EXPLORING UNIQUE COMPUTING APPLICATIONS ENRICHED BY SCIENTIFIC PHENOMENA AND MATHEMATICAL CONCEPTS

Vladimir V. Riabov
Rivier University
420 S. Main Street ▪ Nashua, NH 03060-5086, USA
E-mail: vriabov@rivier.edu

The role of phenomena found in linguistics, physiology, and psychology is analyzed in various computing applications, including cryptology, image processing, and software engineering. Simple mathematical concepts (statistics, probability, graphs, and approximation formulas) contribute to the successful merge of sciences. Java applets and software tools are developed to demonstrate these applications.

The objective of this paper is to review mathematical aspects of various computing applications that involve unique phenomena acquired from other sciences (e.g., linguistics, physiology, and psychology), as well as to explore practical applications with computing tools. The algorithms and codes were examined by undergraduate and graduate students in *Discrete Mathematics*, *Algorithms*, *Computer Science Fundamentals*, *Computer Networks*, *Computer Security*, *Software Engineering*, and other courses taught by the author.

1. Deciphering with the Linguistic Letter Frequency Analysis

Modern methods of deciphering use traditional mathematical concepts from the theory of numbers, Galois Fields, and probability. For several decades the substitution cipher approach with the Letter Frequency Analysis [1, 2, 3] has been also effectively used. To explore this approach, which is based on applying the linguistic properties of an original plaintext, students make some assumptions about the plaintext:

- That the plaintext consists of characters, not some kind of binary code.
- That it is written in some natural language with known linguistic properties (e.g., English).
- That we know the frequency of letters in a typical piece of text in that language.
- That the plaintext is typical of normal English text, and so we expect the same frequencies of letters (approximately, within statistical fluctuations).

As long as we know that there is a 1-to-1, unique mapping from plaintext to ciphertext (and, therefore, from ciphertext to plaintext), we can employ our knowledge of those letter frequencies to crack a substitution cipher. It is important to note that we need a large enough piece of text to give us some expectation that we have a large statistical sample. The longer the message, the better statistical sample we are likely to have.

Known letter frequencies in typical English text may be found on the web [3]. A typical representation of the letter frequencies in traditional English (E, T, A, O, I, N, S …) is shown on the bar chart (see Fig. 1, right). The Java tool [4] allows a student to view the letter frequencies for the ciphertext being examined (Fig. 1, center). Students may display letter frequencies in alphabetic order, or in order by frequency. If one of the characters has a 20% then the language

may be German since it has a very high percentage of E. Italian has 3 letters with a frequency greater than 10% and 9 characters are less than 1% [5].

