

Cracking the NitNat Cipher

Liam Rafferty

Introduction

The Nit Nat Cipher has alluded cracking from Prof. Brown and previous students for 30 years. Brown's method of acquiring the enciphering program is a testament to the Kerchoff Principle, as well as an intriguing story which can be viewed at <http://www.cs.rochester.edu/u/brown/Crypto/assts/projects/challenge.html> . But, as with all ciphers not equivalent to a one time pad, it is flawed.

Quite simply, the Nit Nat Cipher shifts the last six bits of each character according to a key, clearly surmised from NitNat.pl which can be viewed on Brown's page linked to above. The first two bits of the byte, of course, do not effect the regular alphabet, and they're masked out in Brown's recreation of the code in order to avoid special characters. Also, a cursory view of these six bits in the English will show that A is 000001 and a is 100001. And the capital letters in the alphabet have a 0 in that first of six position, whereas the lower case letters have a 1, and are otherwise the same. Clearly a secret message can be read with out proper capitalization, so we can essentially throw out this capitalization bit. (Also, a computer would have to be able to do an rather complicated grammatical analysis to be able to know if a message was properly capitalized, a task more easily performed by a human).

There are five bits left, and from hereon in, they will be referred to as the first, second, third, fourth, and fifth bits. Each of the shifts can be as long as the message, the length of the message shall be referred to as "n" from hereon in. Therefore, the

keyspace is n^5 . For small n , a brute force attack may be viable. But since the Nit Nat Cipher's keyspace deviously grows larger with the length of the message, the size of the keyspace quickly becomes larger than any encountered before in this course, save the one time pad. The astute reader may notice that the keyspace is really only n^4 since there are n elements in the keyspace which will yield the plaintext shifted by a constant. (And of course, a computer would have a hard time finding where the beginning of a message should be.) So in any method, one of the bits, say the first one, should be fixed and then the others shifted against it. There still remains a mighty large keyspace.

The chosen plaintext attack is obvious and the algorithm for it is simple. The known plaintext attack is also obvious, but turns out to be extremely expensive. The ciphertext only attack, the main result of this report, uses a divide and conquer technique using a bitwise index of coincidence.

In this course, an index of coincidence proved to be a fatal blow to the Vigenere Cipher. It allowed the cracker to use statistical knowledge about English on the ciphertext to determine the keylength without having to decipher anything. There is a similar statistical knowledge about the bits in English which allows the cracker to determine each part of the key without being concerned about finding the others. This analysis proves to be particularly devastating to this seemingly strong cipher.

Discussion and Analysis

The first, and easiest, attack performed on any cipher is the chosen plaintext attack. To ferret out the key with clever input, the cracker can give the Nit Nat method a 11111111 followed by lots of nulls. After the encipherment the cracker has only to

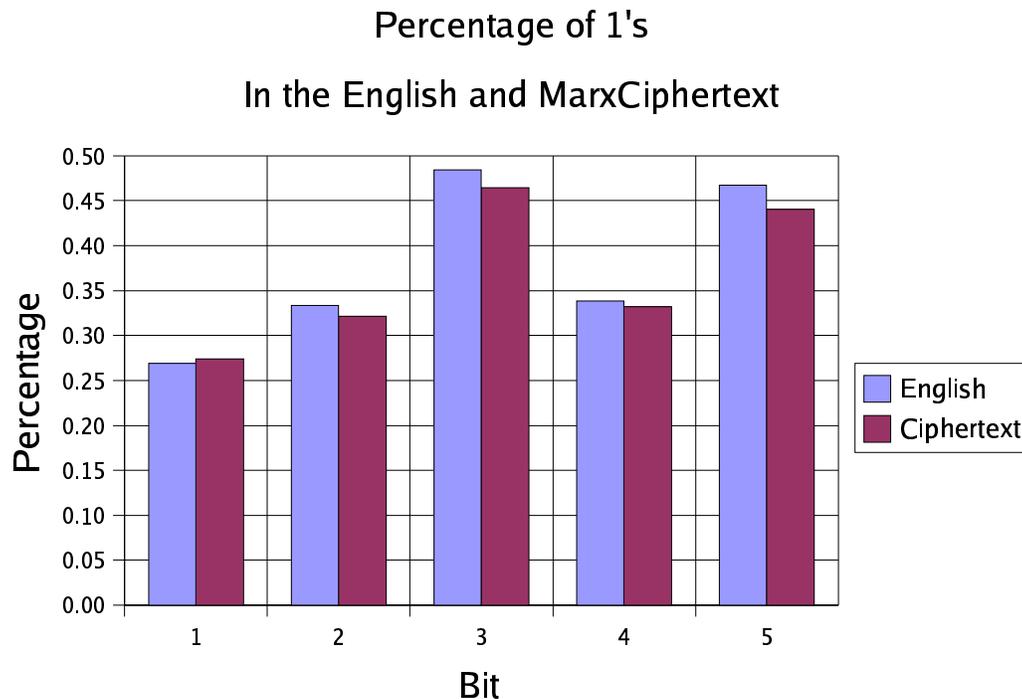
examine how far each 1 went to get the key.

