Algorithm 444

An Algorithm for Extracting Phrases in a Space-Optimal Fashion [Z]

R.A. Wagner [Recd. 5 Mar. 1971 and 30 Aug. 1971]
Department of Systems and Information Science, Vanderbilt University, Nashville, TN 37203

Key Words and Phrases: information retrieval, coding, text compression
CR Categories: 3.70, 5.6
Language: PL/I

Description

Introduction. The algorithm *PARSE* computes and prints a minimum-space form of a textual message, *MS*. The minimization is performed over all possible "parses" of *MS* into sequences of phrase references and character strings. Each phrase reference represents one of a finite collection, *P*, of phrases. The collection, *P*, must be selected before *PARSE* is applied.

Assumptions and requirements. PARSE assumes that the unit of storage is the byte, defined such that one byte can hold either a single character of text or an integer i in the range $0 \leq i < W$. (For IBM 360 equipment, $W = 256 = 2**8$). *PARSE* also assumes that the number of different phrases in the collection *P* is no larger than *W**PHC*, and that each message to be parsed contains fewer than *W**CHC* characters of text. The parameter values $CHC = PHC = 1$ appear appropriate on IBM 360 equipment, when *PARSE* is applied to short messages, such as compiler error messages.

PARSE requires two arguments. The first is the message to be parsed; the second is the table of common phrases which may be used in the parse.

PARSE assumes that an external procedure *HASH* is present; $HASH(MS,I,K)$ is defined as follows: Let $H_1, H_2, \ldots, H_m$ be a sequence of indices such that among them they exhaust all entries $P(H_i)$ such that

$$SUBSTR(MS,I,3) = SUBSTR(P(H_i),1,3).$$

(That is, the H_i's include indices for every phrase $P(H_i)$ which agrees with characters I, $I + 1$, and $I + 2$ of the given message. Other indices may occur among the H_i's, as well.) Then $HASH(MS,I,0) = H_1$, $HASH(MS,I,H_j) = H_{j+1}$, and $HASH(MS,I,H_m) = 0$.

A "hash table" procedure can easily be modified to yield this performance; an equally useable, although slower version returns $MOD(K + 1, M + 1)$ on every call. A procedure *HASH* is included below.

Methods. The method used to determine which phrases to extract from the given message is described in [1]. The resulting parsed message requires least space, assuming that messages are storable only as described in [1]—that is, as sequences of

C ⟨number⟩ ⟨character string⟩
| P ⟨number⟩

representing a literal string of characters, and a reference to a common phrase, respectively.

During the course of the computation, arrays G and H are filled with values of functions g and h, respectively, as defined in [1]. Just before label *BUILD* is reached,

$H(I)$ = length of the best parse of $SUBSTR(MS,I)$, and
$G(I)$ = length of the best parse of $SUBSTR(MS,I)$ among those parses beginning with a character string,

both for $I = 1, \ldots, LENGTH(MS)$.

Internally, *PARSE* uses a single array, Z, paralleling the function arrays G and H, to retain the information needed for re-constructing the parsed form of the message.

$Z(I) = K$, if $G(I) > H(I)$, where K is the number of the "best" common phrase matching *MS* at I, or
$= J$, if $G(I) = H(I)$. ($G(I) < H(I)$ is impossible.)

J gives the index of the end (plus one) of the character string starting at I. In this case, the best parse at I begins with this character string. J satisfies: $G(J) > H(J)$ and for all k, $I \leq k < J$, $G(k) = H(k)$.

Results: To make the printed form of the parsed message more intelligible, *PARSE* prints:
'*C* ⟨number⟩' as '*#ddd*'
'*P* ⟨number⟩' as '*%ddd*'
where "*ddd*" is the 3-digit decimal representation of ⟨number⟩ + 1. In practice, a number representing a character count or phrase index can be stored as an integer, in place of *CHC* or *PHC* characters respectively. Thus, the character string '*ABC*' would be stored as '*C2ABC*', where $\underline{2}$ is a *CHC*-byte integer whose value is 2. The same string would be printed by the *PARSE* algorithm as '#003*ABC*'.

The program *PARSE* returns the number of bytes needed to store *MS*, given the particular set of extractable phrases in *P*.

A sample driver, two sample input streams and associated output follow the procedures *PARSE* and *HASH*.

References

1. Wagner, R.A. Common phrases and minimum-space text storage. *Comm. ACM 16*, 3 (Mar. 1973), 148–152.
2. Bell, James R. The quadratic quotient method; a hash code eliminating secondary clustering. *Comm. ACM 13*, 2 (Feb. 1970), 107–109.

