AD-A119 419 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
USING STRING MATCHING TO COMPRESS CHINESE CHARACTERS,(U)
MAY 82 G GUOAN, J HOBBY N00014-81-K-0330
UNCLASSIFIED STAN-CS-82-914 NL

May 1982

Report No. STAN-CS-82-914

Using String Matching to Compress Chinese Characters

by

Gu Guoan and John Hobby

Department of Computer Science

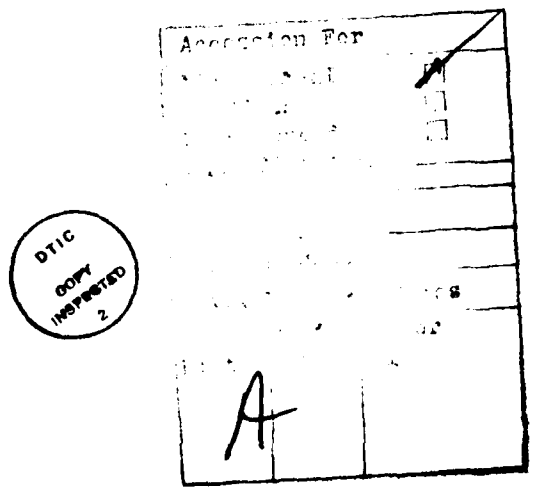
Stanford University
Stanford, CA 94305

Using String Matching to Compress Chinese Characters

John Hobby and Gu Guoan

Abstract

A new method for font compression is introduced and compared to existing methods. A very compact representation is achieved by using a variant of McCreight's string matching algorithm to compress the bounding contour. Results from an actual implementation are given showing the improvement over other methods and how this varies with resolution and character complexity. Compression ratios of up to 150 are achieved for Chinese characters.



This research was supported in part by National Science Foundation grants MCS-77-23738 and IST-7921977 and by Office of Naval Research contract N00014-81-K-0330. Reproduction in whole or in part is permitted for any purpose of the United States government.

0. Introduction

With the development of computer typesetting it has become important to find efficient ways to represent fonts in computers. We can achieve good results at a reasonable cost by adapting ideas from [6] to improve upon existing font compacting schemes. We compare an actual implementation of this new scheme with an implementation of the existing idea of contour coding. Data from Chinese characters has been given special attention in our experiments because such characters are particularly challenging; the new scheme can of course be applied to Western alphabets, where its performance is even better.

Storing and manipulating thousands of high resolution Chinese characters can be quite expensive. For example, the 辞海, a large dictionary of words and phrases, contains 14,782 characters. The 新華字典, a dictionary for daily use, contains about 11,100 characters including some modified and simplified characters. There are about 8,000 commonly used characters. A sample of 21,629,372 characters [1] from writings on political theory, science, literature and the arts, and in newspapers has shown that 100 characters are used 40% of the time, 250 characters are used 60% of the time, 500 characters 80%, 1,000 characters 90%, 2,000 characters 98%, and 3,000 characters more than 99%. A Chinese computer typesetting system needs hundreds of thousands of characters including all the different sizes in four different styles.

In the first section we give a brief overview of the various methods for encoding fonts. We describe in detail a method of contour coding. This is particularly important because the new string matched scheme is based on it and the contour coding method is used for comparison. Finally, we explain the format for string matched compression. String matched compression depends on being able to represent variable length numbers efficiently. The second section describes a rather interesting approach to this problem. Next, we present a fast algorithm based on [6] for actually doing string matched font compression. We then give results including the performance of the method on a variety of characters, its dependence on the size and complexity of the characters, and a comparison with other methods. These results look promising but there is room for improvement. In the final section we deal with these and present empirical results.

1. Methods for Character Encoding

Bitmap encoding

The bitmap is a simple method for encoding characters where one bit is used to specify the contents of each pixel. The advantages of this method are that decoding speed is maximal and only a small buffer is needed. Since the size of the code grows as the square of the linear resolution, bitmap encoding is best suited to low resolution. Using this method, experimental Chinese computer systems have been set up that print characters of 16×16 and 24×24 pixels.