The second attack usually examined is a known plaintext attack. The known plaintext attack is usually easier than a ciphertext only attack, but the Nit Nat Cipher seems to shield itself from such analysis.

The cipher text given by Brown is entitled MarxCipherText. Perhaps a politically/economically savvy cracker could guess that the word “bourgeoisie” was in the text. The first bits of this word are “0 0 1 1 0 0 0 1 0 0” in order for a known plaintext attack to work, every instance of this string of 1's and 0's in the first bit would have to be found and matched with every instance of the comparable string in the second, and so on. The number of times this string appears in the first bit is clearly bounded below by the number of times the word “bourgeoisie” is used, but unfortunately it seems that it can happen relatively often randomly. Let p_1 be the percentage of 1's in the first bit, and p_0 be the percentage of 0's, statistically, the percentage of the text identical to the given string would be $(p_0)^8 \cdot (p_1)^3$ (there are 8 zeros and 3 ones). The resulting probability of this string's occurrence is .0016. Perhaps this seems small, if the text was only 10,000 characters long, then it would only happen 16 times, but it can not be ignored. The probability for the second bit's string is .00072, the third bit's string is .00047, the fourth bit's string is .00019, and the fifth bit's string is .0003. Therefore, the complexity of comparing all of them is each of their probabilities times n multiplied together, which turns out to be $3.11 \cdot 10^{-7} \cdot n^5$. Although this constant coefficient seems small, $O(n^5)$ is still $O(n^5)$. Interestingly, since we don't get to fix one of the bits as we did in the brute force attack, the known plaintext attack is $O(n^5)$ and the brute force attack is $O(n^4)$, the known plaintext attack is more complex than the brute force attack!

The skyrocketing keyspace and being well shielded from known plaintext attacks may make the Nit Nat Cipher seem uncrackable, but it preserves some statistically useful information about English.

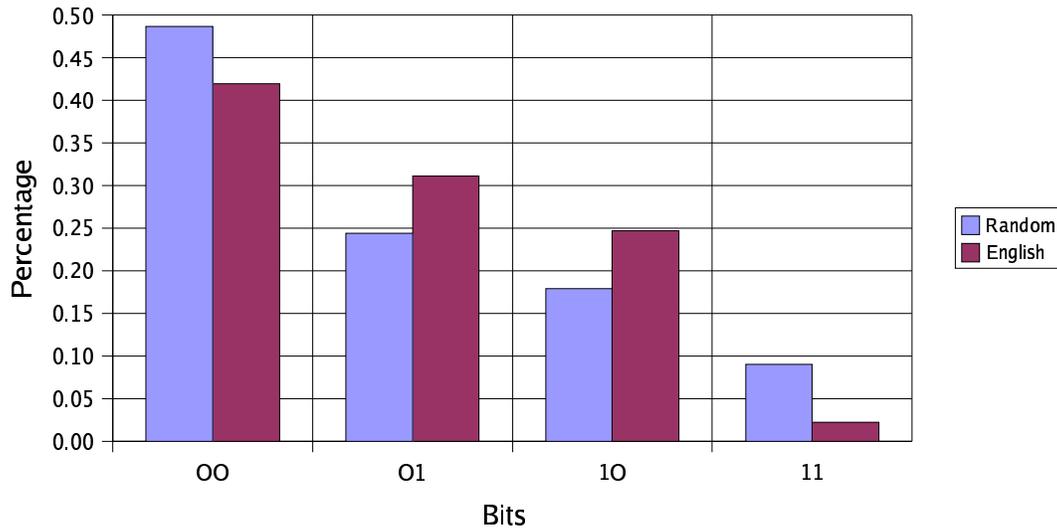
The bitwise percentages of 1's in some plaintext and the MarxCipher text show that the statistical occurrence of 1's is very similar:



Using these percentages, it is possible to calculate the likelihood of, say, the first bit being 0 and the second bit also being 0. Comparing this to the actual occurrence of a 00 case in English shows that there is quite a difference in probabilities.

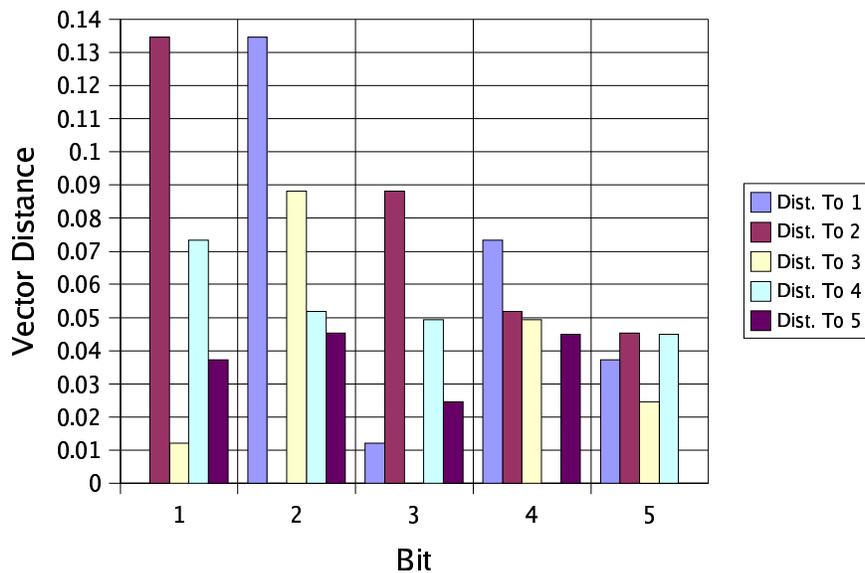
The four cases: 00, 01, 10, 11, can be put into a vector for the random case and the statistical analysis of English. The difference between these vectors, if large enough, can be used to identify English from all the other possible shifts of the first bit against the second bit.