Algorithm (Figures 1–6 follow.)

Fig. 1. The PARSE Algorithm.

```
PARSE:      PROC(MS,P) RETURNS(FIXED BINARY);
            DCL  (MS,P(*)) CHAR(*) VARYING;
            DCL  N;
            DCL  HASH RETURNS(FIXED BINARY);
            DCL (CHC,   /* BYTES PER CHARACTER-COUNT  */
                 PHC)   /* BYTES PER PHRASE-INDEX  */
                 STATIC EXTERNAL FIXED BINARY;

            N=LENGTH(MS);
            BEGIN;
                 DCL(G,H,Z)(N+1) FIXED BINARY;
                 DCL(I,J,K,L,T)  FIXED BINARY;

                 G(N+1)=3; H(N+1)=1; J,Z(N+1)=N+1;
            MSGP:     DO I=N BY -1 TO 1;
                      K=HASH(MS,I,OB);
                      H(I), G(I) = MIN( G(I+1)+1, H(I+1)+CHC+2 );
                      Z(I)=J;
                 /* J HOLDS INDEX OF END+1 OF NEXT CHAR-STRING */
                 M1:  DO WHILE (K>0);
                           L=LENGTH(P(K));
                           IF L ¬> N-I+1 THEN
                           IF L < N THEN
                           IF SUBSTR(MS,I,L)=P(K) THEN DO;
                                T=H(I+L)+PHC+1;
                                IF H(I)>T THEN DO;
                                     H(I)=T; Z(I)=K; J=I;
                                     END;
                                END;
                           K=HASH(MS,I,K);
                           END M1;
                      END MSGP;

                 PUT SKIP EDIT(H(1),N+3,':  ')(2 F(4),A);
                 I=1; GOTO B1;
            BUILD:
                 IF H(I)<G(I) THEN DO;
                      PUT EDIT('%', Z(I))(A,P'999');
                      I=I+LENGTH(P( Z(I)));
                      END;
                 ELSE DO;
                      J=Z(I)-I;
                      PUT EDIT('#',J,SUBSTR(MS,I,J))(A,P'999',A);
                      I=Z(I);
                      END;
            B1:  IF I¬>N THEN GOTO BUILD;
                 PUT EDIT('.')(A);
                 RETURN(H(1));
            END PARSE;
```

Fig. 2. An acceptable HASH procedure.

```
HASH:       PROC(MS,I,K) RETURNS(FIXED BINARY);
     DCL    MS CHAR(*), J FIXED BINARY(31,0),
            (HT (0:200)INIT((201)0),
                  KJ, HP INIT(197),
                  HX,HY,HZ) FIXED BINARY STATIC;
            DCL (CHC,    /* BYTES PER CHARACTER-COUNT   */
                 PHC)    /* BYTES PER PHRASE-INDEX  */
                 STATIC EXTERNAL FIXED BINARY;

            CALL HCMN(K);
            RETURN(HT(HZ));

     HCMN:       PROC(K);
            IF K = 0 THEN
                 IF LENGTH(MS)-I < PHC+1 THEN HZ=-1;
                 ELSE DO;
                      UNSPEC(J)=UNSPEC(SUBSTR(MS,I,PHC+2));
                      HZ=MOD(J,HP);
                      HY=J/HP;
                      HX=0;
                      END;
            ELSE DO;
                 HX=MOD(HX+HY,HP);
                 HZ=MOD(HX+HZ,HP);
                 END;
            HZ=HZ+1;
            RETURN;
            END HCMN;

ENTER:      ENTRY(MS,I,K);
            IF LENGTH(MS) < PHC+2 THEN RETURN;
            KJ=0;
     E1:    CALL HCMN(KJ);
            KJ=HT(HZ);
            IF KJ > 0 THEN GOTO E1;
            HT(HZ)=K;
            RETURN;
            END HASH;
```