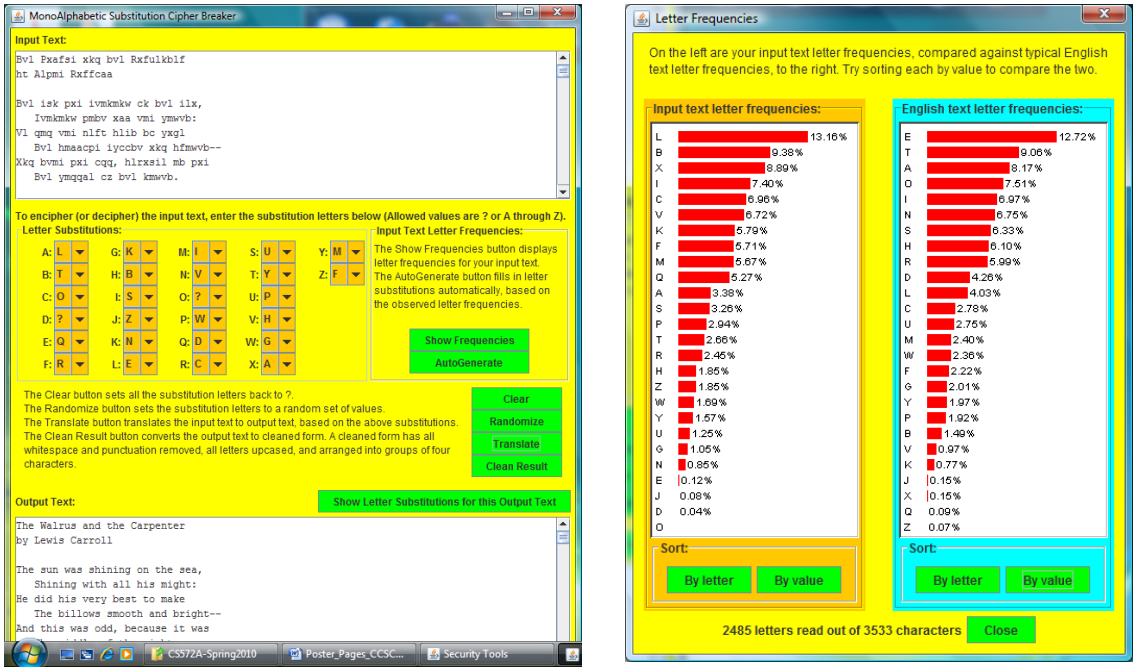


Figure 1. Deciphering the structured ciphertext. MonoAlphabetic Cipher Breaker (*left*) and letter frequencies in typical English (*right*).

The linguistic analysis [5] also shows that common pairs in English are consonants TH and vowels EA. Others are OF, TO, IN, IT, IS, BE, AS, AT, SO, WE, HE, BY, OR, ON, DO, IF, ME, MY, UP. Common pairs of repeated letters are SS, EE, TT, FF, LL, MM and OO. Common triplets of text are THE, EST, FOR, AND, HIS, ENT or THA.

The MonoAlphabetic Cipher Breaker Java applet [4] (see Fig. 1, left) was used for deciphering the structured ciphertext (620 words; 2,485 letters out of 3,533 characters), where the original word spacing, punctuation, and style have been retained. Travis Brant, a CS graduate student, wrote in the assignment report: "…The solution was reached by only using the statistical distribution for the first three characters. Once those were in place, the text was long enough that searching for uncommon words with only one missing character was easily done. Once this practice was put into place, decoding the bulk of the message was reduced to an iterative process of searching for the next nearly-complete word. Decoding this message took about fifteen minutes."

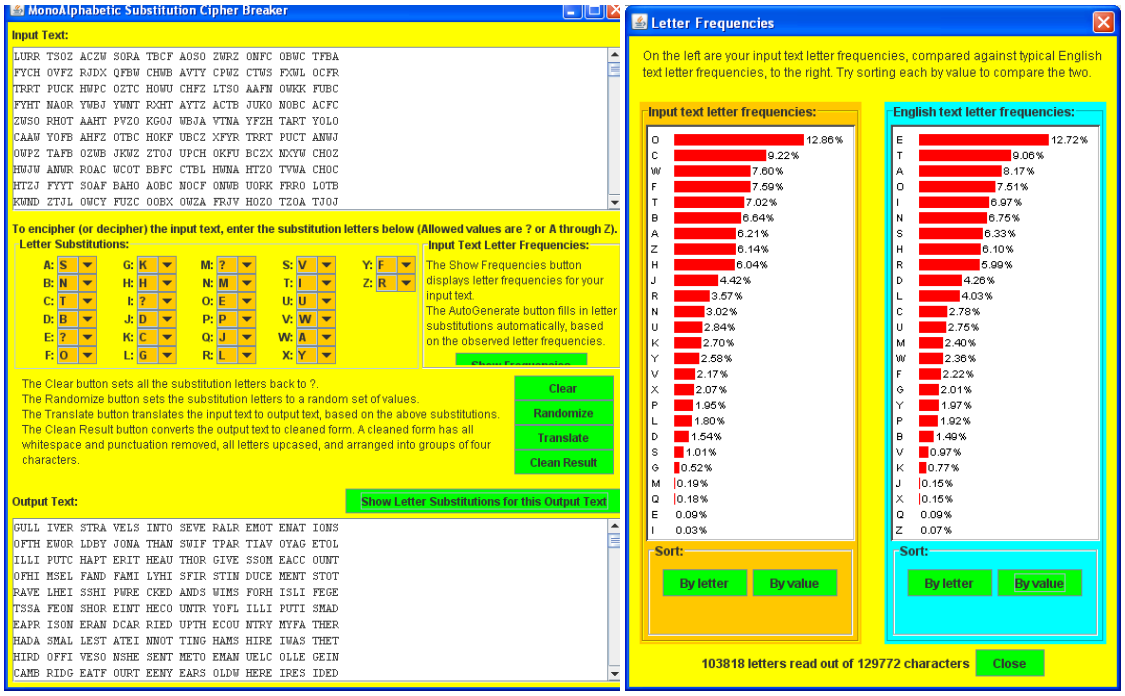


Figure 2. Deciphering the ciphertext organized in groups of four letters. MonoAlphabetic Cipher Breaker (*left*) and letter frequencies in typical English (*right*).

