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

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.

1. Introduction

This article describes a compression technique that allows a more efficient database storage of "Com-
mon Format\textsuperscript{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\cite{1}. The information tool was custom made under specific UCAB-CIDI\textsuperscript{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 \cite{1} \cite{2} \cite{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.

\textsuperscript{1} There is no standard naming for this representation technique used by researchers.

\textsuperscript{2} Universidad Católica Andrés Bello Engineering and Development Research Center (Centro de Investigación de Ingeniería de la Universidad Católica Andrés Bello)
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 automaton 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 is characterized by a position $x_n$ and speed $v_n \in \{0,1,2,\ldots v_{max}\}$. The gap between the nth-vehicle and the vehicle in front of that one is $d_n = x_n - x_{n-1}$. For each time stamp, the array of vehicles is updated according to the following rules[6]:

1. Acceleration
   \[
   \text{If } v_n < v_{max}, v_n \rightarrow \min(v_n, v_{max})
   \]

2. Deceleration due to other vehicles
   \[
   \text{If } d_n < v_n, v_n \rightarrow \min(v_n, d_n - 1)
   \]

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

Figure 2: Traffic jams representation. Left: Empiric data. Right: Computer simulation using NaSch model. Numbers 0 to 5 represent vehicle 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.

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 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. Compression's main objective is to minimize the space required to store data. The main disadvantage of compression is the need for a time consuming decompression process.

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. 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 'IIII' representation.

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.

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.

"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. Additionally, it is not possible to compress some kind of data and the iterative application of those algorithms does not elevate compression ratio.

"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. Iterative applications of lossy algorithms over the same data causes a complete loss of all data.

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 to 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 (31₁₅₀), and for a zero, it refers to number 48(30₁₅₀).

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

  Character 0 text document representation is 00110000₁₅₀
  Character 1 text document representation is 00110001₁₅₀

---

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 “1000001100010...” 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

- .jar
  Initial file size: 252,200 bytes = 246.3 KB
  Final size: 39,658 bytes = 38.7 KB
  Compression ratio: 6.3577695 : 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 = 20.3 KB
- Compression ratio: 12.155388 : 1

5.3. Results Analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>Size (Bytes)</th>
<th>Size (Kilobytes)</th>
<th>Compression ratio to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>.zip</td>
<td>39,658</td>
<td>38.7</td>
<td>6.3593726</td>
</tr>
<tr>
<td>.jar</td>
<td>39,658</td>
<td>38.7</td>
<td>6.3593726</td>
</tr>
<tr>
<td>.tar.gz</td>
<td>39,658</td>
<td>38.7</td>
<td>6.3593726</td>
</tr>
<tr>
<td>Binary Rep.</td>
<td>35,658</td>
<td>34.8</td>
<td>7.0727466</td>
</tr>
<tr>
<td>C.B.R. (zip)</td>
<td>20,748</td>
<td>20.3</td>
<td>12.155388</td>
</tr>
</tbody>
</table>

Table 1. Compression techniques. Ascending ordered by ratio.
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 thought as supercomputer realm to portable computers.

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


---

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 \( \frac{1}{8} \) 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.


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)
Figure 7. Screen to get connection parameters.

Figure 8. Main Menu.

Figure 9. Animation with processed data.

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

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.

```java
public void readFromFileCiclopeg() {
    try {
        InputStream in = new FileInputStream(outCiclopegFile);
        int size=in.available();
        byte b[]=new byte[size];
        in.read(b);
        in.close();
        String s=new String(b,0,size);
        s = s.replace("0", "1");
        s = s.replace("2", "1");
        s = s.replace("3", "1");
        s = s.replace("4", "1");
        s = s.replace("5", "1");
        s = s.replace("", "0");
        StringBuffer sb = new StringBuffer(s.substring(1));
        int i=0;
        while (i<sb.length()) {
            if (sb.charAt(i)=="" ) {
                sb.deleteCharAt(i);
            }
            i++;
        }
        String bitConverterString=sb.toString();
        int top = (bitConverterString.length()/7)+1;
        byte[] bytes = new byte[top];
        String n;
        int j=0;
        while (bitConverterString.length()>7) {
            n=bitConverterString.substring(0,7);
            bytes[j] = Byte.parseByte(n,2);
            bitConverterString=bitConverterString.substring(7);
            j++;
        }
        n=bitConverterString;
        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

```java
public void readFromFileCiclopeg() {
    try {
        InputStream in = new FileInputStream(outCiclopegFile);
        int size = in.available();
        byte[] b = new byte[size];
        in.read(b);
        in.close();
        String s = new String(b, 0, size);
        s = s.replace('0', '1');
        s = s.replace('2', '1');
        s = s.replace('3', '1');
        s = s.replace('4', '1');
        s = s.replace('5', '1');
        StringBuffer sb = new StringBuffer(s.substring(1));
        int i = 0;
        while (i < sb.length()) {
            if (sb.charAt(i) == '-')
                sb.deleteCharAt(i);
            i++;
        }
        String bitConverterString = sb.toString();
        int top = (bitConverterString.length() / 7) + 1;
        byte[] bytes = new byte[top];
        int j = 0;
        while ((bitConverterString.length() > 7) {
            bytes[j] = Byte.parseByte((n - 1) / 2);
            bitConverterString = bitConverterString.substring(7);
            j++;
        }
        n = bitConverterString;
        bytes[j] = Byte.parseByte((n2 - 1) / 2);
        FileOutputStream os = new FileOutputStream("zip_cache");
        ZipOutputStream zos = new ZipOutputStream(os);
        zos.putNextEntry(new ZipEntry("zip_cache"));
        zos.write(bytes);
        zos.closeEntry();
        zos.close();
        InputStream is = new FileInputStream("zip_cache");
        size = is.available();
        byte[] compressedB = new byte[size];
        is.read(compressedB);
        is.close();
        int answer = this.sqlQueryer.handleUpdateInOutanimationId(code, compressedB);
    } catch (FileNotFoundException e1) System.err.println("File not found: "+ file);
    catch (IOException e2) e2.printStackTrace();
}
```

// It creates a stream to read the file
// Measures the size of the file to be read
// It creates a byte array the same size of the file
// Read all bytes from the text file and
// Saves them in b
// Close the file
// It makes a string with all bytes read
// Transformed into chars
// Compression by modeling start
// Replace zeros by 1
// Replace 2 s by 1
// Replace 3 s by 1
// Replace 4 s by 1
// Replace 5 s by 1
// Replace white spaces by 0
// It creates string buffer to perform operations
// This loops erases line breaks
// It transforms the string buffer into a string
// Measures array size
// It makes byte array String n:
// Helping String
// Counter
// Chop the string in 7 bits size pieces
// Transform 1 and 0 sequences in bytes
// Chop the string
// It process the last one
// It ends compression by data modeling
// It starts complemented compressing // (Modeling + entropy-coding)
// It create an external file for data flow
// It wraps the external file into a compress
// Format
// It create header files
// It compresses previously model data
// Close data flow
// Close file
// End of complemented compression
// Reading Compressed file to send it to
// database
// Creates a reading stream for the
// "outCiclopegFile"
// Measure the size of reading file
// It creates a byte array
// file size
// Read all bytes from files and
// Save them in compressedB
// Close the file and ends compressed file
// Reading
// Save in the database

```java
FileNotNullException e1) System.err.println("File not found: "+ file);
```