CPEG 657 Project Report

Benhang Fan, Wanzhu Chen, Wei Zhong

1. Overview

In our project, we have made a mathematic formula search engine and its demo system. The system has a WEB interface and behaves more or less like the Google search engine, which takes one query in the text box as input and returns the ranking results as well as the links to each document. However, our system accepts LaTex as input, the specific purpose of our system is to help users search mathematic formula and its derivations. We would like to target the potential users to be scholars who want to explore a formula: where it comes from and where its derivation is.

2. Existing Systems and The Usefulness of Our System

There does exist some well known mathematic formula search engines (e.g. uni-quation, Wolfra-Alpha and latexsearch.com), but in terms of user experience, after our team members try to use them searching different queries several times, we believe their results and ranking are not satisfactory. For example, when we try to find the derivation of $a^2 + b^2 = c^2$, none of these search engines returns us any useful result.

On the other hand, our search engine focus on mathematic formula and helping users find its derivation, that is, we have a more specific search engine and it can do its job better in its own area. To give an example to illustrate the usefulness of our system, consider the inequation below:

$$(1 + \frac{1}{n})^n < n$$

assume user want to know the derivation of this equation, then the results for each of the pre-existing system are not quite helpful:

- Wolfra-Alpha takes a long time before returning a *Standard computation time exceeded...* error
- uni-quation.com returns: *Sorry, we don't know this formula. Don't give up, try to find solution with pen & paper*.
- latexsearch.com will not even give a result for $(1 + \frac{1}{n})^n$

But our system is able to return documents with queries similar to $(1+\frac{1}{n})^n < n$, like $n > (1+\frac{1}{n})^n$ and $(1+\frac{1}{x})^x = x$ or even $(1+\frac{1}{n})^n = n$.

In terms of searching for mathematic formula derivations, our system performs better and shows more robustness compared with the three base-system listed above from our point of view (but no thorough evaluation has been done yet). However, we have at least demonstrated the potential and practical possibility to have such an mathematic derivation search system.

The sections below will further introduce the implementation of our system.

3. Crawling

For now we only have a manually chosen collection, we manually select a number of conclusions or derivation samples from websites using LaTex (e.g. math.stackexchange.com). It is very likely to have a crawler in the future if we make use of the LaTex source contents in websites using LaTex based contents(e.g. MathJax).

4. Tokenization and Tree Construction

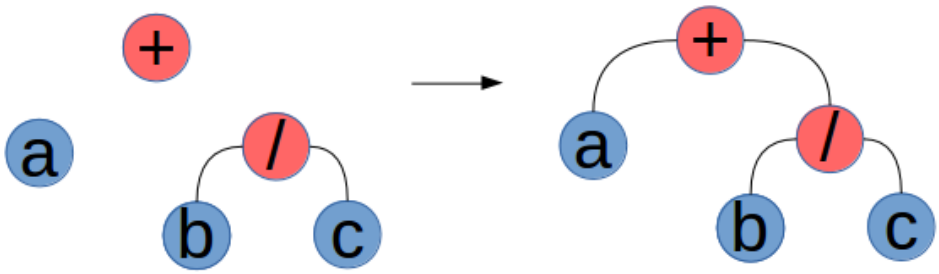
To parse the LaTex language we choose to tokenize a subset of math related LaTex, we use Lex/Flex and Yacc/Bison to tokenize the LaTex Language and construct a "tree" for each equation.

In the level of Lex/Flex, we group LaTex token into variables, different basic math operators, equal class, times class, fraction class and square root. And we omit undefined control sequence for the sake of robustness. Then we give the grammar listed below for Yacc/Bison to parse the LaTex and reduce LaTex grammar into tree structure:

