© [2008] IEEE. Reprinted, with permission, from [Zenon Chaczko, Andrew Chan and Chris Chiu, FILE COMPRESSION USING TYPOGENETIC COMPUTATION, 2008, Third International Conference on Broadband Communications, Information Technology & Biomedical Applications, 2008]. This material is posted here with permission of the IEEE. Such ermission of the IEEE does not in any way imply IEEE endorsement of any of the University of Technology, Sydney's products or services. Internal or personal use of this material is permitted.However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it

FILE COMPRESSION USING TYPOGENETIC COMPUTATION

Zenon Chaczko, Andrew Chan and Chris Chiu

University of Technology Sydney, NSW, Australia

Zenon.Chaczko@uts.edu.au, Andrew.Chan@uts.edu.au, Christopher.Chiu@uts.edu.au

ABSTRACT

Typogenetic algorithms are a break from classical approaches to computation. Based on gene expression and intercellular processes, typo-genetic computation can offer a new approach to the algorithmic problems of system security, data compression and encryption. The method has a potential of much higher compression ratios at the limited computational costs i.e. processing time. This paper presents a formal system based on typogenetics for the purposes of compression. Lossless data compression is an important part of computer science. While the ability to reduce consumption of hard disk space or transmission bandwidth through statistical redundancy has served well in the past, the explosive growth in high quality media content (*.mp3, *.mpg) on the internet in the past few years have highlighted the limitations of traditional statistical techniques for compression.

Keywords: Typogenetics, Lossless data compression

1. INTRODUCTION

With the rapid growth of media rich content in the form of sound, graphics and video, classic lossless compression techniques are becoming ineffective when handling compression of these files. The impact is that often limited and expensive resources such as bandwidth and data storage space are being heavily used. Unlike other techniques typogenetic[1] compression is not based on statistical redundancy thus it has a real potential to be applied as a strong compression computation mechanism. Due to its specific the technique requires from users good understanding of its nature prior to being able to capture its power in the form of computational process.

1.1. Aims and Objectives

The main goal of this research is to investigate ways how the technique of typogenetics could provide a robust, secure, reliable and effective solution for a real-world application. One of many possible applications is a file compression. Although our research involves a range of possible applications of typogenetic mechanisms in the area of smart computation algorithms, the main focus of this paper is on implementation of typogenetics in the file compression domain.

In the first part of the paper, we will navigate through the fundamentals of typogenetics then we move to explore how typogenetics can be actually applied to computation algorithms. We will discuss the growth rate of typogenetics, its impact on theorem coverage, and the importance of theorem complexity [5, 6] to the compression method. The second part of the paper provides an overview of two alternative typogenetic methodologies (the *top-down* and *bottom-up*) that are applicable to the compression problem will be investigated. Benefits and shortcomings of each of these methods are being discussed. The conclusion will offer results of our experimentation, recommendations and projection of future studies. This paper covers considerations and the process of design for a suitable file format applying compression technique. We have thoroughly researched the domain of typogenetics and designed and implemented an experimental framework for a continual refinement, evaluation and improvement of various versions of compression algorithms using this relatively new technique.

2. EXISTING COMPRESSION METHODS

Before the advent of the computer systems, compression was not a well developed discipline if compared to such areas as cryptography, which has been used almost for thousands of years to keep information secret. Encoding data in more compressed forms was seen as a rather useless task since smaller writing or print did the same. With development of information and communication technology, however, the need for compression grew due to the fact that memory and space can be very expensive resources. One of the simpler ways of compression was to run length encoding.

2.1. Run Length Encoding

Run length encoding or ***RLE*** is a common way of encoding repeated symbols in a more succinct form. In RLE the repeated symbols are replaced by a delimiter followed by the number of occurrences. If the delimiting symbol is encountered in the data, the delimiter is repeated to signify it is part of the data. As can be seen, the compressed version on the right on the above example represents a shorter form.

Example

"This ssstring will be compressssssed!" → *"This s!3tring will be compres!6ed!!"*

[1] The term 'typogenetics' first introduced by Hofstadter (1979) represents a formal system which captures the essence of biological genetics.

