by Jeffreys Copeland and Haemer

CHET PHILLIPS

Furuike ya
kawazu tobikomu
mizu no oto.
– Basho

Hard Problems

Last month we talked about generating random text, and this month, as promised, we will talk about generating random haiku. Eventually.

Before we can talk about random haiku, we need to cover some other ground first. Because of the nature of the problems we'll be discussing, this whole column, in some ways, is an exercise for the reader.

NP-complete

There is a class of problems in computing literature called "NP-complete" problems. We need to go back to Turing and his successors to discuss some of the theory involved here.

Assume that we have a set *P* of programs that can be run in polynomial time on a Turing machine. There is also, then, a class of programs *NP* which can be solved in non-polynomial time. If one could show that *P* = *NP* one would be able to show nearly any problem that can be expressed algorithmically is easily solvable with a computer program.

(Yes, we are being sloppy and cutting some corners. If you want a really pedantic treatment, we'd recommend *Computability and Unsolvability* by Martin Davis [McGraw-Hill, 1958, reprinted by Dover, 1982, ISBN 0-486-61471-9] or the more accessible Lewis and Papadimitriou's *Elements of the Theory of Computation* [second edition, Prentice-Hall, 1998, ISBN 0-13-262478-9]. And while we are discussing references, we take a moment to recommend the short story "Antibodies" by British science fiction writer Charles Stross, in which it is proved that *P* = *NP* with disastrous results for our timeline. It's reprinted in *The Year's Best Science Fiction: Eighteenth Annual Collection*, edited by Gardner Dozois, St. Martin's Press, 2001, ISBN 0-31227465-3.)

We sometimes speak of "NP-complete" problems. These are problems least likely to be solvable in polynomial time. Roughly, they are problems without closed form solutions that must be solved by brute force, and may not actually have a solution.

An example of an NP-complete problem is, of course, the Traveling Salesman Program. In this classic puzzle, a salesman must make arrangements to visit a number of cities in his territory, and can only stop in each city once. What's the most mileage-efficient route that can be devised to get him through the full cycle?

The obvious solution is to …

Well, maybe it's to …

Sorry, but it turns out there isn't an obvious solution. The brute-force one is the only one we know about: List the cities, add up the mileage for each permutation of the possible visits, and select the lowest total out of that enumeration.

Similarly, public-key crypto systems depend on the nature of NP-complete problems. The RSA algorithm works because multiplication is much easier than factorization: we can only factor the product of two primes by brute force. This means that as long as we keep picking RSA public keys that increase in size by a decimal digit every five years, we'll

keep ahead of Moore's Law in the difficulty of breaking them.

Another public-key algorithm which depends on brute force is the knapsack algorithm: given a set of pieces cut from a tree trunk of certain thicknesses, which of them will fill a knapsack of a particular length and the same diameter as the original tree trunk?

The Test Matrix

We tripped over a similar problem a few months ago. But unfortunately, it's a variation of the traveling salesman problem, so the only solution we can adopt is the brute force one.

Given a set of products, each with a set of features, what's the minimum list of products on which we need to run the full test set to ensure that we've tested all the features at least once? If we were building printers, the feature set might include duplexers, staplers and long-edge feed vs. short-edge feed. If we were building middleware, the feature set might include base operating system, processor type, and additional installed products. You get the idea.

Let's start with a list of product code names and feature codes, such as:

```
jackson a b z
lincoln a x y z
washington b d e f g
jefferson d f z
madison d e
adams a c b
hamilton x y z
```

(We hasten to add that any resemblance between these product code names and any actual product code names at any company where we've ever worked is merely coincidental. We mention this only because we once worked at a company where there was a code name for the code names, and we know how paranoid and humorless people get about these things.)

Given that input file, how can we figure out which set of rows is the minimum to cover all the features, "a" through "z"? By now, you're cued to answer "Brute force!"

