Automated Cryptogram Solving, Volume III

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

By SHMOO

Volume III covers: Important Preliminaries, Ragbaby, Grandpre, Compressocrat, Condi, Syllabary, Periodic Ciphers (including Vigenere, Variant, Beaufort, Gronsfeld, Auto Key, Running Key, Progressive Key, Interrupted Key, Slidefair, Portax, Gromark, Periodic Gromark, Homophonic, Numbered Key, Quagmire I, Quagmire II, Quagmire III, Quagmire IV, Phillips, Playfair, Seriated Playfair, Two-Square, Tri-Squre, Four-Square, Bifid, Conjugated Matrix Bifid, Trifid, Twin Bifid, Twin Trifid, Digrafid, Headlines, Cryptarithms

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 IX – Miscellaneous Substitutions

Ragbaby Cipher

Main Implementation: Manual Solver with shift assist

Implementation Issues: Resists (so far) automatic solution, support for rarely seen 26 and 36 letter variants.

Ragbaby is (so far) surprisingly resistant to automatic solution. It might fall to a cipher word list strategy, but for the most part, the "vulnerability" is, like Tridigital, weak because it is basically just the word length. Tips are pretty much always given and placing these give alphabet fragments.

As they tend to initially appear in solvers, one gets multiple key fragments from the tip; they are incorrectly shifted against each other. One can instruct the code to change the shifts and see what the results on the plaintext are for such shifts.

The first order issue is to ensure the changed shifts are sound – that no letter appears above or below another one.

Once all shifts are found, then they are implicitly merged and the resulting text can be examined. This either leads to new letters (and new partial alphabets) or it leads to revision of some of the shifts or both.

Ultimately, as text is added, eventually all the partial alphabets are merged and merged correctly. At this point, the solution is largely or entirely revealed.

Grandpre Cipher

Main Implementation: Word list, keyword, with semi-accurate hill climb

Implementation issues: More manual than automatic, but available automation helps.

The solution here is "mostly" manual, but a certain amount of auto-solving like function can really boost the speed of solution and give further entry to the text.

Grandpre, in the ACA, is either an 8x8 matrix or a 10x10 matrix. It is filled in with keywords horizontally and the first letter of each forms a keyword as well. This restriction enables a certain amount of automation.

After the tip is placed, words that might appear in the key square based on the revealed letters is collected. These are tried recursively and (within recursion limits) the result texts are graded and shown. Keep in mind that a decent fraction of the text may be "not assigned" and will end up with low tetragram scores.

None the less, even though some of it is very wrong, useful plaintext fragments often appear.

By telling the program to try certain keywords at certain key square positions, or adding fresh tip material when the plain text is revealed, a series of iterative runs usually produces the answer without too much trouble.

Compressocrat

Main Implementation: Hill Climb with special Compressocrat Tetragram list.

Implementation Issues: See Fractionated Morse, creating a special Compressocrat tetragram list.

Compressocrat is surprisingly analogous to Fractionated Morse, including the idea of reducing it to a Patristocrat of sorts by enciphering the language source text in "the trivial alphabet" and collecting tetragrams for that.

This despite the fact that it compresses the original plaintext a bit and does not reference Morse symbols at all.

Read the Fractionated Morse for clarity – it should be easy to see how the concepts transfer.

As with Fractionated Morse, this Patristocrat on the intermediate text means that the final plaintext is only seen at the very end of a successful hill climb.

Condi

Main Implementation: Semi-solver, uses Cryptarithm concepts. Alternative: Wordlist, keywords.

Implementation Issues: Automatically generating a program from tip and input, supplementing the produced program.

In the end, Condi is mostly about keyword recovery. While the cipher appears simple, using and extending plaintext is surprisingly difficult.

Thus, producing the alphabets with the keywords (even shifting them to be sure one got them all) is one approach that the author has tried. But, again, a multi-word setup defeats it. Moreover, in one instance, the author discovered that because the word list left off an obscure word, that it was only solved by luck – by a word that lead to a nearly identical keyword and yet even with only two letters different, the recovered text was far from complete. Amazing the influence such a slight difference had on the text.

This sort of thing has lead the author to do something so far unprecedented – abandon a potentially workable auto solver and go back to a manual approach.

Condi has enough algebra in it that, with a tip, the enciphering and deciphering can be expressed as additions modulus 26. That can be used to set the whole thing up "as if" it was a Cryptarithm, modulus 26, and see what happens. It is possible to finesse the "starting offset" in tip text to get the equations (see MATANZA's article "Solving a Condi Crypt Using Algebra" in the May June 2012 issue of The Cryptogram).

The author also has an approach that generates a by-hand program to solve cryptarithms. This was incorporated into the manual scheme so that a manual cryptarithm solver is generated for the tip.

