Contents

CHAPTER 55

IMAGE COMPRESSION

55.1 IMAGE COMPRESSION

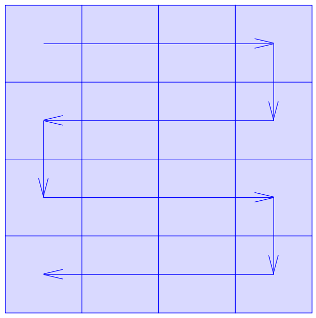
A digitized image is an $m \times m$ matrix of pixels. In this section, we assume that each pixel has a gray value between 0 and 255. So it takes at most 8 bits to store a pixel. If each pixel is stored using the maximum of 8 bits, the total space needed is $8m^2$ bits. We can reduce the storage needs by using a **variable bit scheme** in which different pixels are stored using a different number of bits.

Pixel values 0 and 1 need only 1 bit each; values 2 and 3 need 2 bits each; values 4, 5, 6, and 7 need only 3 bits each; and so forth. When this variable bit scheme is used, we go through the following steps:

1. [Linearize image] The $m \times m$ image matrix is converted into a $1 \times m^2$ matrix using (say) the snakelike row-major ordering in Figure 55.1(a).

2. [Create segments] The pixels are divided into segments such that the pixels in each segment require the same number of bits. Each segment is a contiguous chunk of pixels, and segments are limited to 256 pixels. If there are more than 256 contiguous pixels with the same bit requirement, they are represented by two or more segments.

3. [Create files] Three files $SegmentLength$, $BitsPerPixel$, and $Pixels$ are created. The first of these files contains the length (minus one) of the segments created in step 2. Each entry in this file is 8 bits long. The file $BitsPerPixel$ gives the number of bits (minus one) used to store each pixel in the segment.



(a) Snakelike row-major order

10	9	12	40
12	15	35	50
8	10	9	15
240	160	130	11

(b) Grayscale values

Figure 55.1 A digitized image

Each entry in this file is 3 bits long. The file *Pixels* is a binary string of pixels stored in the variable-bit format.

4. [Compress files] Each of the three files created in 3 is compressed to reduce its space requirements.

The performance of this compression scheme (in terms of achieved compression ratio) depends very much on the presence of long segments that require a small number of bits.

Example 55.1 Consider the 4×4 image of Figure 55.1(b). The gray values in snakelike row-major order are 10, 9, 12, 40, 50, 35, 15, 12, 8, 10, 9, 15, 11, 130, 160, and 240. The number of bits needed to store each of these pixels is 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 8, 8, and 8, respectively. When the pixels are divided into segments of equal bit requirements, we get the four segments [10, 9, 12], [40, 50, 35], [15, 12, 8, 10, 9, 15, 11], and [130, 160, 240].

The file *SegmentLength* is 2, 2, 6, 2 and *BitsPerSegment* is 3, 5, 3, 7. The file *Pixels* contains the 16 gray values in snakelike row-major order. The first three are stored using 4 bits each, the next three using 6 bits each, the next seven using 4 bits each, and the last three using 8 bits each. The first 30 bits of this file represents the first six pixels. These 30 bits are

1010 1001 1100 111000 110010 100011

The space needed to store the three files is 32 bits for *SegmentLength*, 12 bits for *BitsPerSegment*, and 82 bits for *Pixels*. The total is 126 bits. If the image was stored using a fixed 8 bits per pixel, the storage requirements would be $8 \times 16 = 128$ bits. For the sample image, we achieve a modest saving of 2 bits. ■

Suppose that following step 2, we have n segments. The length of a segment and the bits per pixel for that segment are referred to as the **segment header**. Each segment header needs 11 bits of space. Let l_i and b_i, respectively, denote the length and bits per pixel for segment i. The space needed to store the pixels of segment i is $l_i * b_i$. The total space required for the three files created in step 2 is $11n + \sum_{i=1}^{n} l_i b_i$. The space requirements can be reduced by combining some pairs of adjacent segments into one. If segments i and $i+1$ are combined, then the combined segment has length $l_i + l_{i+1}$. Each pixel now has to be stored using $\max b_i,\ b_i + 1$ bits. Although this technique increases the space needed by the file *Pixels*, it reduces the number of headers by one.

Example 55.2 If we combine segments one and two of Example 55.1, the *SegmentLength* file becomes 5, 6, 2 and *BitsPerSegment* becomes 5, 3, 7. The first 36 bits of *Pixels* now represent the new first segment. These bits are

001010 001001 001100 111000 110010 100011

The remainder of *Pixels* is unchanged. The space needed by the files *SegmentLength* and *BitsPerPixel* has decreased by 11 bits as the number of headers is one less than before. The space needed by *Pixels* has increased by 6 bits for a net savings of 5 bits. The total space requirements are now 121 bits. ■

In this section we wish to develop an algorithm to take the n segments created in step 2 and combine adjacent segments so as to produce a new set of segments that has minimum space requirements. After we combine the segments, we can use other techniques such as the LZW method (Section 11.6) and Huffman coding (Section 13.6.3) to further compress the three files.

