Churn algorithm

I have described how Simulated Annealing is a considerably more effective algorithm than Hillclimbing, but is more complicated to program and – in particular – to setup. I have found a way to avoid these difficulties with a related algorithm that I call the Churn algorithm.

To develop Churn, I studied what happened in a typical SA run. In particular I wanted to find the probability that a worse key would be accepted (which is the downward step). This probability is related to how much worse the key is, as is shown in the chart below drawn from the data I collected.

To mimic this situation in an algorithm, I created a series of 100 numbers from the chart to represent probabilities of acceptance.
This series is just the worse scores from zero to 60, but with frequencies in the series to enable representation of probabilities of their acceptance.

```
1,1,1,1,1,1,1,2,2,2,2,2,2,2,3,3,3,3,3,3,3,4,4,4,4,5,5,5,5,5,5,6,6,6,6,6,7,7,7,7,7,8,8,9,9,9,9,10,10,10,10,10,11,11,11,11,12,
12,12,12,13,13,14,14,15,15,16,16,16,16,17,17,18,18,19,19,20,21,21,22,23,23,24,25,26,27,28,29,30,31,33,34,37,39,41,43,48,57,60
```

For example, say a particular key has a score worse by 2. To decide whether to accept this key, the algorithm selects at random one of the 100 numbers from the series. If this chosen number is greater than 2, then the key is accepted. You can see from the series that 86 of the numbers are greater than 2, so the probability that this worse key will be accepted is 86/100, or 86% as the chart requires. Similarly if a key has a worse score by say 42, the probability it will be accepted is just 4%.

To implement the Churn algorithm is now very simple. First in your hillclimber add an array called value[] that contains the 100 integers above.

In your hillclimber, after a change is made to the key, you will currently have a line like:

```
if(score of changed key > original key)
    accept the changed key;
```

As a second change alter the coding to this:

```
x=random(100);
if(score of changed key > original key – value[x] )
    accept the changed key;
```

and you have changed your hillclimbing algorithm to Churn.

You will get an immediate boost in solving performance. This will not be noticeable for quick solving ciphers, like simple substitution, but will be very evident in more difficult Transposition and Polygraphic Substitution ciphers such as the three below:

(a) Incomplete Columnar (25 columns)

```
SECER ISEAS DTLAE HERHL JTSSH FWRTF LHASR EIPMM OMLAS ETNSO HEDSI EDOHO RENHD LAENT STTPO REAME
KHRTD ATCVN ATHLP TTPAI WOAPI NEOEE SHITI ITBAT NFEST ANTES TCRFI OINAO ADNTU LEOUS SIFRO NDIOW
BSIES TEUEO AHSSF GEESS GAHTH HOTAO BIBHW TEAIE SHTHA TEDIU NGIRE SOESU ITUTE BDHTS ORLFN ABTLE
SAUYM SOORA
```

-Churn: solves in average of 0.5 million keys
-Hillclimb: solves in average of 11 million keys

(b) Cadenus

```
MHDNE ARNTU NARBO OYBHN NAIAO TYTTI NEDIT HBLIE NREEO WTDFE RIEAT NSAIB HENOC NNALE TEOEF IEATA
NRGNC ETATU GUMBH ELDCW SIIND EONRS EPHAT IUOHE UDEPE MTLIA TAWEN DDBNE BIALG SISDR SAORL IPEOL
TETAA EEDLH YDIOA MYEEB ADHEH INNTH AFHNL HITTM NWINR GSAOV NREAN HAKCO GEEHD ONENM RATBD RTLIE
ROGTS SEICE
```

-Churn: solves every time, average 50,000 keys (less than half
a second on my computer).
-Genetic: solves every time, 5 million keys.
-Hillclimbing: in 10 runs of 10 million keys each, fails 6 times.

(c) difficult Playfair:

```
AK ZB BQ GR AK IO IW UY CD ZN EN NL UG NP OV DC RZ UC KW OI PL GU IU NE TO AU BL QW UG DC EW ZB
OE KY WZ IK ZF GM BN GM DE DV FK YW KE ZV
```

-Churn: solves every time, in about 50 million keys
(2 minutes on my computer);
-Hillclimbing: no solution at 4,000 million keys (2.5 hours).

Churn programs for solving these ciphers are given in the 'Churn package' in the download section of this website. These programs illustrate different ways of implementing the Churn algorithm, as is explained in the ReadMe file included in the package. Also in the package is this article as a MS Word document. Log Tetragram frequency tables are also in the download section.

Limitations of the Churn algorithm.

