Teaching Support for the Visualization of Selected Recursive Algorithms

Baraník, Róbert and Steingartner, William

Abstract: ***We present in this paper a software project that serves as a teaching tool for the visualization of algorithms taught and explained in the course "Data Structures and Algorithms". This software tool is considered to be a teaching tool for lecturers and students. The visualization is performed by storing the states of particular algorithms in a list of states during the execution of the algorithm and later displaying those states on the canvas and showing in the text area as a pseudo-code. User can enter own input values, both manually and from an XML file, and create output from a canvas or pseudo code. The application provides dual language interface – Slovak and English. Since we consider a universal use of the application, it can be run on the desktop even on a web browser using the WebSwing web server. The application is implemented in Java using JavaFX library for graphics applications.***

Index Terms: ***graphical user interface, recursive algorithms, teaching tool, university didactics, visualization***

1. Introduction

NOWADAYS, there are currently several different applications for visualizing algorithms, mainly sorting algorithms and algorithms for visualizing trees and graphs (for example [10]). However, only a few of applications are suitable for teaching the undergraduate course Data Structures and Algorithms for the study program Informatics at the Faculty of Electrical Engineering and Informatics: either due to the absence of selected recursive algorithms or the lack of localization into the Slovak language, which could help to better understand the principles of presented algorithms. For these reasons, there was a motivation to create an application that could serve both teachers and students as a teaching tool for this. We expect that using our software module could help students to understand the principles of recursive algorithm better, thanks to their visualization.

After selecting the most interesting algorithms that are presented in the course on Data Structures and Algorithms, we focused on the design and development of the teaching tool that is able to visualize and explore the following algorithms:

- Merge sort,
- Max-min,
- Knight's Tour,
- Eight queens puzzle,
- Sierpiński triangle,
- Sierpiński curve.

The application was implemented as a Java application using *JavaFX* library. Our idea was that higher-level visualizing tool can provide visualization as a whole (as a running example) with pausing the algorithm run in any step, and must be able to step forward (in particular steps), and to step back. The application should have several input options, such as manual user input, generated random input by clicking a button, and retrieving input from a file in some universal format (for instance in *XML*). The application should also provide options for visualization output: the output of pseudo-code to a simple (pure) text format or a *PDF* document, or an output of a visualization canvas to an image or, also, to a *PDF* document. Moreover, the application should be localized into Slovak language providing, also, an English user interface. Full documentation can be found in [3].

We consider that our application can successfully serve as a teaching tool for the lecturers actively teaching the course, for the new teachers of the course, for the students in the preparation for lectures and exams, and for the experts who need to simulate these algorithms when writing their software modules or applications.

The structure of the paper is as follows: In section 2, we present the main facts about the algorithms that can be visualized by our software. Section 3 refers to the functionality of the program – it describes its main functions and the user interface. In section 4, we explain the procedure how the application has been designed and developed. We conclude the paper with section 5.

2. Algorithms Overview

Most of the selected algorithms are solved by recursion – defining the process using itself [12]. The Max-min and Merge sort algorithms are well-known *Divide and Conquer algorithms*: they work by recursively breaking down a problem into two or more sub-problems of the same or related type until these become simple enough to be solved

Manuscript received in June 2020. This work was supported by the Project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

Baraník, Róbert. The author is a student of Informatics at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: robert.baranik@student.tuke.sk)

Steingartner, William (corresponding author). The author works at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: william.steingartner@tuke.sk)

directly. The solutions of the sub-problems are then combined to give a solution to the original problem [12], [15].

Merge sort is a recursive sorting algorithm that continually splits an array of size N recursively into halves. If the array is empty or has one item, it is sorted by definition (the base case). If the array has more than one item, the array is split again and the merge sort is recursively invoked on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed [9]. Merging is the process of taking two smaller sorted arrays and combining them into a single, sorted, new array. Continually, in each level of the Merge sort tree, there is the same original number of elements N, only in a different number of fields of different sizes. Merge sort is useful for sorting large amounts of data progressively [6].