Let s_q be the space requirements for an optimal combining of the first q segments. Define $s_0 = 0$. For an instance with $i > 0$ segments, suppose that, in an optimal combining C, segment i is combined with segments $i-1$, $i-2$, $\cdots$, and $i-r+1$, but not with segment $i-r$. The space s_i needed by the combining C is

$$\text{space needed by segments 1 through } i-r + lsum(i-r+1, i) * bmax(i-r+1, i) + 11$$

where $lsum(a,b) = \sum_{j=a}^{b} l_j$ and $bmax(a,b) = \max\{b_a, \cdots, b_b\}$. If segments 1 through $i-r$ are not combined optimally in C, then we change their combining to one with a smaller space requirement and hence reduce the space requirement of C. So in an optimal combining C, segments 1 through $i-r$ must also be combined optimally. That is, the principle of optimality holds. With this observation, the space requirements for C become

$$s_i = s_{i-r} + lsum(i-r+1, i) * bmax(i-r+1, i) + 11$$

The only possibilities for r are the numbers 1 through i for which $lsum$ does not exceed 256 (recall that segment lengths are limited to 256). Although we do not know which is the case, we do know that since C has a minimum space requirement, r must yield the minimum space requirement over all choices. So we get the recurrence

$$s_i = \min_{\substack{1 \le k \le i \\ lsum(i-k+1,i) \le 256}} \{s_{i-k} + lsum(i-k+1,i) * bmax(i-k+1,i)\} + 11 \tag{55.1}$$

Let kay_i denote the value of k that yields the minimum. s_n is the space requirement of an optimal combining of the n segments, and an optimal combining may be constructed using the kay values.

Example 55.3 Suppose that five segments are created following step 2. Let their lengths be [6, 3, 10, 2, 3] and let their bit per pixel requirements be [1, 2, 3, 2, 1]. To compute s_n using Equation 55.1, we need the values of $s_{n-1}, \cdots, s_0$. s_0 is zero. For s_1, we get

$$s_1 = s_0 + l_1 * b_1 + 11 = 17$$

and $kay_1 = 1$. s_2 is given by

$$\begin{aligned} s_2 &= \min\{s_1 + l_2 b_2, s_0 + (l_1 + l_2) * \max\{b_1, b_2\}\} + 11 \\ &= \min\{17 + 6, 0 + 9 * 2\} + 11 \\ &= 29 \end{aligned}$$

and $kay_2 = 2$. Continuing in this way, we obtain $s_1 \cdots s_5 = [17, 29, 67, 73, 82]$ and $kay_1 \cdots kay_5 = [1, 2, 2, 3, 4]$.

Since $s_5 = 82$, the optimal space combining uses 82 bits of space. We can determine this combining by beginning at kay_5. Since $kay_5 = 4$, s_5 was obtained from Equation 55.1 by setting $k = 4$. The optimal combining consists of the optimal combining for segments 1 through $(5-4) = 1$ followed by the segment that results from combining segments 2, 3, 4, and 5. We are left with just two segments, the original segment 1 and the combination of segments 2 through 5. ■

Recursive Solution

Equation 55.1 may be solved recursively for s_i and kay_i. We employ a public static method which sets us the class data members and then invokes a private recursive method that does the actual computation. Program 55.1 gives the private recursive method to compute the s_is and kay_is, as well as a recursive traceback method which computes the optimal combining from the kay_i values.. `l`, `b`, and `kay` are one-dimensional integer arrays that are class data members; `maxLength` and `header` are integer class data members. `maxLength` is the segment length restriction (256), and `header` is the space needed for a segment header (11). The invocation `s(n)` returns the value of s_n and also sets the `kay` values. The invocation `traceback(n)` outputs the optimal combining.

As for the complexity $t(n)$ of Program 55.1, we see that $t(0) = c$ for some constant c and $t(n) \leq \sum_{j=\max\{0,n-256\}}^{n-1} t(j) + n$ when $n > 0$. The solution to this recurrence is $t(n) = O(2^n)$. The complexity of `traceback` is $O(n)$.

Recursive Solution without Recomputations

The complexity of Program 55.1) can be reduced to $O(n)$ by avoiding the recomputation of previously computed s_is. Notice that there are only n different s_is.

Example 55.4 Consider the five-segment instance of Example 55.3. When computing s_5, recursive calls are made to compute s_4, $\cdots$, s_0. When computing s_4, recursive calls are made to compute s_3, $\cdots$, s_0. Therefore, s_4 is computed once, and s_3 twice. Each computation of s_3 computes s_2 once, so s_2 is computed a total of four times. s_1 is computed 16 times! ■

We can avoid the recomputation of the s_i's by saving previously computed s_is in a one-dimensional array `s`. The new code for method `s` appears in Program 55.2. The one-dimensional integer array `s` is a class data member.