That can be enough, but it often leads to too many trial solutions to manage. However, as fly by on the output, observations can be made. One can notice that letters (say, X and Y) are adjacent over and over again and so add the guessed equation: $Y=X+1$. One can add in shrewd inequalities to the generated program such as $E < 17$ (probably true whether E is in the keyword or not) and $W > 19$ when W is suspected of not being in the keyword. Be prepared to throw out some of these ideas; not all will be correct (e.g. W might be in the keyword, the keyword might be significantly shifted). Enough right guesses will drastically reduce the number of possible keys, even without all 26 letters present.

This sort of approach, while it takes a little time, has worked splendidly on recent ciphers.

Another valuable idea is to grade the answers. The actual answer, being a *standard alphabet*, is likely to have a lot of adjacent letters or near adjacent letters that aren't part of the key. So, give a bonus score for every time that has happened (one bonus for adjacent like JK, slightly less for a gap of one like JL) and sort the qualified answers. This usually sifts the correct key (as far as it is known) near the top of the list.

Syllabary

Main Implementation: Mostly, solver's aid. Can sometimes convert to Patristocrat 6x6.

Implementation issues: Unclear. Research needed. But the 10x10 matrix does not help.

Failure Rate: High.

To date, the author has limited success at best with this cipher. The reader might be better placed to seek out what others have done with this cipher.

The author has had some success with a process where the highest frequency syllabary number pairs are arbitrarily assigned to the letters A-Z. But, the remaining numbers (however many there are; maybe as many as 40 or 45 distinct numbers) are arbitrarily smashed down and share the digits 0-9.

This cipher, distorted though it is, is then fed to a special Patristorcrat solver that was created for Checkerboard 6x6 ciphers. It sometimes can fail entirely. But when it works, it suggests a decent set of plaintext to try, even given it doesn't understand the multiple substitution taking place.

Chapter X – Periodic Ciphers

The Cryptogram publishes a great number of periodic ciphers, many are simply variants of each other (one so much so that we call it "Variant").

There are also elaborations and alterations that increase the solving difficulty.

Classification and Commonalities

There is value in exploring the similarities of these many types of cipher before discussing specific systems. The most basic set of periodic ciphers:

- Vigenere
- Variant
- Gronsfeld
- Beaufort
- Porta

These can be solved polymorphically from a common core of code if desired. All that is needed is system-specific encipherment/decipherment. They auto solve well.

Many systems are elaborations that take each of the "main five" above as a base, but with some kind of twist:

- Slidefair (Vigenere and Beaufort)
- Auto key
- Running Key
- Progressive Key
- Interrupted Key

So, in their own way, these too are essentially generic and can be solved from a common core. In some cases, the code is so fast that all of the above "basic set" can be all tried together and still get an intelligible answer from a solution list ordered by scores.

There are some standalone ideas that also use one of the "main five" as a base or just go off on their own.

- Portax (Porta variant)
- Gromark (Gronsfeld)
- Periodic Gromark (Gronsfeld)
- Homophonic
- Numbered Key

Numbered Key is a difficult "solve". Running Key is a bit of a special case – it is a cipher that is just as hard for the legitimate receiver as the cryptanalyst – borders on an unfair problem, but it is still in the ACA's bag of tricks.

The most difficult to solve are the Quagmires – these are periodic ciphers with keyed alphabets. As one "ascends" from Quagmire I through II and III and IV, one finds that the difficulties in auto solving them increases. Quagmire IV is the subject of furious research and Quagmire III is marginal.

Quagmires are distinguished by Roman Numerals and they represent the kind of keying system used. K1 keys the plaintext alphabet, K2 the ciphertext alphabet. K3 uses the same keyword, shifted by the periodic key for both alphabets and K4 shifts two keyed alphabets against each other.

Vigenere, Variant, Beaufort, Gronsfeld, Porta

Main Technique: Column matching or just Hill Climbing

Implementation Issues: Desirable to have a common core. Keep the encipher/decipher routines handy as they may be reused elsewhere.

The first job is to deduce the period if unknown. Some use the Index of Coincidence; the author uses Kasiski. One might have the period as an input parameter in case it is given or in case one wants to override what Kasiski says (if the period finding scheme suggests three, four, six and twelve, or even tries to solve all four, one might just want to say "twelve" and skip the rest).

Column Matching is a semi-exhaustion method.