Comparing the First and Second Bits



There is no theoretical reason why one bit should be better to compare with another, so all the possible combinations were analyzed and are shown in the following graph:

Distance Between English and Randomness

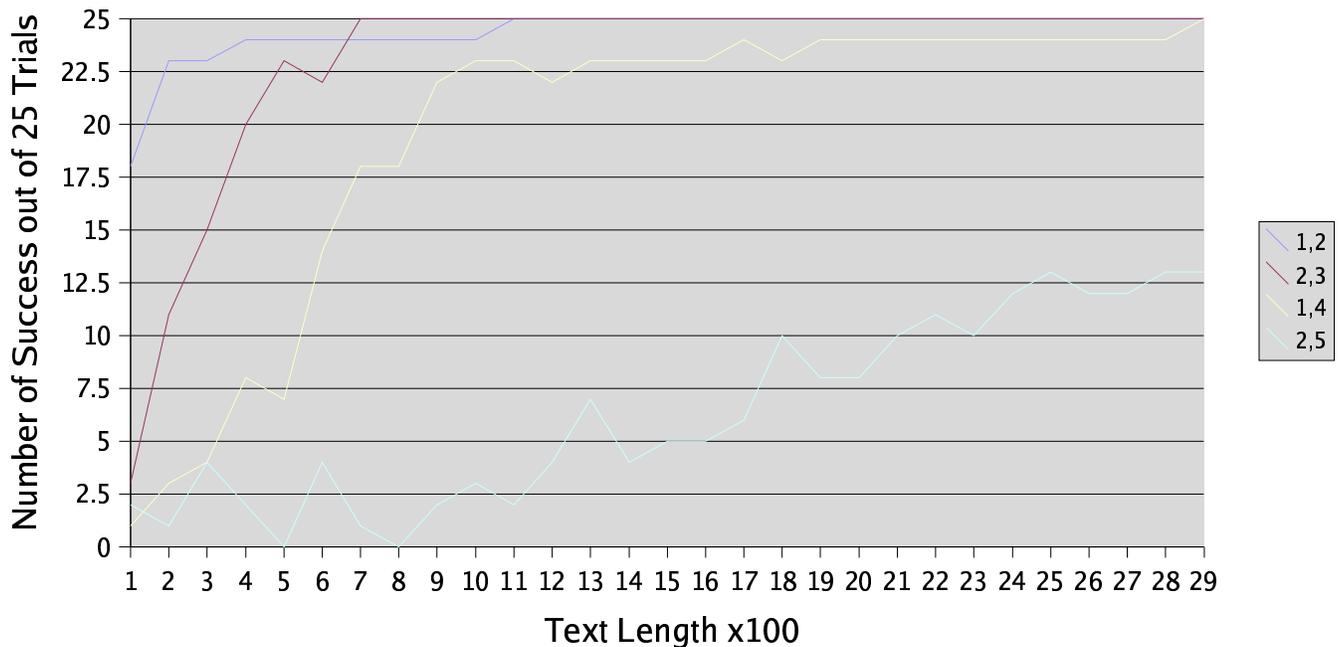


The data indicates that the second and fourth bits should be compared to the first, and the third and fifth bits should be compared to the second. Such an analysis will yield the key

relative to the first bit. Notice the relative differences in vector distance, in particular how small all the distances to 5 are.

These distances were entered into the findTheKey_old.pl and then 25 pieces of text were chopped up and tried for every 100 characters as follows with the analysis.pl method:

Success of Finding the Key for Each Part



The astute programmer will notice that in the analysis program the text is never enciphered, but the astute reader will realize that there's no need to. Every possible shift against the fixed bit is tried so it matters not if it is shifted beforehand. Interestingly this implies that the success of the method is independent of the key and relies solely on plaintext chosen.