To determine the time complexity of Program 55.2, we shall use an **amortization scheme** in which we charge different components of the total time to different entities and then add up the charges for these entities. When computing an s_i, the cost of an invocation `s(j)` is charged to s_j if s_j has not been computed and to s_i otherwise. (This s_j in turn offloads the cost of computing new s_qs to the individual s_qs.) The cost of the remainder of Program 55.2 is charged to s_i. This remainder is $\Theta(1)$ because `maxLength` is a constant (256) and each l_i is at least one. The total amount charged to each s_i is constant, and the number of s_is is n. Therefore, the total work done is $O(n)$.

Iterative Solution

A $O(n)$ iterative solution is obtained if we use Equation 55.1 to compute $s_1, \cdots, s_n$ in this order. This way, when an s_i is to be computed, the needed s_js have already been computed. The resulting code appears in Program 55.3.

```
/** recursively compute s(i) and kay[i] using the
  * dynamic programming recurrence */
private static int s(int i)
{
   if (i == 0)
      return 0;

   // compute min term of Eq. 15.3 for k = 1
   int lsum = l[i],
       bmax = b[i];
   int best = s(i - 1) + lsum * bmax;
   kay[i] = 1;

   // compute min term for remaining k and find min
   for (int k = 2; k <= i && lsum + l[i - k + 1] <= maxLength; k++)
   {
      lsum += l[i - k + 1];
      if (bmax < b[i - k + 1])
         bmax = b[i - k + 1];
      int t = s(i - k);
      if (best > t + lsum * bmax)
      {// found a smaller term
         best = t + lsum * bmax;
         kay[i] = k;
      }
  }

   return best + header;
}

/** output the optimal segment boundaries */
public static void traceback(int [] kay, int n)
{
   if (n == 0)
      return;
   traceback(kay, n - kay[n]);
   System.out.println("New segment begins at " + (n - kay[n] + 1));
}
```

Program 55.1 Recursive computation of s, kay, and an optimal combining

```
private static int s(int i)
{
   if (i == 0)
      return 0;

   if (s[i] > 0)  // s(i) has been computed before
      return s[i];

   // s(i) has not been computed before, compute it now
   // compute min term of Eq. 15.3 for k = 1
   int lsum = l[i],
       bmax = b[i];
   int best = s(i - 1) + lsum * bmax;
   kay[i] = 1;

   // compute min term for remaining k and find min
   for (int k = 2; k <= i && lsum + l[i - k + 1] <= maxLength; k++)
   {
      lsum += l[i - k + 1];
      if (bmax < b[i - k + 1])
         bmax = b[i - k + 1];
      int t = s(i - k);
      if (best > t + lsum * bmax)
      {// found a smaller term
         best = t + lsum * bmax;
         kay[i] = k;
      }
  }

   s[i] = best + header;
   return s[i];
}
```

Program 55.2 Recursive computation avoiding recomputations

EXERCISES

1. When the restriction on segment length is eliminated (i.e., $maxLength = \infty$ in Program 55.1), the time complexity of Program 55.1 is given by the recurrence $t(0) = c$ for some constant c and $t(n) = \sum_{j=0}^{n-1} t(j) + n$ when $n > 0$.

   (a) Use the fact that $t(n-1) = \sum_{j=0}^{n-2} t(j) + n - 1$ to conclude that $t(n) =$

```
public static void vBits(int [] l, int [] b, int [] s, int [] kay)
{
   int n = l.length - 1;   // number of segments
   s[0] = 0;
   // compute s[i] using Eq. 15.3
   for (int i = 1; i <= n; i++)
   {
      // compute min term for k = 1
      int lsum = l[i],
          bmax = b[i];
      int best = s[i - 1] + lsum * bmax;
      kay[i] = 1;

      // compute for remaining k and update
      for (int k = 2; k <= i && lsum + l[i - k + 1] <= maxLength; k++)
      {
         lsum += l[i - k + 1];
         if (bmax < b[i - k + 1])
            bmax = b[i - k + 1];
         if (best > s[i - k] + lsum * bmax)
         {
            best = s[i-k] + lsum * bmax;
            kay[i] = k;
         }
      }

      s[i] = best + header;
   }
}
```

Program 55.3 Iterative computation of `s` and `kay`

$2t(n-1)+1$ for $n > 0$.

(b) Now show that $t(n) = O(2^n)$.

2. Write an iterative version of method `traceback` (Program 55.1). What can you say about the relative merits of the two versions?

3. Write code for steps 1 and 2 of the variable-bit image-compression scheme.

55.2 REFERENCES AND SELECTED READINGS

More information on image compression methods is provide in the book *Image and Video Compression Standards* by V. Bhaskaran and K. Konstantinides, Kluwer Academic Publishers, Boston, 1995. The variable bit scheme of Section 55.1 is from "State of the Art Lossless Image Compression" by S. Sahni, B. Vemuri, F. Chen, C. Kapoor, C. Leonard, and J. Fitzsimmons, Technical Report, University of Florida, 1997.