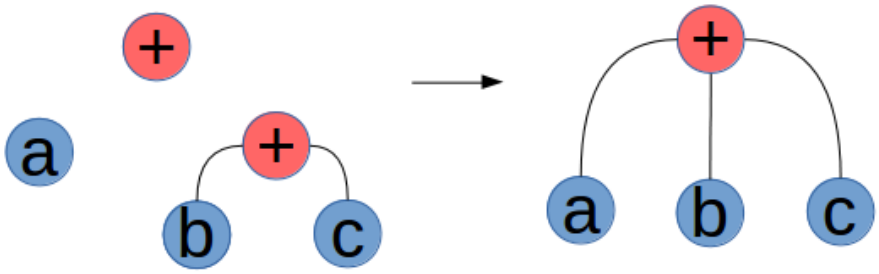
```
0 $accept: doc $end

 1 doc: %empty
 2    | doc query

 3 query: tex '\n'
 4      | '\n'

 5 tex: term
 6    | tex '+' term
 7    | tex '-' term
 8    | '-' tex
 9    | tex EQ_CLASS tex

10 term: factor
11     | term factor
12     | term TIMES factor
13     | term DIV factor

14 body: '{' tex '}'
```

```
15     | '(' tex ')'
16     | VAR
17 factor: body
18       | factor script
19       | ABS_L tex ABS_R
20       | FRAC '{' tex '}' '{' tex '}'
21       | SUM_CLASS body
22       | SUM_CLASS script body
23       | body '!'
24       | SQRT '[' tex ']' body
25       | SQRT body
26 script: '_' VAR
27       | '_' VAR '^' VAR
28       | '_' VAR '^' '{' tex '}'
29       | '^' VAR '_' '{' tex '}'
30       | '^' VAR
31       | '^' VAR '_' VAR
32       | '^' '{' tex '}'
33       | '^' '{' tex '}' '_' VAR
34       | '^' '{' tex '}' '_' '{' tex '}'
35       | '_' '{' tex '}'
36       | '_' '{' tex '}' '^' VAR
37       | '_' '{' tex '}' '^' '{' tex '}'
```

When a grammar is reduced, the tokens is converted to a tree node directly or by attaching sub-trees which is reduced previously to the new root. The following example illustrates how a sub-tree is generated when the *addition grammar* (grammar 6 listed above) is reduced:



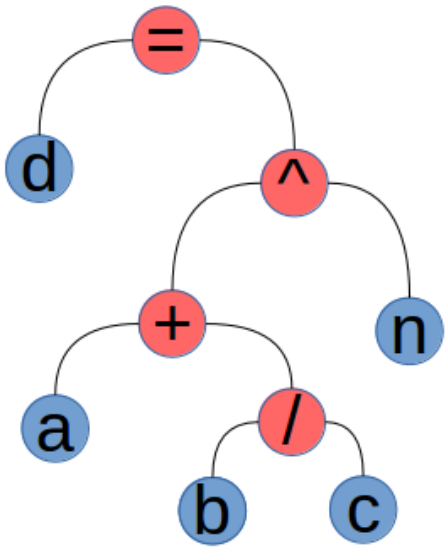
We also notice that some operations may have commutative property, therefore all the sons in two adjacent levels will be attached to the same root in the cases of addition and multiplication:



In this way, we will finally get a tree structured representation for a mathematic equation. Given the following equation for example,

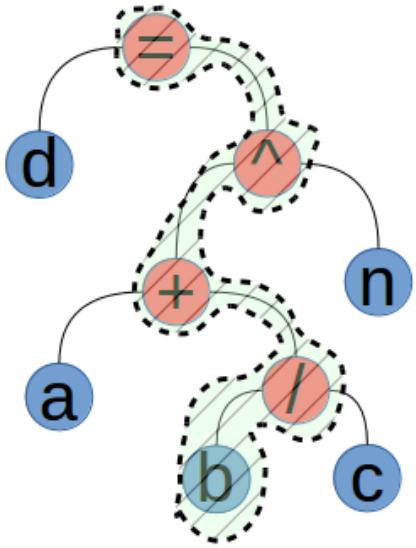
$$(a + \frac{b}{c})^n = d$$

the output of our tree will be:



