:] # only replaced by node in buffer when necessary

def disk_write(self):
    # write node file header: self.file_index, self.n_keys
    with open('data/nodes/node_{0}'.format(self.file_index), 'w') as out:
        json.dump(self.__jsonfy(), out)
        out.close()

def disk_read(self):
    with open('data/nodes/node_{0}'.format(self.file_index), 'r') as infile:
        self.__unjsonfy(json.load(infile))
        infile.close()

```


Chapter 3: Testing Results

Table of test cases. Each test case usually consists of a brief description of the purpose of this case, the expected result, the actual behavior of your program, the possible cause of a bug if your program does not function as expected, and the current status (“*pass*”, or “*corrected*”, or “*pending*”).

3.0 Index Generation Test (5 points)

We use 6 servers with same configuration to generate different indices for different degree of the B Tree (30, 100, 200, 300, 400, 500 respectively) and compared the time consumption of each server. The result was quite astonishing at first but we were able to find a reasonable explanation of that.

With each degree, our Crawler crawls 952 pages precisely and returns the same result when searching a word, which shows the robustness of the whole program.

Following graph is our test results:

Degree	30	100	200	300	400	500
# nodes	758	155	72	55	40	33
Indexing Time(min)	23	72	58	31	7	5
Query Time	Fast	Much Slower	Much Slower	Slower	Fast	Fast
Query (2 nd Time)	Fast	Fast	Fast	Fast	Fast	Fast

3.0.1 The index generation test details

By using <https://en.ipip.net/ip.html> , we were able to find out that the server is located at the west coastline of the US, so we selected servers

nearby at New York provide by DigitalOcean.

shakeseare.mit.edu

Search

IP Data Information

Data	Info
IP	18.18.187.25 (rDNS: the-tech.mit.edu)
Location	United States Massachusetts Cambridge
Owner	mit.edu

ASN	CIDR	Information
