インターフェイスの街角 (23)
シグネチャ法によるお手軽検索システム

増井俊之

高機能なサーチエンジンのおかげで Web ページは比較的簡単に検索できるようになりましたが、受け取ったメールなどの検索には困っている人も多いのではないでしょうか。私もその 1 人ですが、状況を改善するために個人でも気軽に使える検索システムを作ってみました。

全文検索システム

UNIX 上でファイルを探すには、find コマンドでファイル名を検索したり、grep コマンドで内容を検索するのが一般的です。ファイルがどこにあるかが分かっていれば、たいていはこれで間に合いますが、〝どこかにあったはずだが ……〟といったあやふやな記憶しかない場合は探しだすのにかなりの時間がかかります[1]。そのようなときは、Web ページの検索と同様、すべてのファイルに対するキーワードを用いた**全文検索**が有効です。

ファイルの全文検索をおこなう商用システムも市販されていますが、ひろく使われているものはあまりみかけません。フリーのシステムとしては、Namazu[2]が Web サーバーなどでよく利用されているようです。解説によれば、Namazu は〝手軽に使える〟ことを目指しているそうですが、検索速度も重視しているため、〝手軽さ〟はやや犠牲になっているきらいがあります。

Namazu をはじめとするほとんどの全文検索システムでは、高速検索のために単語ごとのインデックスを作成します。たとえば、〝単語〟という単語がどのファイルに含まれているかという情報をインデックス・ファイルに格納します。この方式はきわめて高速な検索が可能になる反面、文書に含まれるあらゆる単語について情報を保持しておかねばならないので、巨大なインデックス・ファイルが必要になります。さらに、分かち書きされていない一般的な日本語の文書を単語に分割するために**形態素解析**プログラムを利用します[3]。そのため、処理に時間がかかり、全体のシステムも複雑になりがちです。システムやインデックス・ファイルが複雑になると扱いにくくなり、ちょっとしたカスタマイズなどもおいそれとはできません。

インデックス・ファイルを用いた検索手法の利点は、なんといっても検索の高速性でしょう。この種の方式は、Web のサーチエンジンのように高速性を優先し、インデックスの作成にかかる手間や時間はさほど問題にしなくてもよい場合に適しているように思います。一方、個人のファイルは、検索回数はそれほど多くないのに対し、更新や移動は頻繁におこなわれます。インデックス・ファイルのサイズも小さければ小さいほどよく、更新の手間も少ないほうが楽です。

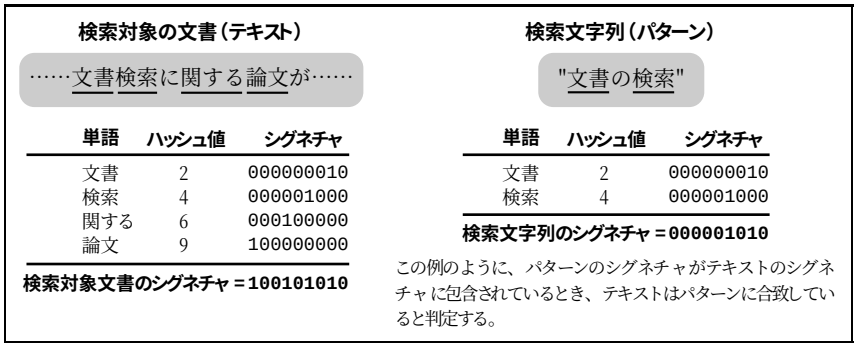
今回は、個人のファイルを対象とした手軽に使える検索システムを紹介します。ごく簡単なインデックスを使っているため検索はあまり速くありませんが、一般的な用途には十分ですし、カスタマイズも容易です。

シグネチャ・ファイルによる検索

文書の全文検索は、grep のようにまったくインデックスを使わない方法と、Namazu や Web のサーチエンジンのようにあらかじめ完全なインデックスを作成する方法の 2 種類に大別できます。前者は、インデックスを作成

1 歳のせいか、最近はこういうことがひどく多くなったような気が ……。
2 http://openlab.ring.gr.jp/namazu/
3 たとえば Namazu では、日本語形態素解析システム「茶筌」などを用いてキーワードの切出しをおこなっています。