5. Comparing Trees

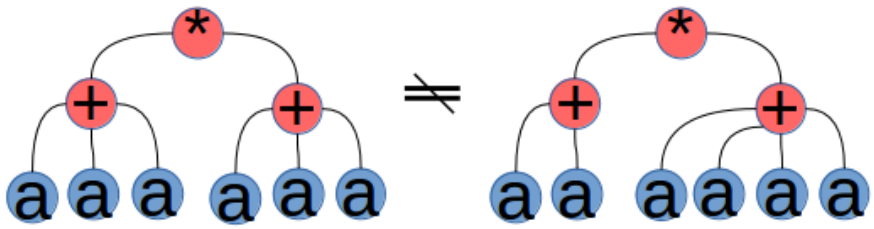
After constructing a tree, we will extract the branch word of a tree. What we call a branch word here, is just the tokens going from the leaves to the root of a tree in order:



For the example above, we will finally get five branch words to be compared with those of other trees. We notice that one math equation can use different symbol sets, so we choose not to distinguish the leaves' actual symbol in the branch word.

We also notice the branch word is not enough to fully distinguish trees, an example would be the equation $(a + a + a) \times (a + a + a)$ and $(a + a) \times (a + a + a + a)$,

they have the same branch words while have different meanings in mathematical language:



Therefore we decide to introduce weights to the branch word. We define the weight of node n_i as the sum of that of its successors, which is given by:

$$w(n_i) = \sum^{n} 1 \qquad n = \{succ(n_i) \cup n_i\}$$

Then we store all the branch words from each tree into file system, using the following format:

```
3X1-3-5-7-9-
```

to represent the weights of each token in branch word, and save a posting file that contains weight information into ./collection directory, the path where the posting file is saved is the token name of a branch word in order (e.g. `./collection/var/frac/add`).

In the format shown above, the first number 3 represents there are 3 identical branch words for this formula, and the string after character X is the weights of the branch word in the following format:

```
w0-w1-w2-.....-wn-
```

To compare two trees, we first go to the directory in the order of query tokens of a branch word, and for each of the posting file in that directory or sub-directories, we use all the branch words from query tree to compare with those of the other in the posting file respectively. And calculate the similarity between two branch words, sum the similarity degree for each related formula in the document collection, then rank all the related formula using the sum score.

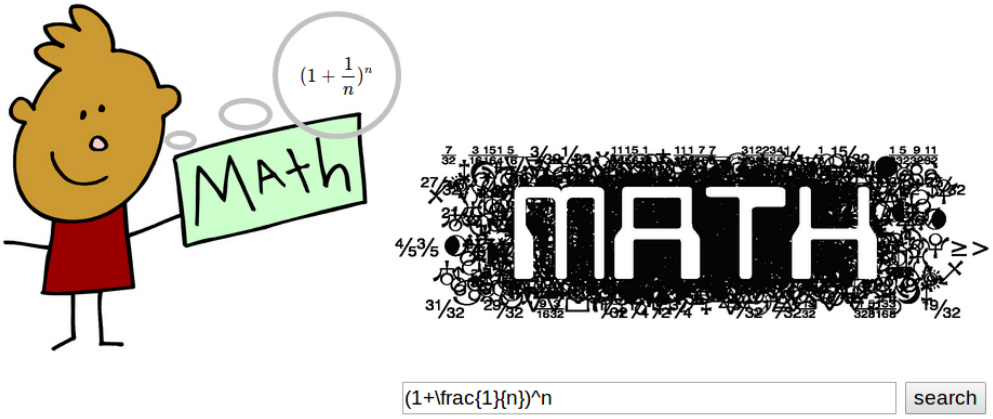
In order to give the detailed formula to calculate the difference degree between two branch words, we have these notations: let m be the number of continuous matches between two branches from the begining of the branch word, n be number of same branch word, l be the length of branch word, for query branch word i and document branch word j, we give the similarity formula:

$$\begin{cases} s_{i,j} & = \min(n_i, n_j) \times \dfrac{m}{l_i} + \dfrac{1}{|n_i - n_j| + 1} \times \dfrac{m}{\max(l_i, l_j)} \\ S & = \sum_{k \in T} s_{i,j}^{k} \end{cases}$$