Clearly, the distance between the English and Random Vectors has a large effect upon the ability to find the proper key. Finding the fifth key is particularly tricky. But, even in the fifth position, the longer the text, the more accurate the key finding

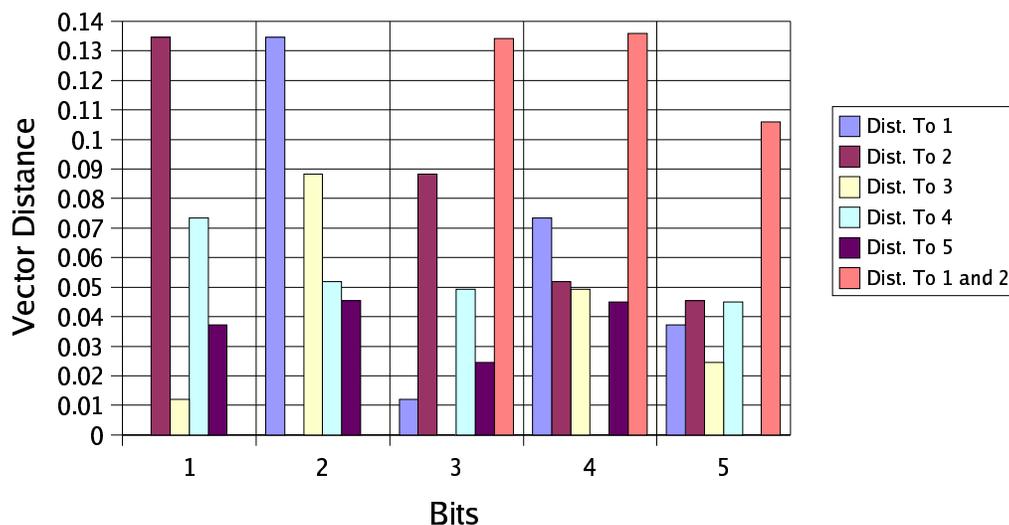
algorithm.

How complex is this algorithm? It's main component makes n comparisons n times for each two bits. So, naively, it's $O(n^2)$. But it should be clear to the reader, that there's no reason to make n comparisons, since a sufficiently large amount, say 10,000 would make the difference between English and randomness obvious (even less if we're only concerned with the second compared to the first). Unfortunately, the 10,000 comparisons must be tried n times, since there's n possibilities for that relative key. But $O(n)$ is most likely as fast as any cracking algorithm could get.

The graph and the trials clearly show that the first and second bits are relatively easy to match up. Perhaps a better method would match the first and second bits, and then match the third, fourth and fifth with the two. The possibilities would be: 000, 001, 010, 011, 100, 101, 110, and 111. The relative distance between English and Randomness of this comparison is add to the following graph:

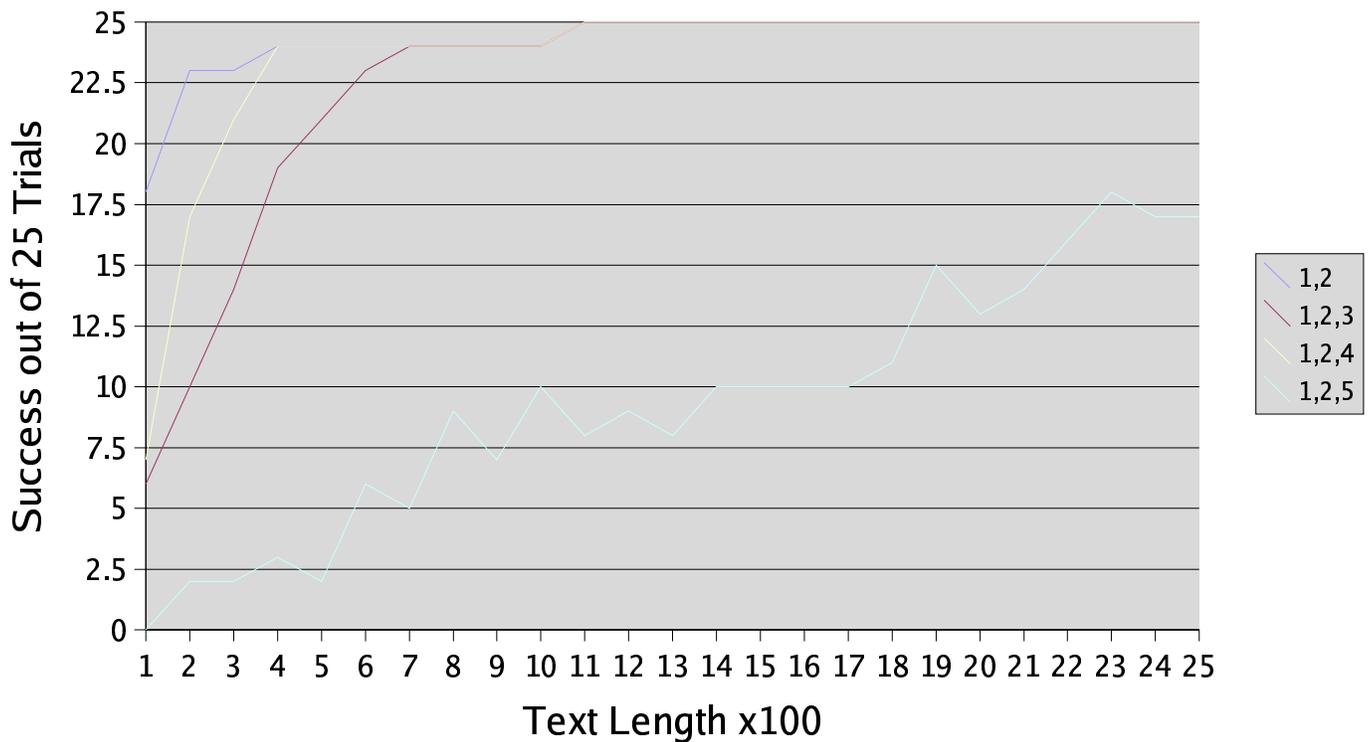
Distance Between English and Randomness

Including the 1 and 2 Comparison



The distance used to find the third and fourth bits is now comparable to the accuracy had in finding the second relative to the first. The distance used to find the last bit is now twice what it used to be, but the pesky fifth bit still creates problems. The analysis using this method per length of text is shown as follows:

