Improving File Compression Using Elementary Cellular Automata

John Albury
Computer Science
University of Central Florida
Orlando, FL 32816-2362
jalbury@knights.ucf.edu

Richard Wales
Computer Science
University of Central Florida
Orlando, FL 32816-2362
richard_wales@knights.ucf.edu

Annie S. Wu
Computer Science
University of Central Florida
Orlando, FL 32816-2362
aswu@cs.ucf.edu

Abstract

We present a novel technique for pre-processing files that can improve file compression rates of existing general purpose lossless file compression algorithms, particularly for files that these algorithms perform poorly on. The elementary cellular automata (CA) pre-processing technique involves finding an optimal CA state that can be used to transform a file into a format that is more amenable to compression than the original file format. This technique is applicable to multiple file types and may be used to enhance multiple compression algorithms. Evaluation on files that we generated, as well as samples selected from online text repositories, finds that the CA pre-processing technique improves compression rates by up to 4% and shows promising results for assisting in compressing data that typically induce worst-case behavior in standard compression algorithms.

Introduction

We present a novel technique for pre-processing files that can improve the file compression rates of existing general purpose lossless file compression algorithms. For a given file, an exhaustive search of all elementary cellular automata (CA) rules is used to find an optimal end state such that storing this end state in a transformed CA file results in better compression than compressing the original file. While the work in this paper focuses on the popular compression algorithms gzip, bzip2, and xz, this approach can be generalized to work with other compression algorithms.

File compression is a topic of growing importance in this era of big data. Most text-based pre-processing techniques tend to work with one or both of the following principles in mind: (1) reducing the size of the file to be compressed (i.e. by using a more efficient encoding) is likely to reduce the size of the resulting compressed file, and (2) encoding the information in a format that will allow greater compression is likely to reduce the size of the compressed file. General compression algorithms suffer when used on random strings of characters as there are no sensible patterns that the file can be easily reduced to. SSH keys are a commonly used item that contain remarkably random data. Since the CA search does not rely on any pattern seeking, we hope to improve the worst-case performance of standard compression algorithms.

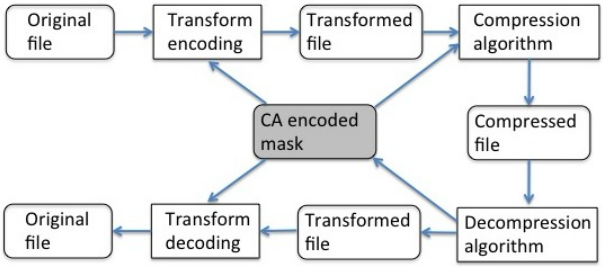

Figure 1: Basic steps of the file pre-processing algorithm. The CA encoded mask (gray box) is found by exhaustively searching every possible CA rule.

Our method differs from previous methods in that it operates at the bit level rather than at the character or word level. Figure 1 shows how our pre-processing method is combined with and used to enhance the capabilities of an existing compression algorithm.

For each file to be compressed, a binary *mask* of the same size as the *original file* is created. This mask is used to encode the original file into a *transformed file* by using a bit-wise XOR operation to subtract the mask from the original file. The transformed file is then compressed by the compression algorithm to form the *compressed file*, and an encoding of the mask is added to the compressed file. Upon decompression, the compressed file is decompressed into the transformed file. The CA is run to obtain the mask that was used to create the transformed file, then a bit-wise XOR operation between the mask and the transformed file is used to recover the original file.

The key to this method is finding a mask that effectively transforms an original file into a transformed file that has greater compression potential than the original file. In addition, the mask must be able to be stored efficiently with the compressed file so that it does not increase the compressed file size significantly. To store the mask efficiently, we take advantage of the fact that a CA provides a space efficient means of storing a complex pattern, such as the mask. By encoding the CA with just a few bytes, upon decoding we

can recover the mask by executing the CA. We show that, given a file to be compressed and enough computational time, in many cases it is possible to find a CA rule and initial start state that will result in better compression. We test our method on 10 randomly generated 2048-bit SSH public keys, 10 randomly generated text files containing lowercase/uppercase letters and numbers, and 10 files selected from online text file repositories (files were selected from the Canterbury Corpus and textfiles.com). We compare the results of using our pre-processing method with the results of using the compression algorithms alone.

