Common Phrases and Minimum-Space Text Storage

by

Robert A. Wagner

Technical Report
No. 70-74
September 1970

Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14850

Common Phrases and Minimum-Space Text Storage

by

Robert A. Wagner*

Abstract: A method for saving storage space for text strings, such as compiler diagnostic messages, is described. The method relies on hand-selection of a set of text-strings which are common to one or more messages. These phrases are then stored only once. The storage technique gives rise to a mathematical optimization problem: determine how each message should use the available phrases to minimize its storage requirement. This problem is non-trivial when phrases which overlap exist. However, we present a dynamic programming algorithm which solves the problem in time which grows linearly with the number of characters in the text.

Key words and phrases: diagnostic messages, error messages, common phrases, minimum space, text storage, optimization, dynamic programming.

CR categories: 5.41, 4.10, 3.73.

*Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14850

Introduction

The PL/C compiler includes an extensive body of diagnostic messages [3]. A high premium has been placed on reducing the total amount of space used by this compiler. These facts prompted an effort to study means of reducing the storage-space cost of storing the diagnostic messages. The approach we took appears to be useful whereever a fixed collection of utterances must be stored.

We chose to select, by hand, a collection of phrases, each of which appeared in several messages. Instead of storing multiple copies of the common characters, only one copy was stored in full. Each occurrence of the common phrase, in messages or other phrases, could be replaced by a "reference" to the phrase. Only two bytes (characters) sufficed to record a phrase reference. Thus we hoped to achieve significant space savings by selecting repeated phrases each more than two characters in length.

An interpreter was designed to print messages and expand phrase references. We then attacked the resulting mathematical optimization problem. Here, we assume that a set of common phrases is given. Only these phrases may be referenced within messages or phrases. The problem is to discover for each message or phrase, that "parse" into non-overlapping phrases which minimizes the message's storage requirement. The present work concentrates on an efficient algorithm we have developed for so parsing messages.

The designers of at least two other fast, compile-into-core compilers (PUFFT [4] and CUPL [5]) have attacked the message-

storage problem. In both instances, the solution -- some form of phrase-reference -- was similar to ours. Neither design-group has reported any automatic techniques for selecting phrases, or for parsing messages.

Analysis of Space Requirements

In order to motivate the parsing algorithm, it will be necessary to describe the formats of phrases and messages. From the message format, and the format of accompanying tables, we will derive the cost function for the optimization problem we are to solve. This function requires a solution in integers to a non-linear optimization problem (Appendix A) in even the simplest case. Solutions to such non-linear problems are usually costly to compute. However, a re-formulation of the problem, using the principles of dynamic programming, produced a computationally efficient procedure. The interpreter which printed messages accepted the following message syntax:

<message> :: = <phrase reference><message>|<character string><message>|<end mark>

<phrase reference> :: = P<number>

<end mark> :: = E

<character-string> :: = C<number><string>

<string> :: = any string of 1 to 256 printable characters

<number> :: = any integer in the range 0 to 255, represented in binary as a single character.

The <number> occurring in a <phrase reference> indicates which of 256 phrases is intended; the <number> appearing in a <character-string> is a count of the number of characters in the <string> , minus one.

We assume throughout that no character-string larger than 256 characters ever occurs. The space requirement for each possible component of a message are then (in 8-bit bytes):

<phrase reference> - 2 bytes

<end mark> - 1 byte

<character string> - 2 + ℓ bytes, where the <string> contains ℓ characters.

A <phrase reference> simply refers to one of a distinguished subset of messages. This selected phrase is substituted for its <phrase reference> by the interpreter. Phrases and messages are syntactically identical. Thus, the interpreter recursively expands any <phrase references> in each referenced phrase.

A more compact encoding for character strings would have been possible. Specifically, we could have reduced the space-cost of a character string to ℓ, the number of characters occurring in its <string>. However, this would have required the message-printer to scan every character of the string, looking for either the phrase reference (P) or the end mark (E). This appeared to be unacceptably slow. Instead, we added two extra characters to each string (the C <number>), allowing us to use the IBM 360 "Move Characters" instruction to move the entire string at once. We refer to these extra characters as "overhead characters," and say that the character-string overhead equals 2 in this scheme.