Fig. 3. A driver for the PARSE procedure.

```
DRIVER:     PROC OPTIONS(MAIN);
            DCL  MS CHAR(256) VARYING;
            DCL NP,M;
            DCL (HASH RETURNS(FIXED BINARY), ENTER)
                 ENTRY(CHAR(256) VARYING, FIXED BINARY, FIXED BINARY);
            DCL PARSE RETURNS(FIXED BINARY);
            DCL (CHC,  /* BYTES PER CHARACTER-COUNT  */
                 PHC)  /* BYTES PER PHRASE-INDEX   */
                 STATIC EXTERNAL FIXED BINARY;

            CHC,PHC=1;    /* COUNT/INDEX SIZE=1 BYTE  */
            GET SKIP LIST(NP,M);
            BEGIN;
                 DCL P(NP) CHAR(M) VARYING;
                 DCL NB,NA,I,J;

                 NB,NA=0;
                 DO I=1 TO NP;
                 GET SKIP LIST(P(I));
                 CALL ENTER(P(I),1,I);
                 END;

                 PUT PAGE LIST('PHRASES, AND THEIR PARSED FORMS');
                 DO I=1 TO NP;
                      PUT SKIP(2) EDIT(I,'  ''' || P(I) || '''')
                           (F(4),A);
                      NA=NA+PARSE(P(I),P);
                      END;

                 PUT PAGE LIST('MESSAGES:');
            L1:  GET SKIP LIST(MS);
                 PUT SKIP(2) LIST( '''' || MS || '''' );
                 IF MS='' THEN GOTO L2;
                 NB=NB+LENGTH(MS)+CHC+2;
                           /* ALLOW FOR STRING-OVERHEAD + END MARK  */
                      NA=NA+PARSE(MS,P);
                      GOTO L1;

            L2:  PUT SKIP EDIT('FINAL STATISTICS:',
                 'WITHOUT PHRASE EXTRACTION:',NB,
                 'AFTER PHRASE EXTRACTION:',NA,
                 'SAVING:',NB-NA,
                 '  (',(NB-NA)*100/NB,'%)')
                      (A,3(SKIP,A,F(5)),A,F(5,1),A);
                 RETURN;
                 END DRIVER;
```

Fig. 4. Sample input files.
(a) Two phrases, four messages. Illustrates heavily overlapping phrases.
(b) Five phrases, 23 messages. These messages are the first 23 numbered error messages from the syntactic analysis section of the PL/C compiler.

A

```
CMS03 LISTING OF INPUT STREAM
 00001
 00002     2,10
 00003    'AAAAA'
 00004    'AAAAAAA'
 00005    'AAAAAAAAAA'
 00006    'AAAAAAAAAAAA'
 00007    'AAAAAAAAAAAAAA'
 00008    'AAAAAAAAAAAAAAA'
 00009    ''
```

B

```
CMS03 LISTING OF INPUT STREAM
 00001
 00002           5,20
 00003    'EXTRA '
 00004    'MISSING '
 00005           'IMPROPER '
 00006    'SEMI-COLON'
 00007    'EXPRESSION'
 00008    'EXTRA ('
 00009    'MISSING ('
 00010    'EXTRA )'
 00011    'MISSING )'
 00012    'EXTRA COMMA'
 00013    'MISSING COMMA'
 00014    'EXTRA SEMI-COLON'
 00015    'MISSING SEMI-COLON'
 00016    'MISSING :'
 00017    'MISSING ='
 00018    'IMPROPER *'
 00019    'MISSING *'
 00020    'EXTRA END'
 00021           'MISSING END'
 00022           'MISSING KEYWORD'
 00023           'INCOMPLETE EXPRESSION'
 00024           'MISSING EXPRESSION'
 00025           'MISSING VARIABLE'
 00026           'MISSING ARGUMENT, 1 SUPPLIED'
 00027           'EMPTY LIST'
 00028           'IMPROPER NOT'
 00029           'IMPROPER ELEMENT'
 00030           'UNTRANSLATABLE STATEMENT'
 00031    ''
```

Fig. 5. Result of applying DRIVER to the cards listed in Figure 4(a). Note that phrase 2 is itself reduced in size by PARSE, while each of the messages are reduced to strings of phrase references alone.

```
PHRASES, AND THEIR PARSED FORMS

   1   'AAAAA'
   8    8:  #005AAAAA.

   2   'AAAAAAA'
   7   10:  #002AA%001.


MESSAGES:

'AAAAAAAAAA'
  5  13:  %001%001.

'AAAAAAAAAAAA'
  5  15:  %001%002.

'AAAAAAAAAAAAAA'
  5  17:  %002%002.

'AAAAAAAAAAAAAAA'
  7  18:  %001%001%001.

''
FINAL STATISTICS:
WITHOUT PHRASE EXTRACTION:   63
AFTER PHRASE EXTRACTION:   37
SAVING:   26  ( 41.3%)
```