Run length encoding

Run length encoding uses rectangular areas one bit thick called runs. It is shown in [2] that if n is the size of the matrix, run length encoding produces output of length $O(kn \log n)$, where k is the number of runs per a scan line. The factor k is really a measure of the complexity of the character pattern. For Roman fonts, k is approximately 4, while it is more than 10 for most Chinese characters. For 10.5 point Chinese characters of size

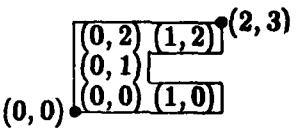
$n = 128$, the compression ratio over bitmap encoding is about $\frac{n^2}{kn\lg n} = \frac{n}{k\lg n} \approx 1.8$. We can see that run length encoding is not particularly effective for typical Chinese character typesetting. However, run length encoding has the advantage that it is almost as fast as bitmap encoding and it also requires very little buffer space.

Differential run length encoding

This method differs from run length encoding in that we represent the positions of a stroke boundary on a scan line by the displacement from the boundary points on the adjacent scan line. This method uses only $O(kn)$ space for $n \times n$ bitmaps since the displacement from adjacent scan lines is usually small.

Contour coding

The basic idea behind the type of contour coding we use here is that the black pixels are surrounded by a bounding contour. The contour can be restricted to an orthogonal grid as illustrated below.



In this example, we give the coordinates of all the black pixels and also of certain points on the bounding contour. If a point on the boundary has coordinates (i, j), then the pixel at its upper right also has coordinates (i, j) but the pixel at its lower left has coordinates $(i-1, j-1)$. This means that the maximum possible coordinates for points on the bounding contour are always one more than the coordinates of the upper right hand pixel. Contour coded files begin with a header giving the number of contours, N, the size of the character box, and the position of the contours within the character box. The specification of the actual contours follows. The file is thought of as a bit string with word boundaries ignored, and it can be diagrammed as follows.

$$\langle N\rangle\langle x_{\max}\rangle\langle y_{\max}\rangle x_1 y_1 d_1 \ldots x_N y_N d_N \sigma_0 \sigma_1 \ldots \sigma_{n-1} \tag{1}$$

The character box contains pixels with x coordinates ranging from 0 to $x_{\max}$ and y coordinates ranging from 0 to $y_{\max}$. The bounding contour for the i^{th} boundary starts at (x_i, y_i) in direction d_i. Quantities in angle brackets represent variable length numbers. (See below.) The variables x_i, y_i, and d_i are fixed length numbers and each σ_i represents either "left", "right", or "straight". The character in the above figure could be represented with $N = 1$, $x_{\max} = 1$, $y_{\max} = 2$, $x_1 = 2$, $y_1 = 3$, $d_1 =$ west, and $\sigma_0 \ldots \sigma_{11} =$ *SLSSLSLLRRLL*. We can tell when a boundary ends by when it closes on itself. If $N > 1$ the string $\sigma_0 \ldots \sigma_{n-1}$ can be broken N pieces, each of which form a closed path. The starting coordinates x_i and y_i each require $1 + \lfloor \lg(x_{\max} + 1) \rfloor$ bits and $1 + \lfloor \lg(y_{\max} + 1) \rfloor$ respectively and the initial directions d_i will fit in 2 bits. The boundary turns σ_i are encoded into 1 or 2 bits as follows.

$$R \mapsto 00, \qquad L \mapsto 01, \qquad S \mapsto 1$$

This header information is not encoded in a particularly efficient manner. One could envision ways of encoding x_i, y_i, and d_i more efficiently by picking starting points with special properties. For Chinese characters where there can be many boundaries some differential scheme might be tried. All these methods are awkward and have little effect because the header is much smaller than the rest of the file anyway.

String matched contour coding

String matched contour coding is a slight modification of the above scheme where $\sigma_0 \ldots \sigma_{n-1}$ are encoded differently. A string matched contour coded file is

$$\langle N \rangle \langle x_{\max} \rangle \langle y_{\max} \rangle x_1 y_1 d_1 \ldots x_N y_N d_N \tau_1 \langle k_1 \rangle p_1 r_1 \tau_2 \langle k_2 \rangle p_2 r_2 \ldots \tau_m \langle k_m \rangle p_m r_m \qquad (2)$$

where

$$\Sigma(q) = q - 1 + \sum_{i \le q} k_i,$$

$$\tau_q = \sigma_{\Sigma(q)} \quad \text{and}$$

$$\sigma_{p_q + j} = \begin{cases} \sigma_{j+1+\Sigma(q)}, & \text{if } r = 0, \\ \mathrm{rev}(\sigma_{j+1+\Sigma(q)}), & \text{if } r = 1; \end{cases} \qquad 0 \le j < k_q.$$