Max-min is a recursive algorithm that returns the maximum and minimum value of a given array. In this approach, an input array is halved until pairs or individual values are formed. The maximum and minimum of the array are determined from them (with an individual value if it is the same value for the minimum and maximum) and the algorithm recursively returns these two values to the "parent" array. The return values of the array pairs are then compared: the maximum of the first array with the maximum of the second array and the minimum of the first array with the minimum of the second array [1]. The maximum and minimum of this "parent" array are then returned recursively, and this is repeated until recursion is complete, when the maximum and minimum of the entire array are obtained.

Knight's Tour is an algorithm for finding a (closed) path of movement of a chess knight on a chessboard, mostly 8×8 in size. The knight can move to a maximum of eight different squares relative to the previous position. This problem is solved if the knight steps on each square of the board according to established rules. An essential characteristic of the solution using recursion is the fact that the individual steps, based on which we proceed to the final solution, are first examined by trial and error and then recorded [14]. In addition to recursive implementation, the algorithm can be solved using Warndorf heuristic [7]: the algorithm searches for another possible field based on the heuristics of the surrounding (maximum eight) positions, which indicates how many possible next steps are from the given field. The algorithm always chooses the lowest possible heuristic to avoid failure.

Eight queens puzzle is the problem of placing chess queens on the board so that they do not endanger each other according to the rules of chess. Chess composer Max Bezzel published the eight queens puzzle in 1848. Franz Nauck published the first solutions in 1850. Nauck also extended the puzzle to the n queens problem, with n queens on a chessboard of $n \times n$ squares. Since then, many mathematicians, including Carl Friedrich Gauss, have worked on both the eight queens puzzle and its generalized n-queens version [14]. The queen can move horizontally, vertically and diagonally to an infinite distance limited only by other pieces and edges of the board. Thus, there must not be two or more queens in the same row, column or diagonal [5]. This algorithm goes through each column, in which it tries to insert exactly one queen. After placing one queen on the board, it moves to the next column and puts another queen starting from the first row. If the queen cannot be placed on any row of the given column, the queen is removed and the algorithm returns to the previous column and moves the queen in this column to the next row. The algorithm ends by placing the last queen in the last column.

Sierpiński triangle is one of the first examples of a fractal – an irregular, fragmented geometric shape that can be divided into parts, each of which is at least approximately similar, a scaled-down copy of the entire geometric shape [8], [13]. It consists of several triangles – 3^{n-1}, where n is the level of triangle recursion. The first level of the triangle consists of a simple one-sided triangle with one vertex pointing upwards. Each subsequent level consists of three triangles of the previous level: one at the top, one at the bottom left, and the last at the bottom right, creating a blank inside, also in the shape of a triangle, but with one vertex down. The algorithm is recursive – when it reaches the lowest level, it draws one simple triangle. The process of creating such a triangle is also very nicely presented in [11].

Sierpiński curve is a shape that fills a square-shaped space. Similar to the Hilbert and Peano curve, the Sierpiński curve maps unit interval onto square [2]. The curve consists of open curves turned in four different directions (bottom, right, top and left) and connected by other lines. The zero-order curve consists of a square standing on one vertex, formed by connecting lines, without open curves. The next level is formed by open curves connected by other lines. In the first level, for example, at the bottom, there is only one open curve connected to the other parts by lines. In the second, there are already four simple open curves connected by lines, and in the next levels, this part is formed by an increasing number of these open curves, which will gradually fill the entire space of the square. Open curves and their connections are constructed by the following rules:

$$Curve : A \searrow B \swarrow C \nwarrow D \nearrow \quad (1)$$

$$
\begin{aligned}
&A : A \searrow B \Longrightarrow D \nearrow A \\
&B : B \swarrow C \Downarrow A \searrow B \\
&C : C \nwarrow D \Longleftarrow B \swarrow C \\
&D : D \nearrow A \Uparrow C \nwarrow D
\end{aligned}
\tag{2}
$$