図 1　シグネチャ法による検索の仕組み



する必要がなく、インデックスのためのディスク領域も不要ですが、検索に時間がかかってしまいます。後者では、高速な検索が可能ですが、インデックスを作成する手間やある程度のディスク領域が必要になります。

これらを折衷したものとして、〝小さなインデックス・ファイルを用いて検索の一助とする〟方法が考えられます。インデックスである程度絞り込んでから grep などを利用すれば、巨大なインデックス・ファイルを用意しなくても、比較的効率のよい検索が可能になります。UNIX 上で個人のファイルを検索する場合、一般には寸秒を争うほどの高速性は必要としないので、このような中間的手法が向いているのではないでしょうか。

シグネチャ・ファイル

中間的なインデックスを使う方法の 1 つに、**シグネチャ・ファイル**という簡易インデックスを利用する**シグネチャ法** [2] があります。これは、検索対象の文書（以下〝テキスト〟と表記）および検索文字列（以下〝パターン〟と表記）の〝特徴〟をビット列で表現した〝シグネチャ〟を用いて必要なテキストを高速に抽出する手法です。

シグネチャは、テキストまたはパターンに含まれる単語から計算されます。テキストまたはパターンの各単語についてハッシュ値を計算し、その値で示されるビット位置だけを〝1〟としたビット列をその単語のシグネチャとします。すべての単語のシグネチャの論理和がテキストまたはパターンのシグネチャとなります。すべてのテキストについてシグネチャを計算した結果をあらかじめシグネチャ・ファイルとして用意しておき、検索の前処理段階においてパターンのシグネチャがテキストのシグネチャに完全に包含されるとき、テキストにパターンが含まれる可能性があると判断します（図 1）。

この段階で一致する部分がないと判定されたテキストには、パターン中の単語は含まれていません。しかし、同じシグネチャをもつ単語は複数存在する可能性があるので、一致する部分があると判定されたテキストがかならずしもパターンを含むとはかぎりません。そこで、テキストが本当にパターンを含んでいるかを判定する処理がさらに必要になります。

シグネチャ法の特徴

シグネチャ法には、以下のような特徴があります。

- **検索が比較的高速**
  テキストとパターンのシグネチャの照合は単純な論理計算であり、高速に実行できます。さらに、シグネチャの計算法をうまく選べば、不要なテキストの大部分をシグネチャによる判定でふるい落とせるため、すべての処理を含めて全体的に高速な検索が可能になります。
- **シグネチャ・ファイルの作成/更新が高速**
  テキストの内容が変化したり追加された場合、シグネチャ・ファイルも更新する必要があります。しかし、シグネチャ・ファイルの構造は単純なので、更新されたファイルのシグネチャを入れ替えたり、新しいシグネチャを追加したりする処理も容易です。すべての単語にインデックスをつける方式では、検索を高速化するためにトライ（trie）や B-tree といった特殊な木構造を使用するのが一般的ですが[4]、これらの構造を利用すると追加や削除に時間がかかります。

4 たとえば、Web のサーチエンジン goo (http://www.goo.ne.jp/) では、キーワード辞書にトライが使われています。

●よぶんに必要なファイルが単純で小さい

シグネチャ法による検索でよぶんに必要なのはシグネチャ・ファイルだけです。各テキストのシグネチャはファイル本体にくらべてかなり小さいので、全単語のインデックスを用意する方法と比較するとディスク領域をそれほど必要としません。

このように、シグネチャ法は grep などを用いた検索とインデックスを利用する方法の中間的な特徴をもっており、個人や小規模なオフィスにおける情報検索に適しています。

検索システムの実装

シグネチャ・ファイルは、以下の手順で作成します。

1. 検索対象のファイル一覧を作る。
2. 各ファイルの重要部分を抽出し、漢字コードを統一する。
3. 重要部分のシグネチャを計算してシグネチャ・ファイルに加える。

以下では、これらの処理を順番に説明します。

検索対象のファイル一覧作成

まず、*.o や*.dvi など、明らかにテキスト検索の対象にはならないファイルを除外するため、検索が必要なファイルの一覧を生成する **listfiles** コマンドを作ります（リスト 1）。