1. The column layout is known once the period is set (including the last partial row).
2. Take the columns pair-wise and try out all 26*26 keys for each column pair.
3. Grade each column by deciphering with the current key pair, using Digram frequencies to grade (take the log of the raw frequency as usual).
4. If there is an odd period, grade the last column using Monogram frequencies.
5. Depending on the period, keep the N best scoring key pair fragments for that given column. N is selected to be computationally reasonable. If N is too small, there is risk that the solution will sometimes be lost or only partial. The author sets N no lower than 5 but no higher than 10. N would be set based on the current period and the empirical computation cost observed.
6. For each saved column key fragment, stitch them all together (recursion works well here) and then tetragram grade the result.
    a. Suppose the period was six and only the best two key fragments were saved. Suppose further the top keys for each column were:

AY	ST	ED
BE	QM	ST

    b. Then the keys tried would be: AYSTED, AYSTST, AYQMED, AYQMST, BESTED, BESTST, BEQMED, BEQMST and they would be stored in order of highest tetragram score.

The alternative is to do a straight-up hill climb. The author hasn't coded that for these, but it should work well.

AutoKey (Vigenere, Variant, Beaufort, Porta)

Main Technique: Column matching or just Hill Climbing for Porta

Implementation Issues: Desirable to have a common core for Autokey that shares code with the Main core above. Porta is more practical to hill climb for this one.

Running Key (Vigenere, Variant, Beaufort, Porta)

Main Technique: None, unsolved problem

Implementation Issues: Encipherment actually destroys information

Failure Rate: No known computer solution

Running Key is a bit of a cheat. One basically takes one half of the cipher and uses it as the key to encipher the other; the resulting ciphertext is half as long.

There is no distinction between the cryptanalyst and the legitimate receiver. This is pure puzzle and (for the author, at least) an unsolved problem precisely because it destroys information.

Possible solution ideas: It could be possible to look at all pairing if "key" and "plain" in the appropriate tableau and, based on the monoalphabetic frequencies, make a predictive pairing for each "cipher" letter. One could then take the top several candidate pairs and begin recursively constructing semi-plausible text, using tetragram or even digram grading to construct likely text. That might feed a manual process. The problem here is a combinatorial explosion. Suppose we used the top five pairings? One would only be able to run such a process in ten character chunks; to do more explodes the computation costs. It is uncertain that tetragram scoring would be effective in such a short string.

But, so far, this looks like a puzzle cipher that favors manual approaches in the end. Considering its low frequency of appearance, the author has not yet invested in solutions; this one is still manual.

Progressive Key

Main Technique: Hill Climbing

Implementation Issues: Solves like the others with only the added work to try (and then adjust for) the progression index. A little slower, but not much.

Interrupted Key

Main Technique: Like Vigenere or by hand

Implementation Issues: This one depends on whether the con constructor is serious about the "interrupted" concept to make the key randomly restart. A contributor can make one by using by two or three copies of the key, most of which are incomplete, which allows regular Vigenere family constructor code to be used. In that case, it becomes a Periodic with a larger key than usual.

Failure Rate: High if it is truly an “interrupted” key

This is another cipher that is almost as much work for the legitimate receiver as the cryptanalyst. However, it should still decipher fairly easily for the receiver compared to others of this class.

Slidefair (Vigenere, Beaufort)

Main Technique: Hill Climbing

Implementation Issues: Not really much like Porta in the end, but accessing Periodic common functions is still nice. More aligned with Portax.

Slidefair is extremely vulnerable and fast. So much so, the author's solver solves all three systems, including Variant. It is proabably only necessary to solve Vigenere and Beaufort as Vigenere and Variant solve identically with the keys being variants of each other.

Portax

Main Technique: Hill Climbing

Implementation Issues: Not really much like Porta in the end, but accessing Periodic common functions is still nice. More aligned with Slidefair.

Portax can have successful tip dragging because of a serious weakness: If the first letter of a plaintext alphabet is from the letters N through Z, the resulting encipherment is independent of the key. There are also limits substitutes, which further constrains tip placement. However, in the author's own code, he has not bothered with this as it is pretty vulnerable to hill climbing. See JA73 Novice Notes for more.

Gromark

Main Technique: Hill Climbing

Implementation Issues: Straightforward

Gromark uses a mixed key, but that is irrelevant for computer solution which is presented with the pseudo random key anyway. The running key is an interesting, Fibonacci-based method, but as it is always given, it is no problem for the computer to reconstruct. Gromark is at the edge of what paper and pencil can do, but is right down the middle of the highway for computer solution.

Periodic Gromark

Main Technique: Hill Climbing of dual keys

Implementation Issues: Author's code is incomplete, needs investigation.

Failure Rate: Fairly high at present.

The correct concept probably alternates the hill climbing of the regular alphabetic key and the period key.

Homophonic

Main Technique: Key Exhaustion

Implementation Issues: None

While this is exhaustion and not with a Factorial Key Set but exhaustively performs 25 to the 4th power executions, that is still only 390625 total iterations, quite tractable on a modern computer. Accordingly, this is one of the most vulnerable periodic ciphers.