The Parsing Problem

Phrases and messages are initially presented in the form of simple strings of characters, with no "overhead" characters present. The length of each phrase or message is also provided. In the required final form, <phrase references> will be permitted to refer to that collection of strings designated <phrases> originally; phrases and messages are otherwise indistinguishable. In particular, all phrases and all messages presented will be stored.

The set of phrases given initially acts like a partial grammar for a context free language. The language consists of a finite set of sentences (the phrase and message strings themselves). The right-hand sides of its grammar rules contain no non-terminal symbols. In addition, the "grammar" may be augmented as needed by additional rules, which correspond to <character string> occurrences. Nevertheless, we will term the translation process which changes the simple initial strings into compact final form "parsing."

Technically, a parse will be a set R of pairs (j,p), where p is a phrase which matches characters $j, j+1,\ldots,j+|p|-1$ of the message text. Here we denote the length of a phrase p by $|p|$. Furthermore, if R contains pair (j,p), R must not also contain a pair (k,q) where $j \leq k < j+|p|$ or $k \leq j < k+|q|$. Each parse corresponds directly to an acceptable final form for a message.

Several procedures have been proposed for solving this problem, apart from the integer programming approach. It seems worthwhile to examine one of these, since it illustrates how such procedures can eliminate the non-linear constraint in the integer-programming formulation, without incurring tremendous time penalties.

The single non-linear constraint in the integer programming formulation is designed to ensure that, if two phrases match the same substring of the message, only one of the phrases is used. Various other methods can be used to achieve the same end. (For example, all acceptable parses could be enumerated, rather than generated from the phrases and positions which make them up.)

Another technique involves repeated transformation of the entire message-text to (supposedly) produce a minimum-space form. Some particular phrase is selected, and all occurrences of that phrase are replaced by references to phrase. The resulting text is then scanned for occurrences of other phrases, without "expanding" any existing phrase references. The scanning is a simple comparison operation which treats the special "P" and "E" marks as ordinary characters. The process is then performed repeatedly, until all phrase occurrences have been extracted.

At each stage, some rule for selecting the phrase when occurrences are to be extracted is needed. One choice for this rule is:

"Select that phrase which is longest, of those phrases remaining." However, this rule fails to yield truly minimal-storage forms of message-text. Consider the message:

ABCDEABCD

where phrases

(1) ABCD and (2) CDEAB

are available.

The "largest first" rule yields:

AB (2) CD, where (2) represents a 2-byte phrase reference to phrase 2. Its cost is 6 bytes, excluding character-string overhead.

The best parse is

(1) E (1): cost = 5 bytes, excluding character-string overhead.

(The two illustrated strings would actually be stored as:

C1ABP2C1CDE (11 bytes),

and P1C0EP1E (8 bytes),

where the underlined numerals represent the character whose 8-bit binary representation equals the underlined number.)

The Optimal-Parsing Algorithm: Simple Case

An efficient algorithm for producing a space-optimal parse of a body of text was developed. The algorithm is basically a generalization of the procedure described above.

Instead of heuristic rules, the algorithm uses a kind of mathematical induction, to ensure that the parse it produces is optimal.

First, by the conditions of the problem, each "message" and "phrase" is presented initially in the form of a string of characters. The total number of characters is also known, for each message and phrase. This representation allows phrases to be compared with substrings of messages easily, to see if a "match" is achieved at each character-position of the message.

It should be clear from the desired final form of messages, that no phrase can "match" the boundary between two messages. Minimizing the storage-space required for each individual message will therefore minimize the space needed for the entire text. We can concentrate, then, on the algorithm which produces a space-minimal parse of a single message, using a predetermined set of phrases as possible phrase-references.

Consider one message, presented initially as a simple string of characters (no character-counts or "flags" present). Number the character-positions in this finite string from 1 to N. Suppose that, for all j, $N \geq j > I$, the following function has been computed:

f(j) = least space needed to store the final form of the characters numbered j, j+1,...,N of the given message.

We will show how $f(I)$ can be computed from this information, the message string itself, and the given set of phrases. Using this rule, the value of $f(1)$ can eventually be determined, and along with this value = "the least space needed to store the entire message," the optimal parse.

Let P = the set of all phrases.

If $p \in P$, let $|p|$ = length of p in characters, when p is in initial (fully expanded) form.