The second ciphertext (25,955 words; 103,818 letters out of 129,772 characters) was organized in groups of four letters and word spacing and punctuation have been removed. The absence of the content clues (word spacing and punctuation) makes it more difficult to decipher the ciphertext, while the larger sample allows greater use of letter frequency analysis (see Fig. 2). Travis Brant reported:

> "… Deciphering this example was much more difficult than anticipated. The lack of preserved whitespace and punctuation made searching for possible word separation difficult. Once statistical analysis was performed on the text, there was little exposed that seemed correct. The only word that stuck out was the end of the first line, "ENAT IONS". I figured that this was as good of a start as any. Next, I caught some word pairings on the first three sections of line thirty, "OCIM ONTI NUED". At this point, I swapped M for C to create "O?IC ONTI NUED". At this point swapping Z for R brought more words forth. A large breakthrough was reached when L was swapped for G, spelling "GULL IVER" as the first words. From here, I looked up a sample of the text from this story "Gulliver's Travels" and saw that the message was the text from Jonathan Swift's work. I used the text from the book to identify and fix the remaining glitches in the decoded text, and was finished. The message was indeed the story of "Gulliver's Travels" by Jonathan Swift. This problem took about one-and-a-half hour of analysis to decipher".

To reduce the time of deciphering this unstructured ciphertext, one student even wrote the customized UNIX scripts and a standard UNIX dictionary to help with the mechanics of the solution [1].

2. Reduction of the Programming Code Complexity

In the second case study, the structured testing methodology [6, 7] and graph-based metrics (cyclomatic complexity, essential complexity, module design complexity, system design complexity, and system integration complexity) have been reviewed by students and applied for studying the C-code complexity and estimating the number of possible errors and required unit and integration tests for the Carrier Networks Support system [8]. Comparing different code releases, it is found that the reduction of the code complexity leads to significant reduction of errors and maintainability efforts.

For each module (a function or subroutine with a single entry point and a single exit point), an annotated source listing and flowgraph is generated as shown in Fig. 3. The flowgraph is an architectural diagram of a software module's logic.

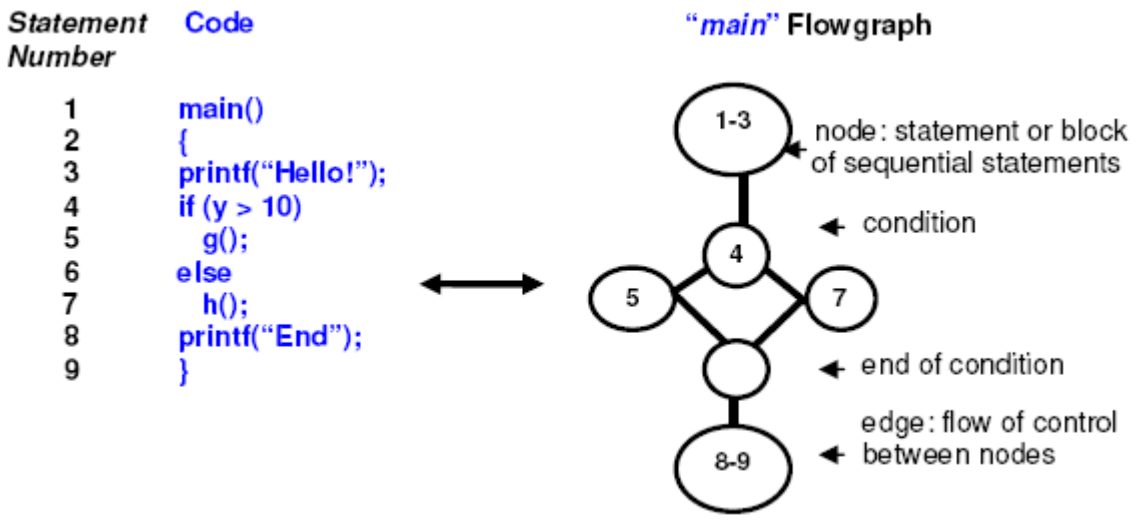


Figure 3: The annotated source listing and the related flowgraph.