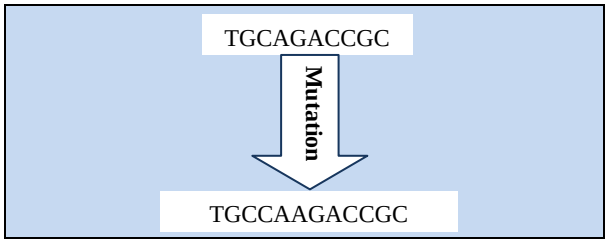

Figure 1: The TGCAGACCGC strand mutates to TGCCAAGACCGC

2.2. Delta Encoding

Delta encoding stores the differences between values to reduce the data required to signify a stream of values. The first value given in a series is a complete value. The values given after the leading value is encoded as the relative difference between itself and the previous value. This method of encoding is particularly important when constant revisions or small updates are being made. Source control systems take advantage of this technique.

Example:
{104, 99, 101, 96, 96} → {104, -5, +2, -5, 0}

2.3. Huffman & Arithmetic Encoding

Based on Shannon-Fano encoding, Huffman coding is of the simplest ways of achieving strong compression. Huffman coding takes advantage of statistical redundancy and uneven distribution of data. A prefix table is created with the most probable symbols encoded as the shortest; and with the least probable being the largest. The original file is then compressed using this lookup table. Arithmetic encoding such as LZW is a more advanced form of Huffman technique, using fractional numbers of bits to achieve greater overall compression. Arithmetic encoding, however, has failed to totally replace Huffman coding because of its computational expense and limitations imposed by use of multiple patents. The weakness of Huffman and Arithmetic encoding occurs in cases of low statistical variation in the data. The overhead of encoding using these variable bit encoding schemes becomes much larger than the amount of compression they provide. Typogenetic Compression aims to fill this gap.

3. INTRODUCTION TO TYPOGENETICS

First introduced by Hofstadter [4], typo-genetics is a system which captures the essence of genetics. A very interesting result of the system is the self-modifying nature of the strands. The self-modifying behavior of the strands forms the foundation of this class of AI technique [2]. There are three main elements of the computational model of typogenetics such as:

- **Bases.** The typogenetic system involves arbitrary lengths of strands comprised of the letters *A, C, G* and *T*. These are also known as bases. All typographical strands are made of these bases. C and T are pyrimidines, A and G are purines bases.
- **Mutation.** The interesting aspect of typogenetics is the fact any strand may have a property of being able to mutate [7, 8] into another. This allows us to start with any strand and create more strands. We shall refer to the starting strand as a *seed.* In a formal system, the seed represents the *axiom.*
- **Enzymes and Amino Acids.** Strands are modified through typographical enzymes. Enzymes have operations such as inserting bases, removing bases, or shifting left and right. These enzymes are derived from the strands themselves through a process of translation. Enzymes themselves are further composed of individual amino acids. Table 3 shows the amino acids names in typogenetics & operations.

3.1. Translation Process

The process of translation involves translating a strand into an enzyme. It involves grouping bases into duplets and mapping them to an amino acid. These amino acids are then accumulated into a sequence which becomes the enzyme[2]. Table 4 how base duplets are translated into amino acids. Besides the mapping between base duplets and amino acids, typogenetics has a structure (Tertiary Structure) which determines the initial starting point or binding preference of the enzyme. The binding preference is calculated from the folds in the enzymes.

The letter at the end of each enzyme on the enzyme table (s, r, l), representing the fold each enzyme creates in the enzyme strand. All strands are initially pointing right, and depending on the fold of the enzyme, the final orientation can be either pointing top, bottom, left or right [4]. Once the final orientation is derived, we can work out the initial binding for the enzyme strand. Enzyme in folding mechanism is depicted in Figure 2, following the folds listed in Table 1.