Approach

Given a file to be compressed, the CA pre-processing method depends on the discovery of a mask that can effectively transform the original file into a transformed file that is more amenable to compression than the original file. We use an exhaustive search of all 256 elementary CA rules to find such a mask and describe the details of the search here.

File Compression and Decompression

Let L be the length in bits of the original file. Our CA search is looking for a non-trivial binary string of the same size to use as the CA mask. To be able to reconstruct the original file from the transformed file, three pieces of information are needed: the CA rule that is used, the number of time steps the CA is run, and the starting configuration of the CA. These pieces of information are used to run the CA and obtain the mask that was used to generate the transformed file, then the original file is recovered using a bit-wise XOR between the mask and the transformed file.

The starting configuration of our CA is specified by two randomly generated integers, *start_one* and *interval_one*, whose values are between 1 and $\frac{L}{5}$. The starting configuration is assumed to be all zeros initially, then a one is placed at the index specified by *start_one*. From there, ones are placed in intervals of *interval_one* until the end of the configuration is reached. Starting from the initial configuration, each rule is applied for M time steps or when the rows of the CA start repeating, whichever comes first. Since in an elementary CA a cell can only interact with the two cells adjacent to it at each time step, it can take many time steps before each cell has interacted with every other cell. For this reason, we set M to be $4L$, with the idea being that complex patterns in the rows (that could result in better file compression) are more likely to emerge after all cells have had a chance to interact with one another several times. We use a Bloom filter (Blustein and El-Maazawi 2002), a space-efficient probabilistic data structure composed of multiple hash tables of varying sizes, to detect whether a particular CA state has been seen before. This allows us to terminate a CA search early if a cycle in the CA has been reached, significantly reducing the time required to run experiments in many cases.

To evaluate the fitness of a mask generated by a CA, a percent compression improvement is calculated. This evaluation is done by performing a bit-wise XOR on the current CA row (the current mask) with the original file and writing the result to a transformed file, compressing this transformed file, adding the encoding of the CA mask, then calculating the percent difference between the size of this file and the compressed size of the original file. This is done for each compression algorithm used, where the compression improvement is only calculated for files compressed by the same algorithm. For each file to be compressed, the best compression improvement, and the time step T in the CA where this improvement was found, for each compression algorithm is saved.

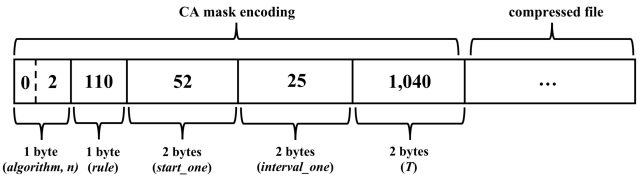

Figure 2: An example of how a transformed file is stored.

Encoding of the Cellular Automaton Mask

In order to reconstruct the original file when decompressing, the cellular automaton needs to be able to reconstructed using just the information encoded with the compressed transformed file. Thus, the rule used, the values of *start_one* and *interval_one*, and the final time step T to which the CA is run must be encoded with the compressed transformed file. The algorithm used to compress the transformed file must also be included in the encoding, since our experiments use three different compression algorithms. We accomplish this via a variable-length encoding to allow for different encoding lengths to be used for different file sizes. This encoding is added to the beginning of the transformed file after it is compressed. The size of this compressed file with the encoding added is the size we use to report our results. Since the maximum values of *start_one*, *interval_one*, and T depend linearly on the length of the file in bits L (the maximum value for both *start_one* and *interval_one* is $\frac{L}{5}$, and the maximum value of T is $4L$), we let each of them be represented by the same number of bytes to simplify the encoding process. There are 256 possible rules in an elementary CA, so the rule used can be represented by a single byte. We only test three compression algorithms, so the algorithm used to compress the transformed file can be encoded using 2 bits. Thus, our encoding of the CA mask is as follows: the first byte of the encoding specifies the compression algorithm used and the number of bytes that *start_one*, *interval_one*, and T will each be represented by (let this value be represented by n); the second byte represents the rule used; the next n bytes represent the value of *start_one*; the next n bytes represent the value of *interval_one*; the next n bytes represent the value T. Then, the compressed transformed file is appended to the encoding.

