Elementary Cellular Automata as Non-Cryptographic Hash Functions

Daniel McKinley

June 2025

1 Introduction

Ten 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 [?]. 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 within a hashed state, and through bitmap file processing shows a clear application to edge detection at shallow depths of iteration. The spacing of the individual tile neighborhoods 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. It is implemented in Java and more images and gifs and code links are available at the website[?].

1.1 Hash Function Background

Hash functions can be described as indexing functions analogous to alphabetical order. If I give you a random word you've never heard before you can still look it up in the dictionary because they are in alphabetical order. It would be more difficult to look it up in a book ordered by numbers of letters because of conflicts when different words have the same number of letters, and this is the difference between different kinds of hashes. An extension of this idea can do the same thing for any kind of data: audio, visual, archives, and more. Databases use hashes to index data and web servers use hashes to distribute workloads. If a site gets a billion visitors a day the work has to be divided up between sub servers by hashing certain reference data such as the client's id data. Database servers keep frequently used data closer to the top of the stack and hashes are used to save significant computing time through using bloom filters to determinine whether requested data is already up front in the cache or that a deeper database has to be called.

There are two kinds of hashes called cryptographic hashes and non-cryptographic hashes. Cryptographic hashes are used in password storage, encryption, and blockchain currency and other secure applications. The hashes described here are almost the opposite of cryptographic because of the transparent inverses, retroactive hashing, minimal avalanche, and linear operation component rules. Other properties that make a good hash like the avalanche property and whether it outputs a certain or fixed size output are addressed below. A survey of literature using automata as a hash function show a focus on the cryptographic kind. [?] [?] [?]

1

1.2   Elementary Cellular Automata Background

Elementary Cellular Automata are a 1 dimensional operation with a set of 256 8 bit binary truth tables numbered 0..255 called Wolfram codes [**?**]. They do this 1D operation in parallel on a binary string, taking in 3 bits at each location, itself and its two neighbors on either side, binary summing these 3 input bits to a value 0..7, and looking up that value's bit in the Wolfram code as the output. Doing this operation repeatedly produces complex behavior falling into four classes of behavior: uniform, periodic/stable, chaotic, and complex. This hash project sieves through all 256 elementary rules, but focuses on 10 in particular in 2 sets of 8 as having a set of ideal hash properties; it uses a small wrapped output space rather than going to infinity on either side.

2   Main Algorithm

This hash algorithm is a kind of lossy compression that operates on 4x4 wrapped cells of binary input data that minimizes errors in compression on a small scale looped over all the input data. Within each cell, row 0 is the input neighborhood and rows 1, 2, and 3 are the ECA rule's output. All 16 possible row 0 inputs are calculated for a given input and then scored so that 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.

Original Image

Frame 1

Frame 2

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 input 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 generated the truth table for a given ECA rule, it is applied to a 2D binary array and bitmap image as follows and with a slight variation is done for 1D input. For every (wrapped) (row,column) location in the input, the local 4x4 cell hash input is the sum of it and its neighbors 2^d away, $(row, column)..((row + 2^d), (column + 2^d))$ where d is depth of iteration. The location's value is replace with the respective minimizing or maximizing codewords that best represent its neighborhood by minimizing error. The comparing of neighbors of powers of 2 distance away is the same stucture as the FFT and Fast Walsh Transforms with rehashes instead of sums. When the iteration's neighbor distance is $log2_{row}$ or $log_2(column)$, whichever is greater, every bit has influenced every other bit and the avalanche property can be analyzed. Since every neighborhood has a unique solution, any size of any given input has a unique hash if hashing in place and not compressing.
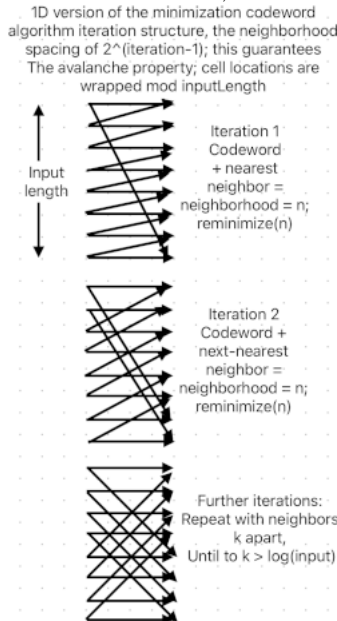
Below is a Fast Walsh-AlgorithmCode.Hadamard Example [?]
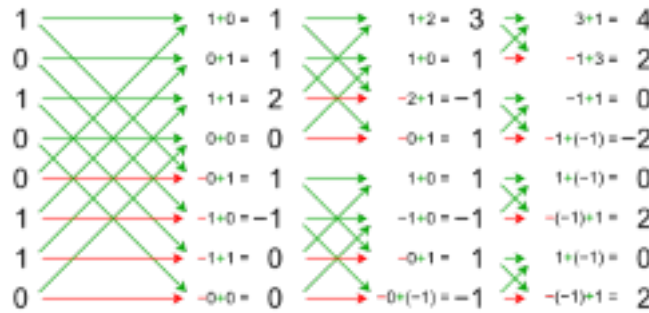
If you did the above with the powers of 2 in reverse order you would get this.
This the 1 dim version, and instead of a sum term it's a rehash.
Find the codewords of the codewords, twice as far apart each time.