3. Functionality

The application is implemented with several functionalities to simplify and improve control and understanding of algorithms. The graphical user interface (GUI, Figure 1) of the application consists of a menu in which a user can switch between the selected algorithms and also switch between two languages of the environment – Slovak and English. When the language option is changed, the text on all elements – title, descriptions, buttons, error messages, and, also, the pseudo-code is changed, except for the usual keywords, such as $\mathbf{while}$, $\mathbf{if}$, $\mathbf{true}$, etc. After choosing one of the algorithms in the menu, the title changes to the name of the selected algorithm and a short description of the algorithm is written on the canvas, which forms the largest part of the interface and on which the states of the algorithms will be visualized later. The pseudo-code is displayed in the text area on the right-hand side of the canvas. The source files for the pseudo-code are uploaded from a folder separate from the application and change based on the visualization status. Below it, there are the buttons for entering input and obtaining output from the visualization.

Pressing one of these buttons opens and displays a screen overlay with an input dialog where a user can select manual input using a text field followed by a validation message, random input by pressing a button, or input from a preprepared *XML* file, or an output dialog, where, after selecting an output source (pseudo-code or canvas), this source can be saved to a simple file (standard text format – *TXT* extension, or graphical format *PNG*) or to a *PDF* document in the selected directory. The key part is the control of the visualization, which consists of several elements: a button to run (play) / pause the process of visualization, buttons for a step back and step forward, a button to return to the beginning, a slider to change the playback speed and a visualization progress bar.

The *WebSwing* web server, which is compatible with the version of the *JavaFX* technology used and can display the application window in the browser, was used to run on the browser. The installation requires a physical server on which *WebSwing* can be run. The *JAR* file and other necessary files must be inserted into the *WebSwing* directory. After starting *WebSwing* and configuring a new profile, the installation is complete and the application can be launched in the browser.

4. Methodological Design

The developed project is structured into several packages. The main package contains *Controller* and *Main* classes, which are typical of JavaFX technology when using the *FXML* file that is used to represent the application interface. The *Controller* class is the most important class of a project – it manages most user actions, such as changing speed, switching algorithms or language, so it is an intermediary between the user and parts of the system. It also contains references to interface elements, such as the canvas, text boxes, and buttons, which are annotated with the *@FXML* annotation [4]. Visualization controls – play/pause, step back / forward, reset, and change speed – are also implemented in this class.

The visualization is performed by storing the states through which the algorithm has passed in the list of states and is also separated from the execution of the selected algorithm. Thus, the playback of the visualization is performed by going through the individual states on a separate thread of the processor. If the visualization is launched, the value of the state index currently plotted is obtained and increased by one. Then, a method for drawing the canvas and writing the pseudo-code is performed on the object of the given algorithm. It returns a *Boolean* value indicating whether the state exists on this index and can be visualized. If it returns the value *true*, it waits for the time based on the launch speed and repeats the process. If it returns the value *false*, it stops the visualization.

The main package also includes the *StageController* class and the *Language* enumeration class, both of which change the language: *Language* has a static variable with the language currently in use and methods to change it, *StageController*, in addition to changing the root of the *FXML* document of the scene to the root of the chosen language, also initializing the scene using an fxml reader and saving the mentioned roots and their controllers.

The *algo* package, which is nested in the main package, contains the packages of each algorithm, the parent *Algorithm* class, from which the individual classes inherit and which contains methods common to multiple algorithms, such as chessboard rendering, the parent *State* class, which contains data for individual states and their visualization, the *Board* class, which stores chessboard data, and the *AlgorithmType* enumeration class. Algorithm packages consist mainly of two classes: the algorithm class (e.g. class *Queen* for the Eight Queens problem) and the state class of that algorithm (*QueenState*). Algorithm classes have several methods that are common to other