ここでは検索するディレクトリ名やパターンがプログラムに埋め込まれていますが、環境によってこれらを変更したり引数で指定できるようにするとよいでしょう。

ファイルの重要部分の抽出

listfiles で生成された各ファイルについて重要な部分を抽出し、コード変換をおこないます（リスト 2）。どの部分が重要かはファイルの内容によって異なるとは思いますが、ここでは機械的に先頭から 1,000 バイトを抽出し、検索処理のために文字コードを EUC に変換しています。

メールやニュース記事の場合はヘッダを除いたり、TEX ファイルなら ``\begin{...}''などのコマンド文字列を取り除く処理をすればよいでしょう。

リスト 1　listfiles（検索対象ファイル一覧を作成）

```
#!/usr/local/bin/perl
require 'find.pl';

# 検索対象のディレクトリ
@dir = (
  '/user/masui/DOC/meibo/',
);

# 検索が必要なファイル名の種別
sub valid {
  local($_) = @_;
  /\.tex$/ || /\.html$/ ;
}

# 検索不要なファイル名の種別
sub invalid {
  local($_) = @_;
  /\.o$/ || /\.dvi$/ ||
# .....
  /~$/ || /\/RCS\// ;
}

sub wanted { # &findで呼ばれる
  # $name = $dir/$_
  # シンボリック・リンクなどは除く
  return if -l $name || ! -f $name;
  # ファイル名をもとに要/不要を判別
  return if ! &valid($name) &
            &invalid($name);
  print "$name\n";
}

&find(@dir);
```

シグネチャの計算

toptext() 関数で抽出されたテキストに対し、ハッシュを計算してシグネチャを求めます（リスト 3〜4）。効率のよいシグネチャを作るにはよいハッシュ関数を使う必要がありますが、このプログラムでは文献 [1] で紹介されている hashpjw() 関数を使っています。

さきほど述べたように、Namazu などの一般的な検索システムでは、テキストをまず単語に分解してからインデックスを作成するため、形態素解析システムが必要です。このとき、誤った形態素解析により検索に失敗する可能性も皆無とはいえません。たとえば、``東京都民''という文字列を ``東京/都民''と分解してインデックスを作成すると、``東京都''では検索できなくなります。あるいは、もとの文脈では ``東/京都/民''だったかもしれません。

ここでは、形態素解析の手間を省いてシグネチャ・ファイルの作成を高速化するため、``あらゆる連続した 3 バイトを単語とみなしてシグネチャを計算する''という手法を

リスト 2　toptext.c（テキスト先頭の重要部分を切り出す）

```
#include <stdio.h>
#include <ctype.h>
#include "toptext.h"

#define ESC 0x1b
#define ishigh(c) ((c) >= 0x80)

typedef enum { BIN, ASCII, JIS, EUC, SJIS } ⇒
        Code;

// 文字コードの判定
static Code checkcode(unsigned char *s)
{
  Code code = ASCII;
  enum {LOW, HIGH} lh = LOW;
  unsigned char c;

  for(;*s;s++){
    c = *s;
    if(lh == LOW){
      if(c == ESC){ code = JIS; }
      else if(isspace(c) || isprint(c)){ }
      else if(c >= 0x80){
        if(code == JIS) return BIN;
        lh = HIGH;
      }
      else { return BIN; }
    }
    else { // コードにより SJIS/EUC 判定
      if(c >= 0x20 && c <= 0xa0){ code = SJIS; }
      else if(c >= 0xa1 && c <= 0xfe){
        if(code != SJIS) code = EUC;
      }
      else { return BIN; }
      lh = LOW;
    }
  }
  return code;
}

typedef enum { SSTART, SESC, SESC2, SHIGH } ⇒
        State;

// 先頭から 1,000 バイトを EUC に変換して抽出
unsigned char *toptext(unsigned char *s)
{
  State state = SSTART;
  static unsigned char buf[2000];
  unsigned char *p = buf;
  unsigned char c,c1,c2;
  int jis = 0;
  Code code;
  code = checkcode(s);
  if(code == BIN) return "";

  for(;p-buf<1000 && *s;s++){
    c = *s;
    if(!jis && isupper(c)) c = tolower(c);
    switch(state){
    case SSTART:
      if(c == ESC){ state = SESC; }
      else if(isspace(c)){ }
      else if(c >= 0x80){
        c1 = c;
        state = SHIGH;
      }
      else { // JIS → EUC 変換
        if(jis) *p++ = (c | 0x80);
        else    *p++ = c;
      }
      break;
    case SESC:
      if(c == '$'){ jis = 1; state = SESC2; }
      else if(c == '('){ jis = 0; ⇒
                         state = SESC2; }
      else { state = SSTART; }
      break;
    case SESC2:
      state = SSTART;
      break;
    case SHIGH:
      c2 = c;
      if(code == SJIS){ // SJIS → EUC 変換
        if(c1 >= 0xe0) c1 -= 0x40;
        if(c2 >= 0x80) c2--;
        c1 = (c1-0x81) * 2 + ⇒
             (c2>=0x9e ? 1 : 0) + 0xa1;
        c2 = (c2 >= 0x9e ? c2-0x9e : ⇒
             c2-0x40) + 0xa1;
      }
      *p++ = c1;
      *p++ = c2;
      state = SSTART;
      break;
    default:
      break;
    }
  }
  *p = '\0';
  return buf;
}
```