Let $ST(j,p)$ be a predicate which is <u>true</u> just when phrase p matches characters $j, j+1, \ldots, j+|p|-1$ of the given message. $ST(j,p)$ is <u>false</u> whenever p is not a phrase, or when any of the characters $j, \ldots, j+|P|-1$ either doesn't exist or fails to equal the character of p its position corresponds with.

To define a rule for computing $f(I)$, consider first the simple case when <u>character-string-overhead = 0</u>.

Let $P(I) = \{p \mid ST(I,p)\}$.

Let $F(I) = \min_{p \in P(I)} [F(I+|P|)+2, F(I+1)+1]$

Take $F(N+1) = 1$ as initial conditions.

Then we claim that $f(I) = F(I)$, $1 \leq I \leq N$, assuming that character-string-overhead = 0.

For, if we assume by induction that $f(j) = F(j)$ for $N \geq j > I$, then if phrase $p \in P(I)$ should be used in the parse at I, then that phrase will "match" characters $I, \ldots, I+|p|-1$ of the message.

The storage space for these characters will be reduced to the 2 characters needed for the phrase-reference to phrase p. The remainder of the message, characters $I+|p|,\ldots,N$, will require $f(I+|p|)$ characters for its storage. But $f(I+|p|) = F(I+|p|)$ by the induction hypothesis. If, on the other hand, no phrase should be used in the parse at I, the one-character string at position I can be stored, followed by the optimal parse for characters $I+1,\ldots,N$ of the message. Since character-string overhead is assumed zero for now, storage of a one-character string requires one character (plus zero overhead characters). Thus, the total storage requirement in this case is $f(I+1)+1 = F(I+1)+1$. Finally, we minimize over all available alternatives to compute $F(I) = f(I)$.

The initial value of $F(N=1) = 1$ was chosen to account for the required <end mark> on the message.

The algorithm presented above is an example of the stage-wise decomposition plus inductive reasoning which is characteristic of dynamic programming algorithms [2]. It should also be recognized that the search for phrases in P(I) required by this algorithm can be greatly narrowed by the use of "hash table" techniques [1]. In our problem, the cost of a <phrase reference> is 2. This places a lower limit of $3 \leq |p|$ on every phrase p which is useful in reducing storage requirements. As a result, the three characters numbered I,I+1,I+2 of the given message may be "hashed," and only phrases beginning with these characters need be compared using the ST(I,p) function.

The Optimal-Parsing Algorithm: General Case

To remove the requirement that character-string-overhead = 0, it proves useful to define a pair of functions inductively. To motivate the definition, consider:

$g(j)$ = the least space needed to store characters $j,\ldots,N$ of the message, provided that the final form of the message begins with a <character-string> , and

$h(j)$ = the least space needed to store character $j,\ldots,N$ of the message, regardless of what the first component of the final storage form of the message is.

We need both $h(j)$ and $g(j)$ to account for the effect that the leading component of the message has when prefixed by another character. If that leading component is itself a <character-string>, the additional character can be absorbed at no extra overhead cost; otherwise, a character-string-overhead cost is incurred. We can now define:

$$G(I) = \min\,[G(I+1)+1,\ H(I+1)+3]$$

$$H(I) = \min_{p\in P(I)}\,[H(I+|p|)+2,\ G(I)]$$

$$G(N+1) = 3,\ H(N+1) = 1$$

The proof that $G(j) = g(j)$ and $H(j) = h(j)$, $j=1,\ldots,N$ follows closely the pattern of reasoning used in proving that $f(j) = F(j)$. Here we take as induction assumption the statement

$G(j) = g(j)$ and $H(j) = h(j)$ for $N\geq j>I$, and we prove successively that

$G(I) = g(I)$, then that $H(I) = h(I)$.

To show that G(I) = g(I), given the induction assumption, we note that the conditions on parses from which g(I) is computed, require that all parses must begin with a <character-string> . The character whose number is I must therefore be a member of a <character-string> . Either character I is added to the beginning of an existing <character string> , or character I must be the sole member of a new <character string> . These alternatives yield space costs respectively of:

g(I+1)+1 (=G(I+1)+1 by the induction assumption.)

or

h(I+1)+3 (=H(I+1)+3).

The smallest of these two alternatives equals g(I). Hence, by definition of G(I), g(I) = G(I).