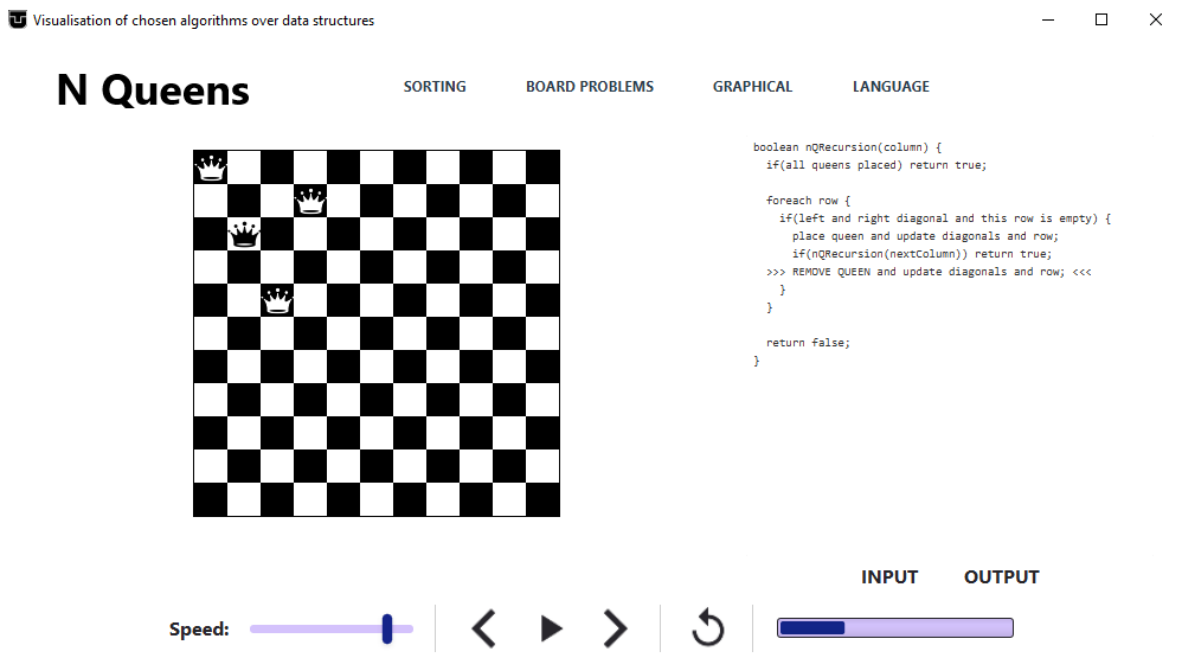

Fig. 1. Application running on desktop

algorithms and inherit them from the parent *Algorithm* class. These include methods:

- *execute* – it executes algorithm code in the background and saves a list of states, which is sent to the *Controller*, to start visualization in its method for playback;
- *drawAllStates* – based on the method parameter (index) determines which state to visualize, executes also the *drawState* and *writePseudoCode methods*, by which it passes the given state directly;
- *drawState* – draws the given state on the canvas based on variables in the state object;
- *writePseudoCode* – writes the pseudo-code stored in the file to a text area, adds changing values from the state variables.

The *io* package contains classes that control input and output data. The most important of these are the *IODialog* and *InputOutput* classes. The *IODialog* contains methods for creating the graphical user interface of input and output dialogues, so it could be said that it is the equivalent of the *Controller* class for input and output dialogues. Based on the dialog type (input or output) and the selected input/output type, this class inserts a dialog, buttons, and text fields in a predetermined location, which, when pressed or changed, send data to an *InputOutput* class object. For feedback, when entering input into a text field, the *InputField* class has been created, which has information about the restrictions of a given input value, such as a minimum and maximum value, and methods for determining the validity of input values. Feedback from the objects of the *InputField* and *InputOutput* classes is finally displayed in the dialog via the methods of the *IODialog* class. *InputOutput* contains methods for input processing and output generation. Based on the type of the algorithm in the *generateInput* method, it generates a correct input or reads values from an *XML* file that the user uploads to the application. It also creates a canvas image or copies the text in the pseudo-code text area when creating the output and applies it to the selected *PNG* or *TXT* file, or pastes the data into a *PDF* using *DocumentBuilder* and the *iText* library. It also checks the format of the selected file using the *commons-io* library. Feedback on each of the methods is sent to the *IODialog* object for display. When the input is correct, it is sent to the *Controller* class for processing, which sends it to the class of the selected algorithm and runs its execute method. Then it is then possible to start the visualization.

