Seppälä, Otto; Korhonen, Ari; Malmi, Lauri

Observations on student errors in algorithm simulation exercises

Published in:
Koli Calling, 17-20.11.2005, Koli, Suomi

Published: 01/01/2005

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Seppälä, O., Korhonen, A., & Malmi, L. (2005). Observations on student errors in algorithm simulation exercises. In *Koli Calling, 17-20.11.2005, Koli, Suomi* (pp. 81-86). TURKU CENTRE FOR COMPUTER SCIENCE. http://www.kolicalling.fi/old_cms/archive/2005/koli_proc_2005.pdf

Observations on student errors in algorithm simulation exercises

Otto Seppälä
Helsinki University of Technology
PL5400
Finland
oseppala@cs.hut.fi

Lauri Malmi
Helsinki University of Technology
PL5400
Finland
lma@cs.hut.fi

Ari Korhonen
Helsinki University of Technology
PL5400
Finland
archie@cs.hut.fi

ABSTRACT

In algorithm simulation exercises, students simulate the steps of a given algorithm by manipulating data structure visualizations on computer screen using a mouse. In contrast to "typical" data structures and algorithms course exercises, these exercises are designed to work on an abstraction level higher than that of the actual implementation. Correspondingly on this higher abstraction level, there exist several misconceptions on how the algorithms work.

We attempted to infer these misconceptions from the students' answers and then implemented corresponding variations of the algorithm to see what amount of the students' answers consistently follow each variation. The results suggest that many students are aware of the ultimate goal of the algorithm but have not studied the algorithm itself well enough. This often leads to different misconceptions that can be modelled and recognized using our approach.

1. INTRODUCTION

Data structures and algorithms are among the most important issues in learning programming. For students, they are often difficult issues, since capturing the dynamic nature of abstract algorithms is not a straightforward task. The traditional way of teaching these topics is programming-oriented, *i.e.*, students on their first or second programming course are introduced to basic structures such as stacks, queues and trees in terms of programming exercises. More algorithms are included in subsequent advanced courses where also algorithm design and analysis principles are discussed.

At the Helsinki University of Technology we have, however, chosen a different approach. Following the first introductory programming course, we give a general course of data structures and algorithms[1] that covers basic structures, important sorting and searching methods, basic priority queues and a number of basic graph algorithms. We deliberately approach the theme on a high conceptual level and aim to give students a broad overview of the field, instead of concentrating on implementation issues.

1.1 Visual Algorithm Simulation

Learning on this higher abstraction level is supported by a number of *visual algorithm simulation exercises* [6], which are carried out in a dedicated learning environment called TRAKLA2 [8]. These exercises are a compulsory part of the course[2] and cover most relevant material on the course. In the exercises, students simulate the working of given algorithms by manipulating the visual representations of the corresponding data structures on a computer screen through context-sensitive drag-and-drop operations and push buttons. An example exercise is:

> "Drag the keys J M R G A B T Z K L from an array in this order into an initially empty AVL tree. Perform appropriate rotations by selecting a node and pushing the correct push buttons indicating which rotation should be executed."

All changes to the data structures are recorded and the resulting states form a sequence. The states of the performed algorithm simulation are then compared with the model solution created by an actual implemented algorithm. The correctness of the student solution is assessed by counting the number of matching key states in the two solution sequences.

TRAKLA2 allows the students to submit their solution many times. However, each time a new random set of initial data is given after getting feedback on the previous submission. In addition, students can view the model answer sequence to the exercise instance at hand in terms of algorithm animation. Again, after viewing the model solution the student cannot submit an answer to the same exercise instance anymore but a new exercise is automatically generated upon request. These two features should help the student to reflect on their understanding and try to find out what went wrong in the answer if it is incorrect. The features can, of course, be used to explore the algorithm by just generating new instances and viewing the model solutions only.

The pedagogical foundation of this approach is to promote the construction of *mental models* [1, 9]. The visualizations act as *conceptual models* of the working of target systems, in our case, composed of algorithms and data structures. These aid students to form an appropriate pattern of the mind of each target system [9]. This pattern must be accurate, consistent, and complete enough in order to be *viable*, *i.e.*, capable of executing and tracking the actions in the conceptual model as well as in the original target system. If the mental model is not accurate enough, a student typically

[1] Some 500 students are enrolled each year including CS majors and minors. The course extent is 5 ECTS credits.

[2] The other compulsory parts are examination (all students), analysis and design exercises in closed labs (CS majors only), and a design project (CS minors only).

makes systematical mistakes while executing the conceptual model. These kinds of errors are symptoms of *misconceptions*. Of course, other types of errors exist as well that are non-systematical such as carelessness mistakes.

There are several challenging topics to be tackled here. First, we should try to recognize and classify various types of misconceptions from the students' answers. Second, we should recognize with some certainty that some specific answer actually belongs to a certain misconception class in order to give advanced feedback on it. Finally, we should try to understand how and why the students form the misconceptions and improve our teaching and support material.

In this paper, we shall discuss our first experiences and results on these lines of research. The paper is structured as follows. In the next section, we discuss recognition of misconceptions in general, including a review of relevant work with algorithms in Section 2.1. In Section 3 we present in some detail data and results from the TRAKLA2 environment. One assignment on the BuildHeap algorithm is used as an example case. Section 5 concludes with a summary of the work and directions for future research.

2. RECOGNIZING THE MISCONCEPTED ALGORITHMS

Any errors made when solving the exercises can either be systematic, carelessness errors or result from randomly trying out the exercise. Our main interest lies with the students who make mistakes in a systematic way, as this is often a symptom of a misconception[3] that could be corrected if recognized.