The algorithm depends on the scoring system correctly distinguishing whether the plaintext from one key is better than plaintext from another. This is done, as explained earlier, by assessing the log tetragraph frequency of plaintexts, using a table of frequencies derived from normal English texts.

This system will fail if the plaintext of the cipher is not like normal English. For example the following plaintext was the solution to a recent cipher:

"Junk jewelry jingles, jolts jerky jackanapes. Jealous jades jot jabberwocky, jerboa jumps jaguarundi."

This is full of unusual tetragraphs, and therefore has a relatively low score of just 432. In contrast, a piece of garbled plaintext that makes no sense will score higher. For example, during the Churn process the garbled text below is formed, and this scores 776 -- much more than the solution:

"honyhepermchundrethirsthemychabyanafethearіothagethishallempibychemliahowfthadoamongu."

Consequently Churn (or any other algorithm) meeting a cipher based on the unusual plaintext will favour other plaintexts which are garbled and incorrect but still have higher scoring tetragraphs than the piece of unusual text.

As a result Churn will be led away from the true ciphertext and produce nothing but garbled plaintexts. Use of other n-graphs, such as digraphs or trigraphs, will suffer from the same problem and will also fail to solve.

A way round this problem is to score with words. Although slower, it holds out a high chance of finding the solution when plaintexts are unusual. For example, the correct plaintext above scores 629 while the garbled one scores just 165.

Scoring with words is best carried out by initially putting all the words from a dictionary into a Trie structure. Then the scoring algorithm looks in the Trie to see whether successive plaintext letters form a word. If they do, the score is incremented by the square of the word length –- and the search process continues from the letter following the discovered word.

Again there is a limit to the success of this method. It will not work unless most words in the plaintext are in the Trie. Plaintexts with unusual words will thus demand a very big dictionary, which can demand more memory than the computer possesses or alternatively can excessively slow down the scoring process. However the scoring method usually succeeds.

A second limitation of Churn, or indeed any other stochastic method, is cipher length. As ciphers get shorter so the amount of information available for scoring gets less, and the effectiveness of scoring diminishes. The result is that ciphers take longer to solve and eventually will not solve at all. This limiting aspect is reached at about 70 to 80 letters, depending on the cipher.

Adjusting for different scoring systems.

For ciphers up to 200 letters long, the best scoring system is based on tetragraphs. For very long ciphers, like the 526-letter Playfair challenge cipher of Simon Singh's 'The Code Book', digraphs provide a faster result. You will find a table of log digraph frequencies in the Downloads section.

The overwhelming number of ciphers I come across are 80 to 120 letters long, and here the best scoring system is provided by log tetragraphs. You will find a table of log tetragraph frequencies, made from 12 books by my friend Hank Gibson in the downloads section of this website, in binary format and also as a text file.

If you have made your own tetragraph table, it is likely that your frequencies differ from mine because of different lengths of texts. Consequently your score for a given plaintext is likely to be different to mine. To use my Churn array of value[] you will need to adjust your scores to be in line with mine. This is simple to do. Score a piece of plaintext with your log tetragraphs, and then work out the average tetragraph score. My average is 10. If yours is 8, then your factor is 1.25; multiply all your scores by this factor whenever you use the Churn algorithm.

Adjusting for ciphers of different lengths.

The 100 integers in the array called value[], that I introduced you to earlier, work well with ciphers with 100 to 120 letters. If you have a much longer cipher to solve, the scores of that cipher will be higher because of the extra length. It is then necessary to bring them back to what they would have been for a cipher of length 110 letters. That is simple to do. If the cipher is 220 letters long, then multiply all scores by a factor of 110/220 = 0.5

If your cipher is less than 100 letter, then scores will need increasing. So for a cipher of length 85 letters, increase the score by a factor of 110/85 = 1.3

Factors affecting solving speed.

The longer the cipher, the faster it will solve because there is more information available to score each key. With this additional information the score is more meaningful and thus decisions on whether to accept or reject a changed key are more likely to be correct.

The dependence of solving time with length is exponential, so as ciphers get shorter they get a lot more difficult to solve.

The other factor that influences solving time is how closely the plaintext resembles normal English text. The nearer the text, the faster will a cipher solve. The relationship is again exponential, with solving time extending markedly as plaintexts move further away from everyday English words.

To give you an idea of the size of this effect, a cipher in normal English of length 130 will solve in 1/50th of the time of a cipher of length 86 when using the Churn algorithm. This effect is normal whatever method is used to solve and, of course, is why the Military put limits on the length of cipher messages.