インターフェイスの街角 (63) — 全文検索システム

増井俊之

2002 年 12 月号で、GETA という検索ライブラリを使用した類似ファイル検索システムをとりあげました。今回は、この GETA を用いて構築した、もうすこし実用的な全文検索システムを紹介します。

検索エンジン GETA

汎用連想計算エンジン GETA (Generic Engine for Transposable Association)[1]は、情報処理振興事業協会 (IPA) が実施した「独創的情報技術育成事業」[2]のもとで、国立情報学研究所の高野明彦氏、日立製作所の西岡真吾氏らにより開発されたシステムです。これを利用すれば、出現頻度にもとづく単語の重みを考慮した全文検索システムが構築できます。

GETA は、以下のライブラリなどから構成されています。

- WAM アクセス・ライブラリ (libwam)
  文書中の単語の出現頻度を示す大規模な疎行列を表現するデータ構造 **WAM** (Word-Article Matrix) を扱うためのライブラリ
- Association Engine (libae)
  WAM を用いて tf・idf 法のような連想計算をおこなうためのライブラリ
- クラスタリング・ライブラリ (libcs)
  文書のクラスタリングをサポートするライブラリ
- その他のユーティリティ・ライブラリとコマンド

これらを使用することにより、全文検索システムを柔軟に構築することができます。

図 1　頻度ファイルの例

```
@/user/masui/DOC/text1
1 です
1 の
1 関連
1 記事
1 検索
1 文書
@/user/masui/DOC/text2
1 に関する
1 技術
1 検索
2 文書
```

2002 年 12 月号では、tf・idf 法による類似ファイル検索手法をとりあげ、GETA を用いた類似ファイル検索システムについても簡単に説明しました。しかし、そのときに紹介した手法では、日常的にファイルの全文検索をおこなうには速度と閲覧性の面で問題がありました。

以下では、これらの点を改善し、時間軸に沿った検索もできるように拡張したシステムを紹介します。

速度の改善

たとえば、text1 の内容が「文書検索関連の記事です」、text2 の内容が「文書検索技術に関する文書」であるとき、GETA では図 1 のような形式の ``頻度ファイル''を作成してから WAM を生成し、検索に利用します。

この図のように文章を単語単位に分割して頻度ファイルを作るには、通常は「茶筌」[3]や「MeCab」[4]などの形態素解析プログラムを使用します。しかし、形態素解析の処

1 http://geta.ex.nii.ac.jp/
2 http://www.ipa.go.jp/STC/dokusou.html
3 http://chasen.aist-nara.ac.jp/
4 http://cl.aist-nara.ac.jp/~taku-ku/software/mecab/

図 2　頻度ファイルを生成するプログラム (makefreqfiles)

```
#!/usr/bin/env ruby
# -*- mode: ruby -*-

require 'find'
require 'kconv'
require 'ftools'
require 'parsedate'

irdir = File.dirname($0)
$: << irdir

require 'fileattr'
require 'timerange'

class FreqfileGenerator
  MECAB = "/usr/local/bin/mecab"
  NKF = "/usr/local/bin/nkf -e"

  def initialize(getaroot)
    @getaroot = getaroot
    @home = ENV['HOME']
    @offset = @home.length + 1
  end

  def generate(dir)
    path = "#{@home}/#{dir}"
    Find.find(path){ |file|
      @file = file
      @fileattr = FileAttr.new(file)
      relpath = file[@offset,file.length]
      @freqfile = "#{@getaroot}/freqfiles/⇒
                   #{relpath}.freq"
      if @fileattr.valid? && freqfile_missing?
        puts @freqfile
        create_freqfile
      end
    }
  end

  def freqfile_missing?
    !test(?f,@freqfile) ||
      File.stat(@freqfile).mtime < ⇒
      File.stat(@file).mtime
  end

  def create_freqfile
    tmpfile = "/tmp/mecabtmp#{$$}"
    mecab = IO.popen("#{NKF} | #{MECAB} ⇒
            > #{tmpfile}","w")
    @fileattr.text.each { |line|
      mecab.puts line
    }
    mecab.close

    freq = {}
    File.open(tmpfile,"r"){ |f|
      f.each { |line|
        word = line.split.first.downcase
        freq[word] = freq[word].to_i + 1 ⇒
                     if word != 'EOS'
      }
    }
    File.delete(tmpfile)

    File.makedirs(File.dirname(@freqfile))
    File.open(@freqfile,"w"){ |f|
      f.puts "@#{@file}"
      freq.each { |word,count|
        next if word[0] == 0x0a
        f.puts "#{count} #{word}"
      }

      TimeRange.rangefreq(@fileattr.time).⇒
        each { |freqline|
        f.puts freqline
      }
    }
  end
end

root = ARGV[0]
exit unless root

fg = FreqfileGenerator.new(root)

fg.generate('IR')
fg.generate('DOC')
```