Success of Finding the Key for Each Part



Although the accuracy of this method increases much more rapidly with text, it is clear that some other method should be created to find the fifth bit since finding the second, third and fourth are so much easier. And once the first four are fixed, shifting the fifth will yield English, which is easily recognizable.

This method though, despite the fifth bit only being found at about 80% for

the text length of the MarxCiphertext, did succeed in finding the key (the other method did not) and the decrypted text is given in the results section of this report.

Conclusion

Perhaps the next logical step would be to have the program inform the user how certain the key is. This could be done by saving the second lowest distance, and if it's too close to the lowest, informing the user.

The findTheKey method seems to be rather efficient and accurate at the text length challenged. And perhaps for the smaller values a brute force attack would be viable.

In any case, the NitNat Cipehr was not nearly as uncrackable as it initially seemed.

Methods

Following is the findTheKey method, the old method that used the first set of statistics is so similar, only this one will be printed.

```
#!/usr/staff/bin/perl

# use:
# perl findTheKey.pl ciphertext

# read the input file into a string
$input = getinput();

# unpack input string to array
@arr = unpack( "C*", $input);

# how many chars is it?
$textlen = @arr;

$mask = 0x01;

my @stats;

$min1_2 = 1;
$min1_2_3 = 1;
```

```
$min1_2_4 = 1;
$min1_2_5 = 1;
```

```
for( $i = 0; $i < $textlen; $i++ ){
```

```
    for($k=0; $k<4; $k++){
        $stats[0][$k] = 0;
    }

```

```
    for ( $byte = 0; $byte < $textlen ; $byte++ )
    {
```

```
        $one = $arr[$byte]>>4 & $mask;
        $two = $arr[( $byte+$i)%($textlen)]>>3 & $mask;
```

```
        if( $one == 0 ){
            if( $two == 0 ){
                $stats[0][0] += 1;
            }else{
                $stats[0][1] += 1;
            }
        }else{
            if( $two == 0 ){
                $stats[0][2] += 1;
            }else{
                $stats[0][3] += 1;
            }
        }
    }

```

```
}
```

```
$dist = (($stats[0][0]/$textlen)-.419783992561333)**2+(( $stats[0][1]/$textlen)-
0.311100779629497)**2+(( $stats[0][2]/$textlen)-0.246629354123453)**2+(( $stats[0][3]/$textlen)-
0.0224858736857163)**2;
```

```
if($dist<$min1_2){
    $min1_2 = $dist;
    $index1_2 = $i;
}

```

```
}
```

```
for( $i = 0; $i < $textlen; $i++ ){
```

```
    for($j=0; $j < 3; $j++ ){
        for($k=0; $k<8; $k++){
            $stats[$j][$k] = 0;
        }
    }

```

```
}
```

```

for ($byte = 0; $byte < $textlen ; $byte++)
{
    $sone = $arr[$byte]>>4 & $mask;
    $stwo = $arr[($byte+$index1_2)%$textlen]>>3 & $mask;
    $sthree = $arr[($byte+$i)%($textlen)]>>2 & $mask;
    $sfour = $arr[($byte+$i)%$textlen]>>1 & $mask;
    $sfive = $arr[($byte+$i)%($textlen)]>>0 & $mask;

    if( $sone == 0 ){
        if( $stwo == 0 ){
            if( $sthree == 0 ){
                $stats[0][0] += 1;
            }else{
                $stats[0][1] += 1;
            }
            if( $sfour == 0 ){
                $stats[1][0] += 1;
            }else{
                $stats[1][1] += 1;
            }
            if( $sfive == 0 ){
                $stats[2][0] += 1;
            }else{
                $stats[2][1] += 1;
            }
        }else{
            if( $sthree == 0 ){
                $stats[0][2] += 1;
            }else{
                $stats[0][3] += 1;
            }
            if( $sfour == 0 ){
                $stats[1][2] += 1;
            }else{
                $stats[1][3] += 1;
            }
            if( $sfive == 0 ){
                $stats[2][2] += 1;
            }else{
                $stats[2][3] += 1;
            }
        }
    }else{
        if( $stwo == 0 ){
            if( $sthree == 0 ){
                $stats[0][4] += 1;
            }else{
                $stats[0][5] += 1;
            }
        }
        if( $sfour == 0 ){
            $stats[1][4] += 1;
        }
    }
}

