# Elementary Cellular Automata as Non-Cryptographic Hash Functions

Daniel McKinley

May 2025

## 1 Introduction

A subset of 10 of the 256 elementary cellular automata (ECA) rules are explored as a non-cryptographic hash function using a lossy compression error-minimization function that operates on input data in 4x4 binary cells [4]. The hash's key properties are that the codewords are unique and evenly distributed, has a lossy inverse, hashed data can be operated on efficiently while hashed, and shows a clear application to edge detection. The loops parallel the nested $2^n$ structure of the Fast Fourier Transform (FFT) and Fast Walsh-Hadamard Transform. General algorithm outline, specific ECA rules, and aggregate properties are discussed along with the appearance of Pi and the Golden Ratio in the error heat map of rule 150's truth table. It is implemented in Java at [1] and more images and gifs are available at the website.

## 2 Main Algorithm

The hash algorithm is a kind of lossy compression that operates on 4x4 wrapped cells of binary input data. Within each cell, row 0 is the input neighborhood and the ECA rule's output is calculated for rows 1,2 and 3. All 16 possible row 0 inputs are calculated and then scored where each bitwise discrepancy between the codeword's output and the input is summed with a weight of $2^r ow$. The input neighborhoods that minimize and maximize the error score are noted as the codeword pair for that 4x4 input and each codeword is 4 bits. Doing this procedure for all $2^{16} = 65536$ possible 4x4 input neighborhoods produces a truth table for a particular ECA rule.

There are $2^{16} = 65536$ binary 4x4 arrays, wrapped by column
$2^4$ possible $row_0$ neighborhoods for a given ECA rule
The ECAfunction(input) is output for rows 1, 2, 3 These 16 outputs are scored for errors by
Summing the discrepancies between originalInput and codewordOutput
Weighted by either $2^{row}$ or $2^{column}$

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

The minimizing and maximizing values of all possible inputs
are noted as the codeword pair of the original binary matrix
for a given ECA rule

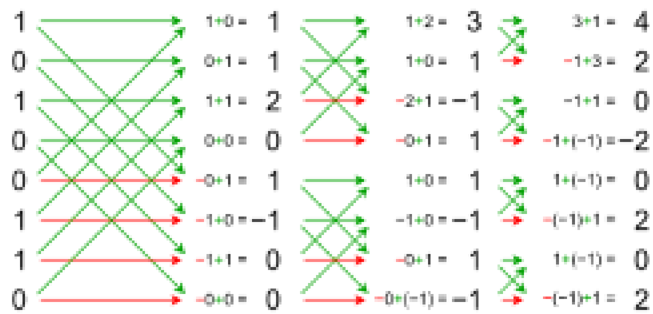
| Input[][] is all 65536 4x4 binary arrays. Each possible binary array's minimum and maximum codewords are found | | | | | |
|---|---|---|---|---|---|
| Wrap | Input[][] | Input[][] | Input[][] | Input[][] | wrap |
| | Input[][] | Input[][] | Input[][] | Input[][] | |
| | Input[][] | Input[][] | Input[][]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

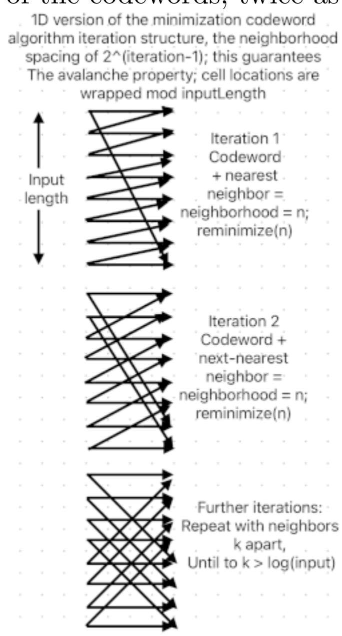
| Each possible neighborhood output from above is scored against the input, the final value is the sum of all individual cell scores. All possible neighborhoods are checked for least and greatest error scores and returned as the codewords. | | | | | |
|---|---|---|---|---|---|
| | err[row][column] = 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[][] | |

Having the minimizing and maximizing codewords for all inputs as above for a given ECA rule, it is applied to a 2D binary array or bitmap image as follows. For every (wrapped) (row,column) location in the input, the 4x4 local cell is the hexadecimal of that location and its right, down, and diagonal right-down neighbors $2^d$ away, $(row, column)..((row + 2^d), (column + 2^d))$ where d is depth of iteration. The value is replace with the respective minimizing or maximizing codewords. This comparing of neighbors of powers of 2 distance away is the same flow chart as the FFT and Fast Walsh Transforms in reverse with reminimizations instead of sums and/or products at every level of recursion. When the iteration's neighbor distance is $log2_{row}$ or $log_2(column)$, whichever is greater, every bit has influenced every other bit and has a unique distinct solution, detailed below.

