TOOLS FOR VISUALIZING TEXT COMPRESSION ALGORITHMS

Sami Khuri
San José State University
Dept. of Mathematics and Computer Science
San José, CA 95192-0103, USA
khuri@cs.sjsu.edu

Hsiu-Chin Hsu
San José State University
Dept. of Mathematics and Computer Science
San José, CA 95192-0103, USA
joy@cs.sjsu.edu

Keywords: visualization, text compression, algorithms, adaptive Huffman coding, dictionary encoding

ABSTRACT

In this paper, we describe visualization tools that were developed to assist students in learning data compression algorithms. The two packages, LZ and AHuffman, animate the three LZ-based algorithms, LZ77, LZ78, LZW, and the adaptive Huffman algorithm. These packages can be used in CS1/CS2, data structures and algorithms courses, or in data compression electives. We also highlight the merits of the packages and the benefits to the students derived from our visualization tools.

1. INTRODUCTION

All over the world, people are developing interactive educational tools. These tools are being used in classroom demonstrations, hands-on laboratories, self-directed work outside of class, and distance learning [5]. The World Wide Web has numerous repositories of animation and visualization packages of traditional algorithms encountered in CS1, CS2, and data structures and algorithms courses. These include sorting, searches, and traversals of trees and graph algorithms. Recently, many computer science instructors of introductory courses have chosen to cover topics that are usually reserved for more advanced courses. One of these topics is data compression. Data compression is popular because of many reasons, two of which are found in [10]: (1) People like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression seems useful because it delays this inevitability. (2) People hate to wait a long time for data transfers. When sitting at the computer, waiting for a web page to come in or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait. The popularity of data compression has lead to the development of electives and honors courses in data compression [7] and to the use of some data compression algorithms in introductory computer science courses. For example, Jiménez et al. [6] present programming projects that have been successfully used in CS1/CS2. These projects include graphics management and human computing interfaces, image processing and data compression. Astrachan [1] uses the implementation of the Huffman coding as an assignment in CS2.

The main purpose of this paper is to present visualization tools that were developed to assist students in learning text compression algorithms. There are two approaches to text compression: statistical and dictionary. In statistical methods, a probability is estimated for each character, and a code is chosen based on the probability. Dictionary coding achieves compression by replacing groups of consecutive characters with indexes into some dictionary. The dictionary is a list of phrases that are expected to occur frequently. Indexes are chosen so that on average they take less space than the phrase they encode, thereby achieving compression.

Our first package, LZ, visualizes three dictionary-based text compression algorithms: LZ77, LZ78, and LZW. The second visualization package, AHuffman, can be used in teaching one of the statistical methods of text compression, namely, the adaptive Huffman coding. The input text being compressed, the building of the dictionary and the resulting compressed text are all simultaneously displayed by running each algorithm, thus enabling the user to see all the intricate details that occur during the compression procedures. LZ and AHuffman take advantage of the Swing components in Java. Both packages are highly interactive and user-friendly and can be used in courses that emphasize various aspects of text compression algorithms.

In the next section, we explain the guidelines we followed in designing and implementing our packages. In the third and fourth sections, we present LZ and AHuffman. Examples of visualizations of the three different LZ-based algorithms, and the adaptive Huffman algorithm are introduced, highlighting the benefits, power and efficiency of our two packages. The interested reader will find a more thorough explanation of the algorithms in [11] and [10], which are the textbooks we have used in teaching the course in data compression. We conclude the paper with some closing remarks and possible extensions of this work.

2. DESIGNING LZ AND AHUFFMAN

Algorithm visualizations depict the execution of an algorithm as a discrete or continuous sequence of graphical images, the viewing of which is controlled by the user [9]. It has been an active area of research since the early 1990s. It has been used with success in teaching computer science courses, designing and analyzing algorithms, producing technical drawings, tuning performance, and documenting programs. Recently, researchers have started to pay particular attention to issues such as the user interface, navigation structures, and platform-dependency of the educational visualizations.

