The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

April 2000
Number 59

In this issue

Floats 1
Satin 4
Tie-ups 5
Classical Cryptography 7
Subscription Renewal 9
Weavable Color Patterns 10
Understanding Icon's Linker 16
Recurrence Relations 18
What's Coming Up 20

Floats

An aspect of weaving that is of great practical importance is the strength and durability of the fabric produced, which depends not only on the kind of threads used and how tight the weave is but also on the manner of interlacing.

The interlacing that gives the strongest fabric is the one used in plain weave, also called tabby, which has a strict 1-over, 1-under interlacement pattern. Figure 1 shows an example.

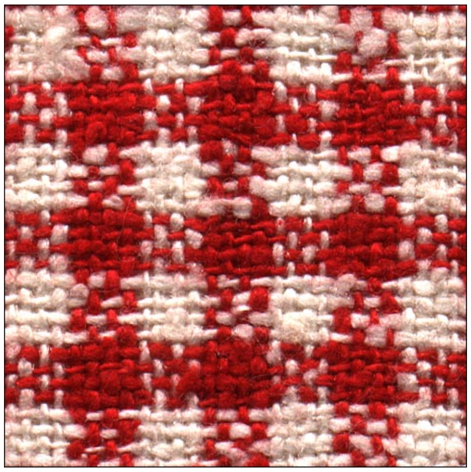


Figure 1. A Plain Weave Fabric

When a thread passes over or under more than one perpendicular thread, the result is called a *float*.

A fabric with long floats does not have the strength and durability of one with short floats or none at all. Not only is there less interlacement, but floats tend to snag.

The importance of snagging depends on the use to which a fabric is put. Long floats sometimes appear on the back of upholstery fabrics, where they cause few problems.

On the other hand, floats allow the creation of textures and patterns that cannot be achieved otherwise. Floats also produce a surface that feels smoother and drapes better than a float-free one. The sheen and luxurious texture of satins is due to their floats.

Figure 2 shows an example in which floats are used to achieve a decorative effect. Notice that one thread has pulled loose where there is a float.

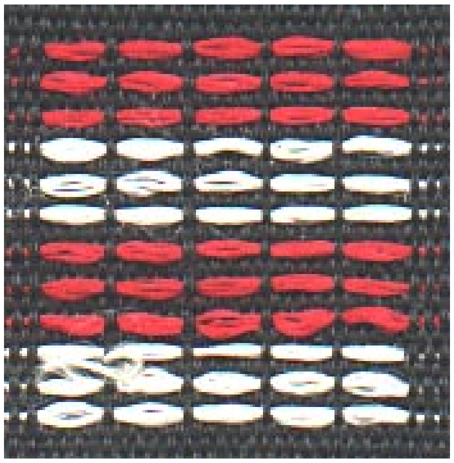


Figure 2. Floats on a Decorative Fabric

Drafts that produce very long floats are undesirable or even unweavable. Unintended floats can occur when designing drafts. For example, in the drawdown shown in Figure 3, there are places where weft threads float all the way across the fabric.

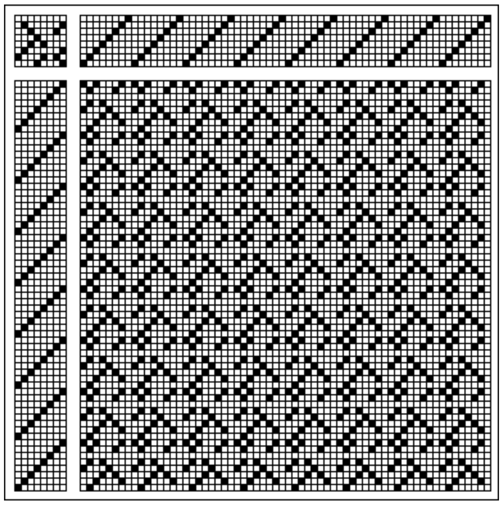


Figure 3. An Unweavable Draft

If an attempt were made to weave this fabric, these weft threads would not be interfaced with the warp at all and would not be attached to the fabric — the fabric would be unweavable.

Weavers avoid such extremes intuitively, but they have to watch for float problems nonetheless. Weavers have methods of modifying drafts with this kind of problem without affecting the appearance of the woven fabric [1]. For example, tiny threads, which are too fine to be noticed, can be used as "incidentals" to add interlacement.

We've ignored the problem of floats in the drafts we've shown in previous articles, since it's not a problem in creating images. Our "virtual" weaving is much easier than real weaving. (There is an interpretation of weaving drafts as patterns in which the interlacement is not a concern. That's on our agenda for a future article.)

Weaving programs have various ways of showing floats. WeaveMaker One uses "float indicators" — over-and-under diagrams for selected warp and weft threads. See Figure 4. The selected threads are indicated by small markers at the top and right edges of the drawdown. We've added arrows to help locate them. The markers can be moved to show the floats for other threads.

A fabric surface simulation, such as the one shown in Figure 5 (also from WeaveMaker One), provides a more intuitive but less precise view of floats.

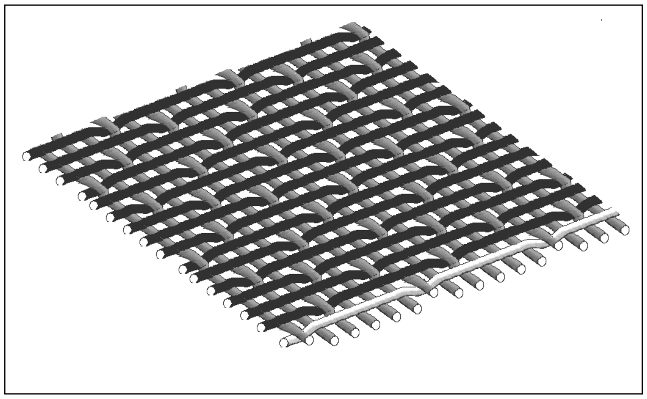


Figure 5. Fabric Surface Simulation

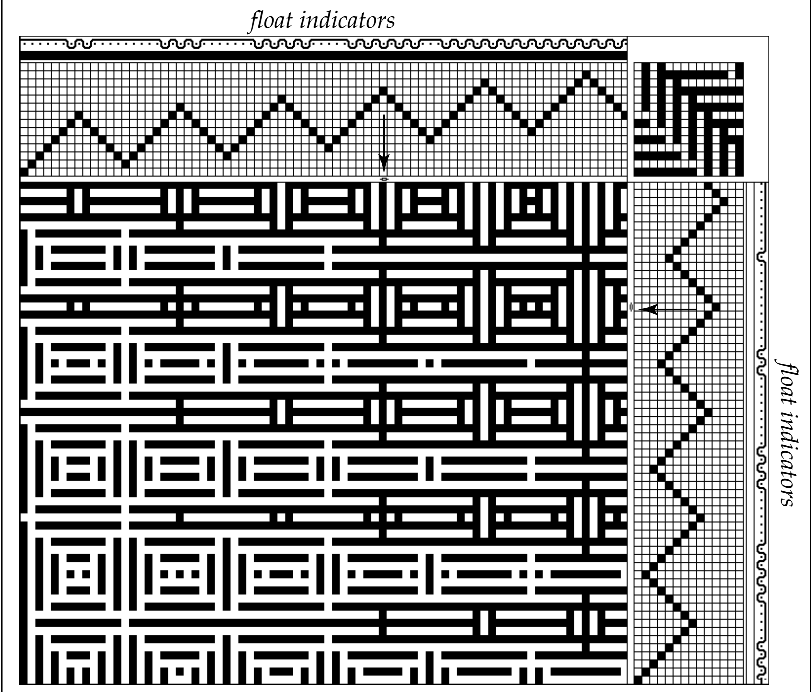


Figure 4. Float Indicators

Creating such diagrams is fairly difficult and few weaving programs attempt it. Realistic rendering of fabric surfaces is in another class altogether.

