Elementary Cellular Automata as an Error Minimization Cryptographic Hash

Daniel McKinley

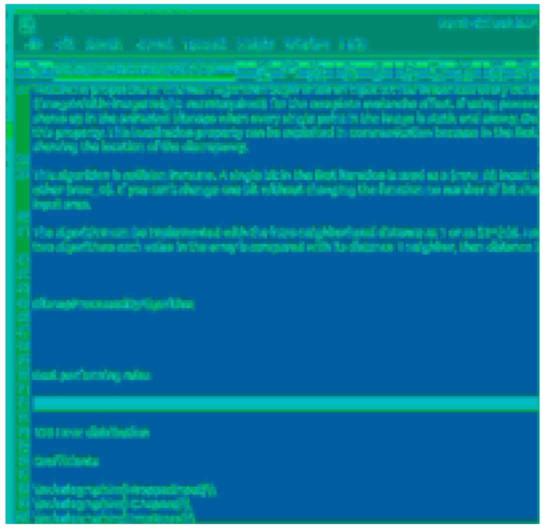
November 2024

1 Introduction

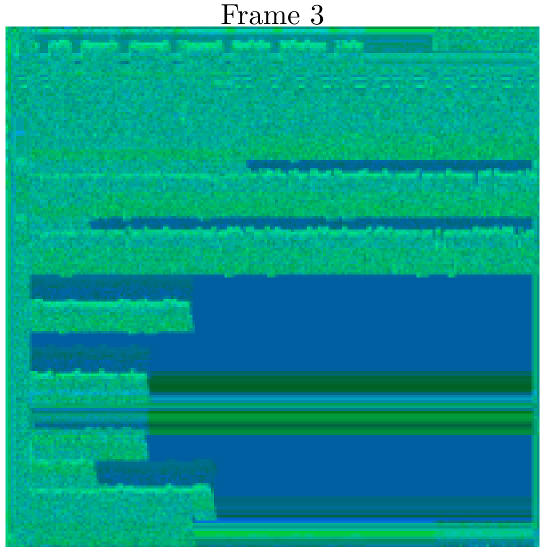
Elementary cellular automata (ECA) are 8 bit extensions of 4 bit logic gate truth tables, done linearly in parallel. [4] Here a subset of 8 of the 256 ECA rules are explored as a hash function and edge detection algorithm in image processing. It utilizes an error-minimization function on 4x4 binary grids. The loops parallel the structure of the Fast Fourier Transform (FFT) and Fast Walsh-Hadamard Transform and the Hadamard matrix shows up as an addition operation. If a Fourier transform breaks a function into sums of proportions of symmetric roots of unity, this hash function breaks a function into a unique set of 4x4 cells of wrapped ECA output with an inverse. The algorithm has applications in image databases and as an edge detection algorithm. General algorithm outline, specific ECA rules, and aggregate properties are discussed. It is implemented in Java at [1] along with more images and gifs at the website.

2 Main Algorithm

Original Image



Frame 3



Frame 6

There are $2^{16} = 65536$ binary 4x4 arrays
Where there are 2^4 possible row_0 neighborhoods for a given ECA rule
Each of these 16 input-ECAoutput arrays are scored by

$$\sum_{r=0}^{3}\sum_{c=0}^{3} 2^r(compressionAttempt_{rc} \oplus original_{rc})$$

The value of the neighborhoods of the minimum and maximum of these 16 sums are noted as the codeword pair of the original binary matrix

The algorithm first creates a truth table for the lossy compression of square binary matrix using 1D ECA. For all $2^{16} = 65536$ possibilities of a 4x4 binary array, wrap the columns so that the grid is cylindrical, use row 0 as the input neighborhood, of which there are $2^4 = 16$ possible. The remaining three rows for all 16 possible row 0 values is the ECA output on the cylinder. For the input row and three output rows, score the XOR difference between it and its binary 4x4 input. The errorScore for a codeword is the weighted sum of discrepancies between this codeword-produced output and the original 4x4 matrix Input[][], where the weight is a coefficient of 2^{row}. The lowest and highest scoring neighborhoods out of the 16 calculated are then two hexadecimal codewords for the binary input neighborhood of size 4x4. All 65536 possible binary 4x4 matrices get a minimum and a maximum codeword.