In designing LZ and AHuffman we followed the guidelines for creating effective visualization tools given in [3], [2], and [8].

- The main goal of visualization is to help instruct. The goal of our packages is to help students achieve a better understanding of the algorithms including the basic steps performed during the execution. Both packages include help files that can be used by the students.
- Visualization control should be simple. For example, since we use only a few buttons and menu choices, the students do not need to spend too much time learning how to use the packages.
- Multiple views of the algorithm should be used. It is our belief that the compression techniques can be easily understood if we display, in separate windows (on the same screen), the text being compressed, the corresponding actions that take place behind the scene, including the construction of the dictionary, and the compression output.
- Strive to draw students' attention to the critical area of the visualization. In our packages, we use different contrasting colors to differentiate between characters to be encoded and those that have already been processed. We temporarily highlight in red the latest entry in the dictionary. We also show changes in the state of an algorithm's data structures by changing their graphical representations on the screen. For example, in the AHuffman package, a leaf is represented by a square, which changes to a circle when the update procedure converts the leaf into an inner node.
- Use a text window to make sure students understand the visualization. In both our packages, we supplement our visualizations with the description of the steps of the algorithms in a separate window entitled "Instructions".
- Provide default visualizations. In our packages, students can start default visualization without any complicated selections. All our visualizations have default input strings on which the chosen algorithm will operate.
- The object-oriented approach was used in the development and implementation of our packages to support their extensibility and reusability in future versions.

3. THE PACKAGE LZ

The first package, LZ, has been designed for visualizing the dictionary-based, data compression algorithms: LZ77, LZ78 and LZW. These techniques consist in identifying and using structures that exist in data. LZ77, LZ78, and one of their popular variants, LZW, are used in many applications, such as the UNIX "compress" and "gzip" utilities and CompuServe's Graphic Interchange Format (GIF). Before introducing three examples that will illustrate the type of visualizations produced by our package, we describe the control components of LZ.

3.1 The Control Components of LZ

As can be seen in Figure 1, Figure 2 and Figure 3, the LZ package has the following control components:

- **File**: The user can close the application by choosing "Exit" from this menu.
- **Algorithms**: The user can switch between the three LZ-based algorithms implemented in this package from this menu.
- **Help**: The user can get instructions for running the package or go over a short tutorial to get a better understanding of the algorithm.
- **Step**: This button enables the user to step through the algorithm. At each step, the screen shows the current state of the algorithm in execution.
- **Run**: This button allows the user to view the behavior of the algorithm by simply setting the process in motion. The results of text compression are displayed immediately in the output window.
- **Reset**: This button allows the user to reset all the variables.
- **Encoding/Decoding**: The user can switch between the encoding mode and decoding mode by selecting the corresponding buttons.
- **Select Demo Inputs**: The user can select one of the predefined input strings from a pop-up window.
- **Input Your String**: By selecting this button, users can type their own input string in the text field.
- **Set Buffer Size**: This option exists for LZ77 only. Users can change the size of the search and look-ahead buffers.

3.2 An Example of the LZ77 Visualization

In the LZ77 approach, the dictionary is simply a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window. The window consists of two parts: a search buffer that contains a portion of the recently encoded sequence, and a look-ahead buffer that contains the next portion of the sequence to be encoded.

To encode the sequence in the look-ahead buffer, the encoder moves a search pointer, from right to left through the search buffer until it encounters a match to the first symbol in the look-ahead buffer. The distance between the search pointer and the first character in the look-ahead buffer is called the offset. The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the length of the match. The encoder searches the search buffer for the longest match. Once the longest match has been found, the encoder encodes it with a triplet <O, L, C>, where O is the offset, L is the length of the match, and C is the codeword corresponding to the character in the look-ahead buffer that follows the match.

