Designing Courseware on Algorithms for Active Learning with Virtual Board Games

Nils Faltin
Department of Computer Science
University of Oldenburg
26111 Oldenburg, Germany
+ 49 441 798 - 3115
Faltin@informatik.uni-oldenburg.de

1. ABSTRACT

We present a method for designing courseware on algorithms for active learning with virtual board games. Our goal is to build algorithm courseware that integrates explanation with animation and makes the student an active participant. We give hints for structuring the material into sections and mixing presentation with exercises. We present our ideas for a new form of visual interactive exercise and a cardboard game prototype with which we tested our ideas.

1.1 Keywords

Courseware design, algorithm animation, active learning.

2. INTRODUCTION

The main goal in learning an algorithm should be that the student gains an intuitive inner understanding of the algorithm's operations. We regard this as a prerequisite for being able to analyze, modify and code the algorithm and to be able to compare it with other algorithms. Learning an algorithm can be a difficult task. Algorithms have a static functional structure but also a dynamic behavior. They operate objects like numbers, links, arrays and invariant properties that are very distant our human living experience.

Teachers and researchers have tried a long time to enhance the understandability of algorithms by using pictures and animations. There has been a lot of progress on the technical side of algorithm animation, as documented by Stasko et. al. in "Software Visualization" [7]. *Animated Algorithms*, a major courseware on algorithms was developed by Gloor, Dynes and Lee [5,6]. It consists of a hypertext version of *"Introduction to Algorithms"* [2], algorithm animations and talking head explanatory movies. But recent empirical evaluations [1,8] have shown that the didactical value of algorithm animations often is low. Participants of the studies complain, that there is too little explanation accompaning the animation and that they do not understand the visual mapping. The study of Stasko and Lawrence [8] showed that if students had to provide own input to an algorithm and had to vary the input and observe the changes in the algorithms behavior, they were able to understand the algorithm better. This was measured in a post-test by comparing groups having access to no animation, a fixed animation or a flexible animation. The active participation of the students turned out to be the key to successful learning, thus confirming the didactical concept of active and explorative learning.

Despite its low effect in the evaluations, dynamic visualisation may still be a powerful media for learning algorithms, if it is embedded differently in the learning process. A common model in algorithm animation is that code and data of the algorithm are in the computer and that graphical abstractions of the data structures are presented in one or more views. The user functions merely as a viewer, having only some sort of VCR-Control (play, pause, step-by-step etc.). But how can the user be given an active role in learning an algorithm? We believe that building interactive visual exercises may be the answer.

Eisenberg [3] advocates a similar idea for the realm of application software. There the user has context knowledge that he can use to guide the algorithm into a direction meaningful for the situation. He wants to give the user the possibility to understand the steps of an algorithm by making them visual and allowing the user to intervene at special decision points.

The challenge is to build algorithm courseware that better integrates explanation with animation and to change the students role from a viewer to an active participant. In the following sections we present our approach to this challenge. We will give hints for structuring the material into sections and mixing presentation with exercises. We will present our ideas for a new form of visual interactive exercise and a cardboard game prototype with which we tested our ideas.

3. PRINCIPLES FOR STRUCTURING THE ALGORITHM PRESENTATION

Learning an algorithm is a difficult task because of all the details that must be understood and that are often interrelated. By splitting the presentation into elementary

ITiCSE '99 6/99 Cracow, Poland

sections which can be understood separately the student can concentrate on these elements. Such a section can also serve as a target for a hypertext link within the course text or in an index or glossary, if the student wishes to clarify a single concept and not to work through the entire lesson. The downside of a strong structuring can be that the total presentation needs more space and time and that the student may have problems bringing the knowledge from the different sections together to a coherent whole. We propose the following structure for the presentation of an algorithm.

Typical sections might cover, for instance:

- the problem to be solved and its practical relevance
- comparison of different algorithms for solving the problem
- the static link structure of data objects
- (numerical) ordering properties between objects and how to obtain them
- main phases of the algorithm
- mapping of link structure to data types of a programming language

It is also possible to divide the presentation along the *functional structure* of the algorithm. That means grouping elementary commands and function calls into a function which in turn can be used by other functions. A trivial example is a swap function which exchanges two values in an array. The algorithm itself can be modeled as a top-level function. Functional abstraction is used for information hiding and for reuse in the context of software engineering but should provide easier learning of algorithms because it facilitates understanding on the next functional level.

Our advice is to present the sections bottom-up, beginning with elementary sections and continuing with sections that build on the elementary sections. This way a concept can be truely internalized before its use. An introduction to the courseware should provide the initial motivation, which is then refreshed by the joy of understanding a section. One may argue, that a top-down presentation makes the value of a section more obvious. But you would have to pay for it with a lot of forward references to concepts the student will not be able to fully understand yet.

Realization details like mapping of link structure, implementation tricks like dummy list end markers and code containing implementation details should be presented after the "logic" behind the algorithm has become clear.

