Compression of Synthetic-Vehicular-Traffic Flow using "Complemented Binary Representation"

■ David Sánchez
davidenrique@gmail.com

Escuela de Ing. Informática
Universidad Católica Andrés Bello
Caracas - Venezuela

Fecha de Recepción: 8 de octubre de 2008
Fecha de Aceptación: 12 de enero de 2009

Abstract

Vehicle traffic studies use Nagel-Schreckenberg (NaSch) based simulations that produce significant-size information files. In order to save that information, common compression techniques are not enough to achieve high compression in minimal time. The present article demonstrates how to compress those synthetic-vehicular-traffic flows using binary representation (data modeling) and .zip codification (generic data modeling with entropy-coding) to get a 12:1 compression ratio.

Resumen

El estudio de tráfico vehicular utiliza entre sus métodos de análisis la simulación sintética basada en el modelo original de Nagel-Schreckenberg (NaSch). Dichas simulaciones arrojan archivos de información de gran tamaño que es necesario almacenar. La aplicación exclusiva de técnicas comunes de compresión no es suficiente para lograr altas tasas de compresión en mínimo tiempo. El presente artículo aborda la compresión de dichos modelos de flujos de tráfico mediante representación binaria (modelado de datos) y codificación .zip (modelado genérico de datos y codificación-entrópica) para obtener tasas de compresión de información de 12:1.

1. Introduction

This article describes a compression technique that allows a more efficient database storage of "Com-

mon Format"[1] synthetic vehicle traffic simulations. To accomplish the technique, it was assumed that the principal cause of the huge "Common Format" file sizes created by NaSch simulators was information represented using statistically redundant data; then, NaSch-obtained information represented using byte flows without statistically redundant data would be an optimal compression technique.

The technique was created during the development of an automatic information tool to analyze synthetic simulations of vehicular traffic[1]. The information tool was custom made under specific UCAB-CIDI[2] requirements.

The compression algorithm was created in order to animate previously computed data (obtained from NaSch simulators) and achieve frame-by-frame analysis of traffic conditions.

The information in this article has a limited radio action to compression of vehicular traffic flow. Nevertheless, the technique can be applied with few modifications to almost any kind of modeling which involves particles flow and require some kind of compression to store data about their behavior.

Particle flow modeling problems are common to gas investigations, biological epidemics, population migrations, fluid research, and many in which, it is possible to create cellular automata that generate representations similar to "Common Format". For all of them, "Complemented Binary Representation" can be a useful compression technique.

This paper also includes explanations about how the information tool works applying the compression technique. The tool was an essential instrument for the experimental demonstration of the capabilities of the compression algorithm.

This research is not a definitive solution, but rather a starting point for further investigations.

2. Previous work and motivation

UCAB-CIDI researchers [1] [2] [3] raised a clear concern about how to automate the processing and organization of synthetic data of vehicular traffic in order to perform the analysis and deductions of its behavior.

The former procedure to perform those analysis and deductions consisted in the following:

Some synthetically data were considered by variations of a model parameters. Data was stored in a text file.

A Cellular Automata (NaSch-model program written in C) was run using the parameters previously stored on the text file. This program is time consuming due to a highly iterative code. Section 2.1 describes the NaSch-model in detail.

1. Some output files were obtained from the previous step. One of them contains processed data; the others are graphical representations of the vehicles' movement. These files are printed and taped together as a continuous-form paper. This representation, detailed in section 2.2, is the principal obstacle to achieving animation and storing. For a 1600 meters two-way street, there are two graphical representation files, each about 60 single-space-letter-size pages, if opened in Microsoft Word, with a font size of 12 pt (Windows).

Steps 1, 2 and 3 were repeated checking graphical representation files.

Generated information was analyzed.

Many simulations are required to obtain a useful representation of a studied problem, in some cases 1000 or more. Under the described manual process, a simple 10-simulation problem took a whole week.

Increasing the number of simulations to useful representation levels quickly turned into an unmanageable task. The huge amounts of files made the analysis a cumbersome process since it was almost impossible to find a particular simulation section.

To solve the problem, a program was developed that included the following:

- A MySQL database to store all the information.
- A Java graphical interface to allow access to processed and unprocessed data. This program also runs the NaSch-based synthetic simulator and creates animations with the results.

1 There is no standard naming for this representation technique used by researchers.

2 Universidad Católica Andrés Bello Engineering and Development Research Center (Centro de Investigación de Ingeniería de la Universidad Católica Andrés Bello)

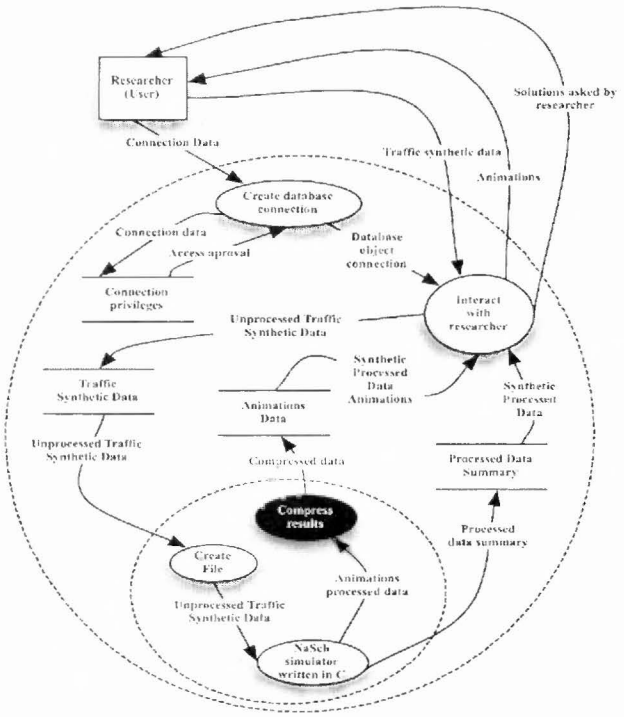

Figure 1. Data Flow Diagram (Level 2). Darkest oval shows compression module location.

The synthetic traffic simulator (NaSch simulator) created by UCAB-CIDI researchers, was modified to be loaded as a C Dynamic Library under a JNI interface. Process intercommunications between C (NaSch simulator) and Java (Graphical interface) were achieved through files.