Input[][] is all 65536 4x4 binary arrays. Each possible binary array's minimum and maximum codewords are found					
Wrap	Input[][]	Input[][]	Input[][]	Input[][]	wrap
	Input[][]	Input[][]	Input[][]	Input[][]	
	Input[][]	Input[][]	Input[][]	Input[][]	
	Input[][]	Input[][]	Input[][]	Input[][]	

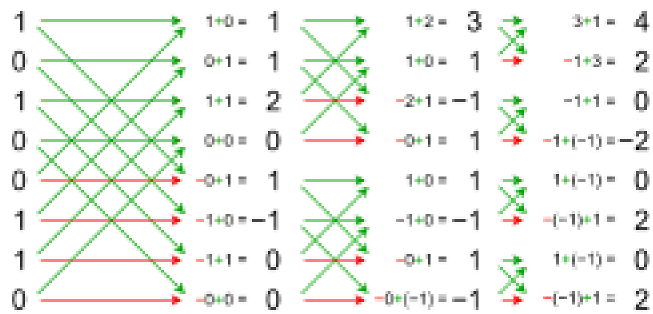
All possible input neighborhoods of a given size are computed with each trial neighborhood placed at row 0. 16 bits input = 4 bit neighborhood = 16 possible codewords. in0..in3 = bitN of each possible codeword. Rows 1, 2 and 3 are then calculated via the standard ECA(rule) operation with wrapped columns					
	ECA[row][column] = Wolfram[rule, row-1, {column-1,column,column+1}]				
Wrap to right	in0	in1	in2	in3	wrap to left
	ECA[][]	ECA[][]	ECA[][]	ECA[][]	
	ECA[][]	ECA[][]	ECA[][]	ECA[][]	
	ECA[][]	ECA[][]	ECA[][]	ECA[][]	

Each possible neighborhood output from above is scored against the input, the final value is the sum of all individual cell scores. All possible neighborhoods are checked for least and greatest error scores and returned as the codewords.					
	err[row][column] = err[][] = (ECA[][] XOR Input[][]), weighted by 2^row				
Wrap	1*err[][]	1*err[][]	1*err[][]	1*err[][]	wrap
	2*err[][]	2*err[][]	2*err[][]	2*err[][]	
	4*err[][]	4*err[][]	4*err[][]	4*err[][]	
	8*err[][]	8*err[][]	8*err[][]	8*err[][]	

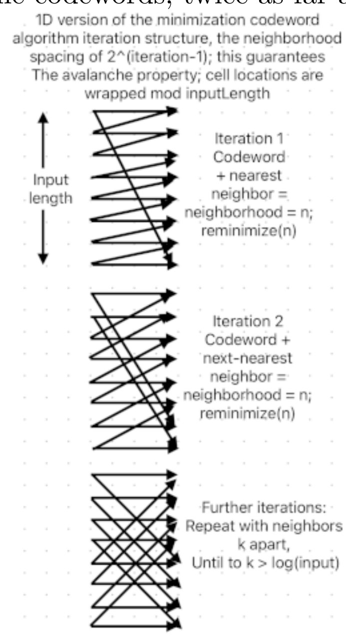
(row,column)..((row+4),(column+4))

For every (row,column) location in input, $(row, column)..((row+2^d), (column+2^d))$ is that value's neighborhood's 4x4 square influence where d is depth of iteration. Every location is part of 16 neighborhoods. Transform the input so that each location is the minimizing and maximizing codewords of overlapping neighborhoods of the ECA rule subset described later. Depths of iteration are the same process with the 4x4 neighborhoods' constituent rows and columns spaced out $2^{depth-1}$ instead of 2^0. Depth 1 is next door neighboring codewords, depth 2 is two spaces away, depth 3 is 4 away. This comparing of neighbors of powers of 2 is the same arrow chart as the FFT and Fast Walsh Transform, except with reminimizations instead of sums and/or products at every level of recursion. These exponential neighborhood expansions guarantee the avalanche property in $log_2(size)$ time. For any given input there is only one possible set of codewords that describe it.

Below is a Fast Walsh-AlgorithmCode.Hadamard Example [3]



If you did the above with the powers of 2 in reverse order you would get this. Below is one axis of this algorithm, and instead of a sum term it's a sum term that gets reminimized. Find the codewords of the codewords, twice as far apart each time.



This transformation has two inverses, one is a lossy conversion from codewords back to output format and the other is an inverse of an iteration which is perfectly invertible. A single rule of the positive 8-tuple of the below is lossy, with an error rate of 3/8. In a rectangular bitmap, the overlapping codeword 4x4 neighbors that influence it, for all 8 of the rules in the subset as well as the 8 max codewords together make a weighted vote on the inverse of the algorithm. Each codeword in the array produces a 4x4 ECA output that is its best guess about the information given to it. Each cell's 4x4 output overlaps with 4 of its neighbors. Each cell of each cell's output is weighted by its relative row and added to its location's tally of votes. At the end if the sum of the votes is positive then it becomes 0, if it is negative it becomes 1. If the inversion is back to output format the votes tally to a single value per location and for an iterational inversion the tallies are made in layers by place value. For a given inversion there is only one possible output.