Numbered Key

Main Technique: Hill Climbing with Duplicate Migration

Implementation Issues: Tuning, effectiveness

Failure Rate: Fairly high as it is not clear how to successfully migrate the multiple substitutes in or out or how and whether to adjust tetragram scores for the reality of multiple substitutes.

The cipher, by design, allows some number of plaintext letters to have duplicate enciphering choices. How many can be determined from analyzing the text. In addition, it is possible to designate some number of the rare letters (J, X, W, Z, Q) as unused in the actual plaintext and so allow those to migrate out of the active solution array.

The Quagmires (Quagmire I, Quagmire II, Quagmire III, Quagmire IV)

Generically, these are the same cipher, but the cipher uses keywords in the plaintext alphabet, the ciphertext alphabet, or both. These differences can show up in coding. The solving difficulty varies. Some existing members are a little ahead of the author for this one and research continues, especially on the Quagmire IV. Because the code varies a bit between them, and these differences can show up in solver code, each is discussed separately as needed.

While the author's code is seldom this good, some ACA members have auto solvers for these variants, particularly Quagmire I, which is slightly easier. Ask around about the state of the art. These ciphers are undergoing active research by members. If Quagmire IV is ever auto solved, its techniques can be easily adapted for the others as they are simpler.

It goes without saying that for all four, getting the period (or a small number of them to try) is the first objective. Everything else proceeds from there.

Quagmire I, Quagmire II

Main Technique: Place tip, hill climbing, process manual commands

Implementation Issues: Full auto solve in one step is rare. Tip is extremely useful. Quag I may be auto solved soon. Quag IV may fall to a Wordlist method.

Failure Rate: Today, this is a hand solve. But, the partial automation can make it much easier. The combination of semi-automation and hand solving is very effective for these.

While these may ultimately be computer solves, today, we are on the cusp of that, especially including the author's understanding. So, the emphasis will be on by-hand aids. But, watch this space.

For Quagmire I and II, one alphabet is unkeyed, the other is keyed. Provide the fixed, unkeyed alphabet once and the keyed alphabet for each position of the unknown keyword (eight alphabets if the period is eight).

For Quagmire I and Quagmire II, one alphabet is unkeyed. Make that alphabet the top, singular one. Then, as one places the tip, put the partial fragments under the fixed letter in the appropriate row for the period found (once the period is known, we know exactly what letters go with what alphabet).

Now, extend the alphabets. If there is a fragment like this: E..S and another S.T (where a "." Means a skipped letter position), merge them and replicate them to each other. Thus, there is E..S.T in both alphabets. Further, if any of E, S, or T appear in the other alphabets, they now get E..S.T in those as well.

What's more, any letters in those other alphabets can all be added to each other. So, if there is an alphabet somewhere with DE in it, the whole fragment extends to DE..S.T in all relevant alphabets. These alphabets are shifted relative to each other (unless they share a common key letter) and those shifts need to be preserved.

Once the extensions are done, we select the alphabet with the most letters in it. We then fill in the missing letters randomly and propagate these to all the alphabets. This will imply that alphabets with no letters in common have so-and-so a shifting relationship. This will often be wrong, but we try it anyway.

We decipher, score, and decide whether this is an acceptable solution, despite the likelihood of errors. We repeat the process N times as a hill climber of sorts and quit.

We also may be able to make out the keyword in the repeated alphabets; that will solve immediately.

The best answers may, despite the keys being incorrect at places, show usable plaintext for further work. These can be added (in the author's implementation) to a list of "by hand" functions that precede the next invocation and hill climb. Eventually, the true keyword or enough text appears to finish the answer.

Note that for Quagmire I, the fact that the shifted alphabets are all unkeyed is an additional weakness. JARROD (John Smith) is in the final tuning stages of what may prove an auto solver for Quag I. ANCHISES also had a method, but it is not clear it works at ACA lengths.

Quagmire III

Main Technique: Place tip, hill climbing, process manual commands

Implementation Issues: Full auto solve in one step is nonexistent. Tip is required for the author's method; it aborts without one.

Failure Rate: Ultimately, this is a hand solve.

The extension rules are different, more limited, and more complex for Quagmire III. See Practical Cryptanalysis for a description. In the author's implementation, the plaintext alphabet is held fixed, but that is arbitrary. It will also fail, in almost all cases, to reveal the keyword.

There are also a couple of new ideas. If there is no shift in a Quag III (that is, the plaintext and ciphertext are one-to-one) this will show up if even one deciphered letter is placed. If one sees (for instance) the letter L in some alphabet directly below plaintext L, then using Quag III rules means that *all* the letters are that way (A=A, B=B, etc.). This is something that should be avoided, but just as in real world cryptography, these weaknesses are sometimes there. One gets no progress on the keyword when this happens, but the compensation is that every character in the column is substituted by itself.