The LZ77 approach implicitly assumes that similar patterns will occur close together. It makes use of this structure by using the recent past of the sequence as the dictionary for encoding. Applications using variations of LZ77 include GNU zip, PKZIP, LHarc, and PNG.

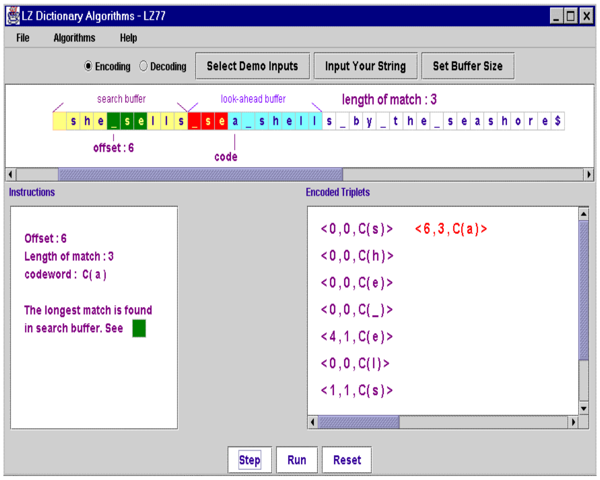

Figure 1. Snapshot of the LZ77 visualization

A sample run of the LZ77 algorithm with a pre-defined string is shown in Figure 1. The input string, as well as the search and look-ahead buffers are shown in the input panel. The search buffer is painted in yellow and the look-ahead buffer in cyan. The panel has a horizontal scrollbar that automatically adjusts the view of the input string. To let users see the process better, we use the green color to represent the longest match found in the search buffer and a red rectangle to indicate the first character of the look-ahead buffer. The display also has a text panel labeled "Instructions" which contains the explanation of the current step of the algorithm. The encoded triplets are displayed in the third panel. The current encoded triplet is shown in red.

For example, suppose the sequence to be encoded is "she sells sea shells by the seashore", and the length of the sliding window is 20. The size of the look-ahead buffer is 10. We use $ to denote the end of input and _ to denote the space. We also use C (x) to denote the encoding of character x. In Figure 1, we see the state of the algorithm visualization after a few steps. We have the following characters: "_sea_shell" in the look-ahead buffer, and as we can see, there is a match of "_se" with offset 6 in the search buffer. So the encoder encodes it with the triplet <6,3,C (a)> (in red), where C (a) is the codeword of character "a".

3.3 An Example of the LZ78 Visualization

The LZ78 algorithm does not rely on the search buffer and keeps an explicit dictionary. It creates a dictionary of the phrases that occur in the input data. When the encoder encounters a phrase already present in the dictionary, the index number of the phrase in the dictionary is used as code. The encoder outputs tokens of the form <pointer to string P, code of symbol C>, where: P denotes a string from the dictionary and C denotes the input character currently being processed. The token then becomes the newest entry in the dictionary. Thus, each new entry into the dictionary is a new character concatenated with an existing dictionary entry.

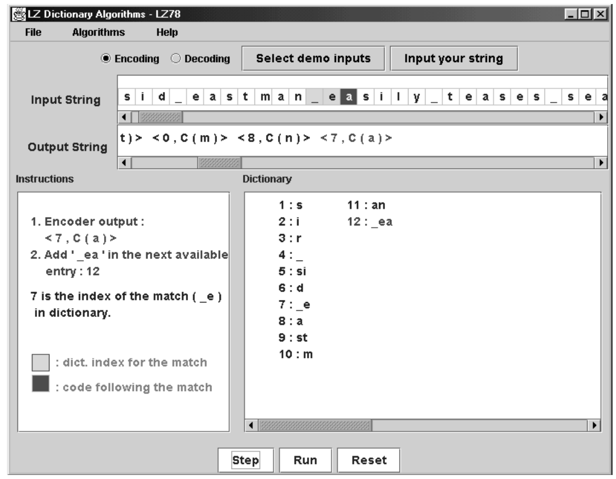

Figure 2. Snapshot of the LZ78 visualization

In the example in Figure 2, we would like to encode "sir sid eastman easily teases sea sick seals". We use $ to denote the end of input, _ to denote the space and C (x) to denote the encoding of character x. The encoder has just finished with "an" from "eastman" and is about to encode the white space "_" that is between "eastman" and "easily". The entry in the dictionary with index 7 is the pattern "_e". The encoder outputs <7,C (a)>, and adds the pattern "_ea" at the next available location in the dictionary which happens to be the location with index 12.