Most weaving programs can produce tabulations or histograms of the number of floats by length. Figure 6 shows a WeaveMaker One histogram for the draft shown in Figure 3:

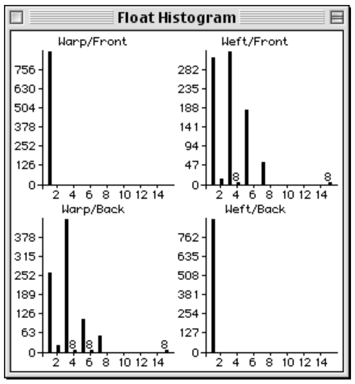


Figure 6. Float Histogram

Floats of length 15 and longer are lumped together without comment. Note that 1 is included, although strictly speaking it is not a float.

Here's a program to produce a float tabulation. It works from an image string of a drawdown. A drawdown image for the back of the fabric is created by exchanging black and white pixels — where a thread is on top on the face of the fabric, it is in back on the back of the fabric. Weft floats are determined by rotating the fabric by 90º.

```
link imrutils

procedure main()
   local front, back, black, white

   front := imstoimr(read()) |
      stop("***cannot create image record")

   black := PaletteKey(front.palette, "black")
   white := PaletteKey(front.palette, "white")

    analyze("Front weft floats", front, white)

   front := imrrot90cw(front)

   analyze("Front warp floats", front, black)

   back := imrcopy(front)
   back.pixels := map(back.pixels,
      white || black, black || white)

   analyze("Back weft floats", back, white)

   back := imrrot90cw(back)

   analyze("Back warp floats", back, black)

end
```

```
procedure analyze(caption, imr, color)
   local counts, length, row

   counts := table(0)

   imr.pixels ? {
      while row := move(imr.width) do {
         row ? {
            while tab(upto(color)) do {
               length := *tab(many(color))
               if length > 1 then
                  counts[length] +:= 1
               }
            }
         }
      }

   if *counts = 0 then fail          # no output

   write(caption)

   counts := sort(counts, 3)

   write()

   while write(right(get(counts), 6),
      right(get(counts), 6))

   write()

   return

end
```

The output for the drawdown shown in Figure 3 is:

```
Front weft floats

     2    16
     3   328
     4     8
     5   184
     7    56
    64     8

Back warp floats

     2    24
     3   440
     4     8
     5   112
     6     8
     7    56
    64     8
```

Reference

1. *Designing for Weaving: A Study Guide for Drafting, Design and Color*, Carol S. Kurtz, Hastings House, 1981.

Satin

Satin usually is thought of as a kind of silk fabric that is characterized by a glossy surface and a smooth texture. Weavers, on the other hand, consider satin to be a weave structure — a system for interlacement — that is not associated with any particular kind of fiber.

The conflict of meaning came into focus with the introduction of fabrics called satins but made with rayon. This use of the word led to a legal dispute, which eventually was settled (in the United States, at least) in 1930 by the Circuit Court of Appeals, which ruled satin was the name of a weave construction and not the name of a textile made from any particular fiber [1].

The common perception remains largely unchanged, however.

As a weave structure, satin is characterized by long floats (which gives satin fabrics their smoothness) and a system of interlacing that avoids the regularity of twills [2].

This is accomplished by having only one warp interlacement with the weft per shaft. These interlacements are arranged so that no two are adjacent on successive treadlings. Figure 1 shows an example of a satin tie-up.

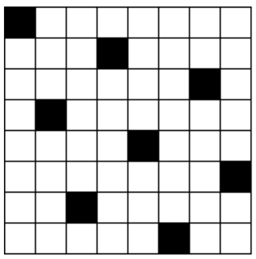


Figure 1. A Satin Tie-Up

The location of interlacement points is determined by a *counter* that depends on the number of shafts used.

Weaving literature is not noted for its clarity and precision. Here are four quotations on the subject from sources dating from 1888 to 1994:

> 1. *Divide the number of harness for the satin into two parts, which must neither be equal nor the one a multiple of the other; again it must not be possible to divide both parts by a third number.* [3]

Harness is used as a collective noun and corresponds to shafts in current terminology.

> 2. *Divide the number of ends (or shafts) on which the satin … is to be woven into two unequal parts, so that one shall not be a measure of the other, nor shall it be divisible by a common number.* [4]

The word ends means the number of warp threads. "Measure of the other" is British English and means (we think) "divisible by the other".

> 3. *Find two numbers which give a sum equal to the number of frames. None of these numbers can be 1; the two numbers cannot divide one another, or by any other number at the same time.* [5]

The word frames is synonymous with shafts.

> 4. *The satin counter cannot be 1, or the interlacement forms a twill. It cannot be one fewer than the number in the unit … , or the interlacement forms a twill in the opposite direction. The counter cannot share a divisor with the number in the unit, or some warp threads interlace more than once and others not at all.* [6]

Grammatical errors, tortured prose, questionable meaning, and definition by elimination aside, it comes down to this:

> Given *n* shafts, find *i* and *j* such that i > 1, $i + j = n$, and the greatest common divisor of *i* and *j* = 1.

Either *i* or *j* can be used as the counter; the smaller one usually is chosen.

Well, that wouldn't make sense to most weavers either, and it's no wonder most of them rely on tables of satin counters. In one sense, tables aren't that bad: there are not that many different counters for the number of shafts that are available for hand looms. Here is a table for 2 to 24 shafts:

shafts	*small counters*	*number*
2		0
3		0
4		0
5	2	1
6		0
7	2 3	2
8	3	1
9	2 4	2
10	3	1
11	2 3 4 5	4
12	5	1
13	2 3 4 5 6	5
14	3 5	2
15	2 4 7	3

16	3 5 7	3
17	2 3 4 5 6 7 8	7
18	5 7	2
19	2 3 4 5 6 7 8 9	8
20	3 7 9	3
21	2 4 5 8 10	5
22	3 5 7 9	4
23	2 3 4 5 6 7 8 9 10 11	10
24	5 7 11	3

Notice that satin requires at least five shafts and cannot be woven with six shafts. By the way, if the number of shafts is a prime, $p > 2$, any number $2 \leq i \leq p-1$ is a valid counter: If $i + j = p$, $\gcd(i, j)$ must be 1 — otherwise the common factor would divide p.

But a weaver who just uses a table of counters, and who doesn't understand the formula or the reason for it, is limited to the designs of others.

Computing satin counters is easy. Here's a procedure that generates the smaller counter from each pair for a given number of shafts:

```
procedure satin_counter(shafts)
   local candidate

   every candidate := 2 to shafts / 2 do
      if gcd(candidate, shafts – candidate) = 1
         then suspend candidate

end
```

The procedure gcd() is in the Icon program library module numbers.

Once the counter is chosen, the tie-up is constructed starting with first position of the first row, adding the counter to that value modulo the number of shafts, using shaft arithmetic [7], to get the position in the second, and so on. Refer to Figure 1 on the previous page.

Here's a procedure that produces a satin tie-up as a row array:

```
procedure satin_tieup(counter, shafts, treadles)
   local rows, m, k

   rows := list(shafts, repl("0", treadles))

   m := 1
   rows[1, 1]  := "1"

   every k := 2 to shafts do
      rows[k, residue(m +:= counter, shafts, 1)] := "1"

   return rows

end
```

The procedure residue() is in the Icon program library module numbers.

References:

1. *Contemporary Satins*, Harriet Tidball, Shuttle Craft Guild Monograph Seven, 1962.