(誌面の都合上、⇒ で折り返しています。以下同様)

理には時間がかかるため、大量のファイルを処理しようとするとかなりの時間が必要です。一方、頻度ファイルを WAM に変換する処理は、形態素解析に比較すれば高速におこなうことができます。

12 月号で紹介した方法では、毎回あらゆるファイルに対して形態素解析処理を実行して頻度ファイルを生成し、それをもとに WAM を生成していました。これに対し、ファイルごとに頻度ファイルを用意し、ファイルが変更されたときだけ頻度ファイルを更新するようにすれば、頻度ファイル用のディスクがよぶんに必要にはなりますが、インデックス作成にかかる時間を大幅に短縮できます。

検索対象ファイルごとに頻度ファイルを作成するための Ruby スクリプト makefreqfiles を図 2 に示します。

ホーム・ディレクトリが /user/masui のとき、

```
% makefreqfiles /user/masui/IR
```

のように検索ディレクトリを指定して makefreqfiles を起動すると、/user/masui/Mail/inbox/1000 のような

図 3　検索対象のファイルの絞込み (fileattr.rb)

```
class FileAttr
  def initialize ...
  def title ...    # ファイルのタイトル
  def text ...     # ファイルのテキストを配列で返す
  def time ...     # ファイル生成日時
  def valid? ...   # 検索対象とすべきかどうかを判定
end
```

図 4　検索環境の設定と WAM 生成

```
% GETAROOT=/user/masui/IR; export GETAROOT
% cd $GETAROOT
% cat etc/ci.conf
handle: all.ir
short-name: ir
dataroot: /user/masui/IR/data/
jma:p: /user/masui/IR/bin/japanese.sh
% find freqfiles -type f -print > freqfilelist
% /usr/local/geta/sbin/mkw ir @freqfilelist
```

図 5　検索結果

```
% search 'wiki'
13236  0.030859   /Users/masui/Mail/backup/1491
31647  0.030555   /Users/masui/Mail/inbox.old/8720
14810  0.030507   /Users/masui/Mail/backup/2919
24108  0.030505   /Users/masui/Mail/inbox.old/13257
18030  0.029959   /Users/masui/Mail/inbox/24590
31707  0.029737   /Users/masui/Mail/inbox.old/8870
23226  0.029689   /Users/masui/Mail/inbox.old/11901
16363  0.029488   /Users/masui/Mail/inbox/20850
18291  0.029488   /Users/masui/Mail/inbox/25099
31723  0.029223   /Users/masui/Mail/inbox.old/8894
%
```

テキストファイルに対し、freqfiles ディレクトリ以下に /user/masui/IR/freqfiles/Mail/inbox/1000.freq という頻度ファイルが生成されます。すでに頻度ファイルがある場合はそれを使うため、形態素解析プログラム MeCab を用いて頻度ファイルを生成するのは、ファイルが新しく生成されたり更新されたりしたときだけです。

12 月号のプログラムでは、find コマンドを用いて検索対象ファイルのリストを得ていました。一方、今回の makefreqfiles では、指定されたディレクトリ以下のファイルを Ruby のライブラリを使用して総当たり方式で探すようにしています。ディレクトリにはさまざまな種類のファイルが存在する可能性がありますが、バイナリファイルはそのままでは形態素解析の対象にはなりませんし、一時ファイルや辞書ファイルなどについては全文検索をおこなう意味はありません。そこで、検索対象とすべきファイルに対してのみ、valid?メソッドが真を返すようにしています (図 3)。

makefreqfiles を実行すると、検索対象としたいすべてのファイルに対する頻度ファイルが freqfiles ディレクトリの下に生成されます。ここで、頻度ファイルのリストを指定して GETA の WAM 生成コマンド mkw を起動することにより、定義ファイル ci.conf で指定した位置に WAM データが生成されます。

ci.conf の内容と WAM 生成処理の様子を図 4 に示します。

作成された WAM を用いて 12 月号で説明したような検索を実行すると、図 5 の結果が得られます。

閲覧性の改善

図 5 の search コマンドでは、検索されたファイルの ID、重み、ファイル名の一覧が出力されます。しかし、このままでは検索結果のファイルの内容すら簡単には確認できません。