A compression module processed the huge text files generated by the simulator. Figure 1 shows the location of this compression module in the information tool. The compression module is the central topic in the present paper.

Because there is no explicit terminology, those text files generated by NaSch simulators are referred to in the present article as "Common Format" text files. The format is described in section 2.2.

2.1. NaSch model

Through cellular automata, researchers create discrete models of space, time and speed. Space is discretized in a way each cell of the cellular automata is occupied by a vehicle only[4]. Time development follows simple rules using stochastic elements[5].

A simple model based on cellular automata which can reproduce many of the characteristics observed in the traffic is the Nagel-Schreckenberg (NaSch) model[6]. In this model, a vehicle state n is characterized by a position x_n and speed $v_n \in \{0,1,2,...v_{max}\}$. The gap between the nth-vehicle and the vehicle in front of that one is $d_n = x_{n+1} - x_n$. For each time stamp, the array of vehicles is updated according to the following rules[6]:

1. Acceleration

If $v_n < v_{max}, v_n \rightarrow min(v_{n+1}, v_{max})$

2. Deceleration due to other vehicles

If $d_n < v_n, v_n \rightarrow min(v_n, d_n - 1)$

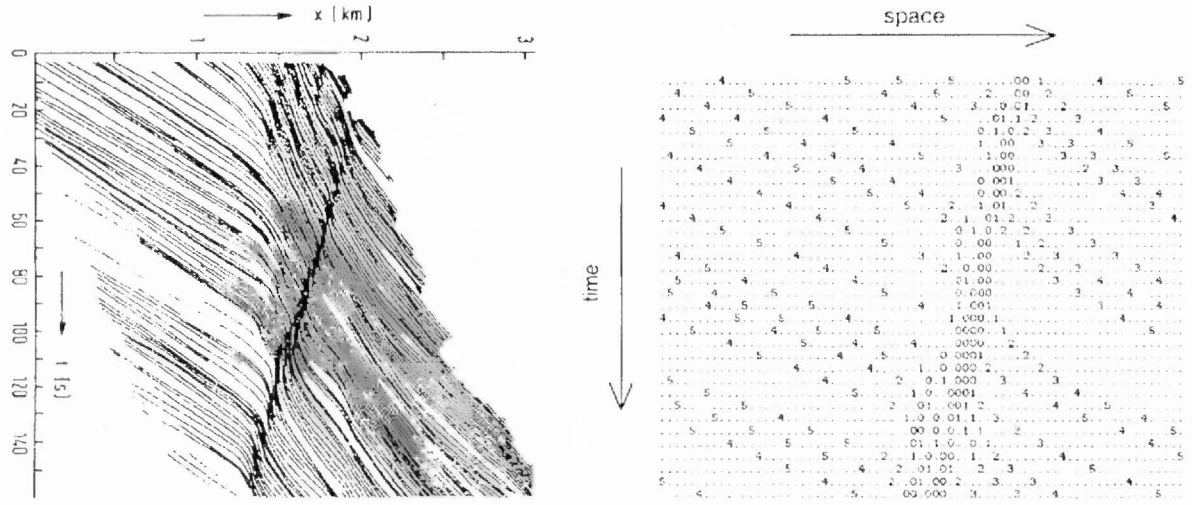

Figure 2 Traffic jams representation. Left: Empiric data. Right: Computer simulation using NaSchmodel. Numbers 0 to 5 represent vechicle speed [4]

3. Random break

If $v_n > 0, v_n \rightarrow max(v_n - 1, 0)$ with probability p

4. Vehicle moving (Driving)

$x_n \rightarrow x_n + v_n$

Rule 1 represents the driver's wish to drive at the maximum allowed speed. Rule 2 prevents collisions and vehicles' entrance to the circulation lane. Rule 3 adds environmental characteristics and incorporates asymmetric acceleration and deceleration. Rule 4 moves the vehicle with the speed determined in the previous steps[4].

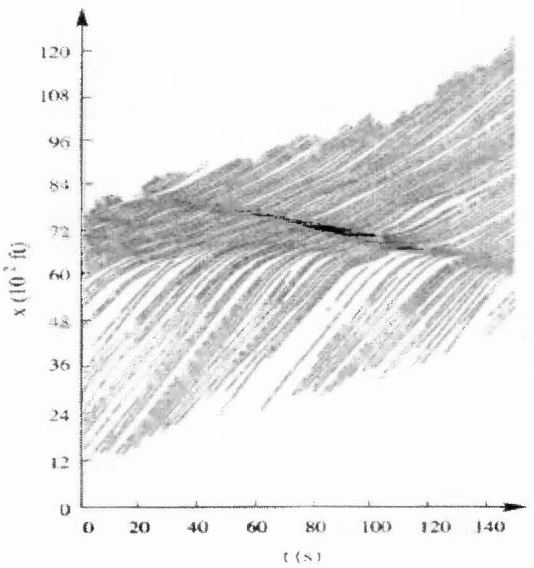

Figure 3. Traffic Jam representation done by Treiterer and Myers in 1974 based on aerial photography[4].

Using a NaSch model and change lane algorithms, traffic researchers can generate computer simulations that match the results acquired through measuring. Figure 3 shows one of those first representations made by Treiterer and Myers in 1974 using aerial photography [4].

Figure 2 shows a comparison between a model obtained through measuring and a synthetic model[4].

2.2. "Common Format" Text Files and their Inconveniences

Even though there is no standard representation for traffic solutions and problems, most researchers have used techniques similar to Figure 3 to explain vehicular traffic behavior.

