Regular Expression Matching with a Trigram Index
or
How Google Code Search Worked

Russ Cox
rsc@swtch.com
January 2012

Introduction

In the summer of 2006, I was lucky enough to be an intern at Google. At the time, Google had an internal tool called gsearch that acted as if it ran grep over all the files in the Google source tree and printed the results. Of course, that implementation would be fairly slow, so what gsearch actually did was talk to a bunch of servers that kept different pieces of the source tree in memory: each machine did a grep through its memory and then gsearch merged the results and printed them. Jeff Dean, my intern host and one of the authors of gsearch, suggested that it would be cool to build a web interface that, in effect, let you run gsearch over the world's public source code. I thought that sounded fun, so that's what I did that summer. Due primarily to an excess of optimism in our original schedule, the launch slipped to October, but on October 5, 2006 we did launch (by then I was back at school but still a part-time intern).

I built the earliest demos using Ken Thompson's Plan 9 grep, because I happened to have it lying around in library form. The plan had been to switch to a "real" regexp library, namely PCRE, probably behind a newly written, code reviewed parser, since PCRE's parser was a well-known source of security bugs. The only problem was my then-recent discovery that none of the popular regexp implementations - not Perl, not Python, not PCRE - used real automata. This was a surprise to me, and even to Rob Pike, the author of the Plan 9 regular expression library. (Ken was not yet at Google to be consulted.) I had learned about regular expressions and automata from the Dragon Book, from theory classes in college, and from reading Rob's and Ken's code. The idea that you wouldn't use the guaranteed linear time algorithm had never occurred to me. But it turned out that Rob's code in particular used an algorithm only a few people had ever known, and the others had forgotten about it years earlier. We launched with the Plan 9 grep code; a few years later I did replace it, with RE2.

Code Search was Google's first and only search engine to accept regular expression queries, which was geekily great but a very small niche. The sad fact is that many programmers can't write regular expressions, let alone correct ones. When we started Code Search, a Google search for "regular expression search engine" turned up sites where you typed "phone number" and got back "`\(\d{3}\) \d{3}-\d{4}`".

Google open sourced the regular expression engine I wrote for Code Search, RE2, in March 2010. Code Search and RE2 have been a great vehicle for educating people about how to do regular expression search safely. In fact, Tom Christiansen recently told me that even people in the Perl community use it (`perl -Mre::engine::RE2`), to run regexp search engines (the real kind) on the web without opening themselves up to trivial denial of service attacks.

In October 2011, Google announced that it would shut down Code Search as part of its efforts to refocus on higher-impact products, and now Code Search is no longer online. To mark the occasion, I thought it would be appropriate to write a little about how Code Search worked. The actual Code Search was built on top of Google's world-class document indexing and retrieval tools; this article is accompanied by an implementation that works well enough to index and search large code bases on a single computer.

Indexed Word Search

Before we can get to regular expression search, it helps to know a little about how word-based full-text search is implemented. The key data structure is called a posting list or inverted index, which lists, for every possible search term, the documents that contain that term.

For example, consider these three very short documents:

```
(1) Google Code Search
(2) Google Code Project Hosting
(3) Google Web Search
```

The inverted index for these three documents looks like:

```
Code: {1, 2}
Google: {1, 2, 3}
Hosting: {2}
Project: {2}
Search: {1, 3}
Web: {3}
```

To find all the documents that contain both Code and Search, you load the index entry for Code {1, 2} and intersect it with the list for Search {1, 3}, producing the list {1}. To find documents that contain Code or Search (or both), you union the lists instead of intersecting them. Since the lists are sorted, these operations run in linear time.

To support phrases, full-text search implementations usually record each occurrence of a word in the posting list, along with its position:

```
Code: {(1, 2), (2, 2)}
Google: {(1, 1), (2, 1), (3, 1)}
Hosting: {(2, 4)}
```

```
Project: {(2, 3)}
Search: {(1, 3), (3, 4)}
Web: {(3, 2)}
```

To find the phrase “Code Search”, an implementation first loads the list for Code and then scans the list for Search to find entries that are one word past entries in the Code list. The (1, 2) entry in the Code list and the (1, 3) entry in the Search list are from the same document (1) and have consecutive word numbers (2 and 3), so document 1 contains the phrase “Code Search”.