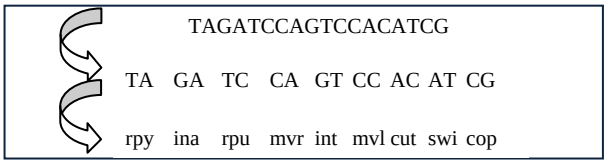

Figure 2: The Translation Process

In Table 1 we can see how the folding mechanism translates into the formation of an initial binding preference which then can be found in Table 2. The duplet AA has been left blank on purpose. This served as punctuation mark on the strand, signaling the end of the code for an enzyme. From this punctuation it is possible to have a strand represent several enzymes. This allows for more complicated strands to be derived. From this we can see the recursive nature of typogenetics. Any given strand acts as both data which is acted upon and also as the program which modifies it. This allows us to take any arbitrary strand, translate it into enzymes, and create another strand. Successive

[2] The details translation process were described by Hofstadter's (Hofstadter, 1979, pp 508 -510)

performances of this "*mutation*" would allow us to create more and strands. The property of self mutating strands into longer versions can be quite useful in the area of compression. If large strings can be represented by smaller strands and mutation numbers, this may give a saving in the total bytes required for storing data.

4. COMPRESSION AND TYPOGENETICS

We have seen how typogenetics can give us a powerful way of lengthening a strand through recursively modifying itself. This section will focus on the how typogenetics might be used to compress strands of data. While mutating a strand is a relatively straightforward matter. Finding the source strand given any arbitrary strand if any exist is a different matter. The situation is very similar to Emil Post's "Post production system." which endlessly generate strings following certain rules. In the Table 3 presents how duplet bases map to various amino acids. The first base is represented by the left column and the second base by the top row.

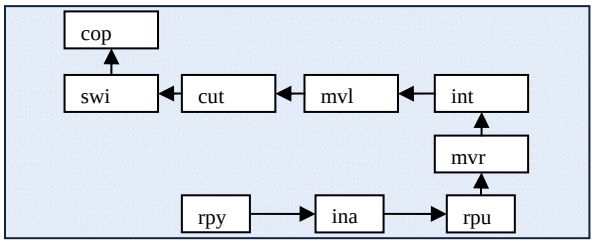

Figure 3: Enzyme Folding

Operator Name **Amino acid**	**Description**	**Folding**
cut	cut strand	s
del	delete a base from strand	s
swi	switch enzyme to the other strand	r
mvr	move on unit to the right	s
mvl	move on unit to the left	s
cop	turn on copy mode	r
off	turn off copy mode	l
ina	insert A to the right of this unit	s
inc	insert C to the right of this unit	r
ing	insert G to the right of this unit	r
int	insert T to the right of this unit	l
rpy	search for nearest pyrimidine to right	r
rpu	search for the nearest purine to the right	l
lpy	search for the nearest pyrimidine to left	l
lpu	search for the nearest purine to the left	l

Table 1 Amino Acid Table

Top	C
Bottom	G
Left	T
Right	A

Table 2 Enzyme Folding Binding Preference

	A	C	G	T
A		cut	del	swi
C	mvr	mvl	cop	off
G	ina	inc	ing	int
T	rpy	rpu	lpy	lpu

Table 3 Translation Table

4.1. Axioms, Rules of Inference and Theorems

In Post's system, all strings start from axioms, which is a set of strings. These strings, following given rules of inference or rules, then produce more strings. This process of string creation in the system continues indefinitely. It is clear after a certain stage that some strings could be produced by the rules, whereas some strings could not be produced by the rules. The strings which were producible are also known as theorems. The non-producible strings are known as non-theorems. In the same way how some strings can be produced, strands which can be produced from a given axiom can also be known as theorems. Non-reachable strands which cannot be produced by typogenetics can also be referred to as non-theorems. In typogenetics however, the rules of inference are not static but are derived from axioms themselves. This makes typogenetics a much more complex system to study.

4.2. Seed and Generations

It follows then that any strand which is a theorem may be encoded in terms of its axiom and the number of times the axiom modified itself. For simplicity, the axiomatic strand is specified as the seed, and the number of modifications as the generation. If a strand is a theorem, it then follows that all theorems have a compressed form represented by their seed and generation. From these two pieces of data, the original theorem can be reproduced. Likewise, if a bit stream is represented by its seed and a byte representing the generation, the original bit stream can be recreated.