The function rev just switches left and right (i.e., $\mathrm{rev}(L) = R$, $\mathrm{rev}(R) = L$, and $\mathrm{rev}(S) = S$). The i^{th} 4-tuple $\tau_i \langle k_i \rangle p_i r_i$ means "The next character in the string $\sigma_0 \ldots \sigma_{n-1}$ is τ_i. Now start at σ_{p_i} and copy the next k_i characters. If $r_i = 1$ then switch left and right as you do the copying." The τ_i could be encoded just as the σ_i for contour coded files but this would be inefficient because we can assume that $\tau_q \neq \sigma_{\Sigma(q)}$. (Otherwise we could use a larger value for k_{q-1}.) The first turn τ_1 is therefore encoded as the σ_i above, while the subsequent τ_i require only 1 bit. The numbers k_i are variable length while the p_i can be fixed length numbers because we require that $0 \le p_i \le \Sigma(i)$. Finally, the r_i are only 1 bit long.

2. Encoding Variable Length Numbers

In the above encodings we need some way of encoding arbitrarily large integers efficiently. All encodings must have unique prefixes so that we can tell where they end. This clearly requires longer codes for larger integers. Simple schemes for doing this are suggested in [5] and [9]. These schemes are asymptotically optimal, however they behave poorly for numbers of the size that our k_i are likely to be. To combat this we look at the probability distribution we get for the k_i in a typical character. A capital "S" 537 pixels high had the distribution on the next page. The column labeled "occurrences" gives the total number of different i for which k_i was in the specified range. In the column labeled "normalized occurrences" this is divided by the size of the range.

The "ideal" way to encode k_i would be to use Huffman's minimum weighted path length algorithm to find the encoding that minimizes the expected average length of the k_i. We must modify this approach if we want the encoding to be efficient for all the distributions of k_i we are likely to encounter. We would also like there to be a simple pattern that would allow us to compute the encoding for arbitrary integers. For these reasons, we abstract the frequency information in the table and pretend that the actual normalized occurences are constant up to a certain value (say about 20) and then they fall off as $1/x^\alpha$. This assumption could be modified without too much difficulty. We will see below that it is very convenient to deal with probabilities falling off as $1/x^\alpha$. In addition, this provides a reasonable fit with the above data for large k_i, with $\alpha \approx 2.75$. We can adjust this parameter to obtain somewhat more reasonable behavior for large numbers.

k_i	occurrences	normalized occurrences
3	1	1.0
4–5	2	1.0
6–7	2	1.0
8–10	14	4.7
11–15	30	6.0
16–22	41	5.9
23–31	25	2.8
32–44	15	1.14
45–63	6	0.32
64–90	4	0.15
91–127	1	0.03
128–180	1	0.02
181–255	0	0
256–361	1	0.01

We can derive the following encoding procedure from these assumptions.

```
function encode(n);
begin
    r ← r0;
    e ← e0;
    if n < t
    then if n < c
        then return the lower e − 1 bits of n
        else return the lower e bits of n + c
    else begin
        x ← t;
        d ← d0;
        while n ≥ x + d
        do begin
            e ← e + 1;
            x ← x + d;
            r ← 2(r − d);
            d ← ⌊rb + .5⌋;
          end;
        return the lower e bits of (2^e − r + n − x)
      end;
end;
```

For the above function we assume the following initialization has taken place.

integer r_0, e_0, c;
real b;
$b \leftarrow (1 - s/2)$;
$r_0 \leftarrow \lfloor t(s-1)/b + 0.5 \rfloor$;
$e_0 \leftarrow \lfloor \lg(r_0 + t - 1) \rfloor + 1$;
$c \leftarrow 2^{e_0} - r_0 - t$;
$d_0 \leftarrow \lfloor br_0 + 0.5 \rfloor$;

