Automated Cryptogram Solving, Volume II

Design Guide to Automatically Solve Almost All of the ACA Cipher Systems by Computer

By SHMOO

Volume II covers: Important Preliminaries, Complete and Incomplete Columnar, Nihilist Transposition, Cadenus, Amsco, Myszkowski, Turning Grille (Grille), Swagman, Bazeries, Nicodemus, Fractionated Morse, Morbit, Pollux

Important Preliminaries

This volume assumes (realistically, *requires*) that the reader has read the Introduction and Simple Substitution Chapters in Volume I. This text may be outright incomprehensible without that background.

Chapter V – Complete Columnar, Incomplete Columnar, Nihilist, Cadenus, Amsco, Myszkowsky Transpositions

Columnar Transposition (Complete Columnar, Incomplete Columnar, Nihilist Transposition)

The Complete Columnar transposition, Nihilist Transposition, and even Incomplete Columnar transpositions can all be solved with the same basic method. In object terms, it would be possible to have a parent object called "Columnar" and "Incomplete", "Complete", and "Nihilist" all be children. Whether that infrastructure is really worthwhile is an exercise for the reader, but there is a great deal of commonality.

They will be treated as one cipher here, with differences as noted. There is little discussion of the ciphers themselves; they are basically all use a numeric key (or a word which functions like a numeric key). The techniques for writing them into a trial solution array is straightforward. In effect, they are all trial decipherments, each one pretending to be correct. Competent grading puts the actual correct key at or near the top.

Main Technique: Exhaustion via Factorial Key Set

Implementation Issues: Does not handle very long keys (e.g. period over 12 or so)

The generic implementation looks like this in pseudo code (period is known or guessed):

```
FactorialKeySet fs = new FactorialKeySet(period);
foreach farray in fs {
   // farray is an integer array that is "period" long and
   //    has the current key to try.
   Pt = Decipher(Ct); // this varies by the system,
                      //   may be a polymorphic call.
   grade = Grade(Pt,tipIf_any,tipLoc_if_Known); // boost score for tip
   if (grade is Good Enough)  {
         Add_to_solution_list(Pt,Key,Grade);
  }
}
```

This is enough to suggest how to program all three. "Grade" would include standard "tetragram" scoring plus a bonus if the tip is discovered in whole or in part.

Cadenus

Main Technique: Exhaustion via Factorial Key Set

Implementation Issues: None. Long keys are effectively prohibited.

Because Cadenus takes off its text differently than the previous ciphers, it may be difficult to have a common or even polymorphic implementation for this compared to the others. But, the generic program above still describes how the solution works.

Amsco

Main Technique: Exhaustion via Factorial Key Set

Implementation Issues: Irregular columns

The issue with Amsco is that there are either one or two letters in each column position. This is probably best managed by having an array of two letter strings that are either "* " or "**" that are then replaced with the ciphertext on each trial. A given Factorial Key Set will have an entirely predictable matrix, with the ** or * columns known. The only added issue is that each key needs to be done twice, once commencing with "*" in the first position and once with "**" in the first position because that is a free choice in the construction. The * and ** alternate thereafter. Note that the cost is low despite this because of the low number of allowed periods.

Myszkowski

Main Technique: Hill Climbing

Implementation Issues: How many duplicate key values to add?

Failure Rate: Moderately high.

Because we can have repeated key numbers, exhaustion is no longer realistic. There is no serious limit; there could be two numbers that have duplicates; maybe three; maybe four. The number of duplicates could be one, two, or even three. There are practical limits on how great these magnitudes are, but even so, there are a lot of potential keys here.

Automated solution for Myszkowski has so far been less certain than many other ACA systems, at least for the author.

The interested solver might consult: Cary N. Davids, NUCLULAR, Computer Column, SO 2013, "Attacking the Myszkowski Without a Tip" which contains a couple of practical suggestions about duplicates. NUCULAR claims the method (eventually) always works.

Chapter VI – Irregular Transpositions: Grille, Swagman

This chapter contains two more singular transposition methods; unique concepts that have little in common with other transpositions other than transposing letters.

Grille (or, Turning Grille)

Main Technique: Hill Climbing

Implementation issues: Handling of odd periods, extensive precomputation

This could in theory be handled by Factorial Key Set type methods, but the actual combinatorics are too great. Hill Climbing is used instead. The "turning" part of the "turning" grille means that every time we assume a grille position, we "see" four letters as we turn. A given number in the key can be in any quadrant.