Our programs always begin the same way.

```
#! /usr/local/bin/perl -w
# find test matrix coverage
# $Id: matrix,v 1.2 ...

use strict;
```

In this case, because we need to exhaustively examine each possible ordering of the products, we'll use the `Permute` package from the Comprehensive Perl Archive Network (`http://www.cpan.org`). Collect `Algorithm-Permute-0.03.tar.gz`, unpack it, and then run

```
perl Makefile.PL
make
make test
make install
```

Back in our program, we include the permutation algorithm with a Perl `use` statement, followed by some global variables.

```
use Algorithm::Permute;

my %products;
my %features;
my @products;
my @features;
```

Why both hashes and matrices? Internally, we'll be storing the list of features in a hash indexed by the product name. Thus we can say `%products{'lincoln'}` to get a particular feature set. Similarly, we'll enumerate the features in a hash. We will find later on in the code that a list of either the product or feature names will be helpful. We could generate them on the fly each time with `sort keys %products`, but if we do that once at the beginning it will save us compute time.

Reading the list is a fairly simple loop.

```
# read the test list, saving the
#  feature list in a hash by product
while (<>) {
   chomp;
   s/(\w+)\s//;
   $products{$1} = $_;
   foreach (split / /, $_) {
      $features{$_}++;
   }
}
```

For each line, we strip off the product name, store the feature list in the hash (as threatened), and then enumerate the features in their own hash. Yes, we could store the features as an array pointer in the hash, but we'd still have to walk the array every time we accessed it.

```
@features = sort keys %features;
@products = sort keys %products;

print "products: @products = ",
   scalar(@products), " items\n";
print "features: @features = ",
   scalar(@features), " items\n";
```

Again, as promised, we list the features and products in their own arrays, and print them out for reference, along with the counts of items in both lists.

Now, using the `permute` method from the Permute package, we need to run through each permutation of the products array, and determine how many items we need before we achieve full feature coverage.

```
# now find the set of minimum permutations
my @min_set = @products;
Algorithm::Permute::permute {
   my @n = coverage(@products);
```

```
    @min_set = @n  if( $#n < $#min_set );
}   @products;
```

The `permute` interface generates each permutation in turn, and executes the block of code for each one of them. To enumerate all the permutations, we'd just insert

```
print "@products\n";
```

as the block. Our `coverage` routine returns the minimum list from the permutation, as we'll see here:

```
sub coverage {
    my @products = @_;
    my @used = ();
    my %cov;
    foreach (@products) {
        push @used, $_;
        foreach (split / /, $products{$_}) {
            $cov{$_}++;
        }
        my @cov = sort keys %cov;
        last if( $#cov == $#features );
    }
    @used;
}
```

We loop through the permutation given us, and stop when we've enumerated the full set of features in the products we've seen so far. We return that subset to the caller.

Once we have `min_set` in hand, we should print it out, so we know what products to test:

```
#  print out the minimum list
print "minimum test coverage from@min_set\n";
foreach (@min_set) {
    print "$_: ", $products{$_}, "\n";
}
```

Which brings us to the first major exercise for the reader. We've assumed that this matrix problem is NP-complete, and we must solve it with a brute-force program. On the other hand, we have a sneaking suspicion that there's a method of attacking the matrix itself that would yield a simple algorithmic solution. If anyone can provide us with that algorithm, it will be our readers.

Haiku

Which brings us to the other hard problem of the month: haiku. Haiku, of course, is the Japanese poetry form in which a verse of 17 syllables–three lines of five, seven and five syllables, respectively–captures the mood of a particular moment with (sometime oblique) reference to the season. One of the earliest, and best loved, practitioners of the form was Matsuo Kisaku (1644-1694), whose pen name was Basho. Our favorite example is the one we chose as this month's epigram, which translates as "An old pond. A frog jumps in. The sound of water." Even the translation is problematic, as demonstrated in Hiroaki Sato's book *One Hundred Frogs: From Renga to Haiku to English* (Weatherhill, 1983, ISBN 0-8348-0176-0, but no doubt out of print long since) which discusses the haiku (and predecessor hokku) forms. Sato finishes with a 100 alternate translations of those seventeen syllables of Basho's.

We set out, once we had finished last month's column, to develop a program to generate random haiku as much as we generated random English sentences. It isn't nearly as easy as the statement of the problem.

We set out, once we had finished last month's column, to develop a program to generate random haiku as much as we generated random English sentences.

We follow the same basic approach of last month's random sentences. We produce lines with words in the specified parts of speech in the specified order. The trick is to ensure the lines are the appropriate number of syllables. This is where the brute-force aspect appears.