The process is controlled by the constants t and s. All numbers less than t are given nearly equal length encodings. Beyond t, encodings get one bit longer each time the number increases by a factor of s. We require that $ts/(2-s)$ be at most t less than a power of two so that $c \leq t$. The variable e is the length of the encoding. We use encodings that appear in numerical order when viewed as binary fractions. As numbers get bigger their encodings acquire more and more leading ones. The variable r gives the number of codes of length e that could be used without interfering with the encodings for smaller numbers. We can achieve the constant factor s by using up a fixed fraction, b, of the available encodings at each step. At the end of each iteration of the while loop, x is the smallest number whose encoding is e bits long and there will be exactly d different numbers with encodings e bits long. We can see that for large n,

$$s = \frac{x+d}{x} = 2\frac{r-d}{r}$$

is stable. This holds when $b = 1 - s/2$ as set in the initialization block. A minimum weighted path length encoding for a probability distribution proportional to $1/x^\alpha$ would exhibit this kind of behavior if $r/2^e$ is proportional to $\int_x^\infty 1/t^\alpha \, dt$. This holds when $\frac{1}{2}s = s^{1-\alpha}$, i.e., $s = 2^{1/\alpha}$.

Consider the case where $t = 10$ and $s = 1.7$. This gives $b = .15$, $r_0 = 47$, $e_0 = 6$, and $c = 7$. We get the encodings

0	00000
⋮	⋮
6	00110
7	001110
⋮	⋮
16	010111
17	0110000
⋮	⋮
28	0111011
29	01111000
⋮	⋮
48	10001011
49	100011000
⋮	⋮

If we run the encoding algorithm on $n = 314$, for example, we get

e	x	r	d	$(x+d)/x$
6	10	47	7	1.700
7	17	80	12	1.706
8	29	136	20	1.690
9	49	232	35	1.714
10	84	394	59	1.702
11	143	670	101	1.706
12	244	1138	171	1.701

which makes the encoding of 314 equal to the lower 12 bits of $2^{12} - 1138 + 314 - 244$ or 101111010100.

Decoding variable length numbers

The process of decoding variable length numbers in the above format is quite straight forward. Since speed is important in the decoding process, it may be desirable to use tables to find the values of k and x obtained below. Here we give a version of the decoding algorithm that uses no such tables. This is more instructive and still quite fast.

```
function decode;
begin
    read e0 − 1 bits into m;
    if m < c then return(m);
    m ← 2m + (read in the next bit);
    n ← m − c;
    if n < t then return(n);
    e ← e0;
    x ← t;
    r ← r0;
    d ← d0;
    k ← 2^e − r − x;
    do begin
        if e > e0
        then m ← 2m + (read in the next bit);
        n ← m − k;
        e ← e + 1;
        x ← x + d;
        r ← 2(r − d);
        d ← ⌊rb + .5⌋;
        k ← 2^e − r − x;
      end
    until n < x;
    return(n);
end;
```

All the variables in the decoding algorithm are analogous to the identically named variables in the encoding algorithm. The variable k keeps track of the quantity that would have been subtracted from n if the encoding algorithm had ended on the current step. The variables m and n are the number read from the input so far and how it would be decoded if we had read the correct number of bits from the input. When x surpasses this number it tells us we have read enough bits.

3. Data Compression Algorithms

We will concentrate mainly on the method for creating string matched contour coded files given the contour information. We can easily find the contour information by using the fact that the contour contains a line between (x, y) and $(x, y+1)$ if and only if the pixel at (x, y) differs from that at $(x-1, y)$ and similarly we look at pixels (x, y) and $(x, y-1)$ to see if the contour contains the line between (x, y) and $(x+1, y)$. We can therefore use shifting and masking instructions to avoid looking at the pixels individually.