4. STRUCTURING THE HEAPSORT PRESENTATION

We applied these principles to the design of a courseware on the well known heapsort algorithm. Heapsort [9] is a standard topic of a typical data structures course. We will give a brief sketch of the algorithm here. More details can be obtained from most textbooks on algorithms.

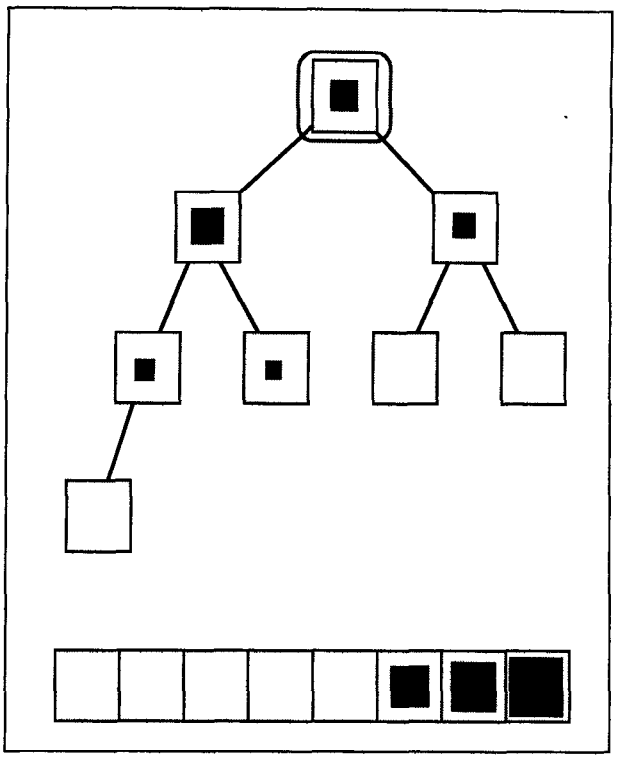

Figure 1. Figure 1 - heaptree with marked node

We decided to divide the heapsort presentation into the following sections

- the sorting problem
- heapsort vs. other sorting algorithms
- complete binary tree
- heap property definition
- `heapify-locally`: obtaining the heap property with three nodes
- the `heapify` function
- `build-heap`: initializing the heap
- removing the largest key
- sorting with heap and list
- storing the heap and the list

Heapsort uses a special data structure, the `heap`, after which the algorithm is named. The heap is a *complete binary tree*, where all levels are completely filled except for the lowest, which is filled from left to right but not necessarily completely. In a heap the key at every node must be greater or equal to the keys of its children. This is the so-called *heap property*. Figure 1 shows a heap tree where key values are denoted by the size of the inner squares. The node that violates the heap property is marked. The following sections deal with obtaining the heap property by comparing and exchanging keys and removing the maximum key from the heap and then repairing the heap, so that it is still a complete binary tree satisfying the heap property. The

`build-heap` and `heapify` functions are used for obtaining the heap property initially and then after removal of the maximum key. Sorting is simply done by removing the largest key from the heap and appending it to the front of the result list. The heap and the list can both be stored efficiently in a single program array.

The division into sections also reflects the functional structure of heapsort. The following hierarchy shows how functions call each other in a typical run. Functions marked with an asterix are called iteratively.

```
heapsort
- build-heap
-- heapify (*)
--- heapify-locally (*)
- sort
-- move-max-to-list (*)
--- move-root
--- move-last
--- heapify (*)
---- heapify-locally (*)
```

We now compare our structure with that of the often cited text book "Introduction to Algorithms" [2], which has the following sections:

- complete binary tree, heap property and storing the heap
- heapify
- build-heap
- sorting with heap

Compared with our structuring, a section of [2] deals with more (closely related) concepts. If a function is only used once its instructions will be embedded only into its surrounding function. The storage of the heap in an array is explained first instead of last and all explanations and functions work with this array.

5. TEACHING AND STUDENT ACTIVITY

Research in learning has shown the importance of motivation and active participation of students. This is not a new insight as is reflected by exercises, which are part of practically every text book and every course on a technical subject. Strangely enough, we have seen few examples where exercises were integrated into courseware in a manner that uses the strengths of this modern media, like animation, sound and interactivity with immediate error feedback. To enhance motivation and active work on the side of the student we implement a section with the following steps.

- stating the problem
- offering a board game
- teaching the standard solution
- offering exercises for practicing the standard solution
- mentioning working and non-working alternatives

To turn back to our heapsort example - How can you obtain the heap property initially by only comparing and exchanging keys in the heap? Instead of telling the standard solution procedure immediately, we let the students explore the possibilities for various procedures themselves. This will make them more aware of the problem and challenge their creativity in problem-solving. To provide such an environment, the data structures can be visualized on the screen in order to allow the students to exchange keys using the mouse. The program can give hints, which nodes of the tree violate the heap property by highlighting them, as shown in Figure 1. We see an analogy between such an exercise and a traditional *board game* like chess or checkers. Distinct objects are moved between specific locations on the board, obeying the rules of the game. The player moves the pieces with a certain goal in mind. Compared with algorithm animation the data structures stay in the computer and the code control is placed into the hands of the student. Each exercise should provide the functions as possible actions, that the student has already learned. Thus exploiting a strength of the new medium courseware in contrast to using paper and pencil. There is no way to start a heapify function on a node on paper, only atomic actions like exchange nodes can be performed there.