There is a lot of hidden regularity in this cipher – an extensive set of precomputed offsets can be deduced for each of the relatively few permitted array sizes (which must be, overall, a square).

Compare these pre-computed values. Can you see how the second is derived from the first? The latter, particularly, helps with operations if your implementation is so inclined. This presumes the "square" is actually implemented as an ordinary single dimension array (which is quite possible to do).

```
static readonly int grilleLayout9[] = {
            0, 1, 2, 6, 3, 0,
            3, 4, 5, 7, 4, 1,
            6, 7, 8, 8, 5, 2,
            2, 5, 8, 8, 7, 6,
            1, 4, 7, 5, 4, 3,
            0, 3, 6, 2, 1, 0
};

static readonly int grilleLoc9[][][] = {
      // num   square location
        { { 0 ,  0 ,  0 },
            { 0 ,  1 ,  5 },
            { 0 ,  2 , 35 },
            { 0 ,  3 , 30 } },
        { { 1 ,  0 ,  1 },
            { 1 ,  1 , 11 },
            { 1 ,  2 , 34 },
            { 1 ,  3 , 24 } },
        { { 2 ,  0 ,  2 },
            { 2 ,  1 , 17 },
            { 2 ,  2 , 33 },
            { 2 ,  3 , 18 } },
        { { 3 ,  0 ,  6 },
            { 3 ,  1 ,  4 },
            { 3 ,  2 , 29 },
            { 3 ,  3 , 31 } },
        {   { 4 ,  0 ,  7 },
            { 4 ,  1 , 10 },
            { 4 ,  2 , 28 },
            { 4 ,  3 , 25 } },
```

```
            {    { 5 ,  0 ,  8 },
                 { 5 ,  1 , 16 },
                 { 5 ,  2 , 27 },
                 { 5 ,  3 , 19 } },
            {    { 6 ,  0 , 12 },
                 { 6 ,  1 ,  3 },
                 { 6 ,  2 , 23 },
                 { 6 ,  3 , 32 } },
            { { 7 ,  0 , 13 },
                 { 7 ,  1 ,  9 },
                 { 7 ,  2 , 22 },
                 { 7 ,  3 , 26 } },
            { { 8 ,  0 , 14 },
                 { 8 ,  1 , 15 },
                 { 8 ,  2 , 21 },
                 { 8 ,  3 , 20 } }
};
```

Swagman

Main Technique: Factorial Key Set on rows.

Implementation Issues: Manual assembly of solution from solved rows

A fuil Swagman implementation would be both difficult and computationally expensive. Fortunately, it is sufficient to pretend it is more or less a columnar cipher. We grade and save individual *rows* separately.

Astonishingly, this works for the price of looking over the ordered solutions and deciding which ones to manually combine for the final answer.

Chapter VII – Substitution with Transposition

Here we consider a couple of systems that perform a substitution operation with a transposition.

Bazieries

Main Implementation: Exhaustion

Implementation Issues: Alternate versions of prose keywords. Zero in the keyword.

Bazeries uses the same numeric keyword twice. It can be any value between one and just short of one million. Transposition and Substitution can be done in any order – they are independent operations.

For the transposition, the numeric key is used. For substitution, a K2 type alphabet is created from the prose name of the numeric key. The easiest way to "get" this is to imagine how one would write out a number for an old-fashioned check. There are a lot of duplicated letters in a prose name, so the keywords often look kind of squished and irregular. This does not affect solving.

Keep in mind that in English (and other languages too, for that matter), there are some well-known variant ways of writing the same number in prose. 1534 can be written "One Thousand Five Hundred and Thirty Four" and it can also be written "Fifteen Hundred and Thirty Four". The connective "and" may or may not appear and, in many cases, it disappears because of the word "Thousand" appearing first.

But, all these variations can be accounted for. If all such prose keys are tried, the numeric keys take care of themselves as they are one-to-one with a given prose key. Note that zeros can create degenerate transpositions, so are often avoided. However, it is possible to write an auto solver that deciphers in spite of the presence of zeros.

Trying out slightly more than a million keys does not tax modern computers and the correct answer usually grades very well.

Nicodemus

Main Implementation: Exhaustion via Factoral Key Set plus special chi square-based concept (hill climb is also possible)

Implementation issues: Larger keys less certain of solution; complex and unique solution method.

Failure rate: Higher as key size grows.

The method the author uses is difficult to easily describe. This may be a cipher implemented very late in the "game" after much success with other systems.

The author has tried a standard hill climber, but then discovered this method adapted from:

practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic/

. . .where the ciphertext, for a given key, has the chi square calculated for each column and the better column scores are saved.