検索結果を手軽にブラウズするには、図 5 のような検索結果を HTML 形式に変換したものを利用するとよいでしょう。

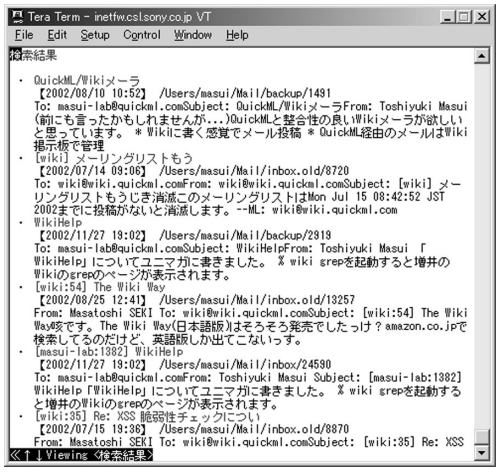
図 6 の search2html コマンドを使うと、単純な形式の HTML ファイルに簡単に変換できます。生成された HTML ファイルの閲覧には、もちろん普通の Web ブラウザを使ってもかまいません。ただし、コマンドラインから検索を実行した場合などは、テキストベースのブラウザである w3m を使うと便利です。

たとえば、図 5 の出力を search2html で加工して作成

図 6　検索結果を HTML に変換するプログラム (search2html)

```
#!/usr/bin/env ruby
# -*- mode: ruby -*-

$: << File.dirname($0)

require 'simplehtml'
require 'fileattr'

search_result = STDIN.readlines

sh = SimpleHtmlGenerator.new

head = sh.head {
  sh.title { '検索結果' }
}

body = sh.body {
  sh.h1 { '検索結果' } +
    sh.ul {
    search_result.collect { |line|
      (id, score, path) = line.split
      if test(?f,path) then
        sh.li {
          fa = FileAttr.new(path)
          sh.a('href' => path){
            fa.title
          }  +
            sh.br +
            fa.time.strftime("【%Y/%m/%d %H:%M】 ") +
            path +
            sh.br +
            fa.text[0..8].join
        } + "\n"
      else
        ''
      end
    }.join
  }
}

html = sh.html {
  sh.doctype + "\n" + head + "\n" + body
}

puts html
```

図 7　search2html の出力を w3m で閲覧



したHTMLファイルをw3mで閲覧すると、図7のようになります。w3mでは、EnterやTAB、Bなどのキー操作でHTMLファイルを軽快にブラウズすることができます。

時間情報を利用した検索

たとえば、〝2年ほど前にA氏から受け取ったメール〟のように、時間を指定して検索をおこないたいこともあります。このような場合、A氏から受け取ったメールをすべて検索してから時間（この例では〝2年ほど前〟）でフィルタリングしたり、あるいは検索対象をあらかじめ時系列に並べ替えておく方法も考えられます。しかし、これでは通常のキーワード検索とは異なる枠組みが必要となって面倒です。また、ファイルサイズなど、時間以外の属性を用いて検索したくなった場合には、そのつど新たな枠組みを用意しなければなりません。

このように、文書に含まれるキーワード以外のものも検索条件として指定したい場合、必要になりそうな検索条件をあらかじめキーワードに変換して頻度ファイルに登録しておけば、通常の全文検索の枠組みで条件を指定することができます。

たとえば、サイズが大きいファイルに対して〝大きなファイル〟という文字列を登録しておけば、〝大きなファイル〟というキーワードによる検索が可能になります。

さらに、ファイルの作成日やメールの送信日を頻度ファイルにキーワードとして追加しておけば、その日付を検索条件とした検索も可能になります。しかし、単純に日付を書き加えただけでは、1999年12月30日に作成したファイルと2000年1月3日に作成したファイルはまったく異なるキーワードをもつことになってしまい、〝1999年の年末あたり〟といった検索条件の指定が難しくなります。