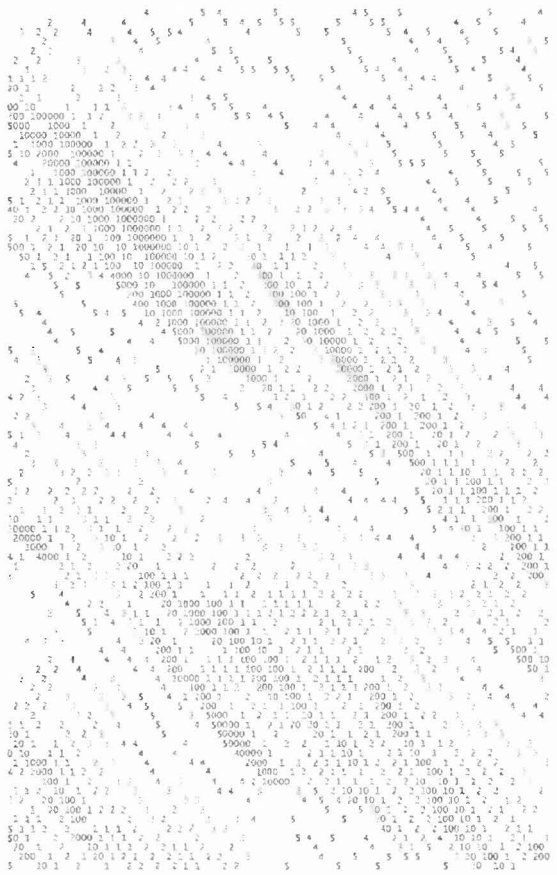

Figure 4. UCAB-CIDI NaSch simulator sample. Numbers 0 to 5 represent speed[1].

As Figure 2 shows, representations obtained through computer simulations use a similar nomenclature. Figure 4 shows the output of one of the UCAB-CIDI NaSch simulators used to analyze vehicular traffic in Caracas (Venezuela)[1].

Those "Common Format" files use numbers, usually from 0 to 5, to indicate vehicle speed and tabulated positions to show vehicle location. Each row represents a time step.

As a consequence of this static representation, the reader needs to imagine how vehicles move. Training is required to assess traffic jams, and the poor format presentation makes it difficult for researchers to analyze the phenomenon.

To avoid those limitations, it was decided to make animations with those models. However, due to simulator operation it is very difficult to generate real time animation (animate at the same time that simulation is being generated). That is why it was decided to create the animation from simulation final results stored in files.

As described earlier, each time the simulator is run, a figure-4-like file is generated. The file includes dozens of pages for each one of the lanes that can be found in a route.

A typical text file for a 5-Km-one-way route[3] has an approximate size of 248 KB (each lane). For a two-way route the researcher gets two 248 KB files, which adds up to 496 KB.

In order to get a quality analysis of the problem, it is necessary to consider at least 100 simulations, each one obtained after variations of the model parameters.

Analyzing a two-way lane, with those 100 simulations would require a database space of $49,600$ KB (48.44 MB). Because of the finite characteristics of today's storage equipment (Databases in personal computers) *is essential some method to compress the information.*

3. Background

In computer science, compression is codified information using less bits (or any other saving information unit) than the original data representation[7]. Compression's main objective is to minimize the space required to store data[8]. The main disadvantage of compression is the need for a time consuming decompression process[8].

Compression's theoretical frame is supplied by "information theory" (highly related to "Algorithm Information Theory"). These case studies were essentially created by Claude Shannon, who published fundamental papers on the topic in the late 1940s and early 1950s [7]. Nevertheless, compression's fundamental concept is at least as old as Romans, who realized that the numeral V needed less space on a stone tablet than the $IIIII$ representation[9].

Compression theory establishes differences between information and data that might not exist in other contexts.

Information is the communication or acquisition of knowledge that allows to expand or specify what it is known about a particular subject[4].

In the traffic study context, the term information would be related to vehicular-flow intrinsic characteristics, which are independent from its representation. The key word for "information" is meaning.

The term data refers to the way information is represented. The medium in which information is contained. The Key word for "data" is representation.

For example, the letter "A" has a known meaning relating to the language context. It could be the first letter of the Roman alphabet, the first open vowel, an article in English, etc. This is information. Data relating to the letter is font, color, size, form, etc.

Depending on how information is treated, there are two main compression techniques. Those based in "lossless" algorithms, which compress data based on statistical redundancy and "lossy" algorithms, which compress data losing fidelity[8].

"Lossless" techniques do not lose information and therefore are preferred to compress critical data. As a disadvantage, those compression-decompression techniques require high level resources in time and computing capacity[9]. Additionally, it is not possible to compress some kind of data and the iterative application of those algorithms does not elevate compression ratio[9].

"Lossy" compression techniques imply removing fidelity. They require a deep understanding of the perceptual limitations and capabilities of receptors (Mostly human senses) to avoid losing relevant information. These techniques are primarily used in video, photography and music, where quality losses could be tolerated by spectators[9]. Iterative applications of lossy algorithms over the same data causes a complete loss of all data[9].

4. Solution

Compression techniques were analyzed over "Common Format" text files. It was necessary to compress the essential information expressed in those files, for them to have smaller footprint, no clearness sacrifice and short decompression time.

"Lossy" techniques were discarded as being unacceptable to lose information. Therefore, a "lossless" technique was applied to the compression problem.

3 It is considered a $1,600$ array to store the vehicles. It is about $5,600$ m, considering 3.5 m average length vehicles)

4 Definition provided by "Diccionario de la Real Academia Española" (Spanish Royal Academy Dictionary).

4.1. Data Analysis

There are two main problems with digital data compression while using "lossless" techniques. These are modeling and data entropy coding. Any representation of the real world exists in its digital form as a symbol (bit) sequence. Data Modeling problem consists in choosing the correct symbols to represent that information and predict occurrence probability of each one of those symbols. Entropy-coding problem consists in codifying each one of the symbols in the most possible compact way [10].

Data modeling is related to the kind of data to be compressed, while entropy-coding is an abstract problem that does not depend on the kind of data to be compressed[10].

While entropy-coding problems are well known, modeling problems are still unknown for many applications[10].

4.1.1. "Common Format" Relevant Information

"Common Format" text files most important information is vehicle position through time in flow lanes. Due to space-time discrete assumptions, interspaced vehicles and their positions are relevant and need to be preserved to ensure high quality.

4.1.2. "Common Format" Irrelevant Information

The "Common Format", shown in Figure 3, includes numbers from 0 to 5 to indicate vehicle speed. These numbers are included to give the reader something he can use to imagine vehicles movement, and are used by the NaSch simulator to create the model frames.

Because the main purpose of compressing the "Common Format" is to make a more explicit and comprehensible animation, it is unnecessary to save speed information. Vehicle movement is going to be shown using over positioned frames. Section 4.1.1 states that all frames have to be preserved.