AS3	18.18.0.0/16	MIT-GATEWAYS - Massachusetts Institute of Technology, US

Third party

IP2Location	MaxMind GEOLite2
United States Massachusetts Cambridge 18.0.0.0 - 18.127.255.255	United States Massachusetts Cambridge 18.0.0.0 - 18.31.255.255

The server configuration is listed as below:

Choose an image ?

Distributions

Container distributions

Marketplace

Custom images

Ubuntu

18.10 x64

FreeBSD

Select version

Fedora

Select version

Debian

Select version

CentOS

Select version

Choose a plan

STARTER

PERFORMANCE

Standard

General Purpose

LTD

CPU Optimized

Standard virtual machines with a mix of memory and compute resources. Best for small projects that can handle variable levels of CPU performance, like blogs, web apps and dev/test environments.

\$5/mo

\$0.007/hour

1 GB / 1 CPU

25 GB SSD disk

1000 GB transfer

\$10/mo

\$0.015/hour

2 GB / 1 CPU

50 GB SSD disk

2 TB transfer

\$15/mo

\$0.022/hour

3 GB / 1 CPU

60 GB SSD disk

3 TB transfer

\$15/mo

\$0.022/hour

2 GB / 2 CPUs

60 GB SSD disk

3 TB transfer

\$15/mo

\$0.022/hour

1 GB / 3 CPUs

60 GB SSD disk

3 TB transfer

\$20/mo


\$0.030/hour

4 GB / 2 CPUs

80 GB SSD disk

4 TB transfer

We created 6 servers with same configuration to explore the search engine performance with different degrees of B Tree.





















ads

Class project / Educational purposes

→ Move Resources

Resources
Activity
Settings

DROPLETS (6)

  ubuntu-s-1vcpu-1gb-nyc1-01	104.248.112.118	500		...
  ubuntu-s-1vcpu-1gb-nyc1-01	134.209.222.102	400		...
  ubuntu-s-1vcpu-1gb-nyc1-01	134.209.214.114	200		...
  ubuntu-s-1vcpu-1gb-nyc1-01	142.93.118.27	100		...
  ubuntu-s-1vcpu-1gb-nyc1-01	157.230.215.145	30		...
  ubuntu-s-1vcpu-1gb-nyc1-01	68.183.111.213	300		...

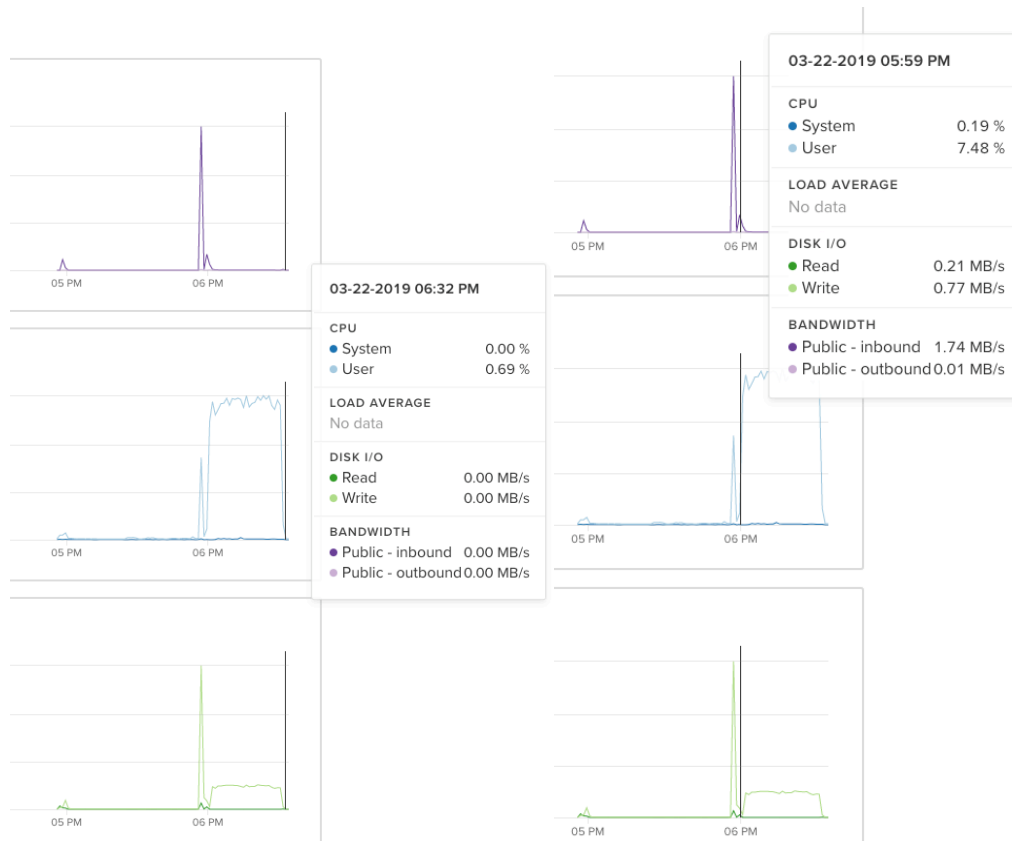
By changing the argument that the Index takes to construct the B Tree, we were able to test B Trees with different degrees in our tests.