The input and output strings in the animation are displayed in panels with horizontal scrollbars which are adjusted automatically as the algorithm executes. The rest of the screen is split into two parts. On the left is the instruction panel that explains the current steps of the algorithm. The dictionary is in the right half of the panel.

3.4 An Example of the LZW Visualization

The LZW, a variant of LZ78, is one of the most widely used compression algorithms. This algorithm is able to work on almost any kind of data and eliminates the necessity of encoding the second element of the token <pointer to string P, code of symbol C>. The encoder only sends the index in the dictionary. The LZW algorithm starts by initializing the dictionary to all the letters of the source alphabet. In our visualization of LZW, we initialize the dictionary with 38 characters: 0 - 9, a - z, the symbol _ for the white space, and the symbol $ for the end of string. These characters are in locations 0 through 37 in the dictionary, as can be seen in Figure 3. The input to the encoder is accumulated in a pattern P as long as P is contained in the dictionary. If the addition of the next letter, C, results in a pattern P+C which is not in the dictionary, then the index of P is transmitted to the receiver, the pattern P+C is added to the dictionary, and we start another pattern with the character C.

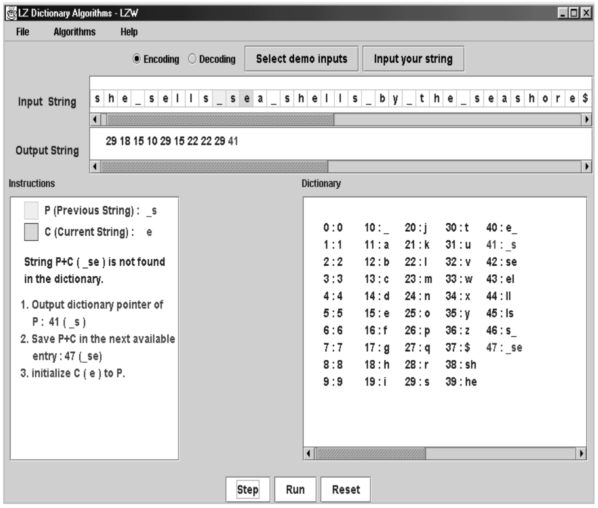

Figure 3. Snapshot of the visualization of LZW

For example, suppose that the sequence to be encoded is "she sells sea shells by the seashore". Figure 3 depicts the state of the algorithm after the encoder has just finished with "s_" from "sells_" and is about to start a new pattern with the white space (that is between "sells" and "sea") as first character. The pattern "_s" is already in the dictionary at index 41 (highlighted in red). So the encoder outputs 41, and adds the pattern "_se" at the next available location in the dictionary which happens to be the location with index 47 (highlighted in blue). The layout we used for visualizing LZW is identical to LZ78's layout described in Section 3.3.

4. THE PACKAGE AHUFFMAN

The second tool we designed, implemented and used in our teaching, is a commonly used method for data compression. The tool can be used to introduce statistical methods of text compression. Statistical methods use variable-size codes, with shorter codes assigned to symbols or groups of symbols that appear more often in the data (have a higher probability of occurrence) [10]. The (regular) Huffman coding, which assigns longer codewords to less probable source symbols, requires the knowledge of the probabilities of the source symbols. If the probability distribution of the source symbols is not known, then we either have:

- to perform a 2-pass procedure, where the statistics are collected in the first pass, and the source is encoded by the (regular) Huffman procedure in the second pass, or
- to use an adaptive Huffman coding procedure, where the Huffman code is constructed based on the statistics of the symbols already encountered.