At a glance, it might seem that saving the speed parameter can provide a smoother animation. However, the NaSch simulator includes discrete space and time in its functioning theory. This means, time only exists in the given frames. An artificially created smoother animation is not only intensive computationally but it would include artifacts to the model. Also, among the NaSch simulator rules there is a random component (Rule 3) with unknown behavior in-between the discrete instants.

The only way to make a smoother and more realistic animation without including information distortion is adjust the NaSch simulator so it uses smaller discrete time units.

In conclusion, there is no need to store 0-to-5 vehicle speed.

4.1.3. "Common Format" Irrelevant Data

"Common Format" files are text files. As such, they include the following irrelevant data:

- Text Headers

  Additional information about the operating system they belong to. This data includes creation time-stamp, modification time-stamp, execution permissions, codification format, etc. All that information would be contained in the database table that will store the compressed objects. Additionally, Relational Database Manager Systems offer transparency over the operating system they are running, so headers can be discarded.

- Break lines or carrier returns

  Returns characters at the end of each line that represent a circulation lane. Database includes a lane length parameter, so this information is redundant.

- Integer number representation

  Codification conventions for text files are preserved according to standards and operating systems. Some of those standards includes ISO 8859, EUC, Windows, Mac-Roman... and even Unicode schemes as UTF-8 or UTF-16[11].

  Most operating systems include a codification based on ASCII [5]. Every time a 1 or a 2 is written, this number has to be represented through an 8 bits codification (a byte).

  Continuing the explanation with ISO 8859 convention (formally known as ISO/IEC 8859), to codify a 1, for example, the code refers to the decimal number 49 (31_{hex}), and for a zero, it refers to number 48(30_{hex}).

  In ISO 8859-1, these characters are converted directly to the binary system[13]:

  Character 0 text document representation is 00110000_{bin}

  Character 1 text document representation is 00110001_{bin}

5 EBCDIC and CDC[12] representations are not currently used

In "Common Format" context, discarding speed as relevant information, a "1" represents a present vehicle and a "0" represents an absent vehicle. An ideal situation would be to represent this single information as concisely as possible.

Just using bits of a byte, an on bit would represent a present vehicle (1) and an off bit would represent an absent vehicle (0). Text files are representing the same information using a byte per character ("1" or "0") under a writing file code.

In other words, each time there is a present or an absent vehicle in a cell lane, "Common Format" Text files are using 8 bits to represent information that should only require only 1 bit (Present or absent). *There is a high data redundancy*.

4.2. Proposed solution

The main idea in solving the problem is to model traffic flow as an array of integer numbers. Those numbers, when read in binary form, would show the same information about vehicles position over time that is printed on the "Common Format" text files.

The solution would be developed in two phases; the first with the modeling technique (Compression by modeling) and the second, by complementing the modeling technique with a known modeling plus data entropy coding algorithm.

- Phase 1

1. Build a modeling algorithm that runs as follows:
   a. The compression module reads the "Common Format" file thrown by a NaSch typical simulator. A sample of these files can be appreciated in figure 3.
   b. Once in memory, all characters "0","2","3","4" and "5" are substituted by "1". All " "(white spaces) characters are substituted by 0. Line-breaks chars are deleted.
   c. Create a byte array.
   d. Divide the string previously obtained, the one that has "1000000100010..." codification, in groups of 7 characters and convert each one to a byte. Add the byte to the byte array.
   e. Store the byte array into BLOB field in a database.

   This algorithm would be referred to as "Binary representation" (Java code can be seen in Appendix B).
2. Verify that results from applying "Binary representation" generate a compression ratio of 7:1.

- Phase 2

3. Modify the "Representation binary" algorithm to complement it with a .zip compression technique, provided by standard Java libraries. .zip compression includes simple modeling (independent from data) and entropy-coding. This phase-2 algorithm would be referred to as "Complemented Binary Representation".
4. Measure results obtained from applying "Complemented Binary Representation".

   The compression module was placed inside a Java program that would be used for further condition and parameter traffic analysis.

   Appendix A explains how the host program works. The compression module is part of that host program and is contained inside a Java class.

4.2.1. Important considerations about into-byte conversion

Bytes are elements used for computer-to-computer communications and also for storing data as BLOBS in databases.

The algorithm created basically replaces "0", "2", "3", "4" and "5" into "1" and white spaces into "0". Then it takes this character string formed by "1" and "0" and segments them in groups of 7. Then each of these 7-elements groups is transformed into a byte. A question that could be raised about it: If 8 bits form a byte, and groups are made of 7 bits. Why is there a bit wasted in each byte formed?

The answer is intrinsically related to the way most modern computers operate. Bytes are used to represent positive and negative numbers. To achieve that representation a "complement by two" convention is used, in which, given n bits, the number interval that can be represented in "complement by two", goes to the interval $[2^{n-1}, 2^{n-1}-1]$ [14]. For an 8-bit representation,

Java Virtual Machine allows integer numbers in the range[-128, 127] [15].

In "complement by two" convention there are 7 bits to represent the numbers from 0 to 127 and all 7 bits combinations are valid. 8 bits strings can represent positive or negative numbers, and thus, not all 8 bits strings are valid under the convention. For vehicular traffic representation this means that not all 8 cells (with mixed vehicles and spaces) have a complement-by-two representation. Some samples:

- Vehicle sequence "10001000" cannot be represented as a byte because it represents the number 136 and is out of the byte representation range (Maximum is 127).
- Vehicle sequence "11100000" can be represented as a byte, because it represents the number -31 and is inside the range representation (Minimum is -128).

As can be seen, there would be a significant overhead on the into-byte conversion algorithm if valid and not valid sequences have to be considered. Sometimes, the algorithm could chop 8 cells, and other times, only 7 cells.

Although this approach would increase compression ratio, it would also increase algorithm complexity, time-consumption and computing overhead. Using only 7-bit sequences compression-decompression speed is incremented due to simpler algorithms.

4.2.2. Complemented Binary Representation Technique

As it was commented on section 4.1 about "Data Analysis" there are two main problems with digital data compression using "lossless" techniques. These are modeling and entropy coding of data. Once the modeling problem was "acceptably" solved (See section 5.1 for experimental results), it raised the need for developing and entropy-coding algorithm. However, the following considerations arose:

- Entropy-coding algorithms are well understood [10].
- Entropy-coding is an abstract problem weakly related to the type of data being compressed [10].
- The Java platform has full compression libraries (generic data modeling + entropy coding). Java platform includes a java.util.zip which allows instrumenting zip, gzip and PKZip compression formats [16].