4.3. Top Down Compression Method

There are two ways of calculating seeds of a typogenetics system. One is the top down approach; the other is the bottom up approach. This section will talk about the top down approach, its merits and its disadvantages. One of the major advantages of top down compression would be small memory footprint required to compress a given strand. Instead of a massive lookup table as characterized in the bottom up approach, a compressing client can derive the seeds of a strand through processing. If a top-down compression technique using typogenetics was to be used. A side effect would be a potentially long execution time. This may affect the usability of this technique. The nature of the typogenetics means that there are many non-theorems. This means that there are strands which may not be able to be derived from smaller strands. Because of this, it is critical when calculating seeds that we need to identify non-theorems easily in a finite amount of time. It then follows that a test for theorem-hood is extremely useful as a first step in compression. In a formal system a test for theorem-hood which can be run in finite time is also known as a decision procedure. Unfortunately for a system as complicated as the typogenetics system, we can be quite certain that no decision procedure as shown by the Church-Turing theorem of undecidability. Although we could possibly test for theorem-hood if only lengthening operators are

present, the system of typogenetics is vastly complicated that reversing this is still a burden. Although there is computation algorithm present in the form of generating and recursive functions in discrete mathematics, this avenue is perhaps outside the scope of this paper but may be appropriate for further study.

4.4. Bottom up Compression Method

The way the bottom up approach works is to methodically work up from the smallest axiom and apply all the appropriate rules of inference to it. Once this is done perform the same on the theorems produced. Eventually all theorems will be derived in this method. With the bottom up approach, there needs to be a means of storing the theorems derived from given axioms. The process of building up this lookup will take time and massive amounts of storage place in order to support larger and larger chromosomes. Although the building of a lookup table will take time, this table needs only be built up once and can be reused if results are cached. This makes the bottom up method quite attractive at compression time.

One of the biggest advantages of the bottom up approach is the ability to have a fast decision procedure if the lookup table is completely built. In reality however, this table can never be "completely built". We can however, be sure that if a strand does exist in our table, it is a theorem, if not it may or may not be a theorem. This is in some ways a partial decision procedure. For the purposes of compression this may be enough. This is a massive advantage over the top down approach for compression.

While building a table sounds all well and good, one of the biggest weaknesses of this method is the fact that enormous strands which are theorems of small seeds may not be found. This means that potentially larger compression opportunities will be missed by the bottom up approach due to memory limitations. While bottom up compression will work well for theorems within the table, larger strands which are theorems cannot be identified easily with this method. While one can create a larger table to accommodate this, this solution is not easily scalable and increases exponentially as seed length and strand coverage is increased.

5. TYPOGENETIC THEORY

If typogenetics is to be applied for compression, there would be a need for the system to generate a large number of theorems or large coverage. This is to increase the chances that a given strand can become a theorem. The translation process in particular has a massive effect on the coverage of the system. From the introduction to typogenetics in chapter it is quite clear that *the Translation Table* has a massive influence on the types of theorems produced by the system. While the translation table is presented in the book GEB, for the purposes of compression this table can be changed to produce different sets of theorems.

From an inspection of *the Translation Table* we can see that there are five main lengthening operators: *ina, inc, ing, int* and *cop* out of a possible 16 base duplets. For a rough approximation we can say that *the growth of a strand* on average is larger than 5 /16 or 31%. In reality however, the growth rate is probably much higher due to the fact that copy mode can lengthen a string dramatically. If strands grew quickly, it follows that there would be less theorems produced by the system under a certain length. Conversely if strands grew slowly, there would be more theorems produced by the system as there are more mutations before a certain length is reached. This means that to produce more theorems, a slower growth is required. While slow growths may allow a typogenetic system to produce more theorems, this comes at a cost to the decompression phase where more generations need to be calculated in order to reach the target strand.