2. "Twills", Icon Analyst 58, pp. 1-2.

3. *Technology of Textile Design*, E. A. Posselt, 1888, p. 25.

4. *The Structure of Weaving*, Ann Sutton, Lark Books, 1982, p. 122.

5. *More About Fabrics*, S. A. Zielinski, Master Weaver Library, Vol. 20, LeClerc, 1985, p. 14.

6. *The Complete Book of Drafting for Handweavers*, Madelyn van der Hoogt, Shuttle Craft Books, 1994, p. 23.

7. "Shaft Arithmetic", Icon Analyst 57, pp. 1-5.

Tie-Ups

Depending on what you read, the tie-up in a weaving draft is the essence of the draft or it's just something that is produced mechanically after a weave structure is designed. Both are true in different contexts.

We've shown how to derive the threading, treadling, and tie-up from a drawdown — an interlacement pattern. This is a case when the tie-up follows as a matter of course.

On the other hand, as we've shown for twills and satins (see the article **Satins** that starts on page 4), tie-ups can be fundamental to design and work for many kinds of threadings and treadlings.

A tie-up also can serve as a motif, such as a diagram of a leaf, that is replicated in the drawdown. Geometric designs also are used in tie-ups to achieve certain kinds of effects. Some tie-ups are specific to certain kinds of weaving.

Many tie-ups are derived from others. And there's the ever-present miscellaneous category.

Basic Tie-Ups

The basic tie-ups are direct, tabby, twill, and satin.

The figures that follow are for eight shafts and treadles. The principles apply equally well to other numbers of shafts and treadles.

A direct tie-up consists of tie-up points in a diagonal line. See Figure 1.

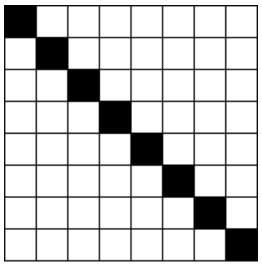


Figure 1. Direct Tie-Up

Direct tie-ups always are square, with the same numbers of shafts and treadles.

We'll consider the effect of direct tie-ups in a later article on the interaction of tie-ups with threading and treadling sequences.

A tabby tie-up, used to produce plain weaves, is a simple checkerboard as shown in Figure 2 .

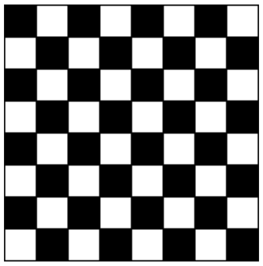


Figure 2. Tabby Tie-Up

We covered twill tie-ups in an earlier article [1]. Figure 3 shows an example for reference.

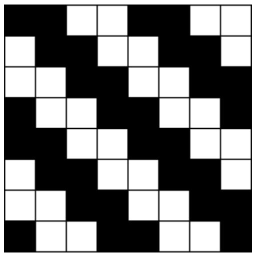


Figure 3. A /2/2 Twill Tie-Up

For classification purposes, tabby might be considered to be a /1/1 twill, but weavers think in terms of texture, and plain weave does not have the diagonal texture associated with twill.

Satin tie-ups break the diagonal effect of twills. See the article **Satin**, which begins on page 4. Figure 4 provides an example to compare with the other types here.

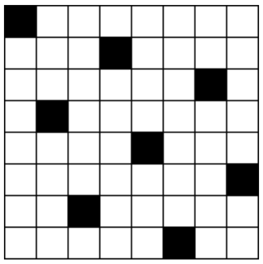


Figure 4. A Satin Tie-Up

Satin can be thought of as a kind of twill in which the shift is greater than one. But, again, it does not have the diagonal appearance expected of twill; it fact, it is designed *not* to have such an appearance.

The basic tie-ups shown above can be characterized by a "under-and-over" row pattern that is circularly shifted by a fixed number of columns for each successive row. A positive shift is to the right; negative to the left. If the sum of the numbers in a pattern is less than the number of treadles, the pattern is extended to fill out the row.

Direct, tabby, and twill tie-ups are circularly shifted by one column for each successive row (in twill tie-ups, positive and negative shifts produce different but complementary effects). In satin tie-ups the shift is chosen to break the diagonal regularity of twills.

Here are the values for the tie-ups in the figures given previously:

figure	*pattern*	*shift*	*type*
1	/1/7	1	direct
2	/1/1	1 (or –1)	tabby
3	/2/2	1	twill
4	/3/5	11	satin

We can represent such tie-ups by strings in which the pattern is separated from the shift by a colon:

- direct: "1/*n*–1:1", where *n* is the number of shafts
- tabby: "/1/1:1"
- twill: "*p*:1"
- satin: "1/*n*–1:*n*+*c*" where *n* is the number of shafts and *c* is the counter

This form of characterization invites the design of other tie-ups that do not fall into any of the basic types given above. Figures 5 through 8 show some examples.

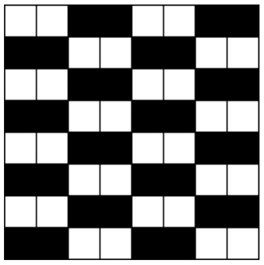


Figure 5. /2/2:2 Tie-Up

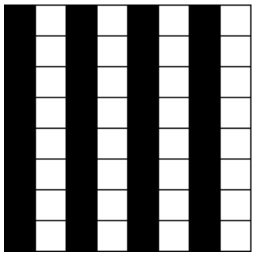


Figure 6. /1/1:0 Tie-Up

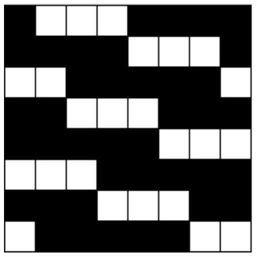


Figure 7. /1/3/4:3 Tie-Up

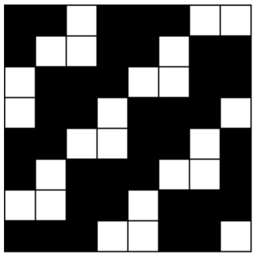


Figure 8. /2/1/3/2:5 Tie-Up

Be aware that simply coming up with a new tie-up using this formula does not insure a weave using it will be interesting or structurally sound.

More to Come

In the next article on tie-ups, we'll consider derivatives of the basic ones.

Reference

1. "Twills", Icon Analyst 58, pp. 1-2.

Classical Cryptography

Cryptography deals with methods for encoding messages to hide their meaning, decoding the results to recover the messages, and "cracking" coded messages when the encoding method is not known.

Classical cryptography deals with the subject from ancient times until approximately World War I. It does not include sophisticated devices developed this century or more recent mathematical methods like public-key encryption.

In our articles on the subject, we'll restrict ourselves to methods that manipulate strings and not attempt to cover various mechanical devices that have been used.

Terminology

Before going on, we need to define some terms. Some of these terms are used inconsistently in the literature; what follows are the meanings we will use.

A message in its unencoded form is called *plain text*. The coded form is called a *cryptogram*. A *cipher* is a method of coding. The process of encoding plain text is called *enciphering*, while the process of decoding a cryptogram is called *deciphering*. Ciphers often use a *key* to parameterize the basic method.

Decryption is the process of decoding a message without knowing the cipher and/or key. We'll be concerned primarily with ciphers and deal with decryption only tangentially.

The Nature of Ciphers

Abstractly, deciphering is the inverse of enciphering — recovering the plain text from the cryptogram. In practice, many ciphers discard information that is not essential to the message or add material that is not relevant or even intended to be confusing to a would-be decryptor. We'll generally deal with the ideal situation in which no information is lost in the process of enciphering and deciphering; that is, when deciphering is the inverse of ciphering:

```
decipher(encipher(pt, key), key) == pt
```

Figure 1 shows the situation schematically.