```
class ShakeCrawler(crawler.Crawler):
    base_uri = 'shakespeare.mit.edu'

    def __init__(self):
        super(ShakeCrawler, self).__init__(self.base_uri)
        self.index = Index(100)
        self.bdict = BDict()
```

Figure 1 Index with B Tree of degree 100

Digital Ocean provides monitors for server. We were able to record the indexing time by refer to the CPU Usage graphs.



After Indexing, we found that 952 pages were indexed (each full.html excluded).

```
[root@ubuntu-s-1vcpu-1gb-nyc1-01:~/shakespeare-search-engine/data# cat link_label
{"link_index": 952, "dict1": {"0": "index.html", "1": "news.html", "2": "allswell/index.html", "3": "allswell/allswell.1.1.html", "4": "allswell/allswell.1.2.html", "5": "allswell/allswell.1.3.html", "6": "allswell/allswell.2.1.html", "7": "allswell/allswell.2.2.html", "8": "allswell/allswell.2.3.html", "9": "allswell/allswell.2.4.html", "10": "allswell/allswell.2.5.html", "11": "allswell/allswell.3.1.html", "12": "allswell/allswell.3.2.html", "13": "allswell/allswell.3.3.html", "14": "allswell/allswell.3.4.html", "15": "allswell/allswell.3.5.html", "16": "allswell/allswell.3.6.html", "17": "allswell/allswell.3.7.html", "18": "allswell/allswell.4.1.html", "19": "allswell/allswell.4.2.html", "20": "allswell/allswell.4.3.html", "21": "allswell/allswell.4.4.html", "22": "allswell/allswell.4.5.html", "23": "allswell/allswell.5.1.html", "24": "allswell/allswell.5.2.html", "25": "allswell/allswell.5.3.html", "26": "asyoulikeit/index.html", "27": "asyoulikeit/asyoulikeit.1.1.html", "28": "asyoulikeit/asyoulikeit.1.2.html", "29": "asyoulikeit/asyoulikeit.1.3.html", "30": "asyoulikeit/asyoulikeit.2.1.html", "31": "asyoulikeit/asyoulikeit.2.2.html", "32": "asyoulikeit/asyoulikeit.2.3.html", "33": "asyoulikeit/asyoulikeit.2.4.html", "34": "asyoulikeit/asyoulikeit.2.5.html"}
```

```
t.CXVI.html": 909, "Poetry/sonnet.CXVII.html": 910, "Poetry/sonnet.CXVIII.html":
  911, "Poetry/sonnet.CXIX.html": 912, "Poetry/sonnet.CXX.html": 913, "Poetry/son
net.CXXI.html": 914, "Poetry/sonnet.CXXII.html": 915, "Poetry/sonnet.CXXIII.html
": 916, "Poetry/sonnet.CXXIV.html": 917, "Poetry/sonnet.CXXV.html": 918, "Poetry
/sonnet.CXXVI.html": 919, "Poetry/sonnet.CXXVII.html": 920, "Poetry/sonnet.CXXVI
II.html": 921, "Poetry/sonnet.CXXIX.html": 922, "Poetry/sonnet.CXXX.html": 923,
"Poetry/sonnet.CXXXI.html": 924, "Poetry/sonnet.CXXXII.html": 925, "Poetry/sonne
t.CXXXIII.html": 926, "Poetry/sonnet.CXXXIV.html": 927, "Poetry/sonnet.CXXXV.htm
l": 928, "Poetry/sonnet.CXXXVI.html": 929, "Poetry/sonnet.CXXXVII.html": 930, "P
oetry/sonnet.CXXXVIII.html": 931, "Poetry/sonnet.CXXXIX.html": 932, "Poetry/sonn
et.CXL.html": 933, "Poetry/sonnet.CXLI.html": 934, "Poetry/sonnet.CXLII.html": 9
35, "Poetry/sonnet.CXLIII.html": 936, "Poetry/sonnet.CXLIV.html": 937, "Poetry/s
onnet.CXLV.html": 938, "Poetry/sonnet.CXLVI.html": 939, "Poetry/sonnet.CXLVII.ht
ml": 940, "Poetry/sonnet.CXLVIII.html": 941, "Poetry/sonnet.CXLIX.html": 942, "P
oetry/sonnet.CL.html": 943, "Poetry/sonnet.CLI.html": 944, "Poetry/sonnet.CLII.h
tml": 945, "Poetry/sonnet.CLIII.html": 946, "Poetry/sonnet.CLIV.html": 947, "Poe
try/LoversComplaint.html": 948, "Poetry/RapeOfLucrece.html": 949, "Poetry/VenusA
ndAdonis.html": 950, "Poetry/elegy.html": 951}}root@ubuntu-s-1vcpu-1gb-nyc1-01:~
/shakespeare-search-engine/data# █
```

3.1 Block Tests

As you will find in our python code modules, we hold a good style in coding, when every module is implemented, we test them before instantiation, which guarantees a good team cooperation.