Cyclomatic complexity, *v*, is a measure of the complexity of a module's decision structure [3, 4]. It is the number of linearly independent paths and, therefore, the minimum number of paths that should be tested to reasonably guard against errors. A high cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain. In addition, empirical studies have established a correlation between high cyclomatic complexity and error-prone software [1]. The results of experiments by Miller [5] suggest that modules approach zero defects when the McCabe's Cyclomatic Complexity is within 7 ± 2. Therefore, the threshold of *v*-metric is chosen as 10. Miller's psychological experiments have led to the reduction of the programming-code complexity.

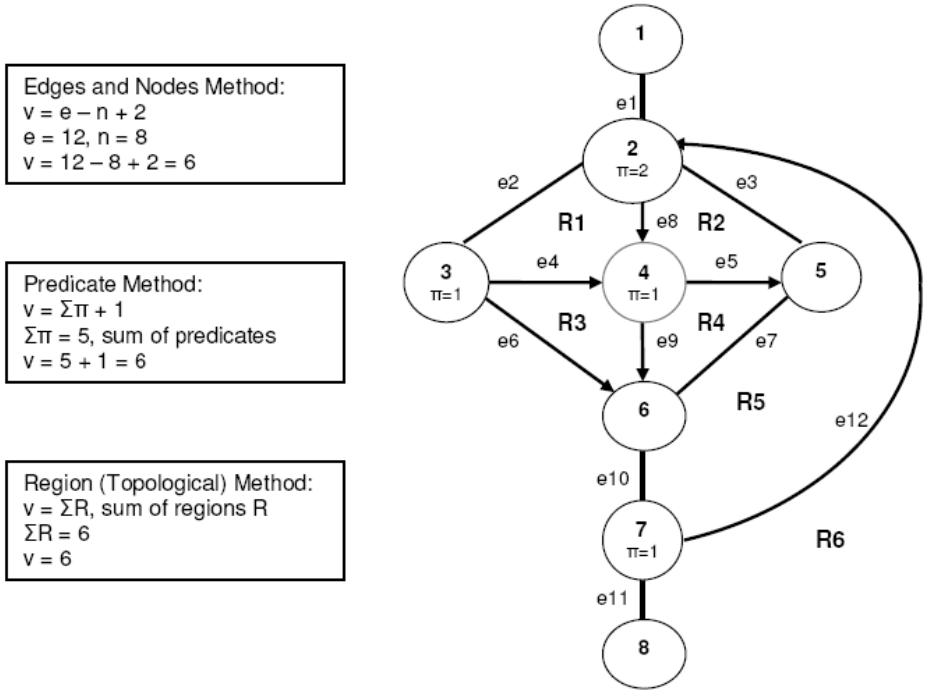


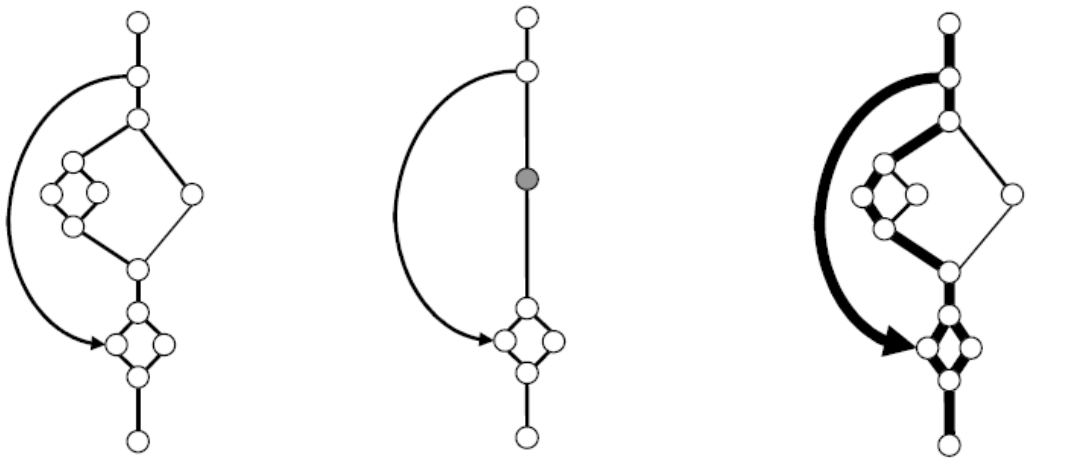
Figure 4: Three methods of evaluating the cyclomatic complexity of the graph.