Kinds of String Ciphers

Most string ciphers fall into one of two categories or a combination of them [1]:

- substitution
- transposition

Substitution replaces characters or combinations of characters with others in plain text. Transposition rearranges the characters.

Substitution Ciphers

Substitution raises the question of the *alphabets* used in the enciphering and deciphering processes. In most textbook examples, the alphabet for the plain text and cryptograms is restricted to the uppercase letters. In practice, there is no reason to be so limited: numbers, spaces, and punctuation often are needed and most methods place no inherent restriction on the characters used.

Another way of dealing with this issue is to leave unchanged characters in plain text that are not in the alphabet used.

In our description of substitution ciphers, we'll use an alphabet limited to letters and just not make substitutions for other characters.

In any event, an alphabet is an *ordered* sequence of characters — a string.

Simple Monoalphabetic Substitution Ciphers

A simple substitution cipher is one in which plain text characters in an *input* alphabet are replaced on a one-to-one basis by characters in *output* alphabets to form a cryptogram. The keys for the cipher are the alphabets.

A monoalphabetic substitution cipher uses only one output alphabet. This is, of course, just what map() does. Procedures to implement a simple monoaphabetic cipher might look like this:

```
procedure encipher(plain, in, out)
   return map(pt, in, out)
end
procedure decipher(crypto, out, in)
   return map(crypto, out, in)
end
```

where in and out are the keys.

It is commonly assumed that in is globally known and the key is just out. In this case the procedures might be cast as

```
procedure encipher(plain, out)
   return map(pt, in, out)
end
procedure decipher(crypto, out)
   return map(crypto, out, in)
end
```

The easiest way to create a simple monoalphabetic substitution cipher is to use an output alphabet that is a rearrangement of the input al-

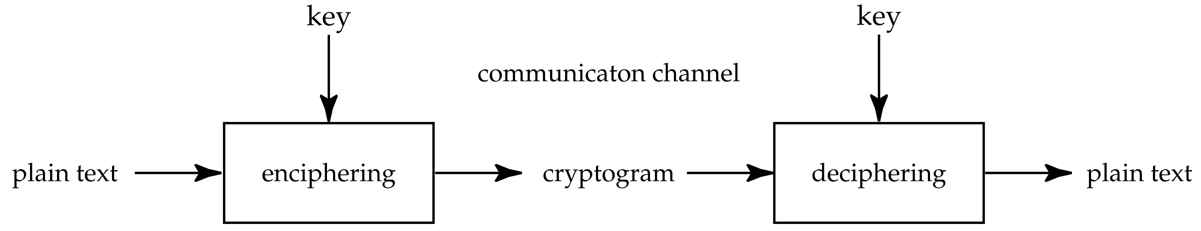


Figure 1. Enciphering and Deciphering

phabet. Reversal is commonly used to illustrate this:

```
input alphabet:     "abc … xyz"
output alphabet:    "zyx … cba"
```

That is, all as in the plain text are replaced by zs, all bs by ys, and so on.

In the case where the output alphabet is a rearrangement of the input alphabet, the key can be the rearrangement method, which usually is simpler and shorter than the resulting output alphabet. Procedures might look like this:

```
procedure encipher(plain, p)

  return map(pt, in, p(in))

end

procedure decipher(crypto, p)

  return map(crypto, p(in), in)

end
```

where p() is a procedure applied to the (known) input alphabet to get the output alphabet. For example,

```
crypto := encipher(plain, reverse)
```

enciphers plain using the reversal of the input alphabet as the output alphabet.

For some such ciphers, arguments to the procedure for forming the output alphabet may be needed. In this case our model can be extended as follows:

```
procedure encipher(plain, p, args[])

  return map(pt, in, p ! args)

end

procedure decipher(crypto, p, args[ ])

  return map(crypto, p ! args, in)

end
```

For example,

```
crypto := encipher(plain, rotate, 3)
```

uses the input alphabet circularly rotated by three characters as the output alphabet. This is known as Caeser's cipher because it was used by Julius Caesar.

The procedure and its arguments can be encapsulated in a single key as follows:

```
procedure encipher(plain, key[])
  local p

  p := get(key)
  push(key, in)

  return map(pt, in, p ! key)

end

procedure decipher(crypto, key[ ])
  local p

  p := get(key)
  push(key, in)

  return map(crypto, p ! key, in)

end
```

One can imagine all kinds of ways of constructing simple monoalphabetic substitution ciphers. They are easily broken, however, by using tables of known letter frequencies for material written in the input alphabet.

Next Time

The next more sophisticated approach is to use more than one output alphabet — so-called polyalphabetic substitution ciphers.

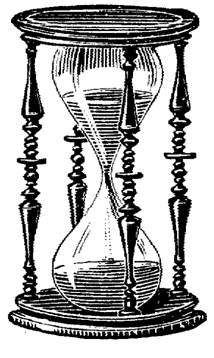
We'll start with this topic in the next article on classical cryptography and then move on to transposition ciphers.

Reference

1. *Cryptanalysis: A Study of Ciphers and Their Solution*, Helen Fouché Gaines, Dover, 1956.

Subscription Renewal

For many of you, your present subscription to the Analyst expires with the next issue. If so, you'll find a renewal form in the center of this issue.



Renew now so that you won't miss an issue.

Your prompt renewal helps us by reducing the number of follow-up notices we have to send. Knowing where we stand on subscriptions also lets us plan our budget for the next fiscal year.

Weavable Color Patterns (continued)

In the first article on this subject [1], we showed examples of small colored patterns that cannot be created by the interlacement of colored threads. We also showed a "forcing" pattern that provides a sufficient basis for the solution of larger patterns in which it is embedded. The forcing pattern is shown again for reference in Figure 1.

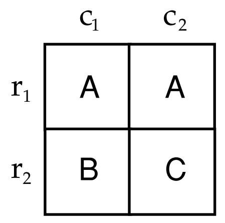


Figure 1. The Forcing Pattern

In Figure 1, c_1, c_2, and r_2 are not constrained but r_1 is completely determined and must be A for the entire pattern in which this subpattern is embedded.

In this article, we'll list the program and describe its more important components.

Data Structures

Columns and rows are treated alike as vectors, for which the record declaration is:

```
record vector(
   index,      # index of this row/column (1–based)
   label,      # row/column label: "rnnn" or "cnnn"
   mchar,      # character used in mapping
   cells,      # colors in row/column cells
   live,       # colors in active row/column cells
   fam,        # color family
   ignored     # non–null if solved or redundant
   )
```

Another record is used to represent the set of vectors that must be the same color:

```
record family
   vset,       # set of vectors
   color       # assigned color (null if not yet set)
   )
```

Two global variables, rows and columns, contain lists of the respective vectors. Each row and column is identified by a unique "mapping" character.

The pattern is represented using the c1 palette with its keys identifying the colors.

Program Organization

The program starts by reading in the pattern, which may be an image file or an image string, and then initializing the data structures.

Next, duplicate rows and columns, as well as solid-colored vectors, are "removed" by marking them "ignored". This may reduce the problem size significantly.

The main loop in the program then iterates over the pattern, developing constraints and setting colors determined by instances of the forcing pattern.

If at any time the pattern can be completely solved by arbitrarily assigning any remaining unspecified colors, the problem is solved. Otherwise, all 2×2 subpatterns are examined for instances of the forcing pattern. If a forcing pattern is found, the colors it forces are set and the loop continues.

If at any point there are no more instances of the forcing pattern, an attempt is made to assign colors to the remaining vectors arbitrarily. If this succeeds, the pattern is solved. If it fails, the pattern cannot be solved.