With adaptive Huffman coding, the binary tree has 2 more parameters (than with the regular Huffman coding). Each node has a weight and a number. The weight of a leaf represents the number of times the symbol has been encountered so far. The weight of an internal node is the sum of the weights of its offspring. Each node is numbered. The numbering of the nodes is used to ensure that the tree will always have the Huffman sibling property.

In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the beginning of the transmission. The tree at both ends is a one-leaf Huffman tree labeled NYT (not yet transmitted). It grows via an update procedure, and is continuously updated with the transmission of every single source symbol. The update procedure consists in adding symbols to the tree, and consequently updating the weights and numbers of the nodes. The purpose of the update procedure is to preserve the sibling property while updating the tree to reflect the latest estimates of the frequency of occurrence of the symbols. The updating procedure is the same for the transmitter as well as for the receiver, so that the whole process is always synchronized.

Before transmission, a fixed encoding (also known as uncompressed code) of the symbols is agreed upon between the transmitter and receiver. The uncompressed code of symbols can be their ASCII code or some other code. It often consists in assigning codes of two different sizes.

To transmit character "a":

- Its current code, as given by the adaptive tree, is transmitted, if it has been previously encountered;
- If it has not been encountered earlier, then the code for NYT (as given by the adaptive tree) is sent first, followed by the fixed (uncompressed) code of "a".

In other words, the receiver knows whether the transmitted symbol is in its uncompressed form (has not been transmitted before) or not, since each uncompressed symbol is preceded by the NYT code which acts as an escape code. So when the receiver encounters it, it knows that what follows is the code of a symbol that is sent for the first time.

In our visualization, we use the 26 lower case letters of the English alphabet and the white space as the input characters. Given an input string, AHuffman will display the process of both encoding and decoding.

As can be seen in Figure 4, the package has four separate windows: the main window, the instructions' window, and two windows that display the tree in different states.

The main window in Figure 4, entitled "Adaptive Huffman Coding", contains three elements:

- **Input String**: a panel with a horizontal scrollbar containing the text to be compressed, "mississippi". The current character being processed (the second "s" in Figure 4) is highlighted in blue.
- **Output String**: a panel with a horizontal scrollbar showing the compressed string. The first part: 011000010000010010, is obtained by compressing "mis". The second part, shown in red: 101, is the code obtained for the second "s" in "mississippi" (as can be seen by the adaptive Huffman tree in the window entitled "Tree in Previous State". A right branch is labeled with "1" and a left branch with "0").

- **Tree**: The major part of the main window is dedicated to the current adaptive Huffman tree.

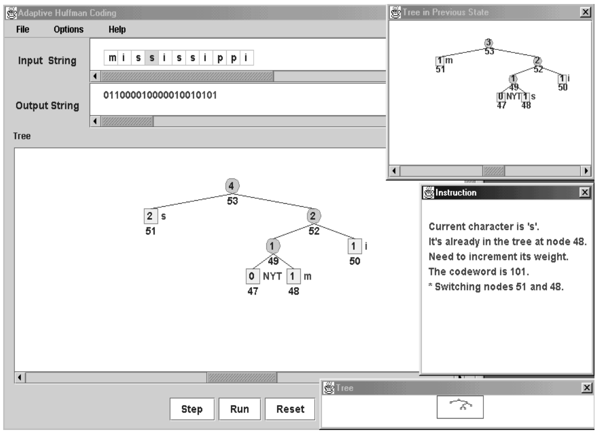

Figure 4. Snapshot of the AHuffman visualization

To keep our visualization packages uniform, we use the same main controls as those of the LZ package. We allow the user to choose between stepping through the algorithm and viewing the current state of all components, running the program until the compression is completed, or resetting all the components to their initial default values. To save the display space, we use a menu **Options** in the main window to let users either select a demo input string, type in their own input string, perform encoding or decoding.