現実には日時を厳密に指定した検索が必要な場合は稀と

図 8　期間を表すキーワードを生成する timerange.rb

```
#!/usr/bin/env ruby
# -*- mode: ruby -*-

class TimeRange
  DAYSEC = 24*60*60
  MONTHSEC = DAYSEC * 30

  WEIGHT = [1,2,4,2,1]
  RANGES = [1,2,3,6,12,24,48]

  def TimeRange.keyword(t,range,offset=0)
    rangesec = range * MONTHSEC
    base = t.to_i / rangesec
    @start = (base+offset) * rangesec
    "R#{range}T#{base+offset}"
  end

  def TimeRange.rangefreq(t)
    RANGES.collect { |range|
      (-2..2).collect { |i|
        "#{WEIGHT[i+2]} ⇒
         #{keyword(t,range,i)}"
      }
    }.flatten
  end

  def TimeRange.rangearg(argstr)
    argstr =~ /^(+)((+))?([my])$/
    first = $1.to_i
    last = ($3 ? $3.to_i : first) + 1
    if $4 == 'y' then
      first *= 12
      last *= 12
    end
    range = last - first

    cursec = Time.new.to_i

    srange = RANGES.find { |r|
      r >= range
    }
    return nil if srange.nil?

    #  puts "first = #{first}"
    #  puts "last = #{last}"
    #  puts "srange = #{srange}"

    startsec = cursec - last * MONTHSEC
    #  puts "startsec = ⇒
       #{Time.at(startsec).asctime}"

    middlesec = startsec + srange * MONTHSEC / 2
    keyword(middlesec,srange)
  end
end

if $0 == __FILE__ then # テスト用
  cursec = Time.new.to_i
#  puts TimeRange.keyword(cursec,1)
  puts TimeRange.rangefreq(cursec)
#  puts TimeRange.rangearg(ARGV[0])
end
```

考えられるので、〝2 カ月ほど前〟とか〝3〜4 年前〟のようにおおよその時期を指定できるほうが便利です。たとえば、1999 年 12 月頃の 1 カ月を〝R1T364〟というキーワードで表現することにします。そして、1999 年 12 月に作成されたファイルにはこのキーワードが 4 つ含まれ、1999 年 11 月や 2000 年 1 月に作成されたファイルには 2 つ含まれるように頻度ファイルを作成しておけば、このキーワードと全文検索システムを利用して期間を指定した検索がおこなえるようになります。

期間を表現するキーワードは、普通のキーワードと重ならなければどんなものでもかまいません。ここでは表現したい期間の長さを月数で表現したもの (R1) と、1970 年 1 月 1 日から数えた時間を期間の長さで割った数字 (T364) を組み合わせたものにしています。

この方式では、2002 年頃の 1 年は〝R12T32〟と表現されます。したがって、2003 年 3 月に約 1 年前のファイルを検索したい場合は、通常のキーワードにこの〝R12T32〟というキーワードを加えて全文検索を実行すればよいことになります。

このようなキーワードの処理をおこなうプログラムを図 8 に示します。頻度ファイルを生成するときは、その生成時刻をもとに、rangefreq メソッドを使用して期間キーワードとその重みを計算して加えます。たとえば 2003 年 1 月 1 日に対して rangefreq を計算すると、図 9 のように期間キーワードとその重みを得ることができます。

通常の単語の出現頻度情報にこれを加えた頻度ファイルをもとにして WAM を作成します。

期間による検索をおこなうときは、コマンドラインで指定した引数から期間キーワードを生成します。

検索コマンドでは、表 1 の引数を用いて時期を指定します。

引数 –2–3m によって〝2〜3 カ月前〟を指定した場合、

図 9　期間を表すキーワード

```
1 R1T399
2 R1T400
4 R1T401
2 R1T402
1 R1T403
1 R2T198
2 R2T199
  ……
2 R24T17
1 R24T18
1 R48T6
2 R48T7
4 R48T8
2 R48T9
1 R48T10
```