5. Conclusion

The application meets all specified requirements, it can visualize all six algorithms based on various types of inputs, step and change speed if necessary, and finally provides an output from the visualization – from canvas or pseudo code. Application is ready to provide a user interface in the Slovak and English language. Our application is suitable for use in teaching, which will certainly be more attractive in this work and students will be able to use it to handle the problem of these

algorithms and recursions much easier and faster. However, the application will need to be further extended, since in this project we focused only on selected algorithms that are part of an appropriate topic in the course Data Structures and Algorithms. In future, we would like to concentrate on other algorithms which are also presented in the course, especially sorting algorithms, the implementation of which would have another great impact in the process of educating the students. User interface can also be modified – it is possible to add certain elements, such as sliders when entering input values, which would simplify and make working with this application easier. One possible improvement is making several versions of visualization with some algorithms, using other types of graphs. Based on the identified possible extensions, we are ready to continue in this work in future and to bring more attractive ways of teaching algorithms and principles of particular structures.

REFERENCES

[1] ALSUWAIYEL, M. H. *Algorithms design techniques and analysis*. World Scientific, 1999.

[2] BADER, M. *Space-Filling Curves – An Introduction with Applications in Scientific Computing*. No. 9 in Texts in Computational Science and Engineering. Springer-Verlag, 2013.

[3] BARANÍK, R. Visualisation of selected algorihms over data structures. Tech. rep., Technical University of Košice, Košice, Slovakia, 2020.

[4] CHIN, S., VOS, J., AND WEAVER, J. *The Definitive Guide to Modern Java Clients with JavaFX: Cross-Platform Mobile and Cloud Development*. Apress, 2019.

[5] COPPIN, B. *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., USA, 2004.

[6] JEON, M., AND KIM, D. Parallel merge sort with load balancing. *International Journal of Parallel Programming 31* (02 2003), 21–33.

[7] LEVITIN, A., AND LEVITIN, M. *Algorithmic Puzzles*. Oxford University Press, USA, 2011.

[8] MANDELBROT, B. B. *The Fractal Geometry of Nature*. Henry Holt and Company, 1983.

[9] MAREŠ, M., AND VALLA, T. *Algorithm Labyrinth Guide*. CZ.NIC, 2017. (in Czech).

[10] MOCINECOVÁ, K., AND STEINGARTNER, W. Software support for visualizing of the graph algorithms in a novel approach in educating of young it experts. *IPSI Transactions on Internet Research 16*, 2 (July 2020), 14–23.

[11] PELÁNEK, R. *Programmer's exercise book*. Computer Press, 2012. (in Czech).

[12] RYCHLÍK, J. *Programming techniques*. KOPP, 1992. (in Czech).

[13] WELSTEAD, S. *Fractal and Wavelet Image Compression Techniques*. SPIE Press, 1999.

[14] WIRTH, N. *Algorithms and Data Structures*. Alfa, 1989. (in Slovak).

[15] WRÓBLEWSKI, P. *Algorithms, Data Structures and Programming Techniques*. Helion, Poland, 2019. (in Polish).

Róbert Baraník is a student of Informatics at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. The title of his bachelor thesis is "Visualization of Selected Algorithms over Data Structures".

William Steingartner works as Assistant Professor of Informatics at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his PhD thesis "The *Rôle* of Toposes in Informatics" in 2008. His main fields of research are the semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with type theory and software engineering.