The second window in Figure 4, entitled "Tree in Previous State", contains the tree constructed in the previous step of the algorithm. It only shows up when the user runs the algorithm in the "Step" mode. We found this window very helpful since users can compare the previous tree with the current tree (in the main window). The third window in Figure 4, entitled "Instruction", gives a description of all the steps that are taken when a single character is processed.

The fourth window in Figure 4, entitled "Tree", is a display tool for users to control the view port of the tree in the main window. This fourth window was created for the sole purpose of better viewing the tree of the main window. With many input symbols, the adaptive tree can grow and become very large to the point of not being completely visible in the main window. The "Tree" window always displays the whole tree but in a smaller scale. The part of the tree inside the red rectangle, is the same as its enlargement in the main window. When the user moves the rectangle in the small window, the view port of the original tree in the main window will be adjusted immediately. In other words, the user can adjust the view port of the main window by using the fourth window instead of having to adjust the scrollbars.

5. CONCLUSION

In this work, we introduce two packages, LZ and AHuffman, that can be used as a supplement to a data compression elective, data structures and algorithms or CS1/CS2 courses. The user can grasp the intricate details of the three LZ-based data compression algorithms and the adaptive Huffman algorithm by using these tools. The user can see the simultaneous animation of the input text being compressed, the building of the dictionary, the resulting compressed output, and read the help files which give further explanations of the algorithm being run. The ease of use of our packages provides students with a powerful visual aid to achieving a better conceptual understanding of these algorithms. Both tools, LZ and AHuffman, are available at http://www.mathcs.sjsu.edu/faculty/khuri/papers.html

Both packages are valuable visualization tools for classroom lectures. Instead of tracing the algorithms by hand, or by overlaying transparencies, one can step through the programs, pause, consult the help files, and answer students' questions. We also use the packages in an open lab environment, where students gain a better understanding of the workings of the algorithms by practicing at their own pace. They can experiment and input their own data, set their own parameters and compare the results.

We also believe that the designers of educational software will find our display tool for controlling the view port of the main window very helpful. Using the display tool is more advantageous than directly manipulating the large figure in the main window through the use of scrollbars.

REFERENCES

[1] Astrachan, O., Huffman coding: a nifty CS2 assignment, *Proceedings of SIGCSE'99*, pp. 371-372, 1999.

[2] Bergin, J., Brodlie, K., Goldweber, M., Jiménez-Peris, R., Khuri, S., Patiño-Martínez, M., McNally, M., Naps, T., Rodger S., and Wilson, J., An Overview of Visualization: its Use and Design. *Proceedings of ITiCSE'96*, pp. 192-200, 1996.

[3] Brown, M. and Hershberger, Fundamental techniques for algorithm displays. In *Software Visualization*, MIT Press, pp. 81-101, 1998.

[5] Gould, D., Simpson, R., and van Dam, A., Granularity in the design of interactive illustrations, *Proceedings of SIGCSE'99,* pp. 306-310, 1999.

[6] Jiménez-Peris, R., Khuri S., and Patiño-Martínez, M., Adding breadth to CS1 and CS2 courses through visual and interactive programming projects, *Proceedings of SIGCSE'99*, pp. 252-256, 1999.

[7] Lelewer, D. and Ng, C., An honors course in data compression, *Proceedings of SIGCSE'91*, pp. 146-150, 1991.

[8] Miller B.P. What to draw? When to draw? An Essay on Parallel Program Visualization. *Journal of Parallel and Distributed Computing*. pp. 1993.

[9] Naps, T. and Chan, E., Using visualization to teach parallel algorithms, *Proceedings of SIGCSE'99*, pp. 232-236, 1999.

[10] Salomon, D. *Data Compression*. Springer Verlag, 1998.

[11] Sayood, K. *Introduction to Data Compression*, Morgan Kaufmann Publishers, 1996.