3 Hash properties

3.1 Two sets of Eight Rules

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. Since every small scall solution has a unique solution every input at large scale has a single hash output. These properties apply to 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. [**?**]. Most properties apply to both with a few exceptions.

0 and 255 are included because of the 4 rows of the output matrix, 1 is neighborhood input and 3 are output, and so still produce an errorScore and a unique solution. In the hashLogicOp-Transform class while 255 minimizations have a valid transform in its truth table, it has to be manually overidden to zero, adding a constant to the O(N). All other rowError codewords with valid operations work seamlessly.

Another reason they're not completely trivial is that a complete codeword set can be encoded as trigintaduonions, hypercomplex rings with degree 5, two past octonions with 32 elements, having no negative members when using hexadecimal codewords. There are 32 unit vectors in degree 5 quaternions, and by encoding in binary the 16th's place is the negative sign so the hexadecimal hash codewords don't produce negative trigintaduonion elements. Having a unique distinct solution, the trigintaduonions lend themselves easily to physical interpretations of the hash sets and can be an orthonormal basis for data structures and tight integration with $GF(2^m)$. I also propose that all degrees of quaternions, octonions, and so on in general be call nions: 4-nion, 8-nion, 16-nion and so on if the degree is called for, or possibly $nion_D$ as a subscript or nion(D), simplifying the prefixes.

3.2 Inverse

There is a hexadecimal version of the hash and a single bit version of the hash. In the hexadecimal version the bitmap raster gets broken down into 4 bit chunks and the single bit version breaks down the raster into single bits. The hex version is faster and is used to generate the images and gifs; the single bit version produces better inverse loss rates. The hexadecimal versions breaking down the raster into hexadecimal display roughly the same experimental inverse as their respective codeword sets, about 3/16 . However the single-hex-single inverse codewords have a loss of about 2 percent on inversion. There may be a better inverse with less or zero loss that uses extra layers of redundancy in processing a raster, however experimental results haven't yielded better than 2%.

The inverse uses a voting system where the codeword neighborhoods add or subtract the appropriate error weight at every location in their zone of influence. The codeword is the algorithm's best guess at what the input was, so the inverse it its neighborhood's errorScore weighted best guesses at the pixels they're relevant to and are tallied and accounted for as output. A codeword votes its lossily reproduced neighborhood with the proper string of positives and negatives for value 0..1,

minimization..maximization, and rowError..columnError. The inverse is deterministic, any given hash has only one inverse.

3.3   Retroactive Hashing

Hashed input can be operated on with the row weighted rule set without the original input and without inverting it. These operations only partially apply to the column weighted set, some of the rule-logic pairs have a uniform operation transform and some do not. This happens through a transformation of logic operation because of the linear nature of the rules used. Rules 15, 51, and 85 simply pass the left middle or right input bit with no other operation or dependency, 170, 204, and 240 are the complements, and 90 and 165 are the parity of the left and right bits and so define properties of the columnWeighted set. Take any 2 of 16 codeword-generated 4x4 cells, logicGate(A) them together and rehash, the result is the same as the original two codewords logicGate(B) together for all values in the truth table. 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 between codewords. For example if I want to retroactively apply a bitmask to a hashed image or IP address without the original input and without lossy and expensive inversion, hash the bitmask to the same depth as the original and lookup the appropriate logic gate operation between the hashed image and the hashed bitmask. These equivalences are easy to brute force because you only have to iterate through codewords and not entire truth tables.

Rows are logic gates, AND = 8, OR = 14, XOR = 6...
Columns are the minMax codewords with rowError weighted hash truth tables
$[0, 15, 51, 85, 170, 204, 240, 255]$ min and $[0, 15, 51, 85, 170, 204, 240, 255]$ max

HashLogicOpTransform truth table
00 15 15 15 00 00 00 00 15 00 00 00 15 15 15 15
01 07 07 07 01 01 01 01 14 08 08 08 14 14 14 14
02 11 11 11 02 02 02 02 13 04 04 04 13 13 13 13
03 03 03 03 03 03 03 03 12 12 12 12 12 12 12 12
04 13 13 13 04 04 04 04 11 02 02 02 11 11 11 11
05 05 05 05 05 05 05 05 10 10 10 10 10 10 10 10
06 09 09 09 06 06 06 06 09 06 06 06 09 09 09 09
07 01 01 01 07 07 07 07 08 14 14 14 08 08 08 08
08 14 14 14 08 08 08 08 07 01 01 01 07 07 07 07
09 06 06 06 09 09 09 09 06 09 09 09 06 06 06 06
10 10 10 10 10 10 10 10 05 05 05 05 05 05 05 05
11 02 02 02 11 11 11 11 04 13 13 13 04 04 04 04
12 12 12 12 12 12 12 12 03 03 03 03 03 03 03 03
13 04 04 04 13 13 13 13 02 11 11 11 02 02 02 02
14 08 08 08 14 14 14 14 01 07 07 07 01 01 01 01
15 00 00 00 15 15 15 15 00 15 15 15 00 00 00 00

