24/01/2016

Final Report

Steganography

TEAM TELA

VERSION 2

Summary

I. Introduction

For the project week, we had to implement a new feature to our steganography project. This new feature is to add an option to compress the message using Huffman Encoding.

In the specification, our client told us that the Huffman compression uses a binary tree to link each symbol of the message to its binary value in the tree (it depends on the frequency of the symbol in the message).

We had to add this feature to our compression/decompression existing tools. The compression part is in Java and the decompression in C.

The client had different levels of compression (going from L1 to L9). We had to be sure that our code won't have regression, which means that the compression tools doesn't have to impact the way we hide the message or the different patterns. Thank to our way of developing the dissimulation tool (using inheritance, interface paradigm etc), the compression modification won't be related and doesn't depend at all of the LSB dissimulation.

To ensure that our code is viable, every level we reached, we did Unit testing to maintain the 80% tested code (on the dissimulation and revelation tool). Unfortunately, we stopped at the Level 6, which permit to compress the message a way more, even if we totally understood how the L7 works and the strategy was put down on paper.

The specifications were clear enough to start thinking about the solutions. The support team was available every day and a competent teacher too.

In this report, we'll explain for each part (compression and decompression), how we implemented functionalities required, how complicated it was to add them and the encountered difficulties.

You'll find at the end of the report our benchmark results or the differences between the procedural language and the object oriented one in this project.

II. Working Strategies

The team was split in two teams as before, to work in parallelism. We kept the same persons working on Java and C to be more efficiency. We decided to work together on the levels, and accept a gap of 1 level between the dissimulation and revelation teams. If the gap had to be bigger, the dominant group would help the one in difficulty. But this wasn't the case. Every level finished was tested before going to the upper one.

We used the school time to work together, solve problems together and write everything found on paper. These notes were used as a logbook to see the progression of every level, how difficult it was etc. Despite the work done in TELA team, we also worked with other groups to share experiences, advices, and compare results. This helped a lot, while the doubt was to found who between the compressing and decompressing was wrong.

The strategy was to work a lot of the time as a team in school hours, and then arrived at home, start thinking about the rapport, doing some paper reflexion or continue developing. On the morning, we take 5-10 minutes to talk about what are the plans, if there is an unexpected problem etc.

We decided to show, at the demo of the mid-week, the L5 level and planned to finish till the level 8. But, as explained earlier, we couldn't because of a lack of time and a lot of time passed on the L5.

The level 6 compresses the message with the transmission from the dissimulation to the revelation the ASCII value of the characters used in the message, the shape of the tree, and then the message compressed with the number of bits used on the last byte (to remove the padding).

We're going to see now how these new functionalities were added to our code, the particularities etc.

III. Java Part

Our code allows to any user to add any compression method, at any time.

Indeed, for the compression, we chose the polymorphism. Every compression method (in our case, the basic and “more” compression” inherits from the abstract class “Compression). To add a way of compression, the user need to create a simple class which inherit from Compression class. Like this case below :

```
public String run() {
    start = System.currentTimeMillis();
    //All the characters of the message, with their value on the tree
    for (CharCode cc : liste) {
        String binaryCode = cc.getCode();
        charTable.put(cc.getCharacter(), binaryCode);

        String charCode = StringConvert.shiftLeftEightZero(binaryCode);
        String charAscii = String.format("%8s", cc.dictionary()).replace(" ", "0");
        String nbBits = String.format("%8s", Integer.toBinaryString(binaryCode.length())).replace(" ", "0");
        sb.append(charAscii);
        sb.append(nbBits);
        sb.append(charCode);
    }
    codeMessage(); //Create the dictionary thanks to the spec format
    padding(); //Add the padding necessary to the message
    String nbBitsUsedInLastByte = String.format("%8s", Integer.toBinaryString(8 - padding)).replace(" ", "0");
    sb.append(nbBitsUsedInLastByte);
    sb.append(messageCoded.toString());

    end = System.currentTimeMillis();
    return sb.toString();
}
```

The « run » method is an abstract method of the mother class

This is the abstract class Compression

```
public Compression(byte[]message) {
    this.message = message;
    messageCoded = new StringBuilder();
    freqCalc = new FreqCalc(message);
    liste = freqCalc.charToCode();
    charTable = new HashMap<>();
    this.message = message;
    String sizeMessage = String.format("%8s", Integer.toBinaryString((liste.size() - 1))).replace(" ", "0");
    sb = new StringBuilder();
    sb.append(sizeMessage);
    start = 0;
    end = 0;
}
```

```
public abstract String run();
protected void codeMessage() {
    for (int i = 0; i < message.length; i++) {
        String code = charTable.get(message[i]);
        messageCoded.append(code);
    }
}

protected void padding(){
    while (messageCoded.length() %8 != 0) {
        messageCoded.append('0');
        padding++;
    }
}
```

Each compress method has the raw message in a byte array and the dictionary obtained during the compression (thanks to the binary tree).

Compression#codeMessage will encode the the message and the dictionnary with the good format (HashMap for the dictionary), and **Compression#padding** will do the 0-padding to the end of the last byte of the message (to complete the byte).