（誌面の都合上、⇒ で折り返しています。以下同様）

とることにします。たとえば ``data''という文字列であれば、``dat''と ``ata''という単語から成り立っていると考えてシグネチャを計算します。

シグネチャはビット列として表現されるため、シグネチャ・ファイルをバイナリ形式にすることも考えられます。しかし、uuencode や MIME のようにシグネチャを ASCII 文字列で表現すればシグネチャ・ファイル全体をテキストファイルにできるので、各ファイルのシグネチャの確認や、エントリの追加/変更が容易になります。

そこで、今回はテキスト形式のシグネチャ・ファイルを使うことにしました。たとえば、ファイル/foo/bar のシグネチャが ``000100100001''である場合、これを 6 ビ

リスト 3　signature.h

```
#ifndef _SIGNATURE_H_
#define _SIGNATURE_H_

#define SIGBYTES 256
#define SIGBITS (SIGBYTES*6)

void calcsig(unsigned char *s, char *sig);
int sigmatch(char *textsig, char *patsig);

#endif
```

リスト 4　signature.c (シグネチャの計算)

```
#include "signature.h"

// ハッシュ関数
static int hashpjw(unsigned char *s, int len)
{
  unsigned h=0, g;
  for(;*s && (len-- > 0);s++){
    h = (h << 4) + (*s);
    if(g = h & 0xf0000000){
      h = h ^ (g >> 24);
      h = h ^ g;
    }
  }
  return h;
}

// シグネチャを計算してsigに格納
void calcsig(unsigned char *s, char *sig)
{
  int i,v;
  for(i=0;i<SIGBYTES;i++) sig[i] = 0x40;
  sig[SIGBYTES] = 0;
  for(;*s && *(s+1) && *(s+2);s++){
    v = hashpjw(s,3) % SIGBITS;
    sig[v/6] |= (1 << v%6);
  }
  for(i=0;i<SIGBYTES;i++)
    if(sig[i] == 0x7f) sig[i] = 0x3f;
}

// シグネチャの比較
int sigmatch(char *textsig, char *patsig)
{
  int i;
  char c;
  for(i=0;i<SIGBYTES;i++){
    c = patsig[i] & 0x3f;
    if((textsig[i] & c) != c) return 0;
  }
  return 1;
}
```

ットずつに分割してから MSB に ``01''または ``00''を追加し、``01000100''(ASCII の ``D'')、``01100001''(ASCII の ``a'')とすれば、``Da''という文字列で表現できます。そして、ファイル名とシグネチャ文字列を 1 行にまとめて、

```
/foo/bar Da
```

という行を/foo/bar に対応するシグネチャ・ファイルのエントリとします。

シグネチャのサイズは大きいほうが効率的に検索できますが、シグネチャ・ファイルも大きくなってしまうので適当なサイズに抑えます。ここでは変数 SIGBYTES を 256 としておきます(リスト 3)。この場合、シグネチャのサイズは $256 \times 6 = 1,536$ ビットとなります。

リスト 5　makesig.c (シグネチャ・ファイルの生成)

```
#include <stdio.h>
#include "signature.h"
#include "toptext.h"

main(int argc, char **argv)
{
  char buf[5000+1],fname[1000];
  char sig[SIGBYTES+1];
  int n;
  FILE *f;

  while(fgets(buf,5000,stdin)){
    sscanf(buf,"%s",fname);
    if(f = fopen(fname,"r")){
      n = fread(buf,1,5000,f);
      buf[n] = '\0';
      calcsig(toptext(buf),sig);
      printf("%s\t%s\n",fname,sig);
      fclose(f);
    }
  }
}
```

