Reading Report - Week 4
CS 221 - Information Retrieval
Prof. Cristina Lopes

Varad Meru
Student # 26648958
Due Date: 01/31/2015

How Google Code Search Worked*: A Summary

By Russ Cox

Summary and Commentary

The author starts with an explanation of the background of his work and its domain. He also noted the absence of real automata in then popular regexp implementations and the motivation for his work. He then jumps into *Indexed Word Search*. He gives details about the use of inverted indexes to maintain the information about the text in any document. The indexes would contain tokens and the document it is seen in. This can be extended to store the location in any document as well. Such structure would generate something like eq. (1), where $x_i, (i = 1 \leq i \leq n)$ is a document, from all the n documents and the locations are preserved for $token_j$.

$$token_j : \{\{x_1, x_2, ..., x_n\}, \{\{x_1 : 1, 4, ...\}, \{x_2 : 23, 56, ...\}, ..., \{x_n : 212, 14, ...\} \tag{1}$$

Then the discussion moves to *Indexed Regular Expression Search* and starts explaining its differences from Indexed Word Search and its complexity. He describes the use of *n-grams* in building an index of substrings of length n. So now a token is represented with only the documents it is listed in, as shown as $regex\_token_j$ in eq. (2), as there is no purpose of location in a regex implementation.

$$regex\_token_j : \{x_1, x_2, ..., x_n\} \tag{2}$$

The queries for such a context are given with the use of ANDs and ORs, which would be suitably placed by partitioning the input query, such as `/Google.*Search/` would be converted into `Goo AND oog AND ogl AND gle AND Sea AND ear AND arc AND rch`. This is a complicated process, especially due to the presence of ORs as well in the regular expression grammar. The rules for the computing the five specific results. The rules are for *empty string*, *single character*, *zero or one*, *zero or more*, *one or more*, *alternation*, *concatenation* and the results that are generated are *emptyable*, *exact*, *prefix*, *suffix*, *match*. He also points to the exponentially large size of the string sets based on size. He then starts describing the `trigram()` function which would generate trigrams given either a string or a set of strings. Some examples on uses of `trigram()` are given in Figure 1

`trigrams`(abcd) = `trigrams`(abcd) = abc AND bcd
`trigrams`(abcd, wxyz) = `trigrams`(abcd) OR `trigrams`(wxyz) &
`trigrams`(abcd) OR `trigrams`(wxyz) = (abc AND bcd) OR (wxy AND xyz)

Figure 1: `trigram()` in action

The explanation of the implementation, deployment and functioning of this implementation is given in the subsequent parts. The search works as given in Figure 2.

```
csearch [-c] [-f fileregexp] [-h] [-i] [-l] [-n] regexp
```

Figure 2: Running the search on code using the build `csearch` binary.

The author concludes with some history of usage of *n-grams* and his thoughts on application of regular expressions, trigrams and its usage in the indexed matcher.

Commentary: This blog introduces the beauty in the design of elegant search engines for specialized applications and catering to a very specific useless, but with scalability, usability and architectural extensibility in mind. It also describes how simple ideas can help in building very successful applications, with very high value.

** How Google Code Search Worked* by Russ Cox. Can be found at `http://swtch.com/~rsc/regexp/regexp4.html`