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.
This the 1 dim version, and instead of a sum term it's a rehash.
Find the codewords of the codewords, twice as far apart each time.




## 3 Hash properties

Within the 256 ECA rules, there are 8 [0,15,51,85,170,204,240,255] using a row weighted errorScore that have the properties of unique codewords for any given input, every codeword occurs the same number of times with relatively even distribution of codewords across the 65536 neighborhoods in the truth table. 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. These properties apply to these rules with both errorScore minimization and errorScore maximization.

There is another overlapping subset of 8, [0,15,85,90,165,170,240,255] whose errorScore weight is $2^{column}$ rather than $2^{row}$. Again, the codewords are distributed perfectly evenly, there are unique solutions for every input, and has a minimizing and a maximizing codeword set. These lists share a few members with the XOR-additive list. [2]. Most properties apply to both with a few exceptions.

This algorithm can be used to make any sized hash. This hash is implemented to either hash-in-place so that the data stays the same size or as compression with the data size quartering each iteration. To implement any size output, first hash to the avalanche point as the fixed data size version, then as compression to the size desired and/or pad with zeroes. Rather than padding with zeroes one can use an inversion to expand the hash to a larger size, however inversion is more computationally expensive than compression. Another option is to hash to the avalanche point and take a subset of it. If using every minMax row-column codeword set, hash to desired size and then hash the hashes.

All 32 Wolfram codes in the 4 minMax row-column sets have a perfectly even distribution of codeword and every input has a unique solution. Each 0..15 codeword is used 4096 times, distributed relatively evenly across the 65536 possible binary 4x4 cells. The minMax row/column set as a whole has unique and distinct hash output with the hash-in-place version, so collisions do not happen. In the compression version individual rules' hashes are minimally collision resistant because the error weight difference means the bottom 2 rows or columns don't change the output much. This is greatly mitigated by using every codeword in the minMax row/column set due to the overlapping and transposed errorScore weights.

Each codeword in the truth table has a lossy inverse of 6/16 with experimental results with the inverse on bitmap images is roughly the same with some variation between codewords including several in the column weighted set that do worse than 1/2. Inverting with every codeword at the same time with overlapping zones of influence decreased the error rate in bitmaps but only to roughly 1/3. The codewords make a weighted vote on every location that cell influences with a positive final total equal to zero and negative total equal to 1.

Hashed input can be operated on with the row weighted rule set without the original input and without inverting it. Because these ECA rules in the 8-subset are linear; rules 15, 51, and 85 pass the left middle or right input bit with no other operation or dependency. Taking any 2 of 16 codeword-generated 4x4 cells and OR them together and reminimize, the result is the same as the original two codewords ANDed together. This shift of logic operations within a hash is uniform within an ECA rule hash and extends to any depth of iteration; the hash algorithm transforms not only the input data but also the relative logic gate. For example if I want to retroactively apply a bitmask to a hashed image or IP address without the original image and without inversion, hash the bitmask and lookup the appropriate logic gate tranformation between the hashed image and the hashed bitmask. These operations only partially apply to the column weighted set, some of the rule-logic pairs have an operation transform and some do not, and no universal combination of gates has a complete set.

This algorithm displays some avalanche properties. The threshold for testing is when every pixel's RGB code has had an opportunity to influence every other pixel, or $log_2(imageWidth)$ or $log_2(imageHeight)$, whichever is greater. Experimentally changing small numbers of pixels and running the required number of iterations produce the property. What happens at that point is that triplets within a rule subset display a substantial number of changes on alternating iterations, and the sum between complementary rules is the total number of cells in the data. This shows that withing a complete codeword set, every single bit is affected by a change. On even iterations within the min codeword set, 15, 51, and 85 have many changes while 170, 204, and 240 don't, and the opposite on odd iterations. The same alternating happens with the column weighted errorScore set triplets to a lesser extent.

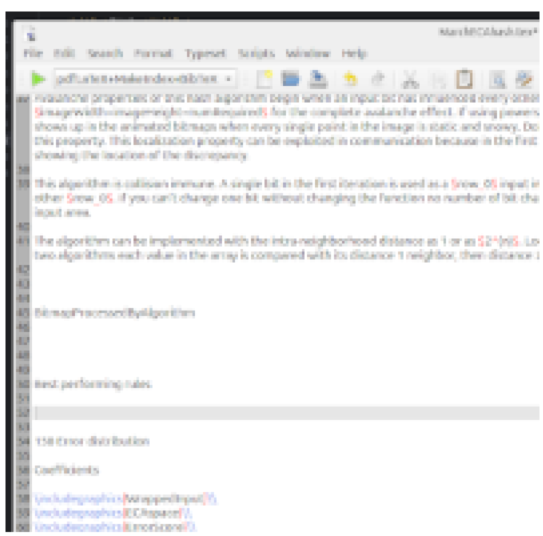
The 32 codewords for any 16 bit input display some symmetries generated by the same algorithm as the left-right-black-white symmetries of the ECA Wolfram code symmetry groups, applied to 4 bits. The left right symmetry is reversing the bit's place order and the black white symmetry is done by reversing the truth table and taking the complement and the left right black white is doing both operations. Applying this to the 4 bit codewords yields some symmetries but no two codeword sets have the same so there are no groups like the 88 independent sets in the ECA.[4]

Visual inspection of hashed bitmaps show a clear application to edge detection. Natural parts of images such as trees appear chaotic while man made surfaces such as countertops, walls, and garage doors tend to converge to a single color that changes over an edge or corner.