シグネチャ・ファイルの作成

listfiles を実行して得たファイルについてそれぞれシグネチャを計算し、それらをまとめてシグネチャ・ファイルを生成します(リスト 5)。

リスト 2〜5 のプログラムをコンパイルし、**makesig** コマンドを作ります。

```
% gcc -g -c makesig.c
% gcc -g -c signature.c
% gcc -g -c toptext.c
% gcc -g makesig.o signature.o toptext.o \
-o makesig
```

シグネチャ・ファイルは、次のようにして生成します。

```
% listfiles | makesig > sigfile
```

図 2　シグネチャ・ファイルのエントリの例

```
/user/masui/DOC/meibo/m/a/Masui.Toshiyuki  zN@DD\hD[DVNL‘Hht@@KDtsAh@w^iT@xZ\aH@PMIQFv?_nBi‘DAnhb
i@QBRNepYPhZhp@ZyH@@D@TZd@fhD@D@f@Pp@|Ri{@jHBD‘YzePDbEFDp‘zHBj~baCb‘@L@lBdW[CLFAS@WR‘Nax‘hN@V\h@
CHd]RhBTctAPQH‘TaPAAPOrB‘YZHcZG~S@BjvPejeAQZHNsfKmdAgOSHA@a@FnrNkHy@bRRJIDh@@RdDvAHFpI@@PphHhEB@
‘E‘HH@@hAY
```

リスト 6　sigfind.c（シグネチャ・ファイルを使った検索実行）

```
#include <stdio.h>
#include <stdlib.h>
#include "signature.h"
#include "toptext.h"

void err(s) { fprintf(stderr,"%s\n",s); ⇒
              exit(0); }

main(int argc, char **argv)
{
  FILE *sigfile,*f;
  char *pat;
  char buf[5000+1];
  char patsig[SIGBYTES+1], ⇒
       textsig[SIGBYTES+1];
  char filename[1000];
  int n;

  if(argc < 3) err("% sigfind pat sigfile");

  pat = argv[1];
  if(strlen(pat) < 3)
    err("Pattern should be longer ⇒
         than 2 bytes");

  sigfile = fopen(argv[2],"r");
  if(sigfile == NULL)
    err("Cannot open signature file");

  calcsig(pat,patsig);

  while(fgets(buf,5000,sigfile)){
    sscanf(buf,"%s\t%s",filename,textsig);
    if(sigmatch(textsig,patsig)){
      if(f = fopen(filename,"r")){
        n = fread(buf,1,5000,f);
        buf[n] = '\0';
        if(strstr(toptext(buf),pat))
          printf("%s\n",filename);
        fclose(f);
      }
    }
  }
}
```

シグネチャ・ファイルを利用した検索の実行

sigmatch() 関数を用いて、検索対象ファイルから生成されたシグネチャ・ファイルとパターンから生成されたシグネチャとを比較し、第 1 段階の検索をおこないます。これに成功したファイルについて、パターンが含まれているか否かを **sigfind** コマンド（リスト 6）で調べれば、最終的な検索結果が得られます。

sigfind は、次のようにして生成します。

```
% gcc -g -c sigfind.c
% gcc -g sigfind.o signature.o toptext.o \
-o sigfind
```

実行結果は、たとえば以下のようになります。

```
% sigfind 'ソニー' sigfile
/user/masui/DOC/meibo/k/i/Kitano.Hiroaki
/user/masui/DOC/meibo/m/a/Masui.Toshiyuki
/user/masui/DOC/meibo/t/o/Tokoro.Mario
  ……
%
```

検索の手軽さ

今回の検索プログラムは全部で 250 行程度と小さく、特殊なツールも使っていないため、インデックスの作成や検索の実行もかなり手軽におこなえると思います。検索速度は Namazu などとくらべるとはるかに低速ですが[5]、ときどき使う程度であれば十分ではないでしょうか。

makesig で生成したシグネチャ・ファイルはテキスト形式なので、どのようなファイルがインデックスに含まれているのかがすぐに分かります。不要なエントリを削除したり、新しいエントリを追加するのも簡単です。たとえば、例に使った名簿ファイル用ディレクトリの私のエントリに関するシグネチャは、図 2 のような行で表現されています。これなら、grep などで取り出したり、不要なエントリを取り除くのも簡単でしょう。