Another item is what might be called an identity merge. Suppose that one alphabet has a J under plaintext M and a second alphabet also has a J under plaintext M? That means that while there is a shift, the two alphabets are identical. So, all letters from one of them may be copied to the other, extending both.

Quagmire IV

Main Technique: None

Implementation Issues: Fully hand solve if at all. But see BION's ideas.

The author, except by hand, does not know how to automate this one.

Extensions for Quag IV are difficult and frankly don't help as much, even with longer tips, because there is comparatively so little that can be inferred. Still, it might be possible to come up with a long running auto solver that does things like take a word list, try a new keyword alphabet for one (say, PT), select the best score from that whatever the CT alphabet is, then fix the best PT alphabet and run all the available CT alphabets. After alternating a few times, perhaps a solution might appear? This is an unknown area, but members are trying things out. BION, in the November-December 2022 Cryptogram, suggests just such an approach, with a clever scheme for grading partial decrypts. The author encourages coders to try it; it is just possible this is a solved problem, unknown to the author.

Chapter XI – Philips, Playfair, Seriated Playfair, Two-square, Tri-square, and Four-square

The main thing these share in common is first, a 5x5 Polybius square, and except for Phillips, they represent some form of digraphic substitution. It turns out that even these minor enhancements over periodics or simple substitution tend to cause auto solving to be more difficult.

Phillips

Main Technique: Hill Climbing with Simulated Annealing

Implementation issues: None, per se. But why is simulated annealing, and its tuning, needed?

Phillips is a frustrating cipher, so far, for this auto solver. The code works, but why is it so hard to write? Simulated annealing is a lot of extra tuning and the author finds the coding obscure even by AI standards.

Yet, the cipher is a keyword block transposed by a fixed scheme into eight known alternatives (given a "base" block) and so should in theory cause no special difficulties. In fact, some of the eight are functionally identical. It should be an elaborate checkerboard. The author is not sure, but perhaps it is the relatively short texts that cause this dilemma. Someone can write a simpler solver, but in any case, the scheme with simulated annealing works.

Playfair

Main Technique: Hill Climbing with Simulated Annealing

Implementation Issues: Swapping rows and columns is sometimes better than individual letter swaps.

Playfair motivated this author to finally get a workable simulated annealing program working. The author found it a significant challenge to get his own version working.

There is this paper: https://www.apprendre-en-ligne.net/crypto/bibliotheque/meta/Playfair-compression.pdf . . .which, upon serious study, lead the author to a working implementation.

Also of interest in the C++ version by ANCHISES (Michel J. Cowan). His article is in Cryptologia and tends to be behind paywalls.

Either can instruct the reader, in detail, about how to create a solver for this cipher that works. The author finds that on any method, tuning the simulated annealing is the hard part. There may also be unstated dependencies between the tetragram grading (almost always included in program samples) and the "temperature" function of the annealing.

Change-and-try adjustment of the temperature array may be required if one uses one's own tetragrams and this is extensive work. The motivation for that extra work is to be able to drop in the "other" tetragram lists in other languages (presumably, collected at similar magnitudes) and so have it work in all ACA languages.

Seriated Playfair

Main Technique: Hill Climbing with Simulated Annealing

Implementation issues: Not hard after basic Playfair is working.

The author found that for some reason, Seriated Playfair is more vulnerable than Playfair.

Two-square, Tri-square, and Four-Square

Main Technique: Solving aid

Implementation issues: Multiple Polybius squares complicates hill climbing. Keyword searches may work, but is also extensive as routes have to be considered.

Toward an Auto Solver

The author, for the moment, has despaired of writing an auto solver for these. However, it is imaginable that something could work along the lines of BION's solver Wordlist solver for Quagmire IV.

The technique would be:

1. Generate two random Polybius squares to start. Call them "left" and "right". In Four-square, these are usually designated II and III; in Two-square the ACA and You calls them 1 and 2.
2. Consult the keyword list and construct every Polybius square for each keyword including all popular routes. Put each of these into the left square, holding the right square constant. Decipher and grade. Keep the best left square.
3. Consult the keyword list and again construct all Polybius squares using all routes. Hold the best left square from step 2 as constant. Try all the new right squares, grade them, and keep the best right square as the new right square.
4. Repeat 2 and 3 some relatively small number of times or until both "best squares" do no change. This is the answer

It is imaginable that one could shotgun this approach, though it is already computationally extravagant, but it is imaginable that the initial state of the right square could result in a different square pair at the end.

Even if this technique ends up taking several hours, one might find it worthwhile because they can be a handful by hand. It would be nice to wake up in the morning from an overnight run and see a solution.