Fig. 6. Result of applying DRIVER to the cards listed in Figure 4(b).

```
PHRASES, AND THEIR PARSED FORMS

  1   'EXTRA '
  9   9:  #006EXTRA .

  2   'MISSING '
 11   11:  #008MISSING .

  3   'IMPROPER '
 12   12:  #009IMPROPER .

  4   'SEMI-COLON'
 13   13:  #010SEMI-COLON.

  5   'EXPRESSION'
 13   13:  #010EXPRESSION.

MESSAGES:

'EXTRA ('
  6  10:  %001#001(.

'MISSING ('
  6  12:  %002#001(.

'EXTRA )'
  6  10:  %001#001).

'MISSING )'
  6  12:  %002#001).

'EXTRA COMMA'
 10  14:  %001#005COMMA.

'MISSING COMMA'
 10  16:  %002#005COMMA.

'EXTRA SEMI-COLON'
  5  19:  %001%004.

'MISSING SEMI-COLON'
  5  21:  %002%004.

'MISSING :'
  6  12:  %002#001:.

'MISSING ='
  6  12:  %002#001=.

'IMPROPER *'
  6  13:  %003#001*.

'MISSING *'
  6  12:  %002#001*.

'EXTRA END'
  8  12:  %001#003END.

'MISSING END'
  8  14:  %002#003END.

'MISSING KEYWORD'
 12  18:  %002#007KEYWORD.

'INCOMPLETE EXPRESSION'
 16  24:  #011INCOMPLETE %005.

'MISSING EXPRESSION'
  5  21:  %002%005.

'MISSING VARIABLE'
 13  19:  %002#008VARIABLE.

'MISSING ARGUMENT, 1 SUPPLIED'
 25  31:  %002#020ARGUMENT, 1 SUPPLIED.

'EMPTY LIST'
 13  13:  #010EMPTY LIST.

'IMPROPER NOT'
  8  15:  %003#003NOT.

'IMPROPER ELEMENT'
 12  19:  %003#007ELEMENT.

'UNTRANSLATABLE STATEMENT'
 27  27:  #024UNTRANSLATABLE STATEMENT.

''
FINAL STATISTICS:
WITHOUT PHRASE EXTRACTION:  376
AFTER PHRASE EXTRACTION:  283
SAVING:   93  ( 24.6%)
```

Algorithm 445

Binary Pattern Reconstruction from Projections [Z]

Shi-Kuo Chang [Recd. 4 Nov. 1970 and 12 May 1971]
School of Electrical Engineering, Cornell University
Ithaca, NY 14850.

Key Words and Phrases: pattern reconstruction, image reconstruction, data compression, picture processing
CR Categories: **3.63, 5.30**
Language: Algol

Description

This procedure reconstructs a binary pattern from its horizontal and vertical projections [1]. The parameters are described as follows. m, n are the dimensions of the binary pattern f. *switch* is an integer variable. fx $[1{:}n]$ is the projection of f on the horizontal axis. fy $[1{:}m, 1]$ is initially set to $(1, 2, \ldots, m)$. fy $[1{:}m, 2]$ is the projection of f on the vertical axis. $f[1{:}n, 1{:}m]$ is the pattern to be reconstructed, initially set to 0.

The projections fx and fy are *inconsistent* if there is no pattern f having such projections. The pattern f is *unambiguous* if there is no other pattern having the same projections as f. Given the projections fx and fy, there are three possibilities: (1) fx and fy are inconsistent; (2) they are consistent but the pattern f is ambiguous; or (3) they are consistent and the pattern f is unambiguous.

(1) Inconsistent Projections. This procedure sets *switch* to -1 and reconstructs a pattern f having the correct horizontal projection fx. Its vertical projection will be different from fy.

(2) Ambiguous Pattern. This procedure sets *switch* to 0 and reconstructs a pattern f having projections fx and fy.

(3) Unambiguous Pattern. This procedure sets *switch* to 1 and reconstructs a pattern f having projections fx and fy. In this case f is unique.

References

1. Chang, S.-K. The reconstruction of binary patterns from their projections. *Comm. ACM 14*, 1 (Jan. 1971), 21–25.
2. Chang S.-K., and Shelton, G.L. Two algorithms for multiple-view binary pattern reconstruction. *IEEE Trans. Syst., Man, Cybern.* (Jan. 1971), 90–94.