For the final step of the data compression process, we will assume that the contour, $\sigma_0, \ldots, \sigma_{n-1}$, has been put into an array and that our goal is to find τ_i, k_i, p_i, and r_i for $i = 1, \ldots, m$. For convenience we will refer to the values of σ_i as L, R, or S, and the values of r_i as "true" or "false". Let σ_{ij} be the substring $\sigma_i\sigma_{i+1}\ldots\sigma_j$ with the understanding that σ_{ij} denotes the empty string when $i > j$. Recall that $\Sigma(i)$ is the number of characters matched by $\tau_1\langle k_1\rangle p_1 r_1 \ldots \tau_{i-1}\langle k_{i-1}\rangle p_{i-1} r_{i-1}$. We will use the function $\mathrm{rev}(\sigma)$ described above and define $\mathrm{rev}(\sigma_{ij}) = \mathrm{rev}(\sigma_i)\ldots\mathrm{rev}(\sigma_j)$. In order to find the shortest encoding (2) we must successively find the position p_i where $\sigma_{p_i}\sigma_{p_i+1}\ldots$ matches as much as possible of $\sigma_{\Sigma(i)+1}\sigma_{\Sigma(i)+2}\ldots$ for $i = 1, \ldots, m$. A straight forward implementation would be $O(n^2)$ but [6] gives a linear time algorithm that can be adapted to this problem. We will show how to use this algorithm to produce the 4-tuples in (2).

It will be convenient to adopt some of the notation used in [6]. Let suf_i be $\sigma_{i,n-1}$ and let head_i be the longest prefix of suf_i that also occurs starting at position j, $j < i$. (In other words, $\mathrm{head}_i = \sigma_{i,i+k} = \sigma_{j,j+k}$, where k is maximal and $j < i$.) Note that the starting position j of the maximal string might not be unique. The algorithm in [6] proceeds by constructing a suffix tree using McCreight's algorithm [7]. In our case we will need two suffix trees, one containing all strings suf_i for $0 \le i \le n-1$ and the other containing $\mathrm{rev}(\mathrm{suf}_i)$ for $0 \le i \le n-1$. The algorithm in [6] starts with an empty tree and inserts successively $\mathrm{suf}_0, \mathrm{suf}_1, \ldots, \mathrm{suf}_{n-1}$. It determines k_i and p_i from the location where $\mathrm{suf}_{\Sigma(i)}$ is inserted in the tree. The algorithm in [6] does not deal with anything equivalent to our r_i.

We can run two copies of this algorithm in parallel inserting $\mathrm{suf}_0, \mathrm{suf}_1, \ldots, \mathrm{suf}_{n-1}$ into suffix tree A and $\mathrm{rev}(\mathrm{suf}_0), \mathrm{rev}(\mathrm{suf}_1), \ldots, \mathrm{rev}(\mathrm{suf}_{n-1})$ into suffix tree B. We can then set k_i and p_i either according to where suf_i is inserted in tree A or where $\mathrm{rev}(\mathrm{suf}_i)$ is inserted in tree B. We choose the option that results in the largest value of k_i. If we use the first option then r_i is false and otherwise r_i is true. Using this method the algorithm on the next page produces the 4-tuples $\tau_1\langle k_1\rangle p_1 r_1 \ldots \tau_m\langle k_m\rangle p_m r_m$ from (2).

The algorithm presented in [6] finds p_i and k_i before τ_i. This has the effect of making p_1 and k_1 automatically 0 and complicates the termination conditions. It is necessary to treat the last triple differently in this case to make sure that k_m is not too large to allow τ_m a meaningful value. In [6] (to simplify the presentation) it is assumed that $\sigma_{n-1} \neq \sigma_j$

```
begin
    j ← i ← 0;
    while i < n
    do begin
        τ ← σ_i;
        if i = n − 1 then output(τ, 0, 0)
        else begin
            while j ≤ i + 1
            do begin
                insert suf_j in suffix tree A;
                insert rev(suf_j) in suffix tree B;
                j ← j + 1;
              end;
            output(τ, k_{i+1}, p_{i+1}, r_{i+1});
          end;
        i ← i + k + 1;
    end;
end;
```

for any $j < n-1$. This ensures that no $\text{head}_i = \text{suf}_i$. In our case this assumption is not necessary because if $\text{head}_i = \text{suf}_i$ then $i = \Sigma(m)$ and we are done.

The decoding algorithm

The decoding problem is how to get from one of the compressed formats (1) or (2) back to the raw bitmap. Decoding the contour coded files (1) is essentially a process of marching around the boundary flipping certain bits in the bitmap. Decoding string matched files (2) combines this with decoding variable length numbers and copying parts of a buffer. We can conceptually break these problems into two pieces: reading (2) and filling a buffer array with $\sigma_0 \ldots \sigma_{n-1}$, and combining the information in such a buffer with that in the preamble to obtain a raw bitmap. (Actually these problems are not entirely separate because when solving the first subproblem the information in the preamble does not tell us when to stop without tracing the contours to see when they close on themselves.)