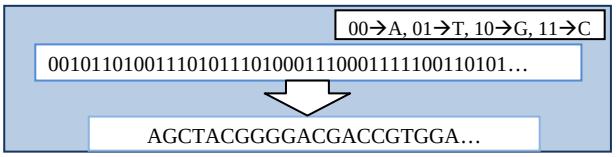

Figure 4: Mapping Binary to Bases

While growth rate has an influence on how many theorems may be produced, the type of theorems produced is also important. The theorems produced by the typogenetic system cannot be too predictable. Particularly in the case of compressing mp3 or mpg files, the variability of the strand sequence will vary between chaotic and ordered. Wolfram describes this as a class 4 system. The *cut, cop* and *off* operators are quite complex in general. The *copy* operator involves copying the length of strand depending on movements of pointer while *cut* splits long strands into smaller parts.

While the cut operator will definitely result in more class 4 results, splitting strands into several parts will add a complexity to the mutation which is out of the scope of this paper. Integrating the cut operator into compression would definitely be an interesting avenue for investigation. The cop and off operators in typogenetics were also quite complex. While an implementation of this was possible, the small axiom lengths we would be investigating and for simplicity, this aspect of typogenetics was ignored. The binding preferences associated with these operators were captured for the tertiary structure, but did not act on the strand in any way.

6. TYPOGENETIC COMPRESSION METHOD

Before compressing a file using typogenetic compression, we need to map a binary string into one of the four bases. This is a relatively straightforward mapping of every 2 bits to a single base. To allow the file format to improve, versioning information for the file will be stored and dispatched. This will simply be a leading byte for the file format.

6.1.1. Compressing File Segments

A high probability of encountering a non-theorem meant that hopes of compressing the entire file as a single strand was not feasible. A more sensible way to compress using typogenetics is to compress only a file. If we are compressing only parts of the file which are deemed compressible, it follows that while decompressing a way of determining the beginning of a compressed section is needed, and the end of the compressed section. A common way to do this is to use a delimiter which will represent the beginning and end of a compressed sequence.

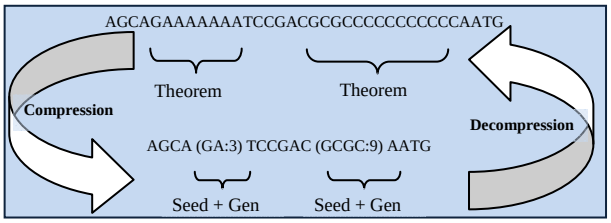

Figure 5: Compression/Decompression Process

If a certain delimiting symbol is chosen, it follows that a way is needed to allow this symbol to be used as data when necessary. A common solution is to repeat the symbol when it represents data. This technique would be required when encoding in our file format. The key to the compressed data is the *seed* and the *generation* of that seed for mutation. This information will be stored between the delimiters and used to decompress the segment. Due to the fact generational data will be either immediately follow the start delimiter or precede the end delimiter, this means that the generation data cannot clash with the delimiter itself. Therefore, the number represented by the chosen delimiter will be a *reserved generation* that cannot be encoded. For similar reasons for a reserved generation, seeds cannot start with a byte signature similar to the delimiter.

6.2. Compression Trade-off

The compressed strand must be smaller than original strand by a certain amount to achieve compression. The original strand must be larger than the two delimiters, the seed, and generation data added together. In many ways using typogenetic compression in this way can be viewed as an extension to run length encoding. Strands which are theorems can be viewed as a run of sorts, and can be compressed into its seed and generation.

7. IMPLEMENTATION ISSUES

After careful consideration it was decided a promising path forward was the bottom up approach, due to the availability of a partial decision procedure and the relative simplicity of the method. In implementation, many unforeseen issues arose. Central to the bottom up approach was the need to pre-calculate all the theorems that will be used during the compression stage. This involved a script which mutated all axioms of length smaller than or equal to 24, the value chosen due to its relative manageability (16 million).