The package java.util.zip was selected to complement the "binary representation algorithm". Advantages of using the pre-made package are:

- The ZLIB compression algorithm and its variants (zip,gzip, PKZip...) implemented in the package are well known by its compression-decompression speed.
- Compression packages are a main feature in the Java platform libraries and are well optimized to run in the Java Virtual machine.
- Possibility to specify the compression strategy (zip, gzip, PKZip...) and speed/strength compression relation.

The resulting algorithm will be referred to as "Complemented Binary representation" (Java code can be seen in appendix C).

5. Experimental results

Tests were conducted on a personal computer (Laptop) with the following hardware-software configuration:

1. Pentium III processor alike (AMD brand).
2. 256 Mb RAM.
3. 5 Gb hard disk space.
4. Xubuntu Linux 6.10.
5. Java SDK 1.5

The nature of the binary representation algorithm offers a constant compression ratio of 7:1 in all performed tests, using many different file sizes. Standard deviation was close to 0.

Other compression techniques that rely on generic data modeling and entropy coding (zip, gzip, PKZip, JAR, .tar.bz2, etc.) have variable compression ratios. Small variations on the files can produce completely different compression ratios, since algorithms could better recognize patterns [9]. As a consequence, compressing a given file is the only way to exactly know how high the compression ratio would be, and the results are usually only valid for that file. Nevertheless, most of alternative compression techniques tests offered average similar results for the same kind of data, presumably because of NaSch model intrinsic characteristics and the intelligence (entropy coding) of the algorithms. The best average compression for most of these techniques is about 6 :1 [17] [19].

As a consequence, and for illustrative purposes, only one test sample (Phase 1 and 2) was included in this article.

5.1. Phase 1

After applying the binary representation algorithm in a "Common format" text file of 252,200 bytes a ratio compression of 7.0727466:1 (7:1) was obtained. Database representation was 35.658 bytes size.

The following results were obtained when compressing the same text file using other techniques:

- .zip

Initial file size: 252,200 bytes = 246.3 KB

Final size: 39,658 bytes = 38.7 KB

Compression ratio: 6.3593726 : 1

- tar.bz2

Initial file size: 252,200 bytes = 246.3 KB

Final size: 32,299 bytes = 31.5 KB.

Compression ratio: 7.8082913 : 1

- tar.gz

Initial file size: 252,200 bytes = 246.3 KB

Final size: 39,668 bytes = 38.7 KB.

Compression ratio: 6.3577695 : 1

- .jar

Initial file size: 252,200 bytes = 246.3 KB

Final size: 39,658 bytes = 38.7 KB.

Compression ratio: 6.3593726 :1

5.2. Phase 2

Binary representation algorithm (Modeling only) returns a 7 : 1 compression ratio. Once applied, the .zip algorithm provided by java.util.zip to the 35,658 bytes file, produced a database representation of 20,748 bytes (Compression ratio 1.7186235 : 1).

In sum, starting from the initial file to the final compressed data:

Initial file size: 252,200 bytes = 246.3 KB

Final size: 20,748 bytes = 20.26 KB

Compression ratio: 12.155388 : 1

5.3. Results Analysis

Method	Size (Bytes)	Size (Kilobytes)	Compression ratio to 1
-	252,200	246.3	0.0000000
.tar.gz	39,668	38.7	6.3577695
.jar	39,658	38.7	6.3593726
.zip	39,658	38.7	6.3593726
Binary Representation	35,658	34.8	7.0727466
.tar.bz2	32,299	31.5	7.8082913
Complemented Binary Rep. (.zip)	20,748	20.3	12.1553880

Table 1. Compression techniques. Ascending ordered by ratio.

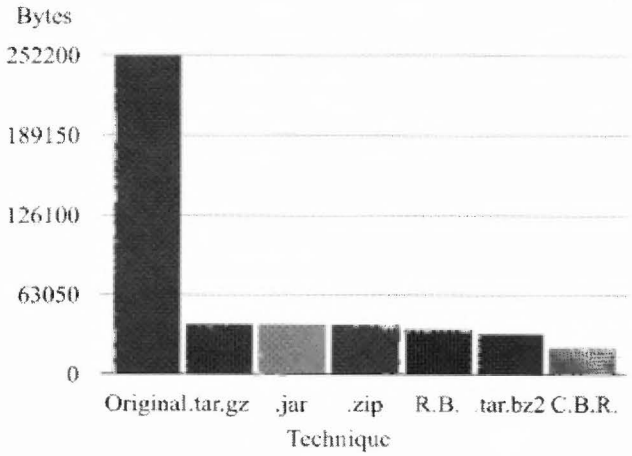

Figure 5 Compression methods considering data size resulting (Less bytes is better). "B.R." means "Binary representation". "C.B.R." means "Complemented Binary Representation (.zip)".

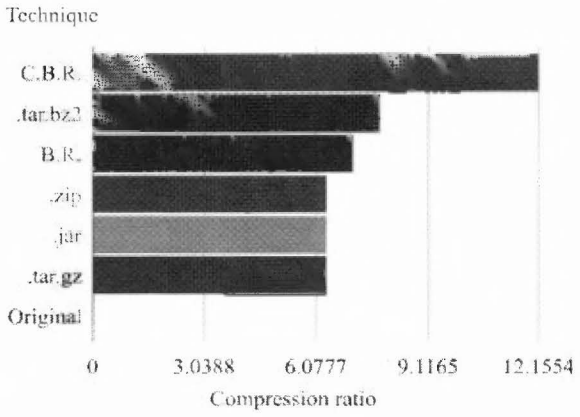

Figure 6. Compression methods. Ordered by compression ratios. (Bigger ratio is better). "B.R." means "Binary Representation". "C.B.R." means Complemented Binary Representation (.zip)".

As can be observed in table 1, the binary representation technique, "B.R.", which only uses data modeling, returns a ratio compression equivalent to other methods which include modeling (although generic) and entropy-coding (.zip, tar.gz, .jar, tar.bz2). Figure 5 shows a visual comparison of techniques considering data size representing the information. Figure 6 shows a visual comparison of techniques considering compression ratio.