An alternate way to support phrases is to treat them as AND queries to identify a set of candidate documents and then filter out non-matching documents after loading the document bodies from disk. In practice, phrases built out of common words like “to be or not to be” make this approach unattractive. Storing the position information in the index entries makes the index bigger but avoids loading a document from disk unless it is guaranteed to be a match.

Indexed Regular Expression Search

There is too much source code in the world for Code Search to have kept it all in memory and run a regexp search over it all for every query, no matter how fast the regexp engine. Instead, Code Search used an inverted index to identify candidate documents to be searched, scored, ranked, and eventually shown as results.

Regular expression matches do not always line up nicely on word boundaries, so the inverted index cannot be based on words like in the previous example. Instead, we can use an old information retrieval trick and build an index of *n*-grams, substrings of length *n*. This sounds more general than it is. In practice, there are too few distinct 2-grams and too many distinct 4-grams, so 3-grams (trigrams) it is.

Continuing the example from the last section, the document set:

```
(1) Google Code Search
(2) Google Code Project Hosting
(3) Google Web Search
```

has this trigram index:

```
_Co: {1, 2}     Sea: {1, 3}     e_W: {3}        ogl: {1, 2, 3}
_Ho: {2}        Web: {3}        ear: {1, 3}     oje: {2}
_Pr: {2}        arc: {1, 3}     eb_: {3}        oog: {1, 2, 3}
_Se: {1, 3}     b_S: {3}        ect: {2}        ost: {2}
_We: {3}        ct_: {2}        gle: {1, 2, 3}  rch: {1, 3}
Cod: {1, 2}     de_: {1, 2}     ing: {2}        roj: {2}
Goo: {1, 2, 3}  e_C: {1, 2}     jec: {2}        sti: {2}
Hos: {2}        e_P: {2}        le_: {1, 2, 3}  t_H: {2}
Pro: {2}        e_S: {1}        ode: {1, 1}     tin: {2}
```

(The _ character serves here as a visible representation of a space.)

Given a regular expression such as `/Google.*Search/`, we can build a query of ANDs and ORs that gives the trigrams that must be present in any text matching the regular expression. In this case, the query is

`Goo` AND `oog` AND `ogl` AND `gle` AND `Sea` AND `ear` AND `arc` AND `rch`

We can run this query against the trigram index to identify a set of candidate documents and then run the full regular expression search against only those documents.

The conversion to a query is not simply a matter of pulling out the text strings and turning them into AND expressions, although that is part of it. A regular expression using the | operator will lead to a query with OR terms, and parenthesized subexpressions complicate the process of turning strings into AND expressions.

The full rules compute five results from each regular expression *r*: whether the empty string is a matching string, the exact set of matching strings or an indication that the exact set is unknown, a set of prefixes of all matching strings, a set of suffixes of all matching strings, and a match query like the above, often describing the middle of the string. The rules follow from the meaning of the regular expressions:

‘’ (empty string)
- emptyable(‘’) = true
- exact(‘’) = {‘’}
- prefix(‘’) = {‘’}
- suffix(‘’) = {‘’}
- match(‘’) = ANY (special query: match all documents)

`c` (single character)
- emptyable(`c`) = false
- exact(`c`) = {`c`}
- prefix(`c`) = {`c`}
- suffix(`c`) = {`c`}
- match(`c`) = ANY

e? (zero or one)

emptyable(e?) = true
exact(e?) = exact(e) ∪ {''}
prefix(e?) = {''}
suffix(e?) = {''}
match(e?) = ANY

e* (zero or more)

emptyable(e*) = true
exact(e*) = unknown
prefix(e*) = {''}
suffix(e*) = {''}
match(e*) = ANY

e+ (one or more)

emptyable(e+) = emptyable(e)
exact(e+) = unknown
prefix(e+) = prefix(e)
suffix(e+) = suffix(e)
match(e+) = match(e)