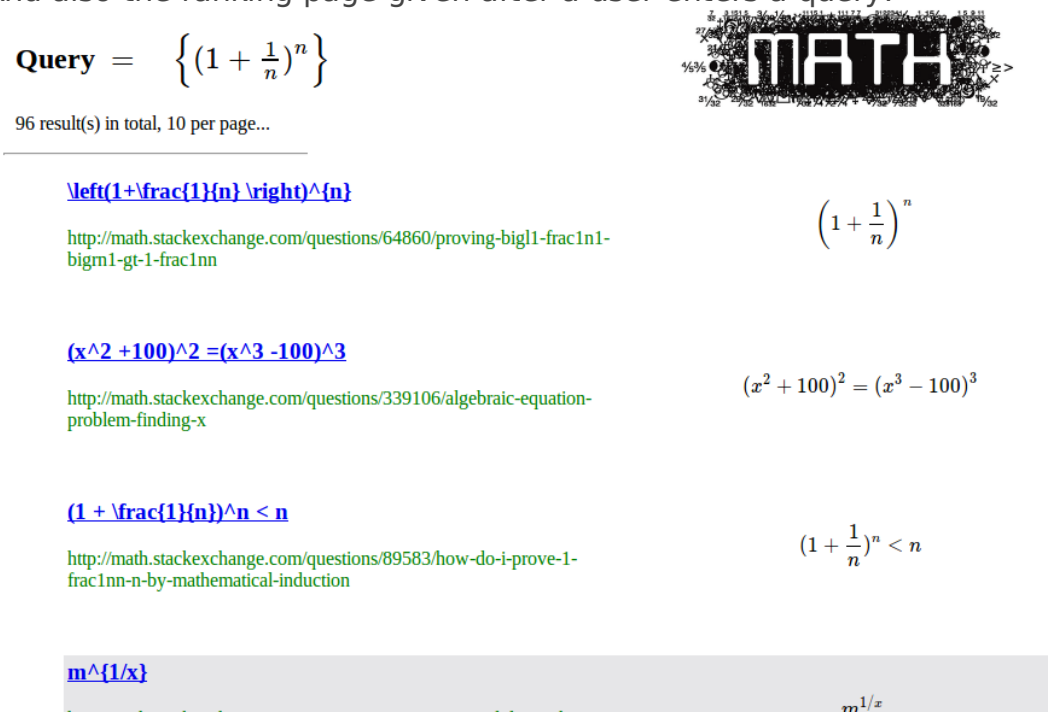
Our ranking program will use formula above to score each formula in the related collection and output the ranked formula to web page interface.

6. WEB Interface

The WEB interface we have now, contains a HTML homepage for user to input query in an input box:



And also the ranking page given after a user enters a query:



For the homepage, we have a manually created HTML file, in which an input tag will pass user's query to our ranking page who will further call our *ma-se* program.

As for the ranking page, our WEB interface uses Apache CGI program as a middle layer to get user query from homepage HTML form, using the POST request. Also the CGI

program will serve as the ranking page http daemon to output the ranking in a HTML form. The POST data will be passed to the standard input of our CGI program.

Then we use *libcurl* open-source library (curl-7.30.0) to unescape the URL encoding, search the query in our collection, return the related formula and ranking page using standard output of CGI program.

The reason to unescape the URL encoding is obvious, when the user submit a HTML (the input box tag is named *q*) form as query:

```
q=a + b
```

Our CGI program will get a string looks like:

```
q=a+%2B+b
```

In our CGI program, we first replace the `+` symbol in POST input by space using a function:

```
void replace_plus(char *str)
{
    while (str[0] != '\0') {
        if (str[0] == '+')
            str[0] = ' ';
        str ++;
    }
}
```

After plus symbols are replaced, the new string would be something like this:

```
q=a %2B b
```

To unescape the URL encoding, the *libcurl* API calling routine is:

```
#include <curl/curl.h>
....
CURL *curl = curl_easy_init();
unescape = curl_easy_unescape(curl, arg, 0, NULL);
```

After we unescape the URL encoding, the string will be exactly what the user inputs in the previous HTML form.

Our CGI program also accepts a GET argument which is passed by GET request in HTTP, the GET argument will specify the start point where the ranking page is returned to users, in the case there are many search results found, we will only return 10 results per page.