A node is the smallest unit of code in a program. Edges on a flowgraph represent the transfer of control from one node to another [3]. Given a module whose flowgraph has *e* edges and *n* nodes, its cyclomatic complexity is $v = e - n + 2$. This complexity parameter equals the number of topologically independent regions of the graph and correlates with the total number of logical predicates in the module [3, 4] (see Fig. 4).

Branching out of a loop. | Branching into a loop. | Branching into a decision. | Branching out of a decision.

Figure 5: Examples of the unstructured logical constructs.

Essential complexity, *ev*, is a measure of unstructuredness, the degree to which a module contains unstructured constructs [3, 6] (see Fig. 5), which decrease the quality of the code and increase the effort required to maintain the code and break it into separate modules. When a number of unstructured constructs is high (essential complexity is high), modularization and maintenance is difficult. In fact, during maintenance, fixing a bug in one section often introduces an error elsewhere in the code [1, 4].



Original flowgraph, v = 5. Reduced flowgraph, v = 3, ev = 3. Superimposed flowgraph, v = 5, ev = 3.

Figure 6: Evaluation of the essential complexity of the flowgraph.

Essential complexity is calculated by removing all structured constructs from a module's flowgraph and then measuring the cyclomatic complexity of the reduced flowgraph [3, 4] shown in Fig. 6. The reduced flowgraph gives you a clear view of unstructured code. When essential complexity is 1, the module is fully structured. When essential complexity is greater than 1, but less than the cyclomatic complexity, the module is partly structured. When essential complexity equals cyclomatic complexity, the module is completely unstructured. The unstructured modules should be recommended for redesigning.

3. Estimating the Number of Projected Errors in the Programming Code

The third case study is designed to introduce Halstead's metrics [10] and explore the physiological phenomenon of human-brain restrictions [11] in estimating the number of projected errors in the programming code. The McCabe IQ™ tool has been used in producing Halstead metrics [10] for codes written in selected programming languages. Supported by numerous industry studies [7], the B-metric of Halstead represents the estimated number of errors in the program.

The concept of event-discriminations was introduced by John M. Stroud, a psychologist, in "The Fine Structure of Psychological Time" [11], where he defined a "moment" as "the time required by the human brain to perform the most elementary discrimination" [10, p. 48]. He reported that, for all waking, conscious time, these "moments" occurred at a rate of "from five to twenty or a little less" per second, which is known nowadays as the Stroud number, S [10]. His study was based on the analysis of the internal processing rate of the brain that correlates with the range of the number of frames per second which a motion picture should have to appear as a continuous picture rather than as single frames. Halstead [10] applied this concept to the evaluation of discriminations in input/output program events. Finally, the Stroud number, S, was used by Halstead in estimating the programming time, $\check{T} = E/S$, where E is the number of elementary mental discriminations required for the program implementation.

The linguistic complexity of various languages (English, PL/I, Algol, Assembly, and others) was studied by Halstead [10, pp. 62-70] for estimating the language level parameter (λ = 2.16) that was used in calculating the mean number of elementary discriminations between potential errors in programming (E_0 = 3000), and, finally, the number of "delivered" bugs, $B = E^{2/3}/E_0$.

4. Exploring How Pictures Resolved in Human Vision Are Represented on a Computer

In the last case study, we explore how pictures resolved in human vision are represented on a computer. In nature, visible light is a continuous spectrum with wavelengths between 370 and 730 nanometers. But the human perception of light is limited by how human eye color sensors work [12]. Perception of color begins with specialized retinal cells containing pigments with different spectral sensitivities, known as cone cells. In humans, there are three types of cones sensitive to three different spectra, resulting in trichromatic color vision. The cones are conventionally labeled according to the ordering of the wavelengths of the peaks of their spectral sensitivities: short (S), medium (M), and long (L) cone types. While the L cones have been referred to simply as red (R, 620-740 nm) receptors, microspectrophotometry has shown that their peak sensitivity is in the greenish-yellow region of the spectrum (see Table 1 and Fig. 7, below). Similarly, the S- and M-cones do not directly correspond to blue (B, 450-495 nm) and green (G, 495-570 nm), although they are often described as such. The RGB color model, therefore, is a convenient means for representing color, but is not directly based on the types of cones in the human eye.