$e_1 \mid e_2$ (alternation)

emptyable($e_1 \mid e_2$) = emptyable(e_1) or emptyable(e_2)
exact($e_1 \mid e_2$) = exact(e_1) ∪ exact(e_2)
prefix($e_1 \mid e_2$) = prefix(e_1) ∪ prefix(e_2)
suffix($e_1 \mid e_2$) = suffix(e_1) ∪ suffix(e_2)
match($e_1 \mid e_2$) = match(e_1) OR match(e_2)

$e_1\ e_2$ (concatenation)

emptyable(e_1e_2) = emptyable(e_1) and emptyable(e_2)
exact(e_1e_2) = exact(e_1) × exact(e_2), if both are known
  or unknown, otherwise
prefix(e_1e_2) = exact(e_1) × prefix(e_2), if exact(e_1) is known
  or prefix(e_1) ∪ prefix(e_2), if emptyable(e_1)
  or prefix(e_1), otherwise
suffix(e_1e_2) = suffix(e_1) × exact(e_2), if exact(e_2) is known
  or suffix(e_2) ∪ suffix(e_1), if emptyable(e_2)
  or suffix(e_2), otherwise
match(e_1e_2) = match(e_1) AND match(e_2)

The rules as described above are correct but would not produce very interesting match queries, and the various string sets could get exponentially large depending on the regular expression. At each step, we can apply some simplifications to keep the computed information manageable. First, we need a function to compute trigrams.

The trigrams function applied to a single string can be ANY, if the string has fewer than three characters, or else the AND of all the trigrams in the string. The trigrams function applied to a set of strings is the OR of the trigrams function applied to each of the strings.

(Single string)

- trigrams(`ab`) = ANY
- trigrams(`abc`) = `abc`
- trigrams(`abcd`) = `abc` AND `bcd`
- trigrams(`wxyz`) = `wxy` AND `xyz`

(Set of strings)

- trigrams({`ab`}) = trigrams(`ab`) = ANY
- trigrams({`abcd`}) = trigrams(`abcd`) = `abc` AND `bcd`
- trigrams({`ab`, `abcd`}) = trigrams(`ab`) OR trigrams(`abcd`) = ANY OR (`abc` AND `bcd`) = ANY
- trigrams({`abcd`, `wxyz`}) = trigrams(`abcd`) OR trigrams(`wxyz`) = (`abc` AND `bcd`) OR (`wxy` AND `xyz`)

Using the trigrams function, we can define transformations that apply at any step during the analysis, to any regular expression e. These transformations preserve the validity of the computed information and can be applied as needed to keep the result manageable:

(Information-saving)

- At any time, set match(e) = match(e) AND trigrams(prefix(e)).

- At any time, set match(e) = match(e) AND trigrams(suffix(e)).
- At any time, set match(e) = match(e) AND trigrams(exact(e)).

(Information-discarding)

- If prefix(e) contains both s and t where s is a prefix of t, discard t.
- If suffix(e) contains both s and t where s is a suffix of t, discard t.
- If prefix(e) is too large, chop the last character off the longest strings in prefix(e).
- If suffix(e) is too large, chop the first character off the longest strings in suffix(e).
- If exact(e) is too large, set exact(e) = unknown.

The best way to apply these transformation is to use the information-saving transformations immediately before applying an information-discarding transformation. Effectively, these transformations move information that is about to be discarded from the prefix, suffix, and exact sets into the match query itself. On a related note, a little more information can be squeezed from the concatenation analysis: if e_1e_2 is not exact, match(e_1e_2) can also require trigrams(suffix(e_1) × prefix(e_2)).

In addition to these transformations, it helps to apply basic Boolean simplifications to the match query as it is constructed: "`abc` OR (`abc` AND `def`)" is more expensive but no more precise than "`abc`".

Implementation

To demonstrate these ideas, I have published a basic implementation, written in Go, at code.google.com/p/codesearch. If you have a recent weekly snapshot of Go installed you can run

```
goinstall code.google.com/p/codesearch/cmd/{cindex,csearch}
```

to install binaries named `cindex` and `csearch`. If you don't have Go installed, you can download binary packages containing binaries for FreeBSD, Linux, OpenBSD, OS X, and Windows.