Procedures

Here's an overview of the procedures in the program:

- active() generates the active vectors in a list.
- addvector() adds a vector.
- chkforce() checks forced colorings.
- dupls() checks for duplicate vectors.
- quad() finds forcing patterns.
- samecolor() links two vectors that must be the same color.
- setcolor() sets a vector to a color and checks the consequences.
- setmaps() resets mapping strings for active vectors.
- setpattern() initializes the data structures.
- solids() checks for families of vectors that are all one color.
- success() reports a successful solution.
- trivial() determines if the pattern can be solved by arbitrary color assignment.

- vectmap() concatenates the mapping characters of active vectors.

Figure 1 shows a procedure call graph for the program.

Program Listing

Here's a somewhat abbreviated listing of the program. Initialization, diagnostics, information logging, and code to display the output have been omitted. The complete program is available on the Web site for this issue of the Analyst.

```
link graphics
link imscolor
link imsutils
link numbers
link options
link random

record vector(        # one row or column
   index,             # index of this row/column (1–based)
   label,             # row/column label: "rnnn" or "cnnn"
   mchar,             # char used in mapping
   cells,             # string of colors in row/column cells
   live,              # string of colors in active row/column cells
   fam,               # color family
   ignored            # non–null if solved or redundant
   )

record family(        # family of vectors that must be the same color
   vset,              # set of vectors
   color              # assigned color (null if not yet set)
   )
global opts           # command options
global imstring       # image string of original pattern specification
global data           # raw cell data
global rows           # list of row vectors
global cols           # list of column vectors
global mapchars       # string of chars used for col & row mapping
global rowvalid       # valid columns in row
global colvalid       # valid columns in column

procedure main(args)
   local n, v
         …                 # process options
         …                 # load image and check validity

   setpattern(imstring) | abort("can't parse pattern string")
   setmaps()               # initialize mapping strings

   while dupls(rows | cols) | solids() do
      setmaps()            # reduce problem size

   # check for quads until no longer worthwhile
   while (not trivial()) & quad(rows | cols) do
      setmaps()            # reduce problem size
```