```

```

        }else{
            $stats[1][5] += 1;
        }
        if( $five == 0 ){
            $stats[2][4] += 1;
        }else{
            $stats[2][5] += 1;
        }
    }else{
        if( $three == 0 ){
            $stats[0][6] += 1;
        }else{
            $stats[0][7] += 1;
        }
        if( $four == 0 ){
            $stats[1][6] += 1;
        }else{
            $stats[1][7] += 1;
        }
        if( $five == 0 ){
            $stats[2][6] += 1;
        }else{
            $stats[2][7] += 1;
        }
    }
}
}
}

```

```

$dist = (($stats[0][0]/$textlen)-0.261515628352764)**2+((($stats[0][1]/$textlen)-
0.158268364208569)**2+((($stats[0][2]/$textlen)-0.109291180888348)**2+((($stats[0][3]/$textlen)-
0.201809598741149)**2+((($stats[0][4]/$textlen)-0.126144410271082)**2+((($stats[0][5]/$textlen)-
0.120484943852371)**2+((($stats[0][6]/$textlen)-0.0185966669050855)**2+((($stats[0][7]/$textlen)-
0.00388920678063086)**2;

```

```

if($dist<$min1_2_3){
    $min1_2_3 = $dist;
    $index1_3 = $i;
}

```

```

$dist = (($stats[1][0]/$textlen)-0.345826478792647)**2+((($stats[1][1]/$textlen)-
0.0739575137686861)**2+((($stats[1][2]/$textlen)-0.174477862813819)**2+((($stats[1][3]/$textlen)-
0.136622916815678)**2+((($stats[1][4]/$textlen)-0.121057148987912)**2+((($stats[1][5]/$textlen)-
0.125572205135541)**2+((($stats[1][6]/$textlen)-0.0203043416064659)**2+((($stats[1][7]/$textlen)-
0.00218153207925041)**2;

```

```

if($dist<$min1_2_4){
    $min1_2_4 = $dist;
    $index1_4 = $i;
}

```

```

$dist = (($stats[2][0]/$textlen)-0.220129103783706)**2+((($stats[2][1]/$textlen)-
0.199654888777627)**2+((($stats[2][2]/$textlen)-0.150382662184393)**2+((($stats[2][3]/$textlen)-

```

```

0.160718117445104)**2+((Stats[2][4]/$textlen)-0.157311708747586)**2+((Stats[2][5]/$textlen)-
0.0893176453758673)**2+((Stats[2][6]/$textlen)-0.00455975967384307)**2+((Stats[2][7]/$textlen)-
0.0179261140118733)**2;

    if($dist<$min1_2_5){
        $min1_2_5 = $dist;
        $index1_5 = $i;
    }
}

print ("\n text length: ".$textlen."||| key: 0 0 ".$index1_2." ".$index1_3." ".$index1_4." ".$index1_5."\n");

#### some subrs...

# read input into a string, toss out carriage returns.
sub getinput
{
    my $fname ;
    $fname = shift(@ARGV);
    $out = "";
    unless (open INFID, $fname)
    {die "\n Can't open $fname for input!\n";}

while ($line = <INFID>)
{
    chop($line);
    $out = $out . $line . " " ;
}
chop($out);
return $out;
}

```

Results

The statistical information is vast and best viewed in it's original spreadsheet location.

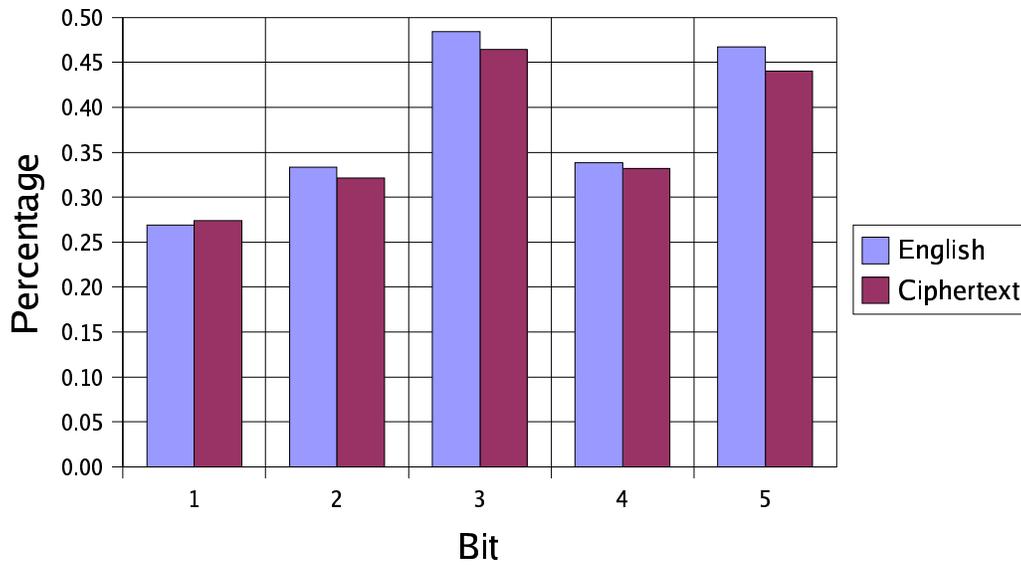
Another excel file was submitted containing all that data.