As can be observed in table 1, complemented binary representation technique, "C.B.R.", has a compression ratio superior to other methods that include modeling (although generic) and entropy-codification (.zip, tar.gz, .jar, tar.bz2). Figure 5 shows a visual comparison of techniques highlighting data size of the information. Figure 6 shows a visual comparison of compression techniques against compression ratio.

Improved compression ratio is possible due to the previous information modeling. Compression time of entropy-coding algorithm is small because it is being done over previously compressed-by-modeling data.

"Complemented Binary Representation" was used in the final solution of the project.

Conclusions

The main reason for the huge size of "Common format" files is the representation of information using statistically redundant data. Understanding the nature of the represented information can achieve an enormous increase in computer efficiency.

Researchers should not scrimp resources in the correct use of computing power. Modeling is the most difficult part of compression techniques since it is related to the nature of the information itself. Most of the time, modeling has to be done by the researcher himself since he is the one who knows what needs to be preserved.

Most of today's computing problems try to be solved by increasing hardware capabilities. However, this article shows how a simple technique can bring tasks previously thoughts as supercomputer realm to portable computers.

In most situations, researchers would opt for a common compression technique, such as .zip, .jar, or .rar without previously analyzing the data being considered. Generic modeling is only marginally useful in most situations. Best results are achieved trough a complete understanding of information nature.

As can be seen, synthetic information acquired through NaSch simulators can be expressed using byte arrays without statistically redundant data. Turning information into this low-level data representation format offers a compression ratio as good as the more sophisticated techniques that include both generic modeling and entropy-coding algorithms.

Even better, applying a common compression algorithm[6] over a previously compressed data by modeling allows outstanding compression ratios [17]. In this case an average of 12:1 compression ratio was achieved. It is rare to achieve this high level compression with a fast lossless algorithm; so once again, the importance of modeling can be overestimated.

Thanks to the inclusion of the "Complemented Binary Representation" compression technique, the information tool developed was able to produce understandable animations of the given problems, using just a portable computer.

Better use of resources can dramatically increase the kind of problems that can be analyzed and solved. Animation of those traffic problems opened the door for future problems that would be too difficult or even impossible to imagine with "Common Format" representation. Among those problems are access roads, distributors, service lanes, passing cars and many others.

This research is not a definitive solution to the vehicle flow compression-modeling problem. Further works can refine the modeling technique using Jots[7] instead of bytes. However, the complexity of the analysis could rise exponentially and it would be more difficult to understand and apply by most researchers.

7. References

[1] A. Aponte and J. A. Moreno, *"Cellular automata and its application ot the modeling of vehicular traffic in the city of caracas,"* Master's thesis, Centro de Investigación y Desarrollo

6 It was used .zip because of its high speed, however, we presume results are valid for similar techniques.

7 Jot is a unit of data equal to a $\frac{1}{F}$ of a byte, where F represents an integer bigger than 8. Jot represents information smaller than a bit[10]. Because the minimum electronic representation is a bit, to achieve its purpose, Jot uses meaning techniques of bit sets. That is why its definition is made using bytes.

de Ingeniería (CIDI), Facultad de Ingeniería UCAB Venezuela, Laboratorio de Computación Emergente LACE, Facultad de Ingeniería UCV Venezuela, 2006.

[2] A. Aponte and S. Buitrago, *"Global optimization in modeling vehicular traffic,"* Centro de Investigación y Desarrollo de Ingeniería (CIDI), Facultad de Ingeniería UCAB Venezuela, Laboratorio de Computación Emergente LACE, Facultad de Ingeniería UCV Venezuela, 2006.

[3] A. Aponte, S. Buitrago, and J. Moreno, *"Inappropriate use of the shoulder in highways. impact over the increase of gas consumption,"* Centro de Investigación y Desarrollo de Ingeniería (CIDI), Facultad de Ingeniería UCAB Venezuela, Laboratorio de Computación Emergente LACE, Facultad de Ingeniería UCV Venezuela, Departamento de Cómputo Científico y Estadística, USB, 2007.

[4] G. Poore, *"Emergent phenomena in vehicular traffic."* , May 2006.

[5] B.-H. Wang, L. Wang, and B. Hu, *"Analytical results for the steady state of traffic flow models with stochastic delay,"* The American Physical Society, 1998.

[6] K. Nagel, D. Wolf, P. Wagner, and P. Simon, *"Two-lane traffic rules for cellular automata: A systematic approach,"* Los Alamos National Laboratory, 1997.

[7] T. Strutz, Bilddaten-Kompression (Image Data Compression) ISBN 3-528-23922-0. Vieweg Braunschweig/Wiesbaden, 3 ed., July 2005.

[8] G. Blelloch, *"Introduction to data compression,"* Computer Science Department Carnegie Mellon University , October 2001.

[9] T. Halfhill, *"How safe is data compression?,"* Byte Magazine, vol. 19, no. 2, pp. 56–74, 1994.

[10] W. D. Withers, *"A rapid entropy coding algorithm,"* Dr. Dobb's Journal, vol. 22, pp. 38–44,78, April 1997.

[11] J. Lewis and N. Dale, *Computer Science Illuminated* ISBN 0763741493. Jones and Barlett Publishers, 3 ed., 2006.

[12] Dale and Orshalick, *Introduction to Pascal and Structured Design.* Mc Graw Hill, 1 ed., 1986.

[13] ISO/IEC, *"Iso/iec 8859 7-bit and 8-bit codes and their extension."* , February 1998.

[14] P. C. Ramon Mata-Toledo, *Introducction to Computer Science.* Shaun, Mc Graw Hill, 1 ed., 2000.

[15] H. Schildt, Java 2: *The Complete Reference.* Mc Graw Hill, 4 ed., 2003.

[16] D. Flanagan, *Java in a Nutshell.* Mc Graw Hill, 2 ed., 1997.

[17] W. Heriman, *"Practical compressor test."* , July 2005.

Appendices

A. Working program Scheme

Screen shots were taken from a Spanish set up because the program was created to work in a "Spanish language" environment.

Before running the program, a MySQL server must be working. The MySQL database used in this article was configured to run every time the computer was turned on.

Once the program is running, the following steps are performed:

1. A dialog shows up to start a local or remote MySQL connection.

   Once confirmed, this dialog creates a data object that keeps the necessary pass phrase to allow the database connection. This object validates keys (Figure 7).