The next step is to teach the standard solution, using for instance text, pseudocode, figures and animations. Then we allow the student to practice the standard solution again with a board game exercise. But now stronger feedback is given guiding the student along the path of the standard solution. If the wrong operation is selected feedback should tell why this will not work and may give a hint to the correct operation.

We observed people working with a card board game prototype and found out that people are quite creative and come up with alternative solutions. It would be good, if the courseware could comment on these solutions. But because of the high development cost of such a Intelligent Tutorial System, we propose to just state a few alternative solutions. Then it can be left as an exercise to the student to analyze these solutions for effectiveness and efficiency, possibly again with a board game. Of course the quality of these alternative solutions can also be stated in a following step.

6. CARDBOARD GAME PROTOTYPES

In order to test our ideas and learn about the learning process we developed a cardboard game prototype of the heapsort data structure.

The prototype consists of a 8 node tree with 4 levels and an 8-element result list placed below the tree. Figure 1 shows the state when two keys are in the result list. The `heapify` function has to be started on the root to rebuild the heap property. Nodes and elements are 6.5 cm wide squares of yellow cardboard and are placed on a white 60 x 60 cm large paper background. The nodes are connected with black lines. Elements and nodes are numbered from 1 to 8 corresponding to the storage mapping of heapsort. There are 8 keys of square green cardboard that can be moved freely on the game board. The smallest key is 2 cm wide, which proved to be the smallest size that can be moved easily with one finger. The width of the keys is increased by 15% from

key to key, giving an area increase of about 32%. According to Weber's law, humans can distinguish weight or size of two objects if they lie at least 10% apart. The players had no problem comparing keys only by looking at the keys' sizes. It works equally well if the keys additionally have numbers written on them.

We introduced the heapsort algorithm and the idea of learning with algorithm board games to a number of computer science students, fellow researchers and laymen. The board game enabled us to explain algorithm steps and visualize them immediately. Likewise, the student could try out own solutions or practice standard solutions. This made it possible to try out our presentation steps described above. The students enjoyed it and made good progress. Several students came up with the same alternative solution to remove the maximum element from the heap. They removed the key at the root and then repeatedly drew up the larger key of the child nodes. This restores the heap property but destroys the completeness property. So we started a sort of dialogue and developed a working alternative solution together with the students. This provided a good starting point for teaching the standard solution and then comparing it with the alternative solution. During the experiments our role frequently changed from teacher to advisor.

We believe that prototype tests give valuable insight and should guide further courseware development.

7. FUTURE WORK

Our research group has a long tradition in the design of courseware with simulation and interactive exercises [4]. The efforts described in this paper are part of a doctoral thesis project on interactive visual exercises for algorithm courseware. At present heapsort is still only a cardboard prototype, but in several master theses courseware with virtual board games is under development. Subjects cover balanced search trees, the ESPRESSO logic minimizer, LR parsing generators and binomial queues. We also plan to asess the usability and learning effect of these courseware products.

We hope that we can prove the value of such exercises and give useful hints how to build them to future authors of algorithm courseware. So that exercises obtain the place they deserve in courseware as they already do in courses and text books.

8. REFERENCES

[1] Byrne, M.D., Catrambone, R. and Stasko, J.T. Do Algorithm Animations Aid Learning? TR GIT-GVU-96-18, GVU, Georgia Institute of Technology, Atlanta, GA.

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L. Introduction to Algorithms. 1990. MIT Press, Cambridge.

[3] Eisenberg, M. The Thin Glass Line: Designing Interfaces to Algorithms. CHI 96 Conference Proceedings (1996). ACM Press, 181 - 188.

[4] Gorny, P. Didaktisches Design telematik-gestuetzter Lernsoftware. In: Koerber, B. and Peters, I.-R. Informatische Bildung in Deutschland, Berlin 1998, LOG IN Verlag. 127-155. English pre-version: http://www-cg-hci.informatik.uni-oldenburg.de/resources/TET.pdf

[5] Gloor, P., Dynes, S. and Lee, I. Animated Algorithms. CD-ROM. 1993. MIT Press.

[6] Gloor, P. Elements of Hypermedia Design. 1997. Birkhaeuser.

[7] Stasko, J., Domingue, J., Brown, M.H. and Price, B.A. (eds.): "Software Visualization". 1998. MIT Press.

[8] Stasko, J. and Lawrence, A. Empirically Assessing Algorithm Animations as Learning Aids. In: [7] 419-438.

[9] Williams, J.W.J. Algorithm 232 (heapsort). Comm. of the ACM, 7:347-348, 1964.