Strand Length	Bits	Total
32	64	1.8*10^19
16	32	4.2*10^9
12	24	16 *10^7

Table 4 Axiom Strand Length and Size

7.1. Implementation Constraints

With the typogenetic system, strands can lengthen themselves and mutate indefinitely. A limit for theorem length needs to be chosen for time constraints. For the purposes of this paper a theorem upper limit of 764 was set as it is the maximum length for a primary key string in SQL Server. Perhaps one of the fundamental issues that arose was memory limitations. Any meaningful calculation of our lookup table would involve hundreds of millions of rows with 16 million axioms. This meant the table could not be reasonably stored in memory. A relational database system would be required to store the theorems. At the time of writing, the database used to store the lookup table had grown to 40GB. This would be a major drawback of this method if a 40GB lookup table was required to compress data. Another problem that became quite evident was the processing time required to calculate the table for all axioms of length 24 or smaller. On a single thread it was calculating on average 100 axioms per second. This meant calculating 16 million axioms on a single thread would take 1.8 days. Distributed processing of the lookup was a must if any progress was to be made.

7.2. Distribution Method

A simple way of distributing calculation was required. A console application with command line parameters was used for simplicity. The console application takes 2 arguments, a lower bound and an upper bound, that was responsible for calculating all axioms between bounds. A batch file started the application concurrently with different parameters. This was an effective way to distribute processing of *the theorem table.* Due to the heavy usage of a database system, the speed of the database system used soon became a bottleneck. The average number of queries to the database exceeded 400 per second. To speed up execution at the database layer, stored procedures were used to speed up queries and calls to the database should be kept at a minimum.

While calculating the initial theorem table, there were certain cases when an infinite loop was met. Upon closer inspection it became apparent that these were caused by self replicating strands which reproduced themselves across several mutations. To prevent this from stalling, processing a cache is required which makes sure that theorems produced by an axiom are not repeated. Hofstadter mentioned these strands in his book. Issues were also encountered with building the initialization table; implementation of the compressing client also had issues for consideration. The design of the compressing client had to be able to handle large files and needed to be able to compress in a reasonable time and better utilise stream processing.

7.3. Stream Processing

As this technique needed to potentially compress some very large files, processing had to work on content which may be in the gigabyte size range. To cater for this, the compressing client cannot cache the contents of the file completely in memory; doing this will result in out of memory errors. To allow the compressing client to compress large files, the content of the target file needs to be processed as a stream, leaving a low memory imprint on the system.

7.4. Compressing Time

Because compression will involve searching for theorems in the database, care must be taken to minimize compressing time. An initial prototype which called the database on every byte of the file was implemented. A 1MB file required over one million calls to the database. The time cost of a set of procedure calls on every byte of a file had an adverse affect on the speed and the design had to be modified.

7.5. Decompressing Client

The decompressing client was perhaps the simplest of the applications. The decompressing client basically consisted of reading in a byte stream, determining start and end points of a compressed region and decompressing accordingly. The decompressing client needs to store two states, *normal mode* and *strand run mode*. Normal mode means that the bytes read from the compressed file should be copied to the output location. Strand run mode means the decompressing client should buffer the bytes in the file, and mutate the seed and number of generations to recreate the original data. The decompressing agent will switch between modes when a delimiter is read. Detection of the chosen delimiting symbol non-repeated would mean the strand should enter *strand run mode*. The next detection of the delimiting symbol would switch back to *normal mode.*

7.6. Results

During implementation phase we have used C#.NET 3.0. The reason for this was mainly the familiarity with the environment, as well as access to lambda expressions which was used extensively in this paper. SQL Server 2005 was used as the database of choice.

7.6.1. 1st Attempt: Negative Compression

The first successful application of typogenetic compression resulted in a negative compression (Log 1 in Appendix). This was achieved with a theorem table with 19 million entries. Compression took a little less than 24 hours. From a glimpse of the compression log, it seems like the implementation of typogenetic compression has merits but requires much more refinement. It is clear that some theorems being detected very regularly, whereas the more chaotic theorems which typogenetics was meant to capture was not evidently seen in this example.

- **Theorem Limit too small:** The limit of 128 for the theorem limit was too small. From the log it can be seen that the theorem limit for the axiom "GA" was regularly being hit. A higher theorem limit would have resulted in a higher compression.
- **Capturing Less Ordered Strands:** It is observed that the captured theorems involved large sequences of base "A." If this was further reduced we can see for most of the theorems had seeds that ended in "GA", hence producing long sequences of "A". If typogenetic compression is to become more useful it needs to capture more variable strands for achieving compression rather than capturing zeros.