To add a functionality, the user has, then to add a condition in the class CompressionProcess:

```
public String compress(boolean isMetrics) throws BadCompressionModeException {
    if(mode.equalsIgnoreCase(CompressionType.BASIC.name())) {
        compression = new BasicCompression(messageInBytes);
    }else if(mode.equalsIgnoreCase(CompressionType.MORE.name())){
        compression = new MoreCompression(messageInBytes);
    }else {
        throw new BadCompressionModeException();
    }
    return compression.run();
  }
```

Using the polymorphism was a good way of work because we can add new functionalities easily and the code is flexible.

IV. C part

A) Dictionary

To implement the compression in the revelation, it wasn't so hard to add. Indeed, thanks to **getopt** for the terminal options, it was quite easy to add the new argument for the compression.

We just had to create a new decompression function which, in a switch, will choose the type of decompression to use.

The different problems occurred were the segmentation fault in the data management.

The toughest task was to build the tree for the L6 compression. Because we had a problem for the stop condition when we built the tree, iterating the index of the current character so went out of the bounds.

We created a parse function to create the different elements mandatory for the decompression sequence.

This function will help to found the elements available to create the dictionary and then retrieve the message and its size.

Then, we just have to use the dictionary to retrieve the symbol corresponding of the code read.

You'll find the structure of the dictionary below.

```
struct elem {
  char letter;
  char *alias;
  int size;
};
```

```
struct dictionary {
  Elem* elem;
  int size;
};
```

This structure contains an array of the elements and its size. The structure Elem contains the letter, it's value on the dictionary, and its size (on one, or more bytes). This structure will help to help the memory easily because we know how much we allow memory thanks to the different size.

B) Acceptation Level 5

In the case of an arbitrary message, we create a dictionary with the struct shown before, containing all the compressed characters and their equivalent in binary, contained in a char[].

```
while ((*i) + 2 < data->size && dictionary->size < size) {

   char letter = data->value[(*i)++];
   int nbBits = data->value[(*i)++];
   int nbBytes = nbBits / CHAR_BIT;
   char alias[nbBytes];

   if (nbBits < 0) {

      print_error("Negative size");
      return NULL;
   }

   int j = 0;
   while ((*i) < data->size && j < nbBytes) {
      alias[j++] = data->value[(*i)++];
   }

   int r = nbBits % CHAR_BIT;
   if (r > 0)
      alias[j++] = data->value[(*i)++];

   alias[j] = '\0';

   if ((*i) >= data->size) {
      return NULL;
   }

   dictionary_add(dictionary, letter, alias, nbBits);
}
```

Then we use this dictionary to decode the compressed message :

```
char dictionary_get(Dictionary d, char *alias, int size) {
  for (int i = 0; i < d->size; ++i) {
    if (d->elem[i]->size == size
      && strcmp(d->elem[i]->alias, alias) == 0) {
      return d->elem[i]->letter;
    }
  }
  return -1;
}
```

C) Acceptation Level 6

To decompress a message with the Huffman, we create the tree thanks to the tree shape given during the compression.

```
Node more_tree(Data data, int *i, int *bits, Symbols symbols, int *j) {

  int k = CHAR_BIT - 1 - ((*bits)++ % CHAR_BIT);
  int value = data->value[*i];
  int current_shape = value >> k & 1;

  if (k == 0)
    *i += 1;

  if (current_shape == 0) {
    Node node = new_node(symbols->values[(*j)++], NULL, NULL);
    if (*j == symbols->size && k != 0)
      *i += 1;
    return node;
  }

  Node left = more_tree(data, i, bits, symbols, j);
  Node right = more_tree(data, i, bits, symbols, j);
  Node node = new_node(0, left, right);
  return node;
}
```

Then, we travel inside the tree to add to the dictionary all the symbols of the message

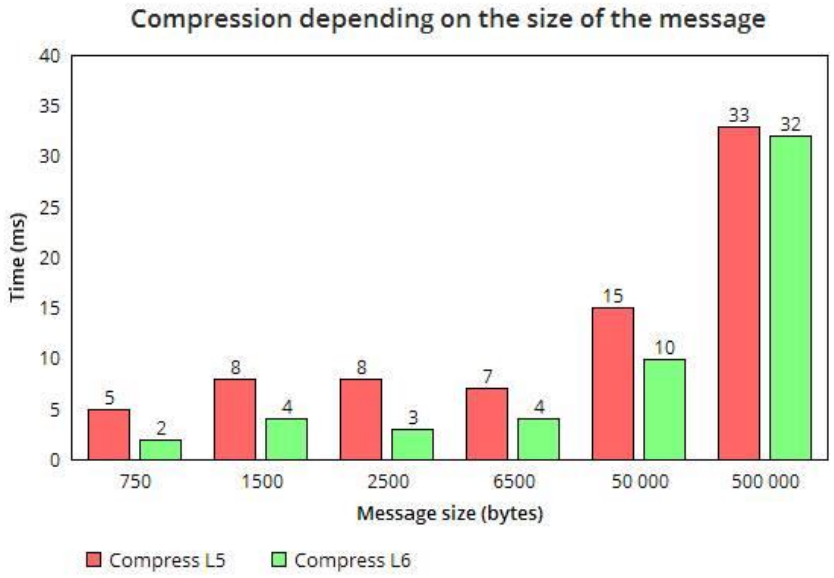
```
void more_dictionary(Node node, Dictionary dictionary, int buffer, int size) {
    if (node->left == NULL && node->right == NULL) {
        buffer = buffer << (INT_BITS - size);
        dictionary_add(dictionary, node->value, int_to_str(buffer, size), size);
        return;
    }

    if (node->left != NULL) {
        more_dictionary(node->left, dictionary, buffer << 1, size + 1);
    }

    if (node->right != NULL) {
        more_dictionary(node->right, dictionary, (buffer << 1) | 1, size + 1);
    }
}
```