   To achieve its work, the dialog (Connection.java class) creates a connection object to the database. If this object (ConnectionData class) is valid, by achieving a satisfactory authentication to the database server, the ConnectionData is automatically crafted into a new object that will handle database queries. This object called SQLQuerier (SQLQuery class) is responsible for database communication using data kept safe inside the connection object (ConnectionData).

   The dialog, as well as the object used to collect data from the user (Connection object), is erased from memory. After this operation, the only way to access the database is through SQLQuerier. It is not possible to modify connection parameters.

2. Program Main Menu.

   SQLQuerier object is inserted into a "Palette" class. This palette is an interface object which allows the user to perform diverse actions such as creating new searching windows (Figure 8).

   Originally, the palette was designed to stay above all windows, but this behavior was changed due to user demand and its actual behavior is like any other window.

3. "Rampa Acceso" (Inner) selection

   This option is in designing stage.

4. "Ida y Vuelta" (Two-way drive)

   "Consultar" (Consulting) menu creates a new window with all data found (Processed or non processed information) as well as a tab to filter information. A sample of this window can be seen in figure 10.

   "Consultar" windows have two "Tabs". The first one can be seen in figure 10. Buttons allow the user to perform different operations on the database. Those operations are:

- Nuevo (New)

  New data to be save in the database. Only simulation entry data is editable.

- Guardar (Safe)

  Save data introduced.

- Eliminar (Delete)

  It allows erase info from the database. Deleting requires a previously selected item. For safety reasons, only one simulation can be erased at a time.

- Simular (Simulated)

  It runs a C simulator library.

- Ver (See)

  It shows an animation with previously processed data by the C simulator.

  A sample animation can be seen in figure 8. This window includes the code that identifies it in the database in the tile. It is possible to have different open windows at the same time.

  In the animation window, each dot represents a vehicle.

- Exportar (Export)

  Export button (Currently disabled) allows selected data to be saved in different formats.

  The Searching ("Buscar") tab allows entering parameters to filter data from the database (Figure 11).

  Once data is introduced, the search is performed and the window shows the result in the "Base de datos" (database) tab. If no parameters are specified, it returns all simulations previously inserted (processed or not) in the database.

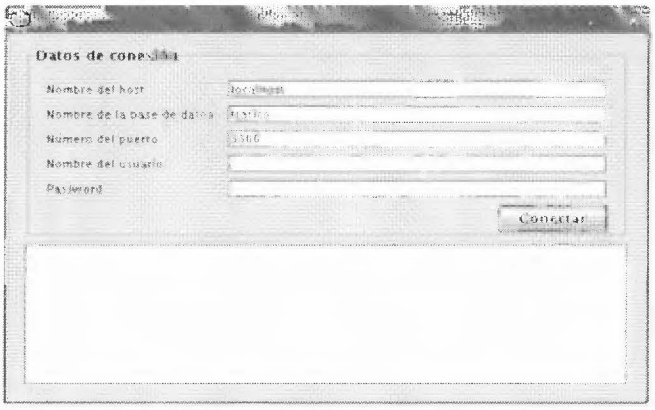

Figure 7. Screen to get connection parameters

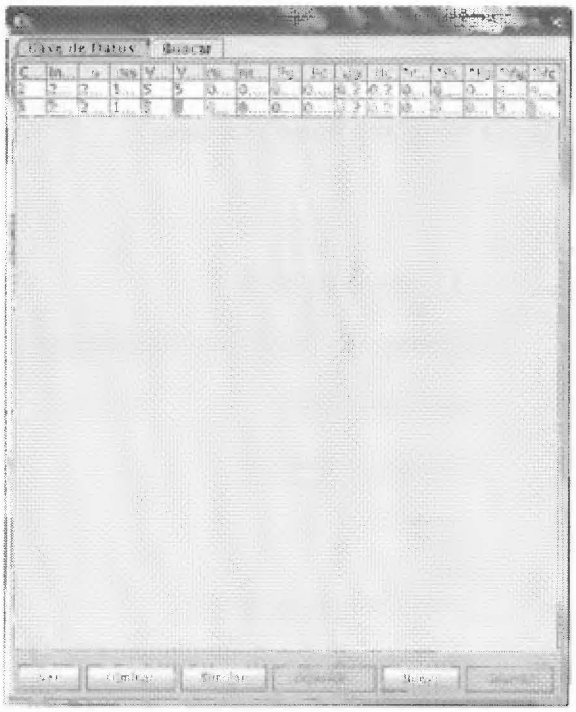

Figure 10. This Window shows all data saved in the database ("Ida y vuelta" (Two-way lane) option).

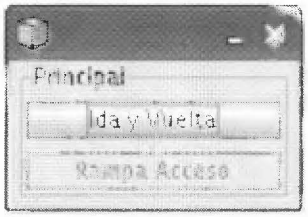

Figure 8. Main Menu

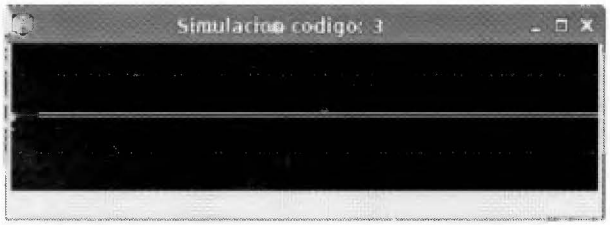

Figure 9. Animation with processed data.

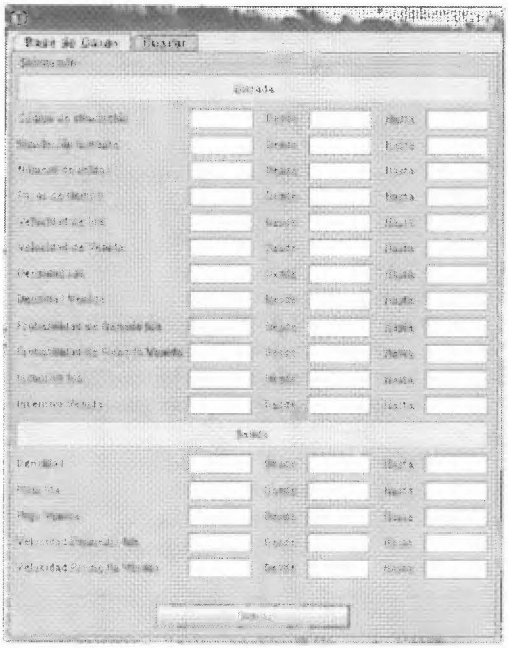

Figure 11. Database Searching tab ("Ida y vuelta" Two-way lane option).