7.6.2. 2nd Attempt: Small Compression Achieved

For the second attempt (Log 2 in Appendix), several things were changed in order to increase compression ratio. Firstly the allowed theorem length was increased to 764. This would allow the algorithm to compress strand runs at a larger compression ratio. The SQL used for the table initialization code was optimized[3] as well as a check to halt mutations if a theorem produced already exists. Calculating the lookup table with strand length of 764 for 24 byte axioms proved much longer than for strand lengths of 128. At the time of compression, the table had only processed 250,000 axioms out of a possible 16 million. The database file size had also increased to 40GB. This made the bottom up method less appealing as a compression method. The increase in strand length increased the effectiveness of the method slightly. Instead of increasing the size, a very small compression was made.

Input File Size (bytes)	Compressed File Size (bytes)	Ratio
671660	671748	0.013 % increase

Table 5: First Attempt Compression Results

Input File Size (bytes)	Compressed File Size (bytes)	Ratio
671660	669150	1% compression

Table 6: Second Attempt Compression Results

7.7. Issues

From inspection of compression in Log 2, it is clear that the same problems plaguing the first attempt still applied to the second attempt. The theorems found in the file were too ordered and instead of the strand length being too small the maximum generation of 256 generations was reached. For typogenetic compression to be effective a lot more work was required on theorizing theorems produced by translation processes.

CONCLUSION

From observing the typogenetics results, potential is there to use typogenetics as a form of compression. However, for typogenetics to work as a mainstream method of compression, more work is required in

[3] Insert Ignore feature of SQL Server was used to reduce database calls by half or more.

understanding the theorems types generated by typogenetics. The ability to compress long strands into small axioms and a number cannot be dismissed. With enough processing power, and more theorem coverage, this technique may become a powerful way of compressing all file types, including files with low statistical redundancy. Perhaps the most important finding is the low coverage of typogenetics for strands of a certain length. The growth factor of typogenetics may need to be slowed and different operators may be needed to increase coverage of this technique. Once a large coverage is achieved, typogenetic compression will become a powerful server side technique. From the first prototype of typogenetics it was apparent that this technique was extremely slow even for small files. Major improvements to the design of the compressing client is required if this technique is to become usable. No user will have the patience to wait 24 hours for a 600KB file to compress. Caching and minimizing database calls for theorem checks will go a long way towards achieving this. From the test table generated, a 40GB database was generated when only 250,000 axioms were calculated. Typogenetic compression needs a better decision procedure to be more effective.

Investigation into further understanding of the theorems produced will be necessary in making the proposed typogenetic compression practical. The top down approach may be required in compressing extremely large strands. The top down approach is seen as a more powerful and scalable way of compressing. If a bottom up approach is to be used to calculate larger axioms, then further efforts need to be made in exploring extremely large databases together with their efficient storage and retrieval. The compression of strings of characters inside a string is also a future topic.

REFERENCES

[1] Bruzzone, A.G., Longo, F., 2005. Modeling & Simulation applied to Security Systems. *Proceedings of Summer Computer Simulation Conference*, pp. 183-188. July 24-28, Philadelphia, Pennsylvania, USA.

[2] Cawsey, A. (1998). *The Essence of Artificial Intelligence.* Europe: Prentice Hall.

[3] Gustavsson, R. & Fredriksson M., (2003). Sustainable Information Ecosystems, Lecture *Notes in Computer Science*, Volume 2603/2003, Springer Berlin / Heidelberg, 123-138.

[4] Hofstadter, D. R. (1979). *Godel, Escher, Bach, an Eternal Golden Braid* (20th Ed). NY Books, p511.

[5] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems* (5th Ed). MIT Press, USA.

[6] Holland, J. H. (1995). *Hidden Order - How Adaptation Builds Complexity.* Basic Books.

[7] Johnson, S. (2002). *Emergence.* Penguin Books.

[8] Mitchell, M. (2002). *An Introduction to Genetic Algorithms.* Massachusetts: MIT Press Cambridge.