```
# ## block test
def test_insertion():
    btree = BTree(2)
    btree.insert(1)
    btree.insert(6)
    # insert between
    btree.insert(4)

    # should split child
    btree.insert(7)

    # another split right side
    btree.insert(9)
    btree.insert(5)
    #

    btree.insert(8)
    btree.insert(10)
    btree.insert(11)
    btree.insert(12)
    # print("inserting 2:")
    # btree.insert(2)

    btree.root.display()
    for c in btree.root.child:
        c.display()

def test_disk_write():
    print('test: disk_write')
    btree = BTree(2)
    btree.insert(1)
    btree.insert(6)
    btree.insert(4)
    btree.insert(10)
    btree.insert(11)
    btree.insert(12)
    btree.root.display()
    btree.root.disk_write()
```

```

# block tests
def test_crawl_html():
    sc = ShakeCrawler()
    # get the links on the first page
    print(sc.get_relative_links(''))

    # when no link on the page
    print(sc.get_relative_links('allswell/allswell.4.1.html'))

def test_crawl_relative_links():
    sc = ShakeCrawler()
    # get the links on the first page
    print(sc.get_relative_links(''))

    # when no link on the page
    print(sc.get_relative_links('allswell/allswell.4.1.html'))

def test_analyze_page():
    sc = ShakeCrawler()
    if not sc.bdict.is_page_crawled(1):
        sc.analyze_page('news.html', sc.bdict.insert_new_link('news.html'))

```

Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the algorithms. Comments on comparing with other known data structures and algorithms. Further possible improvements.

4.0 The efficiency of the B Tree

As stated before the result is a little bit out of our expectation. We were confused at first, but finally came up with an analysis that makes sense.

When of degree around 30, the tree is really high, but the access time can always be guaranteed to $\log(n)$, when of degree around 400, the tree becomes flat, so the height of the tree may be compressed to like 5 levels. So the only cost to consider is the linear access time in each node, which is quite small in a small scale problem.

To analyze this problem quantitatively, we can yield the following

worst-case complexity:

Worst Case: $O(\text{Height} + 2 * \text{Degree} - 1) + O(1)$

Where the height of the B Tree and the degree of B Tree are the decisive factors. Above representation can be then simplified to be:

$O(\log N + M)$

Where N is the number of nodes on the tree, and M is the degree of the tree.

4.1 The Efficiency of LRU

As you may find, when searching a term for the second time, the speed rises, which means our LRU Algorithm works fairly well.

4.2 Further Exploration

We think our search engine is able to handle bigger problems, since no latency was observed when searching and 952 pages were indexed and analyzed in merely 5 minutes with low end server.

The time consumed for query is not carefully benchmarked, but since the query and indexing process is symmetric, we can expect a query performance proportional to that of indexing. But a further exact measurement should be taken. Thus, we made our source, deployment of the search engine server and search engine website open for everyone to make further test.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd Edition, 2009

[2] <https://cs50.harvard.edu/college/>

Author List

3170102881 徐丰河: Coding & Presentation & Documentation

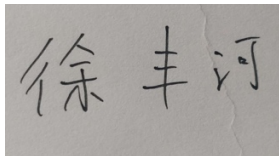
3170103638 王晖: Check & Tests & Documentation

Declaration

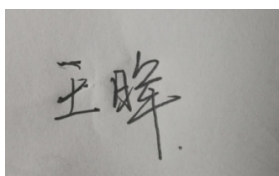
We hereby declare that all the work done in this project titled "Shakespeare Search Engine" is of our independent effort as a group.

Signatures

徐丰河:

A photograph of a piece of paper with the handwritten Chinese characters '徐丰河' (Xu Fenghe) in black ink.

王晖:

A photograph of a piece of paper with the handwritten Chinese characters '王晖' (Wang Hui) in black ink.