Table 1. Characteristics of the cone cells in the human eye [12]

Cone type	Name	Range	Peak wavelength
S	β	400–500 nm	420–440 nm
M	γ	450–630 nm	534–555 nm
L	ρ	500–700 nm	564–580 nm

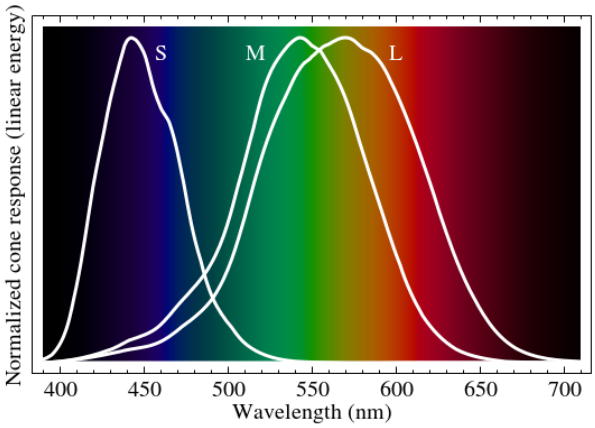


Figure 7: Normalized response spectra of human cones, to monochromatic spectral stimuli, with wavelength given in nanometers [12].

Many species can see light with frequencies outside the human "visible spectrum". Bees and many other insects can detect ultraviolet light, which helps them to find nectar in flowers. Birds can also see into the ultraviolet (300–400 nm), and some have sex-dependent markings on their plumage that are visible only in the ultraviolet range. Many animals that can see into the ultraviolet range, however, cannot see red light or any other reddish wavelengths. Mammals in

general have color vision of a limited type, and usually have red-green color blindness, with only two types of cones [12].

Based on the human perception of light, each pixel is encoded as a triplet of numbers that represent the amounts of red, green, and blue. Therefore, any human-visible color can be made up by combining red, green, and blue. This concept is known as the RGB color model [12, 13]. Each color channel in a pixel is typically represented with a single byte (1 byte = 8 bits), which can represent $2^8 = 256$ patterns. Finally, three channels in a pixel can use $3 \cdot 8 = 24$ bits to represent $2^{24} = 16{,}777{,}216$ patterns of different colors.

The set of simple programs written in *Jython* (Python implemented in Java) [13] can be effectively used for manipulating pictures by making a picture object out of a JPEG file, then changing the pixels (the red, green, and blue components) in the picture. This approach can be used for increasing or decreasing color components, clearing any color component from a picture, lighten or darken the picture, creating a negative, converting to grayscale, mirroring the image, copying a picture to a canvas, creating a collage, rotating (flipping) a picture, reducing red-eye, blurring the image, etc.

The example of creating the negative image with the *Jython* program is shown in Fig. 8, below.

a) Original image (© V. Riabov's photograph) **b)** Negative image

Figure 8: Creating the negative image with the *Jython* program

The *Jython* program code listing and the commands of the code execution are shown in Fig. 9.

```
def negative(picture):
  for px in getPixels(picture):
    red=getRed(px)
    green=getGreen(px)
    blue=getBlue(px)
    negColor=makeColor( 255-red, 255-green,
          255-blue)
    setColor(px,negColor)
```

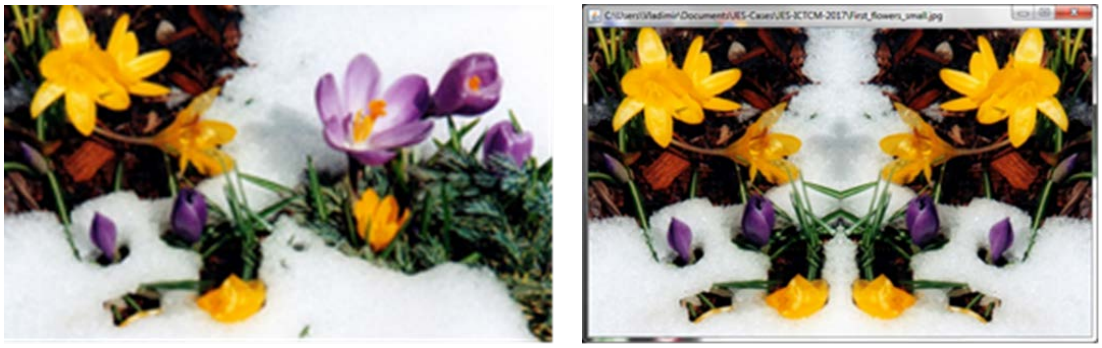
a) "Negative" Program in *Jython*

```
>>> file=pickAFile ()
>>> print file
C:\Users\Vladimir\Documents\JES-Cases\JES-
ICTCM-2017\First_flowers_small.jpg
>>> picture=makePicture (file)
>>> show (picture)
>>>  ======= Loading Progam =======
>>> negative(picture)
>>> repaint(picture)
```

b) JES *Jython* commands

Figure 9: The *Jython* program code listing and the commands of the code execution in the negative image generation case.