An example of a possible encoding is shown in Figure 2. The first 2 bits of the first byte indicate that the compression algorithm used is bzip2 (bzip2, gzip, and xz are represented by 0, 1, and 2, respectively). The last 6 bits of the first byte indicate that *start_one*, *interval_one*, and T will each be rep-

resented by 2 bytes. The second byte indicates that the rule used in the CA was rule 110. The third and fourth byte show that the value of *start_one* is 52. The fifth and sixth bytes show that the value of *interval_one* is 25. Finally, the seventh and eighth bytes indicate that the value of T is 1,040. Using just these values, the CA that was used to generate the mask can be reconstructed and, thus, the original file can be recovered. The length of the CA mask encoding (in bytes) for a given file is given by the following equation: $E(L) = 3\lceil \frac{1}{8} log_2(4L) \rceil + 2$, where L is the size (in bits) of the file to be compressed.

Experimental Results

We evaluate the performance of the CA pre-processing technique by comparing the compression performance of the standard compression algorithms with the compression performance of these algorithms with CA pre-processing. The algorithms used are gzip, bzip2, and xz.

The files on which we perform our experiments are given in Table 1. The first group of files (key1 through key10) are SSH keys (stored as text files), the second group files (random1 through random10) are randomly generated text files containing numbers and lowercase/uppercase letters, and the last group of files are files selected from online text repositories. For the last grouping of files, all are text files, with the exception that mind6 is an ASC file. The xargs file comes from the Canterbury Corpus and the rest of the files in this group come from the Science section of textfiles.com, an online text repository. File sizes range from 0.7 KB to 4.2 KB. For each file, the CA is run with 10 different randomly generated start states (by randomly choosing the values of *start_one* and *interval_one*), and for each run the best compression improvement for each of the compression algorithms is stored. The results of these trials are averaged to give the average compression improvement for each file.

Discussion

Table 1 shows the results for each of the compression algorithms tested.

%Imp. is the percentage of trials (out of the 10 that are run for each file) where our method results in a net positive effect on compression. The results reported take into account the extra length that the CA mask encoding adds. Out of the algorithms we test, bzip2 is the most responsive when looking at this measurement. Aggregating across groups of files, our method improves bzip2 compression in 69.5% of trials for the random files tested (both the SSH keys and randomly generated text files) and in 35% of trials for the non-random files tested. In many cases, compression improvements are found, but these improvements are outweighed by the length of the CA mask encoding. If we ignore the length of the CA mask encoding that is added to the compressed file, compression improvements are found in 66.6% of trials overall, compared with 28.1% when taking the encoding into account.

Δ_{avg} is the average percent improvement in compression when using our method compared with using the standard compression algorithm alone, and Δ_{best} is the best percent improvement in compression for any individual trial. When looking at these measurements, the xz algorithm shows the most promising results. For the randomly generated text files compressed with the xz algorithm, our method shows improvements of up to 4.492%. The bzip2 algorithm also shows promising results in this aspect. For most of the files compressed with bzip2, using our method results in a net improvement in compression, although these improvements rarely are more than 1%.

T_{best} is the time step at which the best individual compression improvement is found (that is, the value of T when Δ_{best} is found). A value of "N/A" means that no CA masks are found that improved compression. Interestingly, values of T_{best} as high as 26,000 are observed, showing that the CA masks being generated are generally non-trivial.

When analyzing the improvement that our method yields for each of the algorithms tested, one trend that stands out is that the compression improvement our method offers has an inverse relationship with the compression ratio of the standard compression algorithm for the file being tested. This could help explain why our method shows such poor results when used with the gzip algorithm compared to the other algorithms. For most of the files we test, gzip produces the best compression. This relationship also seems to hold true from file-to-file when using the same algorithm. Files that, when compressed, have lower compression ratios seem to respond better to our pre-processing technique. The SSH keys and randomly generated text files exhibit this behavior; typically, standard compression algorithms perform poorest on random-like data like these files and this holds true for the files we test as well. These random-like files also show the highest and most consistent improvements when using our pre-processing method compared to the non-random files. This relationship is shown in Figure 3. For each file and compression algorithm pairing, the original compression ratio (without using our method) and the compression improvement when using our method are plotted. As shown in the figure, there is a general trend downward as compression ratio increases. Thus, this method could have intriguing implications for compressing random-like data and other types of data that typically induce worst-case behavior in standard compression algorithms.