Decrypted MarxCiphertext:

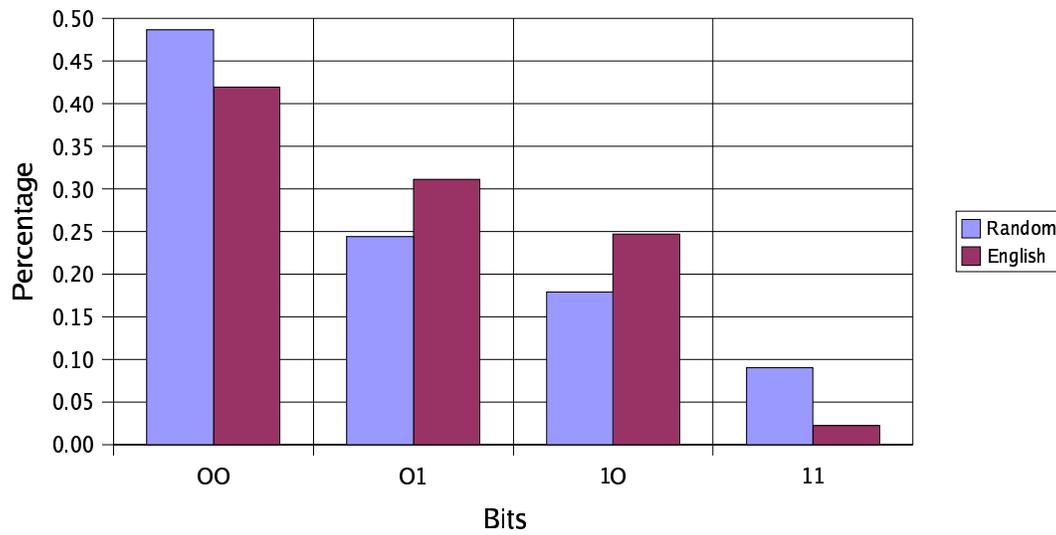
`and` with it` the` antagonism` between` intellectual` and` manual` labor||` after` labor` has` bec
ome` not` only` a` means` of life` but` also` the` Primary` necessity` of life{`` whenl` with` the` d

development of the individual in every sense the productive forces also increase and all the springs of collective wealth flow with abundance only then can the limited horizon of the bourgeois right be left behind entirely and society inscribe upon its banner from each according to his abilities to each according to his needs as the labor process considered as the consumption of the labor power sold to the capitalist shows us two peculiarities the laborer works under the control of the capitalist the latter takes care that the work is carried on properly and that the means of production are put to a suitable use in other words the freedom and independence of the worker during the labor process do not exist secondly the product is the property of the capitalist not of the laborer as the capitalist according to our hypothesis pays the daily value of the labor power it appertains to him to employ this power similarly the other elements essential for the manufacture of the product namely the means of production belong to him consequently the labor process is carried on amongst things which have all been purchased by the capitalist and thus the product is his property capitalist production therefore of itself reproduces the separation between labor power and the means of labor it thereby reproduces and perpetuates the condition of exploiting the laborer notwithstanding this progress the equal right is still encumbered with bourgeois limitations the right of the producers is proportional to the labor they supply the equality consists in measuring this right by an equal standard labor furthermore one worker is married the other is not one has more children than the other etc etc in a higher phase of communist society after the enslaving subordination of the individual to the division of labor shall have disappeared

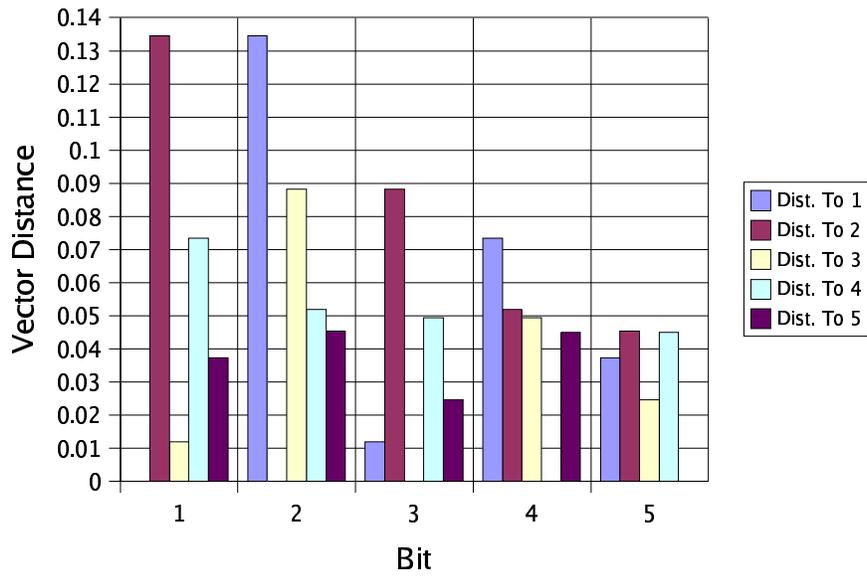
Percentage of 1's In the English and MarxCiphertext