The proof that H(I) = h(I) follows the proof that F(I) = f(I) so closely that its repetition would be pointless.

In our implementation of this dynamic programming algorithm, we record the choice selected by each minimization function for each j. We can then reconstruct the set of optimum choices by reconstructing those G and H values needed for the computation of H(1). Appendix B gives an Algol version of the complete parsing algorithm.

Summary

We have presented a method for storing textual messages which reduces the storage-space they require by allowing them to

refer cheaply to any of a fixed set of common phrases. An efficient algorithm for determining exactly which combination of phrase references and character strings produces the minimal storage representation for each message has also been described. This algorithm does not automatically choose the optimal set of common phrases, however. Nonetheless, we have found the parsing algorithm quite helpful in reducing the storage requirements for error diagnostics printed by the PL/C compiler.

Appendix A

An integer-programming formulation of a simple case of the optimum parsing problem is presented here.

Let P_{ij} = 1 if phrase i matches the given message at position j, and

P_{ij} = 0 otherwise.

Let L_i = length of phrase i.

The P_{ij} and the L_i can be calculated directly from the given set of phrases and the message to be parsed.

Let V_{ij} = 1 if parse i is to be used in the given message at position j, and V_{ij} = 0 otherwise.

Then V_{ij} represents the required parse, in the case where the character-string overhead = 0, just when V_{ij} is the solution to:

maximize $\sum_{i,j} V_{ij}(L_i - 2)$

such that

(1) $V_{ij} = 0$ or $V_{ij} = 1$

(2) $V_{ij} \leq P_{ij}$

(3) $\sum_{i} V_{ij} \leq 1$

(4) $V_{ij} * \sum_{m} \sum_{k=j+1}^{j+L_i-1} V_{mk} = 0$

Condition (2) ensures that a phrase matches the message, if it is used in a feasible parse. Condition (3) ensures that at most one phrase is used at each starting position. Condition (4) imposes the requirement that phrases selected must not overlap. The criterion function represents the total characters saved by phrase references. Maximizing this total-saved minimizes the space needed for the message when character-string overhead is zero.