B. "Binary Representation" algorithm implemented in Java

Due the small size of the algorithm, it was decided to place it inside the Read.java class.

```
public void readFromFileCiclopeg() {
   try {
        InputStream in = new FileInputStream(outCiclopegFile);      //It creates a reading input stream to read
                                                                    //"outCiclopegFile" file
        int size=in.available();                                    //It measures the size of the file is going to be
                                                                    //read
        byte b[]=new byte[size];                                    //It creates a byte array the size of
                                                                    //the file being read
        in.read(b);                                                 //Read all text file bytes and
                                                                    //save them in b
        in.close();                                                 //Close the file
        String s= new String(b,0,size);                             //It creates a string with all read bytes
                                                                    //converted to chars
        s = s.replace('0','1');                                     //Replace 0 s by 1
        s = s.replace('2','1');                                     //Replace 2 s by 1
        s = s.replace('3','1');                                     //Replace 3 s by 1
        s = s.replace('4','1');                                     //Replace 4 s by 1
        s = s.replace('5','1');                                     //Replace 5 s by 1
        s = s.replace(' ','0');                          //Replace al white spaces by 0
        StringBuffer sb = new StringBuffer(s.substring(1));         //It creates a string buffer to perform operations
                                                                    //on data
        int i=0;                                                    //This loops erases all line breaks
        while (i<sb.length()) {
         if (sb.charAt(i)=='') {
            sb.deleteCharAt(i);
         }
         i++;
        }
        String bitConverterString=sb.toString();                    //Transform a String buffer into a new String
        int top = (bitConverterString.length()/7)+1;                //Measures the byte array length
        byte[] bytes = new byte[top];                               //It creates the byte array
        String n;                                                   //Helping String
        int j=0;                                                    //Counter
        while (bitConverterString.length()>7) {                     //Chop the string in 7 bits size pieces
             n=bitConverterString.substring(0,7);
             bytes[j] = Byte.parseByte(n,2);                        //Transforms 1 and 0 sequences in bytes
             bitConverterString=bitConverterString.substring(7);    //String chopped
             j++;
        }
        n=bitConverterString;                                       //It processes the last one
        bytes[j] = Byte.parseByte(n,2);
        int answer = this.sqlQuerier.handleUpdateInOutAnimacionIda(code,bytes);
                                                                    //It saves in the database
   } catch (FileNotFoundException e1) System.err.println("File not found: "+ file);
     catch (IOException e2) e2.printStackTrace();
  }
}
```

C. "Complemented Binary Representation" implemented in Java

```
public void readFromFileCiclopeg() {
  try {
     InputStream in = new FileInputStream(outCiclopegFile);          //It creates a stream to read the file
     int size=in.available();                                        //Measures the size of the file to be read
     byte b[]=new byte[size];                                        //It creates a byte array the same size of the file
     in.read(b);                                                     //Read all bytes from the text file and
                                                                     //saves them in b
     in.close();                                                     //Close the file
     String s= new String(b,0,size);                                 //It makes a string with all bytes read
                                                                     //transformed into chars
                                                                     //Compression by modeling start
     s = s.replace('0','1');                                         //Replace zeros by 1
     s = s.replace('2','1');                                         //Replace 2 s by 1
     s = s.replace('3','1');                                         //Replace 3 s by 1
     s = s.replace('4','1');                                         //Replace 4 s by 1
     s = s.replace('5','1');                                         //Replace 5 s by 1
     s = s.replace(' ','0');                                         //Replace white spaces by 0
     StringBuffer sb = new StringBuffer(s.substring(1));             //It creates string buffer to perform operations
     int i=0;                                                        //This loops erases line breaks
     while (i<sb.length()) {
          if (sb.charAt(i)=='')
             sb.deleteCharAt(i);
          i++;
     }
     String bitConverterString=sb.toString();                        //It transforms the string buffer into a string
     int top = (bitConverterString.length()/7)+1;                    //Measures array size
     byte[] bytes = new byte[top];                                   //It makes byte array String n;
                                                                     //Helping String
     int j=0;                                                        //Counter
      while (bitConverterString.length()>7) {                        //Chop the string in 7 bits size pieces
          n=bitConverterString.substring(0,7);
          bytes[j] = Byte.parseByte(n,2);                            //Transform 1 and 0 sequences in bytes
          bitConverterString=bitConverterString.substring(7);        //Chop the string
          j++;
     }
     n=bitConverterString;                                           //It process the last one
     bytes[j] = Byte.parseByte(n,2);                                 //It ends compression by data modeling
                                                                     //It starts complemented compressing //(Modeling + entropy-coding)
     FileOutputStream os = new FileOutputStream("zip_cache");        //It create an external file for data flow
     ZipOutputStream zos = new ZipOutputStream(os);                  //It wraps the external file into a compress
                                                                     //format
     zos.putNextEntry(new ZipEntry("zip_cache"));                    //It create header files
     zos.write(bytes);                                               //It compresses previously model data
     zos.closeEntry();                                               //Close data flow
     zos.close ();                                                   //Close file
                                                                     //End of complemented compression
                                                                     //Reading Compressed file to send it to
                                                                     //database
     InputStream is = new FileInputStream("zip_cache");              //Creates a reading stream for the
                                                                     //"outCiclopegFile"
     size=is.available();                                            //Measure the size of reading file
     byte compressedB[]=new byte[size];                              //It creates a byte array
                                                                     //file size
     is.read(compressedB);                                           //Read all bites from files and
                                                                     //save them in compressedB
     is.close();                                                     //Close the file and ends compressed file
                                                                     //reading
     int answer = this.sqlQuerier.handleUpdateInOutAnimacionIda(code,compressedB);
                                                                     //Save in the database
    } catch (FileNotFoundException e1) System.err.println("File not found: "+ file);
      catch (IOException e2) e2.printStackTrace();
}
```