In going from the string matched format to the pure contour format we will take as given the ability to get fixed format and variable length numbers from the input file. The algorithm is then very simple. (See next page.) We refer to the entries of the buffer we are generating as $\sigma_0 \ldots \sigma_{n-1}$.

The next subproblem is going from the buffer containing the bounding contours to the raw bitmap. It is easy enough to follow the contour keeping track of the current position and direction. One has to study the formats described above to avoid "off by one errors". For each vertical edge in the contour between (x_0, y) and $(x_0, y+1)$ we have to flip all the bits in locations (x, y) where $x \geq x_0$. We can do this quickly by having another bitmap with one bit for each word in the original. We then flip the right had a part of the word containing the pixel (x_0, y) and the right hand part of the word in the auxiliary table corresponding to the y row. (We assume a two level hierarchy is sufficient.) After all the bounding contours have been traced we need one additional pass to flip all the words whose bits in the auxiliary are still on.

```
begin
    i ← 0; j ← 1;
    repeat
        σ_i ← τ_j;
        i ← i + 1;
        for k ← 0 thru k_j − 1
        do σ_{i+k} ← (if τ_j then rev(σ_{p_j+k})
                      else σ_{p_j+k});
        i ← i + k;
        j ← j + 1;
    until done;
end;
```

Decoding speed

The above algorithm was implemented on a DEC-10 computer at Stanford. Some attention was paid to the speed of the code but more use of assembly language and a few other improvements could probably achieve significant gains. The program started fro the string matched format, created a buffer containing the bounding contours, and us this to regenerated the bitmap. There are two stages to the compution: first the buffer created and used to scan the bitmap, and then the auxiliary table is used to regenera the rest of the bitmap. The complexity of the first stage is linear in the length of the inp (actually $O(n \lg n)$ in general), while the second stage takes $O(n^2)$.

character	stage 1	stage 2	total
計	25 ms.	4 ms.	29 ms.
明	30 ms.	4 ms.	34 ms.
俊	40 ms.	4 ms.	44 ms.
堃	43 ms.	4 ms.	47 ms.
昌	26 ms.	4 ms.	30 ms.
水	24 ms.	4 ms.	28 ms.
田	23 ms.	5 ms.	28 ms.
張	37 ms.	5 ms.	42 ms.
耀	46 ms.	5 ms.	51 ms.
父	22 ms.	3 ms.	25 ms.

We can see that the quadratic term is still quite small at this resolution. Even at four times this resolution about 60% the time is still spent in stage 1. The above results are good considering the amount of data being handled, but in many applications faster decoding may be required. The process is probably simple enough to put into hardware if necessary.

Compression Results for Chinese Characters

The following table gives compression figures for a wide variety of Chinese characters. These characters were designed with Tung Yun Mei's LCCD system [8]. All the results given here were checked with the decoding algorithm to verify that the original bitmap

Chinese character	resolution 105 × 141†			resolution 203 × 275†	
	differential run length‡	contour coded	string matched	contour coded	string matched
美	7184	1819	1291	3477	1965
國		2326	1736	4340	2528
史	5504	1411	1243	2823	1856
坦	6416	1301	1031	2353	1315
福	9792	2026	1453	3580	1911
大	6144	1227	991	2431	1448
學		1577	1508	3091	2107
計		1472	911	2412	1028
示		1710	1211	2866	1592
机	11888	1615	1363	3143	2144
系	8336	2005	1652	3779	2557
教	13584	2019	1962	4171	2983
授	12240	2365	2116	4540	3208
高	7360	1633	1201	3064	1475
納	13744	2333	2208	4459	3199
德	13664	2314	2093	3980	2876
權		2236	1992	4410	2896
鍾		2245	2084	4550	2967
程	10808	1637	1466	3059	2038
序	8128	1441	1299	2881	1936
耀	16928	2485	2264	5022	3574
張		1989	1741	3979	2599
騰	14464	2424	2512	4930	3728
葉		1835	1512	3409	2001
志	7312	1503	1377	2704	1944
堅		1883	1542	3719	2057
設		2152	1710	3972	2534
施	13552	2275	2185	4553	3381
明	6736	1601	1161	3059	1550
華		2083	1791	4013	2549
維	13728	2105	1795	4021	2706
場		2257	1963	4544	2889
假	12800	2355	1991	4597	2902
伏	8224	1613	1327	3077	1993
雀	10096	1636	1380	3040	1954
效	13040	2265	1989	4590	3013
期	9520	1870	1581	3726	2327
象		2427	2122	4828	3328
材	10896	1497	1210	3171	1909
忽	11520	2413	1937	4594	2987