Appendix B

An Algol Procedure to Produce a Space-Minimal Parse of a Text String

```
procedure PARSE(R,L,C,N,P,M); value C,N,P,M;
integer array R,C,P; integer L,N,M;
comment:  C[1],...,C[N] hold the characters in the text string,
          P[i,1],...,P[i,P[i,0]] hold the characters making up
                                 phrase i.
          R[1],...,R[L] will hold the optimum final form of
                        the character string, after PARSE
                        returns.
          All characters are assumed to be encoded as
          integers in the range 0 to 255 inclusive throughout
          this algorithm;
comment:  The flags 'C', 'P' and 'E' are coded respectively
          as the integers 1, 2, and 3 in R;
begin integer array G,H,Q[1:N+1], HSH[0:383];
      integer I,J,K,X,S,T;
      comment:  G[i] will be the least space needed to store
                     the final form of C[i],...,C[N],
                     assuming the final form must begin with
                     a <character string>.
                H[i] will be the least space needed to store
                     the final form of C[i],...,C[N].
```

```
                Q[i] will record a phrase number (when
                     H[i] < G[i]) or a character-string
                     length (when H[i] = G[i]).
                HSH  is a "hash table," holding phrase
                     numbers or zeros.  Its size reflects
                     the restriction that no more than 256
                     phrases are present;
procedure   ENTER(A); value A; integer A;
     comment:   This procedure enters a character into R;
begin   L := L+1; R[L] := A; end ENTER:
Boolean procedure   ST(j,p); value j,p; integer j,p;
     comment:   ST(j,p) is true if phrase  p  matches characters
                j,j+1,...,j+|p|-1  of the text string, and is
                false otherwise;
begin integer  i;
     ST := false;
     if  j+P[p,0] > N+1 then goto L1;
     for i := 1 step 1 until  P[p,0] do
          if  C[j+i-1] ¬ = P[p,i] then goto L1;
     ST := true;
L1:  end ST;
```

```
integer procedure  HASH(Q,B,X); value Q; integer B,X;
    comment:  if Q < 0,
                  B should be an array reference, involving
                  the actual parameter, a variable,
                  substituted for the formal parameter  X by
                  the call.  HASH examines the 3 entries of  B
                  accessed when  X = 1,2, and 3, and returns
                  an index to an entry in HSH which indexes
                  a phrase which begins with those same
                  three integers, if such an entry exists.
                  Otherwise, HSH returns the index to a
                  zero entry in HSH.
              if Q ≥ 0,
                  parameters  B  and  X  is ignored.  The
                  values examined the last time HASH was
                  called with  Q < 0  are used, together
                  with an internally-stored record of the
                  last value returned by HASH to compute
                  the index of the next entry in HSH which
                  indexes a phrase beginning with the three
                  examined characters.  If no such entry
                  exists, the HSH entry contains 0;
    comment:  HASH uses a slightly modified version of the
              quadratic quotient algorithm. [1]
```

```
begin own integer Y,j,A,k;  integer g;
     if   Q < 0 then begin
                 Y := 0
                 for X := 1 step 1 until 3 do
                     Y := 256 * Y + B;
                 j := max(1, entier(Y/HSHL));
                 k := Y; A := 1;
                 end;
L1:    A := A + j; k := k + A
       k := k - (k ÷ HSHL) * HSHL;
       if HSH[k] = 0 then goto L;
       g := HSH[k];
       if Y ¬= 256*(256*P[g,1] + P[g,2]) + P[g,3]
                 then goto L1;
L:     HASH := j;
       end HASH;
```

```
comment:   body of PARSE procedure begins here, with
           hash table initilization;
     HSHL := 383;  comment:  chosen to be a prime ≈ 3/2*256,
                             where 256 = largest allowable
                             M  value;
     for I = 0 step 1 until HSHL do
           HSH[I] := 0;
     comment: enter the phrases into HSH;
     for I := 1 step 1 until M do begin
           K := -1;
           L1:   K := HASH(K,P[I,X],X);
                 if HSH[K] ¬ = 0 then goto L1;
           HSH[K] := I;
           end;
     comment:   Compute G,H and Q;
           G(N+1] := 3; H[N+1 := 1; Q[N+1] := S := 0;
           for I := N step -1 until 1 do begin
                 G[I] := G[I+1] + 1;
                 Q[I] := S;
                 T := H[I+1] + 3;
                 if T < G[I] then begin
                       G[I] := T; Q[I] := S := I end;
                 H[I] := G[I];
                 K := -1
                 for K := HASH(K,C[I+X-1],X) while
                       HSH[K] ¬ = 0 do
                       if ST(I,HSH[K]) then begin
```

```
                    J := HSH[K];
                    T := H[I+P[J,0]] + 2;
                    if T < H[I] then begin
                        Q[I] := J;
                        H[I] := T end;
                    end;
                end;
comment:  Now, using G, H and Q we can construct the final
          space-optimal form of message C;
          L := 0; I := 1;
L3: if I > N then goto L4;
    if G[I] = H[I] then begin
          ENTER(1);              comment:  <character string>;
          ENTER(Q[I] - I);
          for J := I step 1 until Q[I] do
                ENTER(C[J]);
          I := Q[I] + 1; end
    else begin
          ENTER(2)               comment:  <phrase reference>;
          J := Q[I];
          ENTER(J);
          I := I + P[J,0];
          end;
    goto L3;
L4: ENTER(3);                    comment:  <end mark>;
   end PARSE;
```

Acknowledgement

We are grateful to Professor Howard Lee Morgan, one of our co-workers on the PL/C implementation project, who suggested the method described here for storing messages. Professor Morgan also wrote the message-writer interpreter which accepts and converts condensed messages to readable form.

REFERENCES

1. Bell, J.R. The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering. CACM 13, 1, pp. 107-109 (Feb 1970).

2. Nemhauser, G.L. Introduction to Dynamic Programming. John Wiley, N.Y. (1966).

3. Conway, R.W., et. al. PL/C - A High Performance Subset of PL/I. TR 70-55, Computer Science Dept., Cornell Univ., Ithaca, N.Y. 14850, (Feb 1970).

4. Rosen, S., Spurgeon, R.A., and Donnelly, J.K. PUFFT - The Purdue University Fast FØRTRAN Translator, CACM vol 8, number 11 (Nov 1965).

5. Conway, R.W., and Maxwell, W.L. CUPL - An Approach to Introductory Computing Instruction. Dept. of Computer Science, Cornell Univ., Ithaca, NY.