This is more debatable as an idea for Tri-square because three squares are involved. Moreover, the "lower right" square, being at the center of each triplet, is "a little more equal" than the other squares. Whether this helps or hurts remains to be seen. However, it may also mean that the right method does the upper right, then lower right, then lower left, then again the lower right and repeats this block of four instead of the more obvious three. That is much more computationally intense, but if it improves the solution rate, it is obviously worth it.

All of this is, of course, typically defeated by multi-word keys like ILIKECODE, which is one of the reason the author has not yet tried it. Even if only one of the squares is multi-word, defeat seems likely.

Issues for the Solving Aid

A serious solving aid still has significant content compared to most. Particularly, there needs to be code that takes a placed tip and fills in an expanded edition of the squares for two-square and tri-square such that false decodes are not created. The code must be further willing, as it proceeds, to merge rows and columns as new material is added. Experience shows that these sparse squares may have to have dimensions as great as 20x20.

There also needs to be the manual ability to add text extensions from the human and for the rows and columns to be appropriately swapped or merged so the human can reconstruct the keywords.

Throughout, the code must be on the lookout for invalid mergers (where merging of a row means some particular letter over-writes an existing one) and reject these.

Foursquare, by contrast, is easier. The squares do not expand, but contradictions still have to be detected.

Chapter XII – Bifid, Trifid Ciphers, and Digrafid

Bifid

Main Implementation: Hill Climbing with Simulated Annealing

Implementation Issues: Can solve without a tip. Forgiving Simulated Annealing.

The author finds that even a mediocre Simulated Annealing works well for Bifid for some reason. Bifid may well be the first cipher one codes up with Simulated Annealing.

The Chi Square test (see Appendix C) is used to calculate the period when it is not known. The period is often given for these, however. See *Practical Cryptanalysis II* for a discussion of how to drag tips over the ciphertext (there are differences for even and odd periods).

Conjugated Matrix Bifid

The author has not worked on this cipher except to conclude that it is more than a trivial simple substitution on top of a regular Bifid.

Trifid

Main Implementation: Hill Climbing with Simulated Annealing

Implementation Issues: Solves much better with a tip; virtually required

If one can code up Bifid, a Trifid is not that much different (the Chi Square is already done). However, just that small added complexity versus Bifid seems to mean that a tip is required as opposed to optional as it is on Bifid.

Twin Bifid or Trifid

Main Implementation: Especially for Bifid, ignore the "twin" aspect and solve individually

Implementation Issues: Does not really work for Trifid without a true tip; at least not presently.

Failure rate: For Bifid, just solving them separately without a tip works at least two thirds of the time. A solution for either creates a fair amount of placed tip for the other one.

Fortunately, Twin Trifids are not common – Twin Bifids are more popular with ACA constructors.

Digrafid

Main Implementation: Hill Climbing with Simulated Annealing

Implementation Issues: Tip is very significant for this cipher. More successful with it.

Failure Rate: Can fail without a tip.

This is somewhat akin to Bifid or Trifid and so a lot of the same discussion applies, despite the fact that it doesn't "look much" like Bifid or Trifid.

Chapter XIII – Headline Ciphers and Cryptarithms

Headlines and Cryptarithms do not – entirely – fit into tidy auto solving categories. In many respects, both actively resist auto solving, to varying degrees. The author can and does bring substantial auto solving function to bear. But, there is a long way to go before we get to "input cipher output answer" in far too many cases for these.

Headline Cipher ("Headlines")

Main Technique: Combination of Patristocrat solvers and hand-built key reconstruction

Implementation Issues: Ciphertext is allowed to be short and "unfair" (abbreviations, proper names)

Headlines are a past time invented in the bowels of the NSA and somehow became known outside of the agency. So, it does not really correspond to normal ACA ideas about ciphers and solutions.

It is five Aristocrats, plaintext from newspaper headlines, each encoding with a K3M key. It is the same key, except the shift between the "base" K3M standard alphabet has a different shift for each text (well, there may be a repeat shift, but it is rare). It can be related to Quagmire III with a mixed key. The shift is denoted by a keyword, called the setting, which appears when you line up the solved alphabets in the correct alignment.

To claim a solution, one must do more than recover the text (which can often be done standalone). One must recover at least two of:

1. The keyword in the standard alphabet (here a K3M)
2. The "setting" (the word representing the shift between the five K3M alphabets) and
3. The "hat" which represents the keyword used to scramble the K3M.