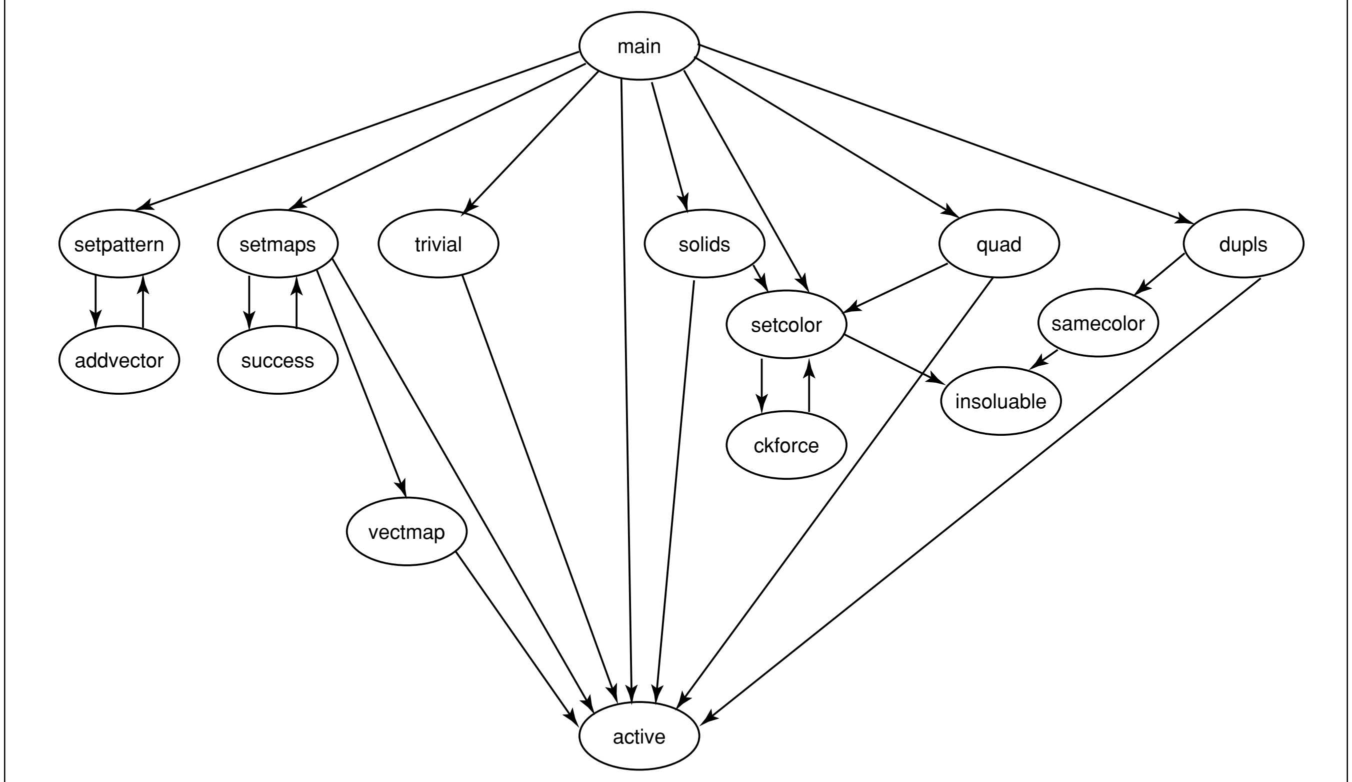


Figure 2. Program Call Graph

```
   every v := active(rows | cols) do        # solve or show
      setcolor(v, ?v.live)                  # impossible
   setmaps()                                # detect solved problem

end

#  active(vlist) –– generate vlist entries that are not being ignored

procedure active(vlist)
   local v

   every v := !vlist do
      if /v.ignored then suspend v
end

#  addvector(vlist, lchar, data) –– add new vector to vlist, labeled
#  with lchar

procedure addvector(vlist, lchar, data)
   local v, f

   v := vector()
   f := family()
   v.index := *vlist + 1
   v.label := lchar || v.index
   v.mchar := mapchars[*vlist + 1]
   v.cells := data
   v.fam := f
   f.vset := set()
   insert(f.vset, v)
   put(vlist, v)

   return

end

#  ckforce(v) –– check forced colorings of vectors intersecting v

procedure ckforce(v)
   local c, cs, vlist

   cs := &cset –– v.fam.color
   vlist := case v.label[1] of {
      "r":   cols
      "c":   rows
      }

   v.cells ? while tab(upto(cs)) do
      setcolor(vlist[&pos], move(1))

   return

end

#  dupls(vlist) –– check for duplicate (identical) vectors in a list;
#  succeeds if it accomplishes anything

procedure dupls(vlist)
   local s, t, v, w, n

   t := table()
   n := 0

   every v := active(vlist) do {
      s := v.cells
      if not (/t[s] := v) then {
         samecolor(t[s], v)
         v.ignored := 1                     # set inactive
         n +:= 1
         }
```

```
      }
   return 0 < n

end

#  dupls(vlist) –– check for duplicate (identical) vectors in a list;
#  succeeds if it accomplishes anything

procedure dupls(vlist)
   local s, t, v, w, n

   t := table()
   n := 0

   every v := active(vlist) do {
      s := v.cells
      if not (/t[s] := v) then {
         samecolor(t[s], v)
         v.ignored := 1            # set inactive
         n +:= 1
         }
      }

   return 0 < n

end
```

```
#  quad(vlist) –– find a 2x2 forcing subproblem; looks
#  for AABC pattern with AA oriented along one vector of vlist;
#  succeeds after finding one quad pattern and forcing colors.

procedure quad(vlist)
   local wlist, a, b, c, s, t, x1, x2, y1, y2, ss, ts

   every put(wlist := [], active(vlist))

   shuffle(wlist)                  # for better chance of quick solution

   every x1 := 1 to *wlist do {
      s := wlist[x1].live          # potential AA vector
      ss := cset(s)
      every x2 := (x1 ~= (1 to *wlist)) do {
         t := wlist[x2].live       # potential BC vector
         ts := cset(t)
         if *(ss ++ ts) < 3 then next
         every y1 := 1 to *s do {
            a := s[y1]
            b := t[y1]
            if a == b then next
            if *(ts –– a –– b) = 0
               then next
            every y2 := y1 + 1 to *s do {
               if s[y2] ~== a then next
               # now have found AA at subscripts y1, y2
               c := t[y2]
               if c == (a | b) then next
               setcolor(wlist[x1], a)
               return              # return after finding and forcing one
               }
            }
         }
      }

   fail

end

#  samecolor(v, w) –– link together two vectors that must be the
#  same color

procedure samecolor(v, w)
  local vfam, wfam, f, x

  vfam := v.fam
  wfam := w.fam
  if vfam === wfam then return

  if \vfam.color ~== \wfam.color then
    insoluble("cannot merge " || v.label || " and " || w.label)

  f := family()
  f.vset := vfam.vset ++ wfam.vset
  f.color := \vfam.color | \wfam.color | &null

  every x := !f.vset do
    x.fam := f

  return

end

#  setcolor(v, c) –– force vector v to color c, checking the
#  consequences

procedure setcolor(v, c)
  local f, fc
  static depth, todo

  initial {
    depth := 0
    todo := set()
    }

  f := v.fam
  fc := f.color
  if \v.ignored & fc === c then return

  if \fc ~== c then {
    f.color := &null
    insoluble(v.label || " cannot be both " || fc || " and " || c)
    }

  f.color := c
  v.ignored := 1                # set inactive
  insert(todo, v)               # but make note to check forcings

  if depth > 0 then             # avoid deep recursion
    return

  # check forcings only if not nested

  depth +:= 1

  while v := ?todo do {
    ckforce(v)
    delete(todo, v)
    }

  depth –:= 1

  return

end

#  setmaps() — recompute mapping strings for ignoring cols
#  and rows

procedure setmaps()
  local v

  rowvalid := vectmap(cols)
  colvalid := vectmap(rows)

  every v := active(rows) do
    v.live := map(rowvalid, mapchars[1+:*cols], v.cells)
  every v := active(cols) do
    v.live := map(colvalid, mapchars[1+:*rows], v.cells)

  if (*colvalid = 0) | (*rowvalid = 0) then success()

  return

end

#  setpattern(im) –– initialize pattern data from image string

procedure setpattern(im)
  local ncols, nrows, i, j, s

  mapchars := string(&cset)

  imstring := im
  ncols := imswidth(imstring) | fail
  nrows := imsheight(imstring) | fail
  data := (imstring ? 3(tab(upto(',')+1), tab(upto(',')+1), tab(0)))

  rows := [ ]

  data ? while addvector(rows, "r", move(ncols))

  cols := [ ]

  every i := 1 to ncols do {
    s := ""
    every j := i to *data by ncols do
      s ||:= data[j]
    addvector(cols, "c", s)
    }

  return

end

#  solids() –– check for families with remaining members all one
#  color; succeeds if it accomplishes anything

procedure solids()
  local f, v, n

  n := 0
  every v := active(rows) | active(cols) do {
    if *cset(v.live) = 1 then {
      setcolor(v, v.live[1])
      n +:= 1
      }
    }

  return 0 < n

end

#  success() –– report successful solution

procedure success()
  local v, r, c

  every v := !rows | !cols do           # set colors for don't–cares
    /v.fam.color := ?v.cells
        …                               # display solution
  exit()

end

#  trivial() –– succeed if this is a trivial case; a trivial case is one
#  that can be solved by coloring remaining vectors arbitrarily
# with any of the colors they contain (color one vector, force
# others, repeat until done)

procedure trivial()
  local c, s, cs, union, isectn

  if (*rowvalid < 3) & (*colvalid < 3) then
    return                      # trivial (2x2 or smaller)
  if (*rowvalid < 2 ) | (*colvalid < 2) then
    return                      # trivial (1xn)

  union := ''
  isectn := &cset

  every cs := cset(active(rows | cols).live) do {
    union ++:= cs
    isectn **:= cs
    }

  if *union < 3 then return     # trivial (bi–level or solid pattern)

# If a pattern can be permuted into a solid color except for
# one diagonal line (or parts of one), then it is trivially solved.

  if *isectn = 1 then {         # if single background color
    c := string(isectn)
    every s := active(rows | cols).live do {
      s ? {
        tab(many(c))
        move(1)
        tab(many(c))
        if not pos(0) then fail # if not a diagonal case
      }
    }
    return                      # trivial (diagonal case)
  }
  fail                          # not a trivial case
end

# vectmap(vlist) –– concatenate mapping chars of active
# vector entries

procedure vectmap(vlist)
  local s, v

  s := ""
  every v := active(vlist) do
    s ||:= v.mchar

  return s

end
```

Output

On completion, the program writes a line indicating whether or not the pattern could be solved. An enlarged version of the pattern then is displayed in a window with row and column color assignments along the top, bottom, and sides. If the pattern could not be solved, the colors just reflect the program state at termination. Figure 3 shows a solved color pattern. This image is much better viewed in color; see the Web site for this issue of the Analyst.

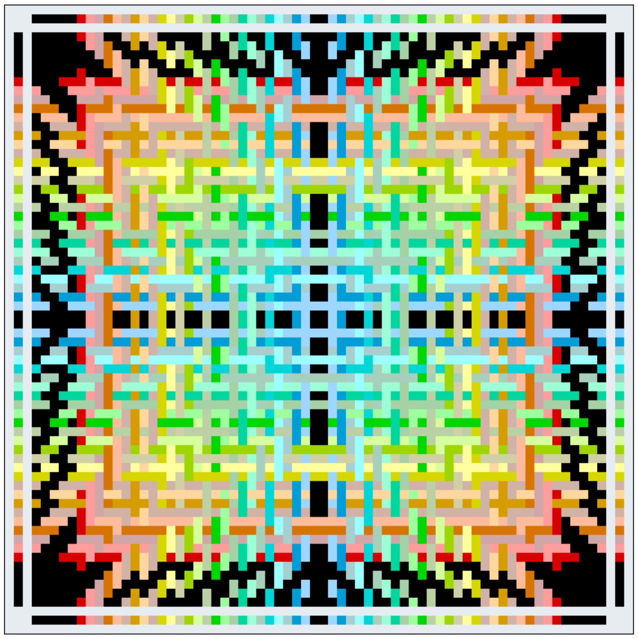


Figure 3. Solution

Command-Line Options

The program supports several command-line options that are not shown in the listing above:

–b	Run in batch mode (no window for results).
–d	Show details of solution on standard error output.
–n	No shortcuts; retain solid and duplicate vectors.
–r	Raw output to standard output of columns, rows, and grid data.
–t	Provide timing information.
–v	Write verbose commentary to standard output.

Comments About the Program

The `c1` palette is used because it is the largest palette all of whose keys are "printable". This simplified program development and debugging as well as the representation of colors in the program output. Because of the use of the `c1` palette, at most 90 colors in a pattern can be discriminated. In practice, weaves rarely have many colors, so this is not a problem for patterns that might actually be woven. However, the colors shown in the result may be slightly different than the original colors, or worse, different colors may be mapped into the same color (yarns used in weaves sometimes differ only subtly in color).