We then take the results and create plausible keys (the restriction that the keys are strictly increasing as numbers is a big help). This can be done exhaustively or through hill climbing for higher periods.

Chapter VIII – Morse-based Ciphers

In the ACA, there are three major cipher types based on the Morse code – Fractionated Morse (the oldest) and two more "recent" systems, the Morbit (popularized in the ACA in 1964), and the Pollux (popularized in 1968). The basic convention of . for Morse dot, - for Morse dash and either one letter x (for the end of a letter) or two (for the end of the word) is documented elsewhere, not repeated here.

Fractionated Morse

Main Technique: Hill Climbing an Aristocrat/Patristocrat with special Morse Tetragrams

Implementation Issues: Potential need for non-English Morse Tetragrams (not needed so far)

Fractionated Morse goes from fairly hard to basically trivial with an interesting trick. (See the definitions in the Introduction about *standard* and *trivial* alphabets to understand the following).

First step (one time cost):

1. Get the same stream of English you used to create your tetragram count.
2. Get rid of all non-English letters as you did before, but preserve the spaces between words.
3. Create an enciphered Fractionated Morse Stream using the *trivial alphabet*. That is, use the key ABCD for the Fractionated Morse "key" and encipher normally.
4. Take a tetragram frequency count of the "cipher" stream produced in step #3. Call it "Morse Tetragrams".

The trivial alphabet, for Fractionated Morse, looks like this:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
.........---------xxxxxxxx
...---xxx...---xxx...---xx
.-x.-x.-x.-x.-x.-x.-x.-x.-    thus, A=…, Z=xx-
```

With Morse Tetragrams in hand, you have now reduced the Fractionated Morse cipher to an Aristocrat of sorts! What you are now looking for is the actual keyword alphabet, but it structurally like a *standard alphabet* of an Aristocrat over the *trivial alphabet*.

Something like this:

```
ZREPUBLICADFGHJKMNOQSTVWXY
ABCDEFGHIJKLMNOPQRSTUVWXYZ
.........---------xxxxxxxx
...---xxx...---xxx...---xx
.-x.-x.-x.-x.-x.-x.-x.-x.-
```

The *standard alphabet* will be keyed, sometimes shifted as above. But, whatever it is, even if it turns out to be unkeyed, your code can now find it just as it does for Aristocrats with no key.

Here's the thing: Because the actual Fractionated Morse alphabet used by the cipher constructor is now the *standard alphabet* in this modified look at things, you solve as you do for Aristocrat keys against the *trivial alphabet*. The letters you discover from this almost plaintext are now subjected to Morse tetragram scoring. The higher scores, as with Aristocrats, are closest to right. Why? Because if the trial *standard alphabet* is right, you have translated the stream to the *trivial alphabet* and you have a

Tetragram frequency count with which to score them for things enciphered in the *trivial alphabet*. It's just that the underlying "language" is Morse code.

Only at the very end, when you have collected the top one or two dozen answers, do you translate back into Morse and then into English (you do this as the *standard alphabet* you uncovered is overlaid directly on the *trivial alphabet* and the trivial is overlaid on the actual Morse Triplets).

The resultant Morse stream is then translated back to English, giving (in the best case) an essentially perfect answer, word spaces included.

Note that even though it might not always be so, this process has so far worked, unchanged, with the same Morse Tetragram list for languages other than English when Fractionated Morse is used to encipher them.

Morbit

Basic Method: Exhaustion with Factorial Key Set

Implementation Issues: None

The technique here is direct and exhaustive, though nine factorial executions can be lengthy on slower computers. One simply tries out the entire nine factorial key space, keeping the highest scoring answers.

Classical solutions distinguish between x and non x elements and then further decide which non x are dots and which are dashes. That can work; indeed, recursion can be used to advantage, but it turns out, exhaustion is straightforward on modern PCs.

Pollux

Basic Method: Exhaustion with Factorial Key Set

Implementation Issues: None

The technique here is direct and exhaustive, though ten factorial executions can be lengthy on slower computers. One simply tries out the entire ten factorial keyspace, keeping the highest scoring answers.

Since ACA keys tend to have 4 dots, 3 dashes, and 3 x-es, this "plaintext" alphabet can be used for the Factorial Key Set.

Classical solutions distinguish between x and non x elements and then further decide which non x are dots and which are dashes. That can work; indeed, recursion can be used to advantage, but it turns out, exhaustion is straightforward on modern PCs.