†Maximum size of bounding box over all characters

‡9 point Sung Dynasty style characters with 144 × 144 typeface

could actually be regenerated. The table entries give the number of bits needed to represent each of the characters below at two different resolutions and in three different formats. The data on the right side of the table are for the same characters at about twice the size. For comparison a bitmap encoding would require at least 105×141 or 14,805 bits for the lower resolution and 203×275 or 55,825 for the higher. The column labeled "differential run length" gives actual data from an ideographic image digitizer. It refers to 9 point Sung Dynasty style Chinese characters from [4]. The type face for these characters is 144×144 pixels, but the type body is probably about the same size for the other characters. The resolutions stated below are merely for the minimum size bitmap that could contain all these characters (i.e. type body). We can see that the string matched format represents these characters most efficiently. The compression ratio relative to bitmap encoding ranges from 5.9 to 16.2 at the lower resolution and from 16.5 to 54.3 at the higher one. The compression ratio relative to simple contour coding ranges from .965 to 1.615 at the lower resolution and from 1.32 to 2.35 at the higher one. There was only one case where contour coding outperformed the string matched format. String matched encoding compares very favorably with the differential run length format that achieved compression ratios relative to the bitmap ranging from .87 to 2.03.

The string matched compression format is particularly well suited to high resolution characters. The relative gain from the string matching procedure over the contour coded format increases with resolution. In addition to this, the relative overhead in terms of decoding time decreases. The table below also refers to the characters we designed with the LCCD system. Resolutions are measured as before. Magnification factors of 1, 2, 3, and 4 were used to achieve the different size bitmaps refered to below.

contour coded bits vs. string matched bits at different resolutions

Chinese character	96×131		186×255		277×376		368×499	
	contour coded	string matched	contour coded	string matched	contour coded	string matched	contour coded	string matched
計	1472	991	2412	1028	3418	1224	4384	1220
把	1301	1031	2353	1315	3409	1483	4421	1733
高	1633	1201	3064	1475	4466	1842	5866	1998
明	1601	1161	3059	1550	4517	1904	6089	2358
堅	1883	1452	3719	2057	5625	2799	7417	3027
葉	1835	1512	3409	2001	5129	2506	6635	2701
場	2257	1963	4544	2889	6598	3586	8694	4204
俊	2355	1991	4597	2902	6764	3454	9008	4146
系	2005	1652	3779	2557	5593	3291	7302	3990
地	2275	2185	4553	3381	6629	3853	8855	4269

Note that in the first line of the table, string matching actually produced a more compact encoding at the highest resolution than at the second highest. It is interesting to compare the string matched compression figures in the above table with other methods for compressing the contour coded format. Recall that the contour coded format used a simple Huffman code with R, L, and S encoded 00, 01, and 1 respectively. This scheme averages about 1.58 bits per turn. A more complicated encoding using all strings of length six over

the alphabet $\{L, R, S\}$ achieves at best .86 bit per turn for a saving of about a factor of 1.8. We can see that the string matched method easily surpasses this at high resolutions. Furthermore the string matched format has much more potential for improvement.

It is interesting to see how the string matched compression method responds to the different kinds of strokes used in Chinese characters. There are five fundamental strokes: dots (·), horizontal strokes (−), vertical strokes (|), "Pie" strokes (ノ), and "Na" strokes (乀). Chinese characters can be divided into two classes: those that consist mainly of horizontal and vertical strokes, and those that consist mainly of dot, "Pie", and "Na" strokes. The former class is more common occurring at a frequency of about 60%. Characters that consist mostly of horizontal and vertical strokes are in general easier to compress, but the string matched method appears to respond particularly well to them. The method responds reasonably well to straight lines at constant slope because any such line will become a repeating pattern of letters from the alphabet $\{L, R, S\}$. Curved boundaries present more problems. Compare the following compression results for isolated strokes. (To facilitate a more direct comparison the bit counts given do not include the preambles.)

kind of stroke	bounding box	contour coded	string matched
horizontal	94×13	284	160
vertical	11×105	248	128
Pie	45×64	433	296

The following table shows how this relates to full characters. The characters on the left consist entirely of horizontal and vertical strokes; those on the right have mostly Pie and Na strokes. These characters are all at the same resolution we have been using: maximal bitmap size 105×141.