The first step is to run `cindex` with a list of directories or files to include in the index:

```
cindex /usr/include $HOME/src
```

By default `cindex` adds to the existing index, if one exists, so the last command is equivalent to the pair of commands:

```
cindex /usr/include
cindex $HOME/src
```

With no arguments, `cindex` refreshes an existing index, so after running the previous commands,

```
cindex
```

will rescan `/usr/include` and `$HOME/src` and rewrite the index. Run `cindex -help` for more details.

The indexer assumes that files are encoded in UTF-8. It rejects files that are unlikely to be interesting, such as those that contain invalid UTF-8, or that have very long lines, or that have a very large number of distinct trigrams.

The index file contains a list of paths (for reindexing, as described above), a list of indexed files, and then the same posting lists we saw at the beginning of this article, one for each trigram. In practice, this index tends to be around 20% of the size of the files being indexed. For example, indexing the Linux 3.1.3 kernel sources, a total of 420 MB, creates a 77 MB index. Of course, the index is organized so that only a small fraction needs to be read for any particular search.

Once the index has been written, run `csearch` to search:

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

The regexp syntax is RE2's, which is to say basically Perl's, but without backreferences. (RE2 does support capturing parentheses; see the footnote at the bottom of code.google.com/p/re2 for the distinction.) The boolean command-line flags are like `grep`'s, except that as is standard for Go programs, there is no distinction between short and long options, so options cannot be combined: it is `csearch -i -n`, not `csearch -in`. The new `-f` flag causes `csearch` to consider only files with paths matching *fileregexp*.

```
$ csearch -f /usr/include DATAKIT
/usr/include/bsm/audit_domain.h:#define BSM_PF_DATAKIT          9
/usr/include/gssapi/gssapi.h:#define GSS_C_AF_DATAKIT    9
/usr/include/sys/socket.h:#define       AF_DATAKIT      9               /* datakit protocols */
/usr/include/sys/socket.h:#define       PF_DATAKIT      AF_DATAKIT
$
```

The `-verbose` flag causes `csearch` to report statistics about the search. The `-brute` flag bypasses the trigram index, searching every file listed in the index instead of using a precise trigram query. In the case of the Datakit query, it turns out that the trigram query narrows the search down to just three files, all of which are matches.

```
$ time csearch -verbose -f /usr/include DATAKIT
2011/12/10 00:23:24 query: "AKI" "ATA" "DAT" "KIT" "TAK"
2011/12/10 00:23:24 post query identified 3 possible files
```

```
/usr/include/bsm/audit_domain.h:#define BSM_PF_DATAKIT          9
/usr/include/gssapi/gssapi.h:#define GSS_C_AF_DATAKIT    9
/usr/include/sys/socket.h:#define       AF_DATAKIT      9               /* datakit protocols */
/usr/include/sys/socket.h:#define       PF_DATAKIT      AF_DATAKIT
0.00u 0.00s 0.00r
$
```

In contrast, without the index we'd have to search 2,739 files:

```
$ time csearch -brute -verbose -f /usr/include DATAKIT
2011/12/10 00:25:02 post query identified 2739 possible files
/usr/include/bsm/audit_domain.h:#define BSM_PF_DATAKIT          9
/usr/include/gssapi/gssapi.h:#define GSS_C_AF_DATAKIT    9
/usr/include/sys/socket.h:#define       AF_DATAKIT      9               /* datakit protocols */
/usr/include/sys/socket.h:#define       PF_DATAKIT      AF_DATAKIT
0.08u 0.03s 0.11r  # brute force
$
```

(I am using `/usr/include` on an OS X Lion laptop. You may get slightly different results on your system.)