We'll show you:

```
#! /usr/local/bin/perl -w
# $Id: haiku,v 1.5 ...
# write code in the fall /
# generate random haiku /
# amuse our readers
```

Our usual beginning, including a self-referential comment.

```
use strict;
use Lingua::EN::Syllable;
```

This time, we include the `Syllable` module, which allows us to determine the number of syllables in a given word.

We need a selection of words, as before:

```
    # Adverb selection
my @A = ( "badly", "happily", "sadly" );
    # adjective selection
my @a = ( "bright", "large", "odd",
    "green", "blue", "new", "random", "correct" );
    # verb selection
my @v = ( "generate", "sleep", "amuse", "write",
    "watch", "jump", "leap", "love", "correct" );
    # noun selection
my @n = ( "fall", "spring", "summer", "winter",
    "code", "haiku", "error", "Silent Bob",
    "frog", "snow", "leaves" );
```

This time, rather than encapsulate the possible order of the parts of speech in the algorithm, we'll provide it in a table:

```
  # possible parts of speech in each line
my @order = ( "vaan", "van", "Avn", "anv",
  "Avan", "Avaan", "AvA", "aaa" );
```

This will make it easier to select a random word order, since we want more variance in structure for our haiku than we did for our sentences. The members of our `@order` array are made up of the names of our word arrays, for ease of reference.

We must generate a line of words. We select a random parts-of-speech order, and try to assemble the specified number of syllables (in this case, stored as `$n`) from those lists of words.

```
sub line {
   my $n = shift;
   my $result;
   while( 1 ) {
      $result = try($n, $order[rand @order]);
      last if( length($result) );
   }
   print $result;
}
```

We make an attempt, and keep trying until we get it right. We are reminded of the episode early in T.H. White's *The Once and Future King* where Merlin transforms Arthur into an ant, and lets him watch the ant colony try to arrange their nest. They do this by picking things up and dropping them at random, rather than having an overriding plan.

We also need our `try` routine, which is the "brute" of our brute-force approach.

```
sub try {
   my ($n, $order) = @_;
   my $syl = 0;
   my $word;
   my @words = ();
   foreach my $x (split //, $order) {
      $word = $n[rand @n]  if( $x eq 'n' );
      $word = $v[rand @v]  if( $x eq 'v' );
      $word = $a[rand @a]  if( $x eq 'a' );
      $word = $A[rand @A]  if( $x eq 'A' );
      push @words, $word;
      $syl += syllable($word);
      last  if( $syl > $n );
   }
   return "" if( $syl != $n );
   return join(" ", @words) . "\n";
 }
```

We pass in a specified number of syllables and parts-of-speech order, and assemble words. We count syllables as we go (relying on the `syllable` package), and we short-circuit the loop if we have assembled too many syllables. If we do not have the specified number of syllables at the end, we return the empty string, and pull back our head to bang it against the wall again. If we finished the loop successfully, we concatenate the words in our line, and return them as a single string suitable for printing.

All that remains is to generate the appropriate lines:

```
line(5);
line(7);
line(5);
```

What does this get us? Not, unfortunately, classic haiku lines such as "probable user error" or "please correct and resubmit." But we do get haiku such as:

```
watch correct haiku
badly write green correct snow
badly love large spring

sleep correct blue spring
generate large random snow
jump random winter
```

It may not be great literature, but it does end up providing amusement.

Random Colors, Too

As we've mentioned before, Copeland is red-green color blind, which has provided us–and Copeland's children, Allie and JJ–with no small amount of amusement. Last month, longtime reader Dr. Neil G. Cuadra wrote to us about a Web site put together by Robert Dougherty and Alex Wade at Stanford, (`http://vischeck.com`). If you upload a picture to the Web site, you can readjust its colormap to see how it would appear to someone with a color deficiency. Check it out.

Also, constant British reader Paul Livesey provided the prompt answer to one of last month's puzzlers: The episode of British cult TV show *The Avengers* in question was "Love All." We remembered it as black-and-white, and that mis-cue caused Paul some problems, too. It was actually an early color episode.

Next month, as usual, we'll be back with a new adventure in computing. Until then, happy trails. ⊷

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.