Chinese character	contour coded	string matched	Chinese character	contour coded	string matched
山	854	361	人	919	659
三	841	470	父	1415	1025
王	959	550	火	1335	1115
田	1243	607	永	1395	1105
昌	1497	707	走	1413	1182

It is clear that the complexity of a character has a great effect on the number of bits needed to represent it. To see how the characters above compare to other Chinese characters we give the following table of the stroke count distribution of 6,763 frequently used characters from [4].

stroke count	number of characters	percentage
1–3	93	1.4%
4–6	753	11.1%
7–9	2014	29.8%
10–12	2037	30.2%
13–15	1183	17.5%
16–18	499	7.4%
19–21	145	2.1%
22–24	35	0.5%
25–27	4	0.1%

Potential Improvements

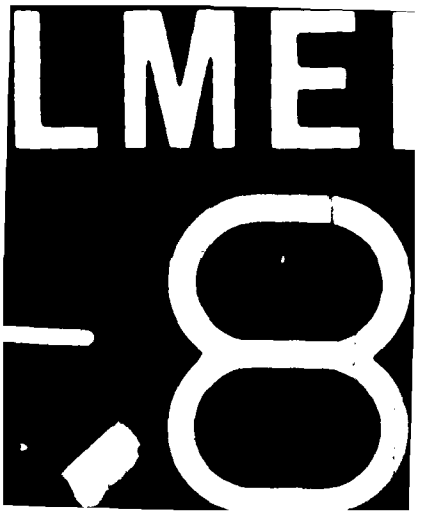
There are several ways the compression figures for Chinese characters might be improved. First of all, since the characters are composed of simple strokes it would be better to represent characters as a union of strokes instead of just giving the overall outline. This would avoid many discontinuities in the bounding contours. Secondly, we should take into account the distribution of p_i. They are most likely to refer to positions near the current position (i.e., $\Sigma(i)$) and possibly tend to cluster in other places as well. This would be particularly important if the characters were being expressed as the union of their strokes. Some kind of escape sequence or simple paging scheme could probably do this without much additional overhead. The preambles in the existing format are blatantly inefficient. It would be particularly important to improve this if we had separate boundaries for each stroke. Finally, if we relax our standards for perfect replication we could probably achieve significant gains through a kind of smoothing process. In the string matching procedure we could look for ways to lengthen the matches by changing a few boundary pixels. This could be made not to damage the appearance of the characters and would achieve better compression without increased decoding time.

Acknowledgement
We would like to thank Donald E. Knuth for his support.

References

1. (Table of Chinese Character Frequency) 土京新華印刷厂 1975.
2. P. H. Couegnoux, Character Generation by Computer, *Computer Graphics and Image Processing* **16** (1981) 240–289.
3. (internal report) *Shanghai Printing Technology Institute* (1981)
4. (The Fundamental Collection of Chinese Character Codes for Information Interchange of P.R.C.) GB–2312–80 (1981)
5. Even, S., and Rodeh, M. Economical Encodings of Commas Between Strings, *Commun. ACM* **21** (April 1978), 315–317.
6. Even, S., Pratt, V., and Rodeh, M. Linear Algorithm for Data Compression via String Matching, *J. ACM* **28**,1 (January 1981), 16–24.
7. McCreight, E. M., A Space Economical Suffix Tree Construction Algorithm, *J. ACM* **23**, 2 (April, 1976), 262–272.

8. Tung Yun Mei, LCCD, A Language for Chinese Character Design, *Software* **11** (December 1981), 1273–1292.
9. Knuth, D. E., Supernatural Numbers, *The Mathematical Gardner*, Wadsworth International, Belmont, Ca., 1981, 312–325.


LME
.8