As a larger example, we can search for `hello world` in the Linux 3.1.3 kernel. The trigram index narrows the search from 36,972 files to 25 files and cuts the time required for the search by about 100x.

```
$ time csearch -verbose -c 'hello world'
2011/12/10 00:31:16 query: " wo" "ell" "hel" "llo" "lo " "o w" "orl" "rld" "wor"
2011/12/10 00:31:16 post query identified 25 possible files
/Users/rsc/pub/linux-3.1.3/Documentation/filesystems/ramfs-rootfs-initramfs.txt: 2
/Users/rsc/pub/linux-3.1.3/Documentation/s390/Debugging390.txt: 3
/Users/rsc/pub/linux-3.1.3/arch/blackfin/kernel/kgdb_test.c: 1
/Users/rsc/pub/linux-3.1.3/arch/frv/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/mn10300/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/media/video/msp3400-driver.c: 1
0.01u 0.00s 0.01r
$

$ time csearch -brute -verbose -h 'hello world'
2011/12/10 00:31:38 query: " wo" "ell" "hel" "llo" "lo " "o w" "orl" "rld" "wor"
2011/12/10 00:31:38 post query identified 36972 possible files
/Users/rsc/pub/linux-3.1.3/Documentation/filesystems/ramfs-rootfs-initramfs.txt: 2
/Users/rsc/pub/linux-3.1.3/Documentation/s390/Debugging390.txt: 3
/Users/rsc/pub/linux-3.1.3/arch/blackfin/kernel/kgdb_test.c: 1
/Users/rsc/pub/linux-3.1.3/arch/frv/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/mn10300/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/media/video/msp3400-driver.c: 1
1.26u 0.42s 1.96r  # brute force
$
```

For case-insensitive searches, the less precise query means that the speedup is not as great, but it is still an order of magnitude better than brute force:

```
$ time csearch -verbose -i -c 'hello world'
2011/12/10 00:42:22 query: ("HEL"|"HEl"|"HeL"|"Hel"|"hEL"|"hEl"|"heL"|"hel")
  ("ELL"|"ELl"|"ElL"|"Ell"|"eLL"|"eLl"|"elL"|"ell")
  ("LLO"|"LLo"|"LlO"|"Llo"|"lLO"|"lLo"|"llO"|"llo")
  ("LO "|"Lo "|"lO "|"lo ") ("O W"|"O w"|"o W"|"o w") (" WO"|" Wo"|" wO"|" wo")
  ("WOR"|"WOr"|"WoR"|"Wor"|"wOR"|"wOr"|"woR"|"wor")
  ("ORL"|"ORl"|"OrL"|"Orl"|"oRL"|"oRl"|"orL"|"orl")
  ("RLD"|"RLd"|"RlD"|"Rld"|"rLD"|"rLd"|"rlD"|"rld")
2011/12/10 00:42:22 post query identified 599 possible files
/Users/rsc/pub/linux-3.1.3/Documentation/filesystems/ramfs-rootfs-initramfs.txt: 3
/Users/rsc/pub/linux-3.1.3/Documentation/java.txt: 1
/Users/rsc/pub/linux-3.1.3/Documentation/s390/Debugging390.txt: 3
/Users/rsc/pub/linux-3.1.3/arch/blackfin/kernel/kgdb_test.c: 1
/Users/rsc/pub/linux-3.1.3/arch/frv/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/mn10300/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/powerpc/platforms/powermac/udbg_scc.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/media/video/msp3400-driver.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/net/sfc/selftest.c: 1
/Users/rsc/pub/linux-3.1.3/samples/kdb/kdb_hello.c: 2
0.07u 0.01s 0.08r
$

$ time csearch -brute -verbose -i -c 'hello world'
2011/12/10 00:42:33 post query identified 36972 possible files
/Users/rsc/pub/linux-3.1.3/Documentation/filesystems/ramfs-rootfs-initramfs.txt: 3
```

```
/Users/rsc/pub/linux-3.1.3/Documentation/java.txt: 1
/Users/rsc/pub/linux-3.1.3/Documentation/s390/Debugging390.txt: 3
/Users/rsc/pub/linux-3.1.3/arch/blackfin/kernel/kgdb_test.c: 1
/Users/rsc/pub/linux-3.1.3/arch/frv/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/mn10300/kernel/gdb-stub.c: 1
/Users/rsc/pub/linux-3.1.3/arch/powerpc/platforms/powermac/udbg_scc.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/media/video/msp3400-driver.c: 1
/Users/rsc/pub/linux-3.1.3/drivers/net/sfc/selftest.c: 1
/Users/rsc/pub/linux-3.1.3/samples/kdb/kdb_hello.c: 2
1.24u 0.34s 1.59r  # brute force
$
```

To minimize I/O and take advantage of operating system caching, `csearch` uses `mmap` to map the index into memory and in doing so read directly from the operating system's file cache. This makes `csearch` run quickly on repeated runs without using a server process.