Algorithm

procedure *Pattern Reconstruction* (*switch, m, n, fx, fy, f*);
integer *m, n, switch;* **integer array** *fx, fy, f*;
comment The parameters are defined as follows: *switch* is an output parameter with values -1, 0, or 1 according as the projections are inconsistent (*switch* $= -1$), the pattern is ambiguous (*switch* $= 0$), the pattern is unambiguous (*switch* $=$ 1). m is the column dimension of the binary pattern f, and n is the row dimension of the binary pattern f. m and n are input

Author's present address: Institute of Mathematics, Academia Sinica, 910 Nankang, Taiwan, Republic of China.

parameters. The array fx [1:n] is the projection of the binary pattern f on the x axis. fx is an input array. The array fy [1:m, 1:2] contains 1, 2, . . . , m in column 1 initially, and column 2 contains the projection of the binary pattern f on the y axis. fy is an input array, and it is modified by this procedure. The array f [1:n, 1:m] contains 0 initially and contains the reconstructed binary pattern finally;

```
begin
  integer ix, iy, j, number;
  procedure Sort;
  begin
    integer limit, ind, i;
    limit := m - 1;
S1:
    ind := 0;
    for i := 1 step 1 until limit do
    if fy [i, 2] < fy [i+1, 2] then
    begin
      integer t1, t2;
      ind := 1;
      t1 := fy [i+1, 1];  t2 := fy [i+1, 2];
      fy [i+1, 1] := fy [i, 1];
      fy [i+1, 2] := fy [i, 2];
      fy [i, 1] := t1;  fy [i, 2] := t2
    end;
    limit := limit -1;
    if (limit > 0) ∧ (ind = 1) then go to S1
  end Sort;
  procedure Merge;
  if fy [number, 2] < fy [number+1, 2] then
  begin
    integer n1, n2, t1, t2;
    n1 := number;
S2:
    if n1 > 1 then
    begin
      if fy [n1, 2] = fy [n1-1, 2] then
      begin n1 := n1 - 1;  go to S2 end
    end;
    n2 := number + 1;
S3:
    if n2 < m then
    begin
      if fy [n2+1, 2] = fy [n2, 2] then
      begin n2 := n2 + 1;  go to S3 end
    end;
S4:
    t1 := fy [n1, 1];  t2 := fy [n1, 2];
    fy [n1, 1] := fy [n2, 1];  fy [n1, 2] := fy [n2, 2];
    fy [n2, 1] := t1;  fy [n2, 2] := t2;
    if (n1 < number) ∧ (number+1 < n2) then
    begin n1 := n1 + 1;  n2 := n2 - 1;  go to S4 end
  end Merge;
  comment The procedure Sort orders fy, and the procedure Merge
    reorders fy. The main procedure now follows;
  switch := 1;
  Sort;
  for ix := 1 step 1 until n do
  begin
    number := fx [ix];
    if number > 0 then
    begin
      for j := 1 step 1 until number do
      begin
        iy := fy [j, 1];
        fy [j, 2] := fy [j, 2] - 1;
        f [ix, iy] := 1
      end;
      comment One column of f is reconstructed;
      if number < m then
      begin
        if (switch=1) ∧ (fy[number, 2] < fy[number+1, 2])
        then switch := 0;
        comment The above condition indicates that the
          pattern f is ambiguous, and the switch is set to 0;
        Merge;
        comment fy is reordered before we start to reconstruct
          the next column;
      end
    end
  end;
  for j := 1 step 1 until m do
  if fy [j, 2] ≠ 0 then switch := -1;
  comment The above condition indicates that the projections are
    inconsistent, and the switch is set to -1;
end Pattern Reconstruction
```

Remarks on Algorithm 445 [Z]

Binary Pattern Reconstruction from Projections [by Shi-Kuo Chang, *Comm. ACM 16* (Mar. 1973), 185–186]

John Lau [Recd. 22 July 1971]
Department of Computer Science, University of British Columbia, Vancouver 8, B.C., Canada

Key Words and Phrases: pattern reconstruction, image reconstruction, data compression, picture processing
CR Categories: 3.63, 5.30
Language: Algol

The procedure works well for all consistent patterns, ambiguous or unambiguous. However, when fx and fy are inconsistent, the procedure can construct a pattern f[1:n, 1:m] with fx satisfied, only if all elements of fx have values between 0 and m. If any of these elements is greater than m, a program interrupt would usually be caused by "value of subscript outside declared bounds" when the program executes the lines

```
for j := 1 step 1 until number do
begin
  iy := fy[j, 1];
  fy[j, 2] := fy[j, 2] - 1;
  f[ix, iy] := 1
end;
```

and execution of program would then be terminated. Even if a pattern could be constructed in this case, it would not be able to satisfy fx entirely.