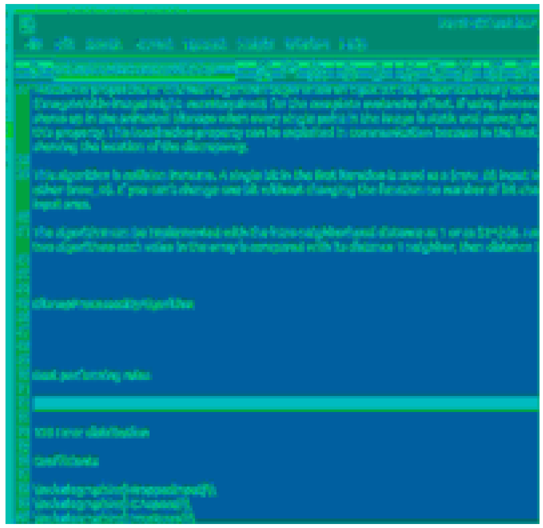
# 4 Implementation and Testing

The algorithm is prototyped on 2-byte RGB *.bmp bitmap files with most photos taken on an iPhone and converted with GIMP. Animated *.gif files and more images are available at my website. You can see the areas of the image hashing 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 with future iterations supporting 3 and 4 byte RGB codes.

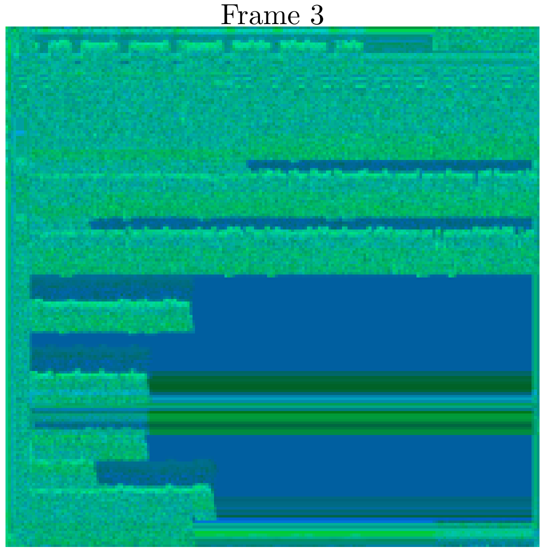
There are several tests run to verify the integrity of the algorithm. One indicator is that every minimizing codeword set has a single gate solution across all 16x16 elements in the truth table, though this is less obvious with the column weighted set. The set of images and gifs produced at least visually verify that something is happening. Looking at the codeword truth tables some show clear obvious patterns and applying the same left-right-black-white symmetry as the 88 ECA symmetry groups [4] yields some equalities between codewords. There may be a better inverse algorithm, the individual codeword hashes invert with roughly the same error rate as its truth table, however the expected synergy between codewords when inverting the entire set has not been as much as expected. Some of the rules' truth tables show obvious linear patterns.



Original Image



Frame 3



Frame 6

## 5 Other rules, shapes, and sizes

The size of the input array can be easily be any power of 2 squared, with only size 4 being fully tested and 8 showing the same properties. Within this prototype project, calculation of truth table lengths of $2^{16}$ using all 4x4 binary grids are acceptable, lengths of $2^{64}$ using 8x8 grids would be challenging at this point. At size 8 there are $2^8 = 256$ possible codewords, which means that while you can't calculate the whole truth table at once you can calculate the codewords individually on the spot. The 8 tuple's uniqueness and distribution properties apply at size 8 by testing random codeword tile; it is only exhaustively tested for size 4. The internal hash logic transforms can also still be easily calculated for size 8 because you only have to work with codewords rather than the entire set of inputs.

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 of these 10 via the property of XORadditiveness [2]. However none outside of these 10 have unique solutions or even distributions in either row or column weighted, maxxed or minned.

## 6 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

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

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 -5 place

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

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

Throw in the 1's and 2's place makes for 7, 11, and 10 accurate digits
If you compare that accuracy to the method of computing pi by the edges of increasing numbers of triangles
it takes dividing 2Pi into 32 triangles to get 7 digits. If takes 32 iterations of the Wallis product

## 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.