This algorithm both minimizes the errorScore in a lossy compression, and maximize the errorScore. There are dual min-max versions of most variables in the implementation.

3 A subset of the 0-255 ECA

There are 8 [0,15,51,85,170,204,240,255] of the 256 8 bit ECA truth tables that display the properties of unique codewords for any given input and perfectly even distribution of codewords. 0 and 255 are included because in the 4 rows of the output matrix, 1 is neighborhood input and 3 are output, and so still produce an errorScore and therefore a unique solution. This subset works with both errorScore minimization and errorScore maximization. This list shares most of its members with the XOR-additive list. [2] These two min max 8-tuples are implemented as sets of relevant Wolfram codes in the hash.

4 Hash properties

Avalanche properties of this hash algorithm begin when an input bit has influenced every other input bit. The avalanche property better applies if instead of the FFT-like powers of two spaced neighbors you use sequential neighbors. This property shows up in the animated bitmaps when every single point in the image is static and snowy. If using powers of two the avalanche effect begins at the greater of $log_2(width), log_2(height)$. Once a given input is in minMax codeword form no further data loss occurs and the inverse always produces the input. The lossiness is only when converting from codeword form back into output form. This localization property can be exploited in communication because in the first few iterations the data is irreversible enough to be imperfectly unencryptable while showing the location of the discrepancy. Doing fewer iterations than the avalanche's diffusion threshold but more than zero limits this property however the edge detection does not need the avalanche effect.

This algorithm does not produce collisions. On their own codeword tuples are not distinct even if an individual tuple is; however wrapping the codewords over themselves the tuple becomes unique; the rare artificially produced small neighborhood collisions disappeared when the same sample was wrapped, eliminating the possibility of any collision. An extra wrap loop on each cell adds factor of 16 in both time and memory for an absolute no-collision set. Without that loop, random testing of collisions produced an one in a hundred trillion on a 5x5 matrix that disappeared when the neighborhood was wrapped. Small scale local collisions may be a possibility, large scale collisions are unlikely, and optionally no collisions.

The 8 Wolfram codes each have 256*256=65536 locations in the truth table. For each truth table, each codeword is used in 4096 locations. In terms of area, 1/256 pairs of locations share a codeword. So for 1 element of the 2 8-tuples collisions are common. When all 16 elements are bundled for a gridpoint in the bitmap input collisions are rare because 1/256 has to line up perfectly 6 times (some elements can be cancelled). When the optional loop is used the wrapping array eliminates any collisions.

Intuitively this transform can be compared to the Fourier transform where a function is turned into combinations of sin waves. This hash function turns an input into combinations of 4x4 binary ECA tiles. The tiles that make it up are all linear; all the rules in the subset are linear. The Wolfram codes are symmetries of pass left bit, pass middle bit, and pass right bit, resulting in shift left shift right and no change. So once an input is made into sets of these tiles, the codewords and the inverse is only an approximation of the original using these linear tiles. Any non-linear information disappears from the data. Edges are linear and so get minimized, what happens visually is that the input is made into a contour map with high contrast.

Codeword addition is each codeword's 4x4 ECA output is laid out flat like a piece of paper with another codeword, added, and then the combination is itself reminimized. In constructing the addition tables for codewords, the result was a table that is exactly the logical operation row AND column. The AlgorithmCode.Hadamard matrix can be constructed using this operation by summing the ones in (row AND column) taking it mod 2 for the boolean AlgorithmCode.Hadamard matrix. Experimentally, in the inverse phase the binary AlgorithmCode.Hadamard value of the codeword was substituted for the codeword's 4x4 ECA output. The result had roughly the same 97% decompression rate as the the first way. Each tuple rule's entire codeword output set was seperately processed as a weighted voting neighborhood without the overlap of the algorithm's main implementation. The result was 164/256 positively correlated AlgorithmCode.Hadamard value to vote value. Each cell of each rule's Wolfram code does better than one half, then overlaps with 15 others al doing better than 1/2.