Rows are logic gates, AND = 8, OR = 14, XOR = 6...
Columns are the minMax codewords with columnError weighted hash truth tables
$[0, 15, 51, 90, 165, 204, 240, 255]$ min and $[0, 15, 51, 90, 165, 204, 240, 255]$ max

-1s represent no operation valid on every member of the 65536 length codeword-gate truth table

HashLogicOpTransform truth table
00 -1 -1 00 -1 00 00 00 15 -1 -1 -1 15 15 15 15
01 -1 -1 -1 -1 01 01 01 14 -1 -1 -1 -1 14 14 14
02 -1 -1 -1 -1 02 02 02 13 -1 -1 -1 -1 13 13 13
03 03 03 -1 -1 03 03 03 12 12 12 12 12 12 12 12
04 -1 -1 -1 -1 04 04 04 11 -1 -1 -1 -1 11 11 11
05 05 05 -1 -1 05 05 05 10 10 10 10 10 10 10 10
06 -1 -1 06 -1 06 06 06 09 -1 -1 -1 09 09 09 09
07 -1 -1 -1 -1 07 07 07 08 -1 -1 -1 -1 08 08 08
08 -1 -1 -1 -1 08 08 08 07 -1 -1 -1 -1 07 07 07
09 -1 -1 -1 09 09 09 09 06 -1 -1 06 -1 06 06 06
10 10 10 10 10 10 10 10 05 05 05 -1 -1 05 05 05
11 -1 -1 -1 -1 11 11 11 04 -1 -1 -1 -1 04 04 04
12 12 12 12 12 12 12 12 03 03 03 -1 -1 03 03 03
13 -1 -1 -1 -1 13 13 13 02 -1 -1 -1 -1 02 02 02
14 -1 -1 -1 -1 14 14 14 01 -1 -1 -1 -1 01 01 01
15 -1 -1 -1 15 15 15 15 00 -1 -1 00 -1 00 00 00

3.4 Input Sizes

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 at the beginning or the end. 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.

3.5 Avalanche and Collisions

This algorithm does not display the avalanche property; 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. Experimental results with bitmaps show that even a 1 bit change alters at least one codeword set solution and that this change propagates forward with discrepancies tracking through the process like trees. The truth tables of any single codeword for any input is not unique, however if checking the entire set of 32 codewords at once the set is unique. Since the set of truth tables is distinct, any given input of any size or shape is unique and collisions don't happen for a given size. Tracking the propagation shows that at least one codeword in the set is changed with any bitmap and that it branches from there with every iteration. It could be described as a minimal avalanche instead of maximal avalanche, just enough to prove uniqueness and propagation of small initial changes. The uniqueness of codeword set solutions is always but not exclusively determined by codewords in the column weighted set.

3.6 Codeword Symmetries

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 instead of 8. 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 between rules but no two codeword sets have the same so there are no similar groups to the 88 independent sets in the ECA.[**?**]

3.7 Edge Detection

Visual inspection of hashed bitmaps show a clear application to edge detection. Natural parts of images such as trees and shrubs appear chaotic while man made surfaces such as countertops, walls, and garage doors tend to converge to a single color that tends to change 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 or any array in general with future iterations supporting 3 and 4 byte RGB codes.

There are several tests avalailable to verify the integrity of the algorithm. One indicator is that every minimizing codeword set has a single internal hash logic gate solution across all 16x16 elements in the truth table, though this is less obvious with the column weighted set because it is incomplete. The set of images and gifs produced visually verify that the algorithm is processing as predicted. Loops that check for uniqueness and even distribution are easily verified in the 4x4 size. There may be a better inverse algorithm, however the individual codeword hashes invert with roughly the same error rate as its respective truth table.

The O(N) of the hash is highly dependent on the size of the input, number of bits per pixel, and level of raster processing whether hexadecimal or single bit. Once the codeword truth tables are calculated, an individual tile becomes a small sum and constant term array lookup, however this is multiplied times the area of the input, possibly each of every 32 codewords, and possibly times $\log_2 sizeInput$ to get to the avalanche point . The inverses are more expensive per iteration, since the sum and lookup becomes a generate and iterate, then a final vote tally.

5 Other rules, shapes, and sizes

The codeword sets can be generated with an size array, 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 for any input individually on the spot. The codeword set's uniqueness and distribution properties seem to apply at size 8 by testing random codeword input tiles; 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. Future iterations of the project can implement tile rectangles, allowing for easier 3D hashing.

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 XOR-additiveness [**?**]. However none outside of these 10 have unique solutions or even distributions in either row or column weighted, maxxed or minned.