Chapter 8.11
Explaining Algorithms:
A New Perspective

Tomasz Müldner
Acadia University, Canada

Elhadi Shakshuki
Acadia University, Canada

ABSTRACT

This article presents a novel approach for explaining algorithms that aims to overcome various pedagogical limitations of the current visualization systems. The main idea is that at any given time, a learner is able to focus on a single problem. This problem can be explained, studied, understood, and tested, before the learner moves on to study another problem. Toward this end, a visualization system that explains algorithms at various levels of abstraction has been designed and implemented. In this system, each abstraction is focused on a single operation from the algorithm using various media, including text and an associated visualization. The explanations are designed to help the user to understand basic properties of the operation represented by this abstraction, for example its invariants. The explanation system allows the user to traverse the hierarchy graph, using either a top-down (from primitive operations to general operations) approach or a bottom-up approach. Since the system is implemented using a client-server architecture, it can be used both in the classroom setting and through distance education.

INTRODUCTION

Today, an *algorithm* means a specific set of instructions for carrying out a procedure or solving a problem, and algorithms play an intrinsic role in Computer Science. Because of their importance, researchers have been trying to find the best way to teach and learn algorithms. One of the best-known approaches has been to use visualization, especially its understanding as "the power or process of forming a mental image of vision of something not actually present to the sight" (Petre, Baecker, & Small, 1998a).

Visualizations use various kinds of multimedia, including graphics to represent data, animation, and video to convey the temporal evolution of a computer algorithm (Stasko & Lawrence, 1998), and voice for explanations, also called auralization (Brown & Hershberger, 1998). Visualizations often come in a form of local programs executed in a single machine. They also come as distributed Web-based programs, using various kinds of server-side scripts to produce visualizations. In this case, the remote user can use applets or

Java Server Pages to access these visualizations, choose the required algorithm, select input data, and so on.

There have been many papers describing the use of animation to software explanation. This article will not review these usages, but interested readers are referred to the following two best-known anthologies (Gloor, 1992, 1998). A typical approach used for algorithm visualization is:

1. Take the *description* of the algorithm (usually this description comes in the form of the code in a specific programming language, such as C).
2. Graphically represent *data* in the code using bars, points, and so forth.
3. Use animation to represent the *flow of control.*
4. Show the animated algorithm and hope that the learner will now *understand* the algorithm.

Step 2 is often automatically generated from the source code, possibly with the help of specifying "interesting events" (Brown & Sedgewick, 1998).

There are various problems with the previous approach. First, providing a *graphical representation* of an algorithm is just another way to show the *code* of the algorithm — instead of using a textual programming language, we use a graphical language. Executing the visualization, we *simulate* the code written in a textual language, using a graphical language and the representation is typically at the *low level of abstraction* that shows the low-level steps.

To *explain* an algorithm, too much emphasis may be placed on meta-tools (graphics and animation) rather than the problem at hand. Indeed, some studies found that the effect of using animation is either neutral or even negative (Stasko & Lawrence, 1998). Dijkstra (1989) even feared "permanent mental damage for most students exposed to program visualization software." However, Crosby and Stelovsky (1995) found that students who interacted with an algorithm animation performed significantly better than students who listened to a lecture. Recently, an overview evaluation of the educational effectiveness of algorithm visualization has been given in Hundhausen, Douglas, and Stasko (2002).

In this article, we describe an alternative and systematic procedure to explain algorithms, and this is why we talk about explaining algorithms. Our approach is based on results of evaluations of existing algorithm visualizations, some findings from Cognitive Psychology and Constructivism Theory regarding the active use of algorithm explanation systems, and experience from software engineering and verification regarding the use of multiple levels of abstraction and properties, such as invariants, to explain algorithms and justify their correctness. However, unlike some more radical believers of Constructivism, and following the experience with design patterns (Gamma, Helm, Johnson, & Vlissides, 1995), we believe that algorithm explanation should be prepared by experts, who have intimate knowledge of the algorithm and the best way of coding it.

Specifically, we use a hierarchical, tree-based algorithm abstraction model. In this model, a single abstraction is designed to explain one operation using the Abstract Data Type (ADT) defined for this specific abstraction, with the top abstraction explaining the algorithm. For example, the root abstraction for a selection sort algorithm is designed to explain this algorithm, using the ADT operations. In this example, one of the ADT operations is a function smallest() that finds the smallest element in the sequence. A visual representation is used by the student to help him or her understand the basic properties of this abstraction; for example, invariants of the selection sort. The lower levels of abstraction focus on operations that are considered primitive at the higher levels. In the selection sort example, the low-level abstraction provides ADT with primitive operations to represent the function smallest(). Accordingly,