The maximum number of colors could be increased to 256 by using the c6 palette. This would make some of the keys used "unprintable" unless they were written with escapes.

The problem with color discrimination could be removed by using a custom palette [2].

Because characters are used to identify rows and columns, the maximum size of an image the program can handle is 256×256. This limit is a result of the coding techniques used but not of the method.

If you looked closely at the code, you may have wondered about the line

```
  shuffle(wlist)
```

at the beginning of quad(). This deliberately disorganizes the vectors on the theory that adjacent vectors are more likely to be similar than randomly chosen ones, so that if one is unproductive, it's likely the next one is. (Recall that quad() returns after finding the first forcing pattern.) This heuristic has not been tested.

An alternative would be to sort the vectors by the number of colors they contain on the theory that vectors with more colors are more likely to have forcing patterns.

The program only determines if a color pattern can be woven and, if so, assigns colors. It does not consider floats, which can render the pattern "unweavable" from the standpoint of fabric integrity. See the article **Floats** that starts on page 1.

In most cases, if there is one solution, there are many different solutions. However, even if a program were structured to provide all solutions, there usually would be so many that it would be impractical to find one with the shortest floats. Finding a solution with the shortest floats (or with other constraints) is a much harder problem than determining color weavability.

A related problem is determining the minimum changes that would be needed to render an unweavable pattern weavable.

Timings

As mentioned in the first article on weavable color patterns [1], the combinatorial nature of the problem makes the efficiency of a solution method of paramount importance.

We have three solution methods to compare: the brute-force, try-all-possibilities method, a 2SAT algorithmic solution, and the heuristic solution described here.

We have a general idea of the relative efficiency of the different methods from trying many cases, but we have not conducted systematic timing tests. To give at least one representative example, we used all three methods on a 64×64 pattern (the heuristic method is so fast on smaller problems that it's not possible to get meaningful comparisons with the other methods). Here are the results in CPU seconds on a 400 MHz Linux PC:

heuristic	0.21
2SAT	3.51
brute force	> 1696560.

We don't know how long it would have taken for the brute-force program to complete; we had to terminate the job because of an equipment upgrade. 1,696,560 seconds is about 19.6 *days* — and that's CPU time.

What Remains

The programs give color assignments for weavable color patterns. In order to actually weave a pattern, it's necessary to have a draft — threading and treadling sequences and a tie-up.

We'll show how to convert color thread assignments to a draft in the next article in this series.

Acknowledgment

Will Evans wrote the 2SAT version of the solution.

References

1. "Weavable Color Patterns", Icon Analyst 58, pp. 7-10.

2. Graphics Corner — Custom Palettes", Icon Analyst 58, pp. 10-14.

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

Understanding Icon's Linker

The Icon compiler produces *ucode*, which consists of instructions for a virtual machine [1]. Ucode files need not comprise a complete program. For example, a module consisting of one or more procedures can be converted to ucode for use in various programs.

Icon's linker combines ucode files and produces executable *icode* files.

Scope Resoluton

The linker performs several tasks, one of the most important of which is resolving the scope of undeclared variables.

If there is a global declaration for a variable in any of the ucode files the linker combines, undeclared variables by that name become global; otherwise they become local.

There are three kinds of global declarations: global, procedure, and record. Except for global, only one global declaration is allowed for a variable.

Elimination of Unreferenced Code

Another function Icon's linker performs is the elimination of global declarations for variables that do not appear explicitly in a program. For example, in the program

```
procedure main()
   while write(*read())
end
procedure uc(s)
   return map(s, &lcase, &ucase)
end
```

the variable uc does not appear in the code and the procedure uc() is eliminated by the linker; the code is unreachable.

The elimination of unreferenced code allows the use of modules in which some procedures are needed without adding the excess baggage for those that are not.

The reduction in the size of icode files can be substantial, especially in programs that use graphics. For example, the simple program

```
link graphics
procedure main(args)
   WOpen("image=" || args[1])
   Event()
end
```

produces a 762-byte icode file. If unreferenced code is not deleted, the icode file is 509,811 bytes! Of course, for more complicated programs that use more of the graphics facilities, the savings are less.

The difficulty with eliminating code that is not explicitly referenced is that it is not necessarily unreachable. String invocation [2] allows any procedure to be referenced by the string name of a variable as opposed to a variable itself. Here's a program contrived to illustrate this:

```
procedure main()
   while write("uc"(read()))
end
procedure uc(s)
   return map(s, &lcase, &ucase)
end
```

In this program there is no occurrence of the variable uc; instead the procedure uc() is called using the string "uc". The linker, on the other hand, eliminates the procedure uc(). The program terminates with a run-time error because there is no code for uc() in the icode file.

The program above is silly, but this program, which applies a procedure name given on the command line, is not:

```
procedure main(args)
   while write(args[1](read()))
end
procedure uc(s)
   return map(s, &lcase, &ucase)
end
```

What happens when this program is executed depends on what's given on the command line. For example, if the program is named xform,

```
xform trim
```

works properly and trims the input file. But

```
xform uc
```

results in a run-time error.

Unfortunately, the symptoms of this problem are mysterious. Looking at the program, uc() is there. Icon novices (and sometimes, in more complicated situations, experienced Icon programmers) search in vain for the cause of such an error. Students learning Icon typically jump to the conclusion that there's a bug in Icon.

About all the advice we can give on this problem is to file it on a list of things to check when a program mysteriously malfunctions because a an expected procedure is not present.

When implicit references to procedures are known to occur, the problem can be avoided by using the invocable declaration. The easiest and safest way is to include

```
invocable all
```

in the program. This prevents Icon's linker from eliminating code. The invocable declaration also can be used with a list of procedures that may be invoked implicitly, as in

```
invocable "uc", "lc"
```

Note the quotation marks.

Listing the procedures that are invocable is, of course, prone to error, especially in large programs that are developed over time.

An alternative, which we prefer, is to add explicit references that do nothing but prevent the linker from removing needed code. For the example given above, an expression consisting of just the variable uc can be added to the main procedure:

```
procedure main(args)

   uc

   while write(args[1](read()))

end

procedure uc(s)

   return map(s, &lcase, &ucase)

end
```

The variable uc, standing alone, has no effect on program function.

Linking Information

Icon's linker can provide information about what it does. The command-line option –v*n* controls the "verbosity" of its standard error output as follows:

–v0: no advisory output; equivalent to –s
–v1: default output
–v2: show space allocation in icode file
–v3: also list discarded globals

Typical –v2 output, with space in bytes, is:

```
bootstrap      176
header         108
procedures    3344
records          4
fields           0
globals        208
statics          0
linenums       920
strings        380
total         5140
```

The list of discarded globals shown by –v3 can be quite long. Here's part of the results from linking the small graphics program shown earlier:

```
discarding procedure    args
discarding record       bev_record
discarding global       bev_table
discarding procedure    BevelTriangle
```

… [*435 similar lines omitted*]

```
discarding procedure    XPMImage
discarding procedure    XPM_Key
discarding procedure    XPM_RdStr
discarding procedure    XPM_Nth
```

Reference

1. "An Imaginary Icon Computer", *Icon Anayst* 8, pp. 2-6.

Supplementary Material

Supplementary material for this issue of the *Analyst*, including images and program material, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia59/

Recurrence Relations

A recurrence relation gives the terms of a sequence as a function of previous terms. For example, the Fibonacci sequence is given by the recurrence

$$a_n = a_{n-1} + a_{n-2}$$

with the initial terms $a_1 = a_2 = 1$ to get the sequence started. Different initial terms produce different but related sequences.