Time cost

While these initial results are promising for compressing random-like data, the time taken to run the program is less than desirable and is impractical for large files at this time. Although the time it takes to compress and decompress a file once a CA solution is found is reasonable, the time required to find a CA solution can be very long in the present system. The runs typically take several days to complete, which is unfeasible for most applications. This will need to be improved by narrowing the search to a subset of rules, as opposed to performing the exhaustive search that the present system uses.

Conclusions

In this work, we present a new method for pre-processing files to enhance the performance of standard compression

		bzip2				gzip				xz			
File	Original Size	%Imp.	Δ_{avg}	Δ_{best}	T_{best}	%Imp.	Δ_{avg}	Δ_{best}	T_{best}	%Imp.	Δ_{avg}	Δ_{best}	T_{best}
key1	1,675 bytes	100%	+0.573%	+0.990%	21,628	0%	-0.444%	-0.307%	N/A	0%	-0.557%	-0.557%	N/A
key2	1,675 bytes	80%	+0.398%	+0.781%	11,104	0%	-0.545%	-0.461%	N/A	0%	-0.334%	-0.279%	N/A
key3	1,679 bytes	10%	-0.246%	+0.071%	9,686	0%	-0.536%	-0.383%	N/A	0%	-0.418%	-.279%	N/A
key4	1,675 bytes	60%	+0.028%	+0.711%	11,759	0%	-0.513%	-0.307%	N/A	0%	-0.559%	-0.559%	N/A
key5	1,675 bytes	100%	+0.572%	+0.919%	10,317	0%	-0.545%	-0.537%	N/A	0%	-0.306%	-0.279%	N/A
key6	1,675 bytes	50%	+0.007%	+0.284%	10,084	0%	-0.467%	-0.383%	N/A	0%	-0.278%	-0.278%	N/A
key7	1,675 bytes	60%	+0.106%	+0.709%	3,098	0%	-0.567%	-0.460%	N/A	0%	-0.557%	-0.557%	N/A
key8	1,679 bytes	90%	+0.404%	+0.780%	4,101	0%	-0.514%	-0.383%	N/A	0%	-0.599%	-0.559%	N/A
key9	1,675 bytes	90%	+0.220%	+0.711%	13,817	0%	-0.383%	-0.307%	N/A	0%	-0.599%	-0.559%	N/A
key10	1,679 bytes	70%	+0.282%	+0.777%	4,750	0%	-0.420%	-0.306%	N/A	0%	-0.556%	-0.556%	N/A
random1	2,048 bytes	90%	+0.144%	+0.241%	16,377	0%	-0.429%	-0.390%	N/A	100%	+2.12%	+2.404%	13,306
random2	2,048 bytes	40%	+0.024%	+0.543%	8,145	10%	-0.416%	+0.325%	8,145	100%	+4.16%	+4.492%	9,122
random3	2,048 bytes	90%	+0.241%	+0.543%	19,694	10%	-0.312%	+0.391%	15,875	10%	-0.149%	+0.248%	15,874
random4	2,048 bytes	20%	-0.199%	+0.302%	8,161	20%	-0.201%	+0.715%	26,789	100%	+2.727%	+2.871%	1,256
random5	2,048 bytes	50%	+0.054%	+0.361%	2,082	0%	-0.416%	-0.260%	N/A	100%	+3.741%	+4.471%	22,515
random6	2,048 bytes	100%	+0.331%	+0.781%	16,321	10%	-0.280%	+0.780%	16,321	100%	+4.222%	+4.481%	2,956
random7	2,048 bytes	100%	+0.211%	+0.362%	8,156	20%	-0.189%	+0.783%	8,719	100%	+0.827%	+1.217%	15,951
random8	2,048 bytes	80%	+0.228%	+0.541%	9,248	0%	-0.429%	-0.325%	N/A	0%	-0.247%	0.000%	N/A
random9	2,048 bytes	50%	+0.018%	+0.422%	13,456	10%	-0.377%	+0.260%	13,456	0%	-0.025%	0.000%	N/A
random10	2,048 bytes	60%	+0.174%	+0.421%	1,876	0%	-0.324%	-0.259%	N/A	100%	+0.538%	+0.733%	3,790
ast500hr	786 bytes	70%	+0.389%	+1.296%	1,340	0%	-1.733%	-1.556%	N/A	0%	-1.186%	0.000%	N/A
fs417	2,018 bytes	70%	+0.177%	+1.027%	1,787	0%	-0.762%	-0.684%	N/A	0%	-0.571%	-0.357%	N/A
genetic	1,873 bytes	90%	+0.581%	+1.378%	14,242	0%	-0.732%	-0.610%	N/A	0%	-0.746%	-0.746%	N/A
mind6	3,216 bytes	0%	-0.741%	-0.370%	N/A	0%	-0.770%	-0.660%	N/A	0%	-0.264%	-0.208%	N/A
unifid	1,200 bytes	0%	-1.581%	-1.581%	N/A	0%	-1.649%	-1.461%	N/A	0%	-1.439%	-1.439%	N/A
xargs	4,227 bytes	20%	-0.068%	+0.227%	9,512	0%	-0.572%	-0.515%	N/A	0%	-0.607%	-0.607%	N/A
goddard	896 bytes	0%	-0.968%	-0.538%	N/A	0%	-1.288%	-0.947%	N/A	0%	-1.203%	-0.6329	N/A
ast-dorn	2,613 bytes	60%	+0.149%	+0.908%	4,775	0%	-0.627%	-0.512%	N/A	0%	-0.674%	-0.674%	N/A
ast-prog	1,672 bytes	20%	-0.361%	+0.425%	12,091	0%	-0.785%	-0.664%	N/A	0%	-0.440%	-0.400%	N/A
taxonomy	3,271 bytes	20%	-0.076%	+0.254%	24,613	0%	-0.643%	-0.531%	N/A	0%	-0.693%	-0.693%	N/A