エントリを追加するときは、

```
% find / -mtime -1 -print | makesig
```

などとして新しいファイルに対するシグネチャだけを計算し、既存のシグネチャ・ファイルに含めることができます。あるいは、ファイルを保存するたびに、そのシグネチャをシグネチャ・ファイルに追加するようにしてもいいでしょ

5 私の環境では、合計 100MB 程度のファイルの検索に約 10 秒かかります。

う。こういった点においても、このシステムはかなり手軽に使えるのではないでしょうか。

検索システムの拡張

今回はシグネチャを用いた検索システムの基本部分だけを紹介しましたが、構造が単純なので以下のような拡張も簡単です。

- 新しい順にファイルを並べ替える
  更新時刻順にシグネチャ・ファイルをソートすれば、新しいファイルがさきにみつかるようになります。
- メールなどのメッセージへの対応
  メールのヘッダと本文は簡単に区別できますから、toptext.c に本文だけを取り出す処理を加えれば、よぶんなヘッダの検索に時間をとられずにすみます。
- AND 検索、OR 検索
  sigfind.c にすこし手を加えれば、複数の引数を用いた AND 検索や OR 検索が可能になります。

このような単純なシステムでは、オプションをたくさん用意して汎用性を高めるよりも、状況に応じてプログラムを書き換えるほうが効率的かもしれません。たとえば、ヘッダを取り除くメール専用の makesig を作り、メールを保存してあるディレクトリに置くといった方法も考えられます。

検索システムの``超''活用

UNIX などで多数の文書を管理する場合、通常は種類に応じて適当な名前を付けたディレクトリやフォルダにファイルを置いているのではないでしょうか。私も、名簿や論文、手紙、FAX などのファイルを異なるディレクトリに保存し、さらに種類や差出人に応じてサブディレクトリを作って分類しています。これは、目的とするファイルを簡単にみつけるためで、たとえば FAX 文書なら FAX 用のディレクトリを、名簿なら名簿用のディレクトリを調べればすみます。さらに、内容を推測しやすいファイル名にしているので、該当するディレクトリで ls や find を使ってファイルを探したり、grep で中身を調べたりすることもできます。

現在のところはこれが一般的な文書管理手法だと思いますが、優れた検索システムがいつでも使えるのなら、こういった分類に費やす手間は無駄なような気もしてきました。求めるファイルがみつかりさえすれば、苦労してディレクトリ構成やファイル名を考えて整理する必要はありません。どこに分類すればいいのだろうと悩むこともなくなるでしょう。適切なディレクトリに正しい名前のファイルを作り続けるのはかなりの負担ですから、ファイル管理の考え方を根本的に改めるほうがいいのかもしれません。

プログラムのなかでは、ファイル名やディレクトリ名を使う必要があるかもしれません。しかし、これも考えを変えて、fopen() などの代わりに検索/ファイルオープン用の findopen() といった関数を用意すればいいでしょう。たとえば、名簿に情報を追加するには、

```
f = findopen("名簿 増井俊之");
fprintf(f,"Phone: 123-4567");
fclose(f);
```

のようなプログラムを使ったり、シェルでは、

```
% cat ‘sigfind ’名簿 増井俊之’‘
```

などとして、ファイル名の代わりに検索コマンドだけを使う方法も考えられます。

これは、ファイルシステムをデータベースと同じような感覚で利用していることになります。このような考え方は以前からありましたが、いままでひろく使われたことはないようです。UNIX などの階層型ファイルシステムに対する検索機能を強化すれば、一般的なファイルシステムとデータベース管理システムの両方の長所をあわせもつシステムが構築できるかもしれません。

（ますい・としゆき　ソニー CSL）

[参考文献]

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley, 1986

[2] C. Faloutsos and S. Christodoulakis, “Description and performance analysis of signature file methods for office filing”, *ACM Transactions on Office Information Systems*, Vol. 5, No. 3, pp.237–257, July 1987

[3] 増井俊之「シグナチャと曖昧検索を用いた文書検索システム」、第 18 回 jus UNIX シンポジウム論文集、pp.9–16、日本 UNIX ユーザ会、1991 年 11 月