The source code for these tools is at code.google.com/p/codesearch. The files illustrating the techniques from this article are `index/regexp.go` (regexp to query), `index/read.go` (reading from index), and `index/write.go` (writing an index).

The code also includes, in `regexp/match.go`, a custom DFA-based matching engine tuned just for the `csearch` tool. The only thing this engine is good for is implementing grep, but that it does very quickly. Because it can call on the standard Go package to parse regular expressions and boil them down to basic operations, the new matcher is under 500 lines of code.

History

Neither *n*-grams nor their application to pattern matching is new. Shannon used *n*-grams in his seminal 1948 paper "A Mathematical Theory of Communication" [1] to analyze the information content of English text. Even then, *n*-grams were old hat. Shannon cites Pratt's 1939 book *Secret and Urgent* [2], and one imagines that Shannon used *n*-gram analysis during his wartime cryptography work.

Zobel, Moffat, and Sacks-Davis's 1993 paper "Searching Large Lexicons for Partially Specified Terms using Compressed Inverted Files" [3] describes how to use an inverted index of the *n*-grams in a set of words (a lexicon) to map a pattern like `fro*n` to matches like `frozen` or `frogspawn`. Witten, Moffat, and Bell's 1994 classic book *Managing Gigabytes* [4] summarized the same approach. Zobel et al.'s paper mentions that the technique can be applied to a richer pattern language than just `*` wildcards, but only demonstrates a simple character class:

> Note that *n*-grams can be used to support other kinds of pattern matching. For example, if *n* = 3 and patterns contain sequences such as `ab[cd]e`, where the square brackets denote that the character between `b` and `e` must be either `c` or `d`, then matches can be found by looking for strings containing either `abc` and `bce` or `abd` and `bde`.

The main difference between the Zobel paper and our implementation is that a gigabyte is no longer a large amount of data, so instead of applying *n*-gram indexing and pattern matching to just the word list, we have the computational resources to apply it to an entire document set. The trigram query generation rules given above generate the query

```
(abc AND bce) OR (abd AND bde)
```

for the regular expression `ab[cd]e`. If the implementation chose to apply the simplifying transformations more aggressively, it would use a smaller memory footprint but arrive at

```
(abc OR abd) AND (bce OR bde)
```

instead. This second query is equally valid but less precise, demonstrating the tradeoff between memory usage and precision.

Summary

Regular expression matching over large numbers of small documents can be made fast by using an index of the trigrams present in each document. The use of trigrams for this purpose is not new, but neither is it well known.

Despite all their apparent syntactic complexity, regular expressions in the mathematical sense of the term can always be reduced to the few cases (empty string, single character, repetition, concatenation, and alternation) considered above. This underlying simplicity makes it possible to implement efficient search algorithms like the ones in the first three articles in this series. The analysis above, which converts a regular expression into a trigram query, is the heart of the indexed matcher, and it is made possible by the same simplicity.

If you miss Google Code Search and want to run fast indexed regular expression searches over your local code, give the standalone programs a try.

Acknowledgements

Thanks to everyone at Google who worked on or supported Code Search over the past five years. There are too many to name individually, but it was truly a team effort and a lot of fun.

References

[1] Claude Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, July, October 1948.

[2] Fletcher Pratt, *Secret and Urgent: The Story of Codes and Ciphers*, 1939. Reprinted by Aegean Park Press, 1996.

[3] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis, "Searching Large Lexicons for Partially Specified Terms using Compressed Inverted Files," *Proceedings of the 19th VLDB Conference*, 1993.

[4] Ian Witten, Alistair Moffat, and Timothy Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 1994. Second Edition, 1999.