For the author, solution consists of these steps:

1. Solve the five "Aristocrats.
    a. Using regular automated means as much as possible, until at least three or four of the headlines are solved. That can usually be managed in several "rounds". The first "round" is often garbage, but one might see a word or some almost-word and try that as a tip. Manual means if all else fails.
    b. Once those are solved, recover any one of the 26 letter chains (see classic K3 keyword recovery elsewhere) by combining chains from the three or four solutions. One might automate the initial chains from the solved alphabets, but the author does the mergers manually.
    c. "Decimate" the 26 letter chain to uncover the other 26 shifted alphabets.
    d. With the other alphabets known, spot a single CT/PT pair and use that to get the right shift to solve the remaining ciphers. It can be trial and error if necessary.
2. With all the alphabets and their shifts known, stack them up in order (PT over CT or, if that fails, CT over PT) and figure out what the "setting" is (some vertical line will reveal it).
3. Use the UVWXYZ method (see below) to figure out the keyword. This is at least as much art as science.

4. Once the keyword is known or guessed:
    a. try out various methods the cipher might have been "blocked" for mixing, looking for short fragments in common with any of the 26 alphabets.
    b. Once you have spotted short fragments in a given alphabet (two or three letters, typically), see if it is the entire alphabet (it will be out of order). This confirms the keyword recovered in 3 is correct.
5. Optionally, use the transposition information recovered in 4 to recover the hat (the author usually does not bother as there is enough to claim a solution).

The UVWXYZ Method

This is written up in the Cryptogram also, but the concept is simple: It is likely that most of UVWXYZ are not in the keyword. If that is mistaken, it will usually be obvious which are in the key later on.

The following presumes that this will all be handled by code to assist in the work.

For each of the 26 alphabets, treat (arbitrarily) the UVWXYZ fragment as the PT alphabet and, for each shifted alphabet, put the corresponding letters from the current shifted alphabet above UVWXYZ. Note that this entirely disregards the alphabet mixing. In fact, it works around it. This is the point in the end.

Examine all shifted alphabets for interesting and plausible texts. Maybe it will be IJKLOP. Maybe it will be EPUBLIC (a likely keyword fragment). Don't worry about the implausible ones – there will be many of those.

The result might also be IJXLOP. That X suggests W might be in the keyword. Drop it and see if it works better with UVXYZ.

Take that interesting text and put it in the PT row and put the same alphabet's CT versions above it. This sometimes works and sometimes doesn't.

Add and subtract from the UVWXYZ fragment and from the second text. This again is trial and error and there may be a point where it stops working. Eventually, though, the keyword is guessed even if incomplete. Sometimes, the extensions that ought to work do not work. This often means it is wrong, but it also sometimes means nothing. Perhaps the irregularity of the transposition matters even here. Whatever the reason, these fragments do not always contain the whole alphabet. But it can be enough to spot a keyword. The next step will double check and confirm the keyword even if parts of it were never seen under UVWXYZ's ideas.

Proving the Keyword

To prove the keyword, proceed as follows:

1. From before, collect all the decimations.
2. Try out the would-be keyword in a variety of blocks, creating what sooner or later will be the actual keyblock.
3. Determine which in #2 is correct by examining each would-be keyblock for short bits of key – three characters are about right – that occur in one of the decimated alphabets. The length of three is chosen because many of the blocks end up with two or three letters in them.

Remember, the actual alphabet is transposed, so other things may be adjacent to the matching parts.

4. Once a three-character candidate is detected, see if all the fragments, in any order, can be found in the rest of the cipher alphabet. If it can, you have proved the keyword and even know how it was transposed.

Cryptarithms

Main Technique: Recursive solving of constructed additions; custom code

Implementation Issues: Varies by the problem. Many fall outside of the general recursion, requiring custom code. Persistence will always get a solution, however.

Cryptarithms are a class of puzzle and not quite a true cipher, though members of the ACA do not really care. They are fun puzzles and have existed in organized form in the ACA since no later than 1938.

The basic concept is that they are simple substitutions of letters over the ten digits of the regular decimal numbers (with occasional forays into bases between 9 and 16). The solution involves math, logic, and sometimes a little exhaustion, to come up with what the constructor guaranteed to be the sole mathematically sound solution.

Types of Cryptarithms

Over the years, many things have been published. Currently popular are:

1. Square roots
2. Cube roots
3. Division
4. Multiplication
5. Addition
6. Subtraction
7. Equations (a pair of Additions; a pair of Subtractions; one of each)
8. Sudokus

A wide variety of infrequently or one-off items such as Magic Squares or special problems of various sorts may be seen.