表 1　時期を指定するための引数

引数	意味
−1m	1 カ月前
−2−3m	2〜3 カ月前
−3−4y	3〜4 年前

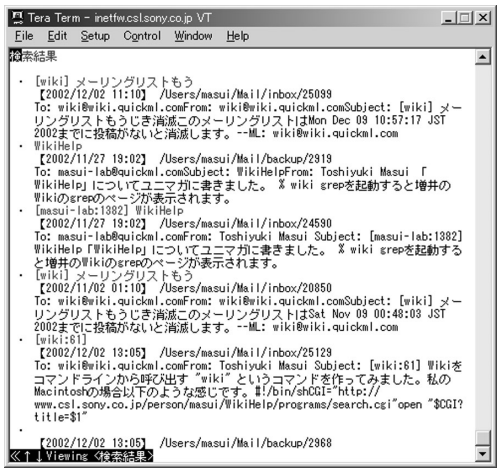
図 10　期間指定が可能な検索プログラム tsearch

```
#!/usr/bin/env ruby
# -*- mode: ruby -*-

irdir = File.dirname($0)
$: << irdir

require 'timerange.rb'

SEARCH = "#{irdir}/search"
SEARCH2HTML = "#{irdir}/search2html"
W3M = "w3m -T text/html"

p = IO.popen("#{SEARCH} | #{SEARCH2HTML} ⇒
    | #{W3M}","w")

if ARGV[0] =~ /^-(-(-)?[my])$/ then
  ARGV.shift
  time_keyword = TimeRange.rangearg($1)
  p.puts time_keyword if time_keyword
end

if ARGV.length > 0 then
  ARGV.each { |arg|
    p.puts arg
  }
else
  p.puts STDIN.readlines
end

p.close
```

2 カ月の期間を表現する期間キーワードのうち現在時刻から 2〜3 カ月前にもっとも近いものを使用します。この計算は、rangearg メソッドでおこないます。

たとえば、2003 年 3 月 1 日の時点で引数−2−3m を用いて〝2〜3 カ月前〟を指定すると、期間キーワードとして〝R2T194〟が得られます。そこで、通常の検索キーワードにこの期間キーワードを加えて全文検索をおこなえば、2003 年 3 月 1 日から数えて 2〜3 カ月前のファイルが重点的に検索されます。

図 11　tsearch −2m wiki による検索結果



期間の指定

search2html と期間キーワードを用いて、期間指定可能な検索コマンド tsearch（図 10）を使うことができます。

このコマンドに〝2 カ月前〟を意味する−2m オプションを付け、

```
tsearch -2m wiki
```

のようにして図 5 と同じ検索を実行すると、図 7 とは異なる結果が得られます（図 11）。

おわりに

Google などの全文検索システムは多くの人が毎日のように使っていますし、その重要性はひろく認められています。しかし、計算機上での通常の作業で全文検索を活用している人はあまり多くなさそうです。自分のファイルやメールに対しては、検索システムを活用するよりも、整理手法を充実させるほうが容易で効果的だからかもしれま

せん。事実、全文検索システム Namazu の開発者である高林 哲氏は、自分のファイルに対して Namazu を使うことはなく、普通のファイル整理手法を利用しているそうです。

一般的なファイル整理の手法では、適切な名前をもつ木構造のディレクトリを作成し、cd や ls などのコマンドで階層的に移動します。インデックスの作成処理や全文検索システムの起動は、cd や ls コマンドなどの操作にくらべるとひどく重いのが普通です。したがって、現在のところは手作業による整理のほうが効率的なのは否めません。しかし、全文検索がもっと手軽に使えるようになれば、整理のための手法も変化してくるのではないでしょうか。

今回紹介した検索システムのおかげで、私は cd や ls を実行する機会がすこし減ったように思います。とくに、昔作った資料を探したり、以前にもらったメールに返事を出したりする場合、ファイルやメールを一覧表示してそのなかから目的のものを選ぶ作業よりも、キーワードを指定して全文検索を実行したほうが早いことが多いようです。そんなこともあって、ちょっと前に受け取ったメールへの返信には、全文検索システムを使う機会が増えてきました。

全文検索アルゴリズム自体は、十分に実用的な域に達しています。それを有効に活用するためのこなれたインターフェイスについて、もうすこし考えてみる必要がありそうです。

今回紹介したシステムは、私の Web ページ[5]で公開していますので、ぜひご利用ください。

（ますい・としゆき　ソニー CSL）

5 http://www.csl.sony.co.jp/person/masui/GETA/