The number of initial terms required is determined by how far back in the sequence terms are specified — called the *order* of the recurrence relation. For example,

$$a_n = a_{n-1} + 2a_{n-3}$$

is a recurrence relation of order 3 and requires three initial terms, a_1, a_2, and a_3, to specify the sequence it produces.

The examples given above are linear recurrence relations with constant coefficients — LRRCs for short — and are instances of the general form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k} \qquad (1)$$

where only the first powers of previous terms are used and the coefficients are constant.

There are other kinds of recurrence relations. For example,

$$a_n = a^2_{n-1} + a^2_{n-2} + a_{n-4}$$

is a quadratic recurrence of order 4, while

$$a_n = a_{n-1} + na_{n-2}$$

is a linear recurrence of order 2 but with a non-constant coefficient.

LRRCs are important in subjects including pseudo-random number generation, circuit design, and cryptography, and they have been studied extensively. LRRCs also have periodic residue sequences [1], which is the main reason for our interest in them. Despite the importance of LRRCs and the work done on them, much about them remains unknown. Very little of a general nature is known about nonlinear recurrence relations. We'll focus mainly on LRRCs.

LRRCs

LRRC Canonical Form

Equation 1 above shows the canonical form for LRRCs. This form does not provide for a constant term, as in

$$a_n = a_{n-1} + 1$$

The reason for not having a constant term in the canonical form has to do with manipulations of LRRCs in which a constant term would require special handling.

A linear recurrence of order k with a constant term can be converted to a linear recurrence of order $k+1$ in canonical form. Consider the example above:

$$a_n = a_{n-1} + 1 \qquad (2)$$

From this it follows that

$$a_{n-1} = a_{n-2} + 1 \qquad (3)$$

Subtracting Equation 3 from Equation 2, we get

$$a_n - a_{n-1} = a_{n-1} + 1 - a_{n-2} - 1$$

and hence

$$a_n = 2a_{n-1} - a_{n-2}$$

which is in the required canonical form.

Problems Related to LRRCs

There are many interesting problems related to LRRCs. In the article on residue sequences, we touched on the properties of their residue sequences. Other problems of interest are:

- computing the sequence for an LRRC
- determining if a sequence can be represented by an LRRC and, if so, finding it
- solving an LRRC to produce an explicit formula for its nth term

An LRRC Generator

An LRRC can be completely characterized by two lists: one containing its coefficients and another containing its initial terms. For an LRRC of order k, both lists are of length k. For example, the recurrence relation

$$a_n = a_{n-1} + 2a_{n-3}$$

has the coefficient list [1, 0, 2]; the initials list, as always, determines the actual sequence. For example, the initials list [1,1,0] produces the sequence

1, 1, 0, 2, 4, 4, 8, 16, 24, 40, 72, 120, …

Following the model for the Fibonacci sequence given in the article on residue sequences [1], here's a general-purpose generator for LRRCs:

```
procedure lrrcseq(terms, coeffs)
  local i, term

  suspend !terms

  repeat {
    term := 0
    every i := 1 to *coeffs do
      term +:= terms[i] * coeffs[–i]
    suspend term
    get(terms)
    put(terms, term)
    }

end
```

Finding LRRCs

Many sequences can be represented by LRRCs, even if the recurrences are not obvious.

The *difference method* often works and it can be done by hand or with a simple program [2]. This method starts with a row containing the terms of the original sequence. The second row consists of the differences of successive terms in the first row, and so on. The rows are labeled Δ^0, Δ^1, Δ^2, … . Here's an example:

```
Δ⁰   1    7   18   34   55   81  112  148  189  …
Δ¹      6   11   16   21   26   31   36   41  …
Δ²         5    5    5    5    5    5    5  …
Δ³            0    0    0    0    0    0  …
```

If a constant row appears, as it does in this example, the process is complete, there is an LRRC, and it can be obtained by using Equation 4 below, which is a consequence of the way the differences are computed:

$$\Delta^k a_n = \sum_{i=0}^{k} (-1)^i \binom{k}{i} a_{n+k-i} \qquad (4)$$

where $\binom{k}{i}$ is the binomial coefficient

$$\binom{k}{i} = \frac{k!}{(k-i)!\,i!}$$

To get an LRRC in canonical form, it is necessary to go to a row of zeroes; Δ^3 in this case. Therefore, by Equation 4

$$\Delta^3 a_n = \sum_{i=0}^{3} (-1)^i \binom{3}{i} a_{n+3-i} = 0$$

Expanding this, we get

$$\binom{3}{0} a_{n+3} - \binom{3}{1} a_{n+2} + \binom{3}{2} a_{n+1} - \binom{3}{3} a_n = 0$$

and hence

$$a_{n+3} - 3a_{n+2} + 3a_{n+1} - a_n = 0$$

from which we get the LRRC

$$a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3}$$

The initial terms are, of course, the first three in Δ^0.

Here's a program to produce LRRCs by the method described above. The sequence is read from standard input.

```
link lists
link math

procedure main()
  local sequence, order, sol, i, original, initials, c

  original := [ ]

  while put(original, integer(read()))

  sequence := copy(original)

  order := 0

  until c := constant(sequence) do {
    sequence := delta(sequence)
    order +:= 1
    if *sequence = 0 then
      stop("No recurrence relation found")
    }

  if c > 0 then order +:= 1

  initials := original[1+:order]

  sol := [ ]

  every i := 1 to order do
    put(sol, (–1 ^ (i + 1)) * binocoef(order, i))

  write("recurrence of order ", order)
  write("coefficients: ", limage(sol))
  write("initial values: ", limage(initials))

end

procedure delta(seq)
  local deltaseq, i

  deltaseq := [ ]

  every i := 2 to *seq do
    put(deltaseq, seq[i] – seq[i – 1])

  return deltaseq

end

procedure constant(seq)
  local c

  c := seq[1]
```

```
      if !seq ~= c then fail
      else return c
   end
```

The output for the sequence given earlier is

```
recurrence of order 3
coefficients: [3,–3,1]
initial values: [1,7,18]
```

Any recurrence derived from a finite number of terms is, of course, conjectural.

Explicit Formulas for LRRC Terms

Any sequence that leads to a 0 Δ sequence can be represented by a polynomial in n. Conversely, all polynomials in n can be represented by a single LRRC; the coefficients of the polynomial only affect the initial terms for the LRRC.

This follows from another equation that results from the method of differences:

$$a_{n+m} = \sum_{k=0}^{n} \binom{n}{k} \Delta^k a_m \qquad (5)$$

From this, we can obtain an explicit formula for the nth term of the corresponding LRRC. Setting m to 1 in Equation 5 gives

$$a_{n+1} = 1\binom{n}{0} + 6\binom{n}{1} + 5\binom{n}{2} + 0\binom{n}{3}$$

(1, 6, 5, and 0 are the leading terms in Δ^0, Δ^1, Δ^2, and Δ^3.) This evaluates to

$$a_{n+1} = 1 + \frac{7}{2}n + \frac{5}{2}n^2$$

Implementing this is similar to finding LRRCs by the difference method. We're out of space, so we'll leave it as an exercise.

References

1. "Residue Sequences", Icon Analyst 58, pp. 4-6.

2. *The Encyclopedia of Integer Sequences*, N. J. A. Sloane and Simon Plouffe, Academic Press, 1995, pp. 10-13.

$$e - 1 = 1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}}}}}$$

What's Coming Up

In the next issue of the Analyst, we plan to have another article on tie-ups, an article on creating drafts for weavable color patterns, and a second article on classical cryptography.

Continuing the series related to periodic sequences, we plan an article on continued fractions, and in particular those for quadratic irrationals.

For the **Graphics Corner**, we expect to have an article on an interactive application for constructing custom palettes.

The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–analyst@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA

and

Bright Forest Publishers
Tucson Arizona