A great fraction of the problems have several additions and subtractions in them. Consider this division problem from November-December 2023:

C-5. Division. (One word, 0-9) by SHMOO

CAUGHT / ALE = ACA; - CHUI= ILLH; - IERR = CRCT; - CHUI UUR

This sets up as:

```
     _______ACA
ALE ) CAUGHT
    - CHUI
      ----
       ILLH
```

```
-  IERR
   ----
    CRCT
-   CHUI
    ----
     UUR
```

General Recursive Method

With the clever introduction of a “guard digit” (a special character treated as zero, but never allowed to be substituted), one can turn most of this into an array of additions, with the bottom row the sum. For this ND 2023 C-5 problem it becomes:

ØØUURØØCRC0ØILL

ØCHUIØIERRØCHUI

ØCRCTØILLHØCAUG

Note that Ø is representing that special character with the Norwegian slashed “O”. It could be any character at all, but this choice has explanatory power. This 3xN array can be fed to a unified, recursive solver that proceeds up to down and right to left, preserving carries.

The recursion process should note whether the next position is a number or a letter of the alphabet.

If it is a letter of the alphabet, it is replaced with the next available unassigned number. Then, the entire puzzle is scanned and wherever that letter appears, the current number takes its place. One then recurses to the next letter down.

If it is already assigned (that is, it is a number or a guard digit) and not the bottom of the column, recursion continues.

If one reaches the bottom of the column, then the substituted values on top should be used to create the sum, including any carry from a prior column. If the contents of the bottom of the column are a letter, and if the number represented by the sum is not assigned, assign it, propagate it as before, and recurse, moving to the top of the next column on the left.

If the number represented by the sum is already assigned to something else, this level of recursion has failed and the computation returns to the prior level to unassign the last letter, where found and all others “to the left” and reassign it to something else, reassigning leftward with the new value (if all assignments have been made, roll back farther).

If the number represented by the sum agrees with an already assigned value, continue the recursion with the next column to the left.

Note that guard digits are never assigned but always treated as if they were zero. Note further that with proper construction, a carry into a column of guard digits, indicates a failed sum even if the number at the bottom of the guard digit column to its right is correct. Sometimes, more than one guard digit is needed to make things like up properly.

Note that guard digits allow all additions and subtractions to be reduced to a single addition and solved recursively. Note further that cube roots, square roots, and divisions can be recast as additions. If there is enough material in the additions, it will solve without considering the multiplications involved in the arithmetic.

There are thus three outcomes:

1. There are multiple solutions usually because some of the letters are not assigned. Solution will involve investigation of the divisions and multiplications themselves.
2. There is no solution. Everything is in the array assigned, but everything is assigned without consuming all letters.
3. There is a single solution. This is the most common and best outcome. If this happens, the solution is complete.

In #1 and #2, one can investigate (for instance) the dividend and the divisor for further restrictions on the sums in question. One might even write code that used the available multiplications in their entirety or in part to further restrict the available letter values and, perhaps, provide enough added restraint to enable a solution to emerge. Square and cube roots can be helpful as well, because the left most value of the root is seen squared or cubed, respectively, in the first subtracted item.

Custom Code

If the problem cannot be reduced to the general addition array as above, or supplemented readily to finish the answer, another way out is to write custom code. This can be done in any language; Python is a good choice because its list handling is easy and intuitive, but any language can be made to work.

Basic functions are:

1. Fixed assignment. Maybe a letter is known to be one value. Common candidates are 0, 1, and 9.
2. Range assignment. Iteratively, all values from 0 to 9 are assigned. If it is known the number cannot be zero, iterate from 1 to 9.
3. Sum a list, remembering the carry. In Python, it might be expressed as: (C,carry2) = addList((U,U,carry1)) Python can return lists and the syntax shown enables a list of two results (the sum and the carry) to be split into its constituent parts.
4. "IF" logic. After this: (R2,carry3) = addList((U,H,carry2)) . . .one might want to know if R2 is equal to the already assigned R and whether carry3 was zero as it should be in this particular problem. The concept is that a failed check should basically be a "continue" which skips all logic "beneath" it.
5. ALLDIFFERENT. A list of already assigned variables, provided as a list, could be checked to ensure each are different as required by the Cryptarithm rules. As in "IF", all future execution "beneath" the ALLDIFFERENT call that returned false would be skipped.

This technique, even if naively done (e.g. right to left from whatever the problem gives) makes it quite easy to set up and perform the additions. One then can write supplemental logic to produce the multiplications or divisions outright.

In square roots or cube roots, one can go farther and make assignments based on assigning the root from 1 to 9 and then squaring or cubing it and assigning all the result letters accordingly. That will get three or four letters of the bat; some combinations might fail at once. In this root, for instance, DA'FF'OD'IL gives root BULB. Moreover, "DS" is what is subtracted from the DA part. So, B*B would be

(10*D)+S, exactly. If B*B is less than 10, it is immediately eliminated. Survivors assign three letters before any other work, a sizable reduction in computation.

Even with only the IF and ALLDIFFERENT logic, the code tends to be quite exclusionary and may execute far faster than naïve expectations because it skips a lot of the deeper logic.