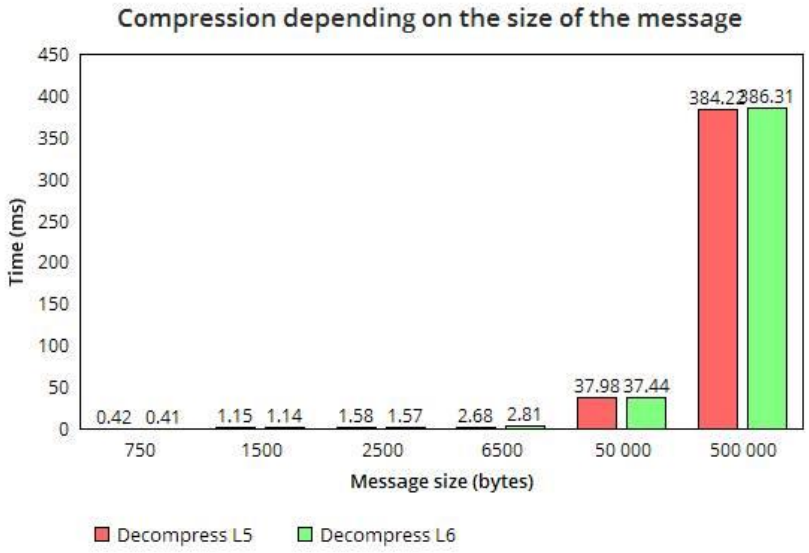
D) Unit test

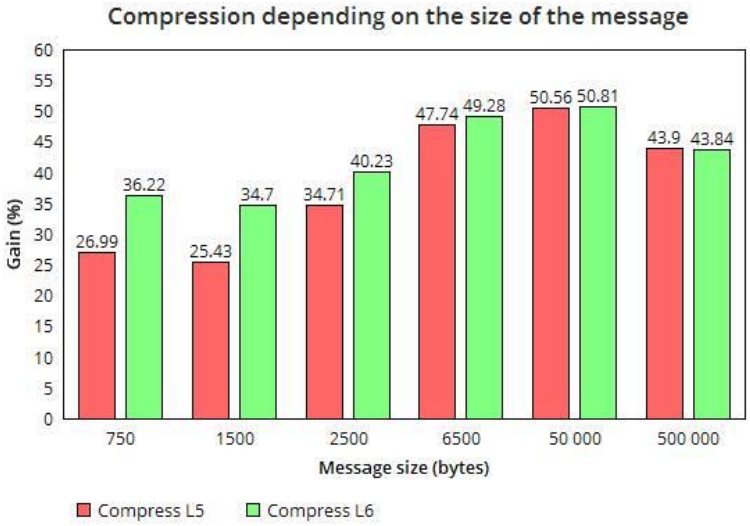
To ensure a viability of our code, we tested every compression method in separate files, and the tree creation, thanks to CTest, already used in our previous delivery.

V. Benchmarks

We did some benchmarks to test our code and show how the compress time will increase. So we varied different parameters of the message to compress. See below the different graphs with their results.







These results show different things. First of all, the more the message is big, the more the time of compression will increase. The type of compression has a role also because the L6 is more efficient than the L5.

The size of the message has a place in the execution time when its size is ~100 000 bytes. For the same message, we note that the revelation time is bigger than the dissimulation one. Because decompressing has to create the tree with the shape, or construct the dictionary while parsing the binary content (depend of the level of compression).

The gain for the compressed message is better when the size of the message is approximately 50 000 bytes. For example, with a message of 1 character, the gain will be negative because the character will be coded in 8 bits and the dictionary for this symbol will be bigger. That's why this benchmark helps knowing what's the limit of our algorithm.

VI. Conclusion

In conclusion, this project helped us a lot, finding defaults and limits in our program and the opportunity of the Java language (ImageRaster little bit slow comparing of working on the image bytes directly). We also knew how to use the object paradigm to keep our code flexible. Thanks to that, we added the compression functionality without harming our code easily.

We also knew how compression data works, and we implemented it, so this project increased our skill and knowledge, which in not negligible.

The cohesion in our team helped a lot, because we were motivated about completing the more level as possible.