The GET argument is passed to our CGI by environment variable QUERY_STRING:

```
include <stdlib.h>
....
char *get_str = getenv("QUERY_STRING");
```

The CGI program of our project can be found in the `./httpd` of our project directory, you can also find some images to be displayed in our WEB interface and the *libcurl* library binary to be linked with CGI program in that directory.

7. An example to go over our system

First, get the source code:

```
$ git clone https://github.com/t-k-/math-se.git
$ cd math-se
```

Uncomment this line in inter-def.h:

```
1 | #define _MY_DEBUG
```

which will enable the debug outputs.

Build the source code by typing:

```
$ make
```

(The Makefile attempts to install the CGI binary to /var/www/cgi directory which you should make first, also it installs a search homepage into /var/www/html directory to be used by a local Apache server, you also need to install Apache first)

Now, suppose we manually get a piece of formula derivation from a Website, we first parse the LaTex using our parser *ma-pa* and input the formula $\left(a+\frac{1}{b}\right)^2$:

```
$ ./ma-pa http://example.com
(a+\frac{1}{b})^2
  └──  ^ (7 )
     |──  + (5 )
     |  |──  a (1 )
     |  └──  / (3 )
     |     |──  1 (1 )
     |     └──  b (1 )
     └──  2 (1 )
branch word: var(w=1), add(w=5), sus(w=7).
branch word: var(w=1), frac(w=3), add(w=5), sus(w=7).
branch word: var(w=1), frac(w=3), add(w=5), sus(w=7).
branch word: var(w=1), sus(w=7).
```

The program will output the tree constructed from that formula as well as the branch words.

Type Enter to terminate the *ma-pa* program and input this equation as document into our collection file system:

```
$ ./co-in
record f31ae4bade734553c3647a4dca076e8c @@ ./collection/var/add/sus/post
record f31ae4bade734553c3647a4dca076e8c @@ ./collection/var/frac/add/sus
record f31ae4bade734553c3647a4dca076e8c @@ ./collection/var/sus/posting.
```

Now you can see our ./collection directory will have some posting files and document links:

```
$ tree collection/
collection/
└── var
    ├── add
    │   └── sus
    │       ├── f31ae4bade734553c3647a4dca076e8c
    │       └── posting
    ├── frac
    │   └── add
    │       └── sus
    │           ├── f31ae4bade734553c3647a4dca076e8c
    │           └── posting
    └── sus
        ├── f31ae4bade734553c3647a4dca076e8c
        └── posting
```

To search a formula in our collection, run the *ma-se* command. The following command searches for the formula $\frac{1}{b}$ in our collection:

```
$ ./ma-se '\frac{1}{b}'
  └──  / (3 )
     ├──  1 (1 )
     └──  b (1 )
branch word: var(w=1), frac(w=3).
branch word: var(w=1), frac(w=3).
query:
./collection/var/frac 2X 1-3-
under ./collection/var/frac:
referred_by ./collection/var/frac/add/sus/posting
2X1-3-5-7-   f31ae4bade734553c3647a4dca076e8c
query branch word: num=2, length=2, weight=[1 3 ]
doc branch word: num=2, length=4, weight=[1 3 5 7 ]
weight matches m=2
score=250.
writing rank file...
score=250: ./collection/var/frac/add/sus/f31ae4bade734553c3647a4dca076e8
$ cat rank
(a+\frac{1}{b})^2
http://example.com
```

As you can see from the output, the search program will compare the branch words between query formula and document formula, then calculate a score for each related document, and write the ranking to a file called *rank* which ranks the formula we just input($\left(a+\frac{1}{b}\right)^2$) to the top. The rank file will later be used by our CGI program to be displayed in a nice HTML file.

If you have the Apache (version 2.4.7 in our case) installed properly, you can open the browser and use our WEB interface through the URL:

http://127.0.0.1/search.html

8. Conclusion

Through this project, we have gained a basic concept and practical experience on search engine system. We have not only applied the knowledge we acquire in class, but also tried some inovative idea and extended our knowledge by exploring the potential to build something new and better. We are amazed by the usefullness of a search engine can become, and some of our group members are willing to work on this project futher and hopefully they will be able to create an unique and useful search engine in the future.

9. Slides

The slides of our presentation can be accessed from:
http://thoughts-of.me/deck.js/slide2.html