Table 1: Compression improvement results for each of the compression algorithms tested.

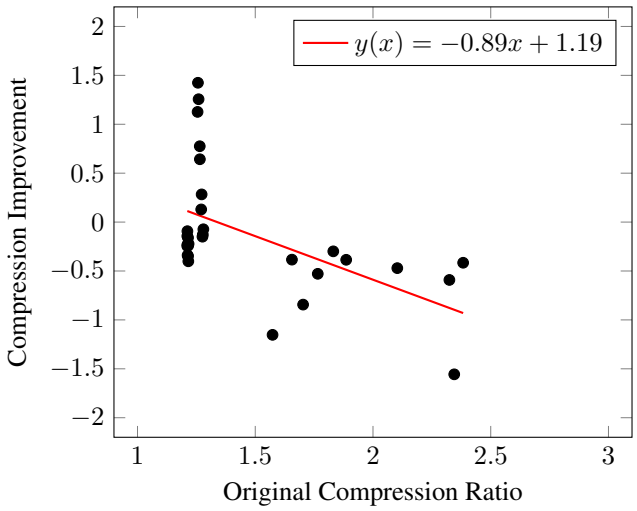

Figure 3: The relationship between the compression ratio on the original file using the standard compression algorithm and the compression improvement that is gained when using the CA pre-processing method.

algorithms. Unlike previous methods which operate at the character and word level, the CA operates at the bit level. As a result, this method can be generalized to multiple file types beyond just text files. We test the ability of the CA pre-processing method to enhance gzip, bzip2, and xz compression results on randomly generated text files, SSH keys, and selected files from online text repositories. A small range of file sizes were tested due to the computation necessary to explore the massive search space available. Initial results show that in many cases our method improves compression, with some results showing up to a 4% improvement compared to the standard compression algorithm, but in other cases it can actually worsen compression due to the inability of our method to find an effective CA mask and the overhead of encoding the CA with the compressed file. Interestingly, this method seems to provide the most benefit in cases where the standard compression algorithm performs the worst. Thus, if the computational cost of the method can be reduced, our method could be used to improve the worst-case performance of widely used compression algorithms.

References

Blustein, J., and El-Maazawi, A. 2002. Bloom filters – a tutorial analysis, and survey. Technical report, Dalhousie University.