Comparing the First and Second Bits

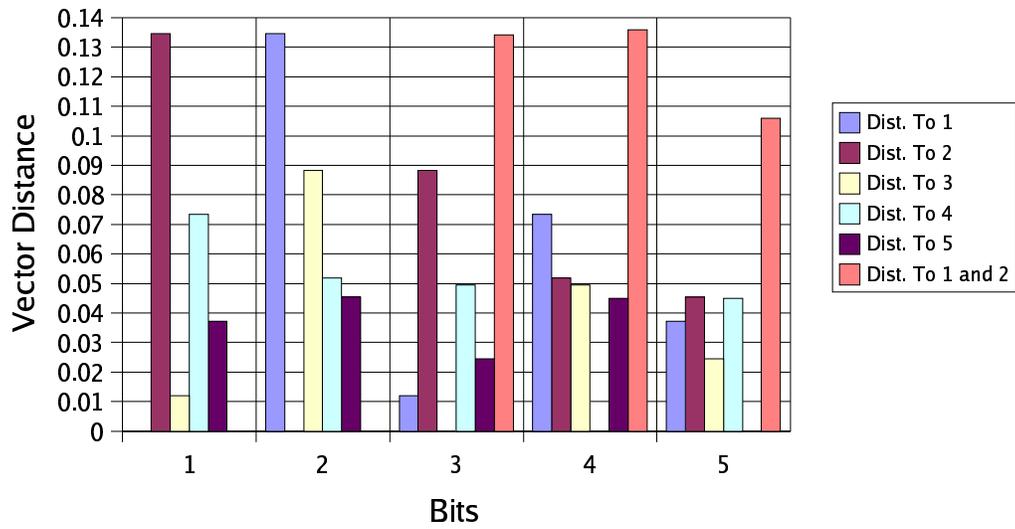


Distance Between English and Randomness

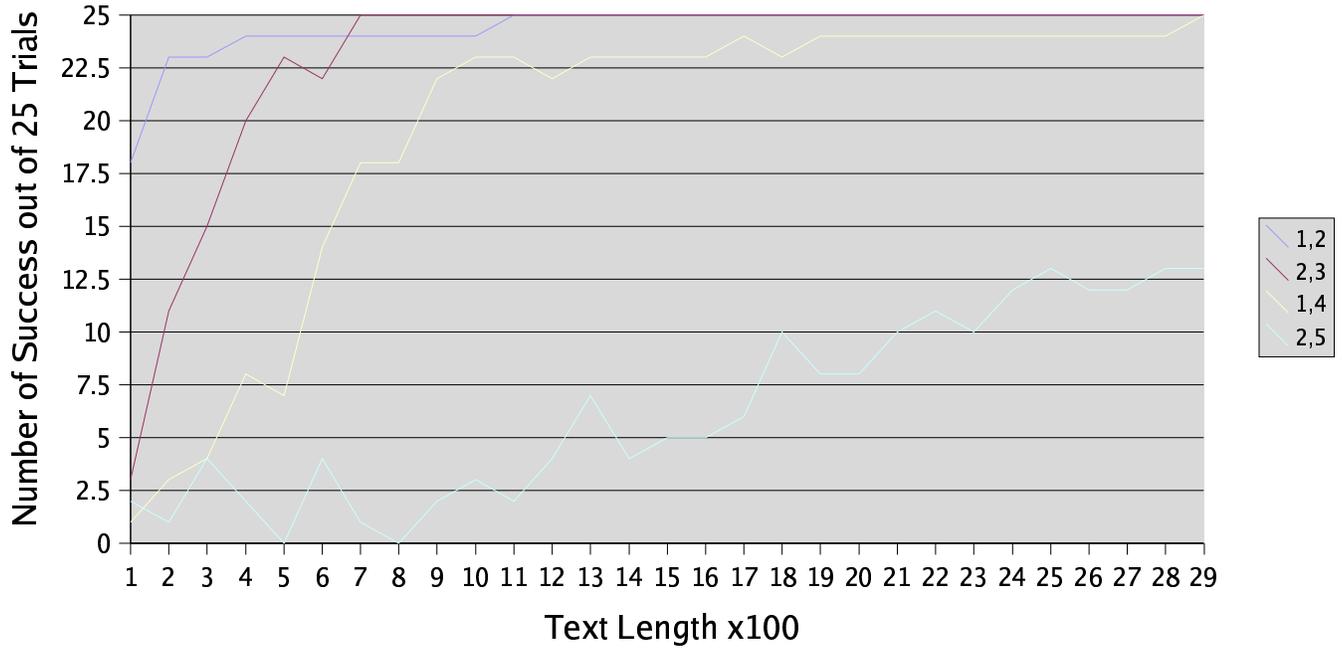


Distance Between English and Randomness

Including the 1 and 2 Comparison



Success of Finding the Key for Each Part



Success of Finding the Key for Each Part