The other example (below) demonstrates the creation of the mirror image symmetrical to the vertical line.



a) Original image (© V. Riabov's photograph) **b)** Symmetrical image

Figure 10: Creating the mirror image (symmetrical to the vertical line) with the *Jython* program

The *Jython* program code listing and the commands of the code execution for creating the mirror image are shown in Fig. 11.

```
def mirrorVertical(source):
  mirrorpoint=getWidth(source)/2
  for y in range(1,getHeight(source)):
    for xOffset in range(1,mirrorpoint):
      pright=getPixel(source,
xOffset+mirrorpoint,y)
      pleft=getPixel(source, mirrorpoint-xOffset,y)
      c=getColor(pleft)
      setColor(pright,c)
```

a) "mirrorVertical" program in Jython

```
>>> file=pickAFile ()
>>> picture=makePicture (file)
>>> show (picture)
>>>  ======= Loading Program =======
>>> mirrorVertical(picture)
>>> repaint(picture)
```

b) JES Jython commands

Figure 11: The *Jython* program code listing and the commands of the code execution in the mirror-image creation case.

These exercises have been effectively used in the introductory Computer Science courses that motivated non-CS majors to explore in depth the image-processing techniques and create simple programs and tools. *Jython Environment for Students* (JES) is available on the CD in the back of the Guzdial's textbook [13] or can be downloaded from http://coweb.cc.gatech.edu/mediaComp-plan/MediacompSoftware/.

After in-class discussions of the case studies, each student continued working on a selected case analyzing algorithms, creating computer codes (in Java/Jython or C/C++), running them at various parameters, comparing numerical results with known data, and presenting the findings to classmates. In the course evaluations, students stated that they became deeply engaged in course activities through examining the challenging problems related to the advanced concepts from the described theories and practical applications.

REFERENCES

[1] Riabov, V. V., and Higgs, B. J. *J. Computing Sciences in Colleges*, 2010; **25**(6): 245-247.
[2] Kaufman, C., Perlman, R., and Speciner, M. *Network Security: Private Communication in a Public World*, 2nd edition. Upper Saddle River, NJ: Prentice Hall, 2002.
[3] Frequencies. Online: http://www.simonsingh.net/The_Black_Chamber/frequencyanalysis.html.
[4] Riabov, V. V., and Higgs, B. J. "Algorithms and Software Tools for Teaching Mathematical Fundamentals of Computer Security." Proceedings of the 23rd Annual International Conference on Technology in Collegiate Mathematics (Denver, CO, March 17-20, 2011), Prentice-Hall, 2011, Paper S062, pp. 208-217.
[5] Decrypting Text - code breaking software: Frequency analysis. Online: http://www.richkni.co.uk/php/crypta/freq.php
[6] McCabe, T. J. *IEEE Transactions on Software Engineering*, 1976; **2**(4), 308-320.
[7] Watson, A. H., and McCabe, T. J. *NIST Special Publication*, No. 500-235. NIST, 1996.
[8] Riabov, V. V. *J. Computing Sciences in Colleges*, 2011; **26**(6): 86-92.
[9] Miller, G. *Psychological Review*, 1956; **63**(2), 81-97.
[10] Halstead, M. H. *Elements of Software Science*. New York: Elsevier North Holland, 1977.
[11] Stroud, J. M. "The Fine Structure of Psychological Time." In: *Annals of the New York Academy of Sciences*, Vol. 138, Interdisciplinary Perspectives of Time, 1967, pp. 623–631.
[12] Color Vision. Online: https://en.wikipedia.org/wiki/Color_vision
[13] Guzdial, M. *Introduction to Computing and Programming in Python: A Multimedia Approach*. Upper Saddle River, NJ: Pearson, Prentice Hall, 2005.