[9] Waldrop, M. M. *Complexity: The Emerging Science at the Edge of Order and Chaos.*

[10] Ring, J.; Madni, A.M., (2005). Key Challenges and Opportunities in 'System of Systems' Engineering, IEEE International Conference on Systems, Man & Cybernetics Vol 1 Issue , 10-12 Oct., pp. 973-978.

APPENDIX

GAAAATAAAAAAAAAAAAAA compressed GAAAAT:14
CAAAGAAAAAAAAAAAAAAA compressed CAAAGA:14
GAAAAGCATAAAAAAAAAAAAAAAA compressed GAAAAGCAT:16
GACAAAAAAAAAAAAAAAAA compressed GAC:17
GAAAAAAAAAAAAAAA compressed GA:14
GACAAAAAAAAAAAAA compressed GAC:13
GAAGCTTAAAAAAAAAAAAA compressed GAAGCTT:13
GAAAAAAAAAAAAAAA compressed GA:14
GAAAAAAAAAAAAAAAAAAAAA compressed GA:21
GAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAA compressed GA:126
CCCCGAAAAAAAAAAAAAAA compressed CCCCGA:14
GAAAAAAAAAAAAAAAAAAAAA compressed to GA:21
GAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAA compressed GA:126
CCCCGAAAAAAAAAAAAAAA compressed CCCCGA:14
GAAAAAAAAAAAAAAAAAAAAA compressed GA:21
GAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAA compressed GA:126
CCCCGAAAAAAAAAAAAAAA compressed CCCCGA:14
GAAAAAAAAAAAAAAAAAAAAA compressed GA:21
GAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAA compressed GA:127

Log 1: First Attempt at Compression

GATGAGTAAAAAAAAAAAAAAAA compressed GATGAGTA:16
CAGAAAAAAAAAAAAAAA compressed CAGA:14
GAAAAAAAAAAAAAAA compressed GA:14
GAAAAAAAAAAAAAA compressed GA:13
CGAAGAAAAAAAAAAAAAAA compressed CGAAGA:14
GATAAAAAAAAAAAAAAA compressed GAT:15
GATAAAAAAAAAAAAA compressed GAT:13
GAAGAAAAAAAAAAAAAAA compressed GAAG:15
TGAAGATAGAAAAAAAAAAAAAAA compressed TGAAGATG:9
GACATGAAAAAAAAAAAAAA compressed GACATG:14
CTTAGAAAAAAAAAAAAA compressed CTTAGA:13
GAAAAAAAAAAAAAAA compressed GA:14
GATAAAAAAAAAAAAA compressed GAT:13
GAAAAGAAAAAAAAAAAAA compressed GAAAAG:13
GAGAATAAAAAAAAAAAAAA compressed GAGAAT:7
GAAAAAAAAAAAAAA compressed GA:13
CCTATTAAGAAAAAAAAAAAAAAA compressed CCTATTAAGAA:13
TTGAAAAAAAAAAAAAAA compressed TTGA:14
GAAAAAAAAAAAAAAAAA compressed GA:16
TAGATGAAAAAAAAAAAAA compressed TAGATG:13
TGAAGAAAAAAAAAAAAAAA compressed TGAAGA:14
GAATAGAGAAAAAAAAAAAAAAAA compressed GAATAGAG:16
CGAGGATAAAAAAAAAAAAA compressed CGAGGAT:13
CCCGGAGAAAAAAAAAAAAA compressed CCCGGAG:7
TCAAGAAAAAAAAAAAAAAA compressed TCAAGA:14
CCTTGAAATAGAAAAAAAAAAAA compressed CCTTGAAATAG:7
GAAAAAAAAAAAAAA compressed GA:13
GATAAAAAAAAAAAAA compressed GAT:13
GAAGTATAAAAAAAAAAAA compressed GAAGTT:13
GCACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC compressed GCA:41
GCTCCCCCCCCCCCC compressed GCT:13
TGTCGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC compressed TGTCGC:42
CAGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC compressed CAGC:40

Log 2: Second Attempt at Compression