5 Bitmap Implementation

The algorithm is prototyped on 4-byte, 3-byte, and 2-byte RGB *.bmp bitmap files, including the first several images which come from a Linux screenshot then into *.bmp by GIMP. The program converts the image's raster in a rectangular hexadecimal matrix. Iteration 0 is the minimizing codewords for each location as is and then further iterations operate on neighboring hexadecimal codewords. Animated *.gif files are available at my website. You can see the areas of the image cloning itself 2 by 2 and slowly dissolving into avalanche territory that eventually just looks like noise everywhere. The current image was chosen because you can clearly see parts of the image doubling itself as the avalanche property slowly takes over. The code works with any size bitmap. Future work on the project would include other image formats as it is a color code hexadecimal conversion.

6 Testing

7 Other rules, shapes, and sizes

The weight in the errorScore sum can be 2^{row} or 2^{column}, both produce the same set of 8 tuples with the unique solution and even distribution properties, though other ECA rules' properties don't necessarily carry over. This transposition and reflections across an axis may be enough to produce unique tuples without the loop, however checking this is ongoing.

The size of the input array can be easily be any power of 2 squared. Calculation of Wolfram code lengths of 2^{16} are acceptable, lengths of 2^{64} would be challenging. At size 8 there are $2^8 = 256$ possible codewords that grow exponentially but not doubly exponentially because the codeword is only the first row of 8, which means that while you can't calculate the whole truth table at once you can calculate the codewords on the spot. The 8 tuple's uniqueness and distribution properties may or may not apply at size 8, it may not; it is only exhaustively tested for size 4.

Out of the other 0-255 ECA rules, some do better than these particular 8 at lossy compression, losing only 3/16 bits instead of 6/16 bits with these 8. In particular the Pascal rules 90, 165, 102, 153, 105, 150 rank near the top, connected to several these 8 via the property of XOR-additiveness (cite wolfram atlas). However none outside of these 8 have unique solutions or even distributions in either row or column weighted, maxxed or minned.

8 Rule 150

If while running this algorithm for rule 150 you produce a heat map of the errors in reconstructing the lossy compression, you get this. To seven binary digits, five past the decimal place, Phi and Pi show up in these ratios. Seven binary digits is a 2% error rate. Seven digits is enough for an ASCII operation. Precision to seven decimal places shows up here and in reconstructing the 8-tuples above. If you label a five point star and proceed backwards starting at 0, you get {3,1,4,2,0} which is roughly the same seven digit precision. Some of those mentioned here are seven binary digits some are accurate to nine. While seven digits by itself is not impressive, 7 digits across three constants is notable.

```
rule 150
Specific: 150

minErrorMap
29482 29482 29476 29464
30940 30904 30958 30958
17486 17486 17516 17576
5532 5592 5502 5582

minProportions[][]
1.0000 0.9527 1.6828 5.3283
1.0497 1.0000 1.7664 5.5929
0.5942 0.5661 1.0000 3.1663
0.1877 0.1788 0.3158 1.0000
2.621312044429018

a = row2 / row 3 = 3.166305133767173
a = (row2 / row 3) - PI = 0.024712480177379703
accurate to the binary 2^-5 place

b = row1 / row 0 = 1.0496675261229476
b = (row1 / row 1) - (PI/3) = 0.002469974926349927
accurate to the binary 2^-9 place

c = (row0+row1)/(row2+row3) = 2.621312044429018
c = (row0+row1)/(row2+row3) - PhiSquared = 0.003278055679122982
accurate to the binary 2^-8 place

As well as the 1's and 2's place makes for 7, 11, and 10 accurate digits
```

The ratios of the row sums of the heat map produce this set of equations to 5 binary decimal places. (1,0) is Pi/3, (2,3) is Pi, (row0+row1)/(row2+row3)=(Phi+1)=Phi^2, combine these to get row2 and row3 specifically

	Row 0	Row 1	Row 2	Row 3
Row 0	1	3/Pi	(r2,c0)^-1	(r3,c0)^-1
Row 1	Pi/3	1	(r2,c1)^-1	(r3,c1)^-1
Row 2	(Pi/Phi^2)*(1+Pi/3)/(1+Pi)	(3/Phi^2)*(1+Pi/3)/(1+Pi)	1	Pi
Row 3	(1/Phi^2)*(1+Pi/3)/(1+Pi)	(3/(Pi*(Phi^2)))*(1+Pi/3)/(1+Pi)	1/Pi	1

References

[1] Daniel McKinley. github.com/dmcki23/, 2024.

[2] Todd Rowland and Eric W. Weisstein. Additive cellular automaton.

[3] Wikipedia contributors. Fast walsh–algorithmcode.hadamard transform — Wikipedia, the free encyclopedia, 2024. [Online; accessed 4-April-2025].

[4] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.