Brown and VanLehn [3] define this systematicity as follows: *A child's errors are said to be systematic if there exists a procedure that produces his erroneous answers.* As there are only so many ways to systematically solve a problem wrong, it should be possible to model each of these procedures. Such an approach was used by Brown and Burton in the Buggy[2] system and refined by Burton with Debuggy[4].

2.1 Buggy and Debuggy

The Buggy and Debuggy systems were designed to assist in mathematics education, revealing the kinds of misconceptions learners have about performing place value subtractions. Sison and Shimura [11] describe and compare various approaches and systems for student modeling. The following description of Debuggy is based on their paper.

Debuggy models the skill to be measured as a network of subskills Incorrect implementations of a skill can then be created by replacing one or more of the subskills with their buggy counterparts from a *bug library*. The system aims to find any misconceptions students might have by applying the following procedure: First a behavior set for each student is built from answers to a set of exercises testing the same skill. The system then tries to recreate all answers that deviate from the correct ones by replacing a single subskill in the procedural network by a bug. If the altered network can explain at least one answer, the system will add it to an initial hypothesis set H. This set is then reduced by removing any bugs that are subsumed by others. Any remaining erroneous answers are then tested with compound bugs created as pairs from the remaining hypothesis set. Finally the system tries to alter some of the remaining bugs with heuristic perturbation operators to increase the number of explained answers. When all these steps are finished the bugs are ranked by their ability to explain the student's answers, the number and type of predicted misconceptions and the amount of perturbations made to the original bugs.

Besides Buggy and Debuggy there exist a number of different approaches to student modeling that are more sophisticated than the ones described. For a comprehensive list of different approaches you can consult the paper of Sison and Shimura [11].

2.2 Our Approach

Our approach is similar to that used in Debuggy, with a few important exceptions. The first one is the data, the student's answers to the exercises, which are already sequences of smaller steps giving us more information on the underlying mental model. We also work in a different domain and a different set of students. What is similar, is the aim to find the set of *algorithm variants* that result from the different misconceptions and then using these variants for automatically recognizing possible misconceptions in each answer sequence.

To know more of the types of errors the students make, we browsed through the students answer sequences looking for any typical mistakes for each exercise. Each time we encountered a sequence that did not match any previously recognized variants, we attempted to infer and implement the corresponding algorithm. The new set of algorithm variants was then tried against the data, narrowing down the number of cases without explanations.

Implementation of erroneous algorithms by hand is laborious, but as we will see in section 3, some algorithms devised by the students defy the type of approach used by Debuggy as some of the new algorithms are not based on the original algorithm at all. Working on the algorithms also gives valuable insight on the different misconceptions less easily grasped by just reading the final results of an automatic algorithm.

2.3 Inferring from the model answer

One of the problems in making inferences on the students answers is that they have to be based on assumptions of the underlying algorithms. We can never fully eliminate the possibility of mapping a correct algorithm seeded with some carelessness to a specific misconcepted algorithm. Slips can also interfere with the selection between incorrect variants.

While we cannot eliminate this problem altogether, we can make it less probable. After having found the most likely variants, we can apply our knowledge of them to generate problem instances where the erroneous procedures better deviate both from the correct procedure and from each other.

2.4 Grading the algorithms

The process of deciding on which algorithm variant best resembles the student sequence is similar to checking the validity of the students' answers (explained in Section 1.1). We run each of these algorithm candidates with the same input to get a number of new sequences. These sequences are then compared to the student's sequence. The comparison procedure iterates over the candidate sequence, selects a

[3] In some cases, systematic error-like behavior can emerge from other sources as well, *e.g*, misreading the exercise definition.

single state at a time, and iterates over the student's states trying to find that very same state. The candidate that best explains the student sequence is the one that finds a match that involves a user state farthest away in the student sequence.

This is different from evaluating normal student solutions, but necessary as the number of steps in each of the algorithm variants may vary. Thus algorithms with "smaller" steps would be favored as more steps would often be explained. For this reason the student sequence (which has a constant amount of states) is used as the measure.

In most of the cases, there exist a number of variants that match the student sequence equally well. In these cases, we have selected the most *conservative* option available. Basically this will be the candidate that best resembles the original algorithm or the most general one available.

2.5 Implications of easy goal verification

One of the most defining features of the problems in Data Structures and Algorithms is that the ultimate goal of the algorithm is often something easily accomplished by any non-algorithmic approach. Whether the goal has been reached can also be verified with ease in many of the cases. Sorting a data structure is an obvious example: regardless of any education in sorting algorithms anyone can both verify if an array of data is sorted or not and if necessary can also perform a series of steps that sort the array.

Interestingly, this easy verification works also to our advantage. One of the clearest clues of a misconcepted algorithm is reaching a legal final state without following the requested algorithm. This often reveals either a solution based on a misconception or a slip followed by a corresponding fix later on. Therefore verification can be used to sieve out likely candidates for misconcepted algoritms as it is a clear sign of not only blindly imitating an algorithm.

2.6 Error categories

The results from studying a number of exercises indicate the following categorization of the erroneous answers.

1. *Slips*, errors created by carelessness. These are always non-systematic. Such errors include skipping data elements in iterations, non-systematic off-by-one errors, and any solitary missing operations. In some cases the student may have also recognized the error and tried to remedy it later on.

2. Problems related to the mapping from the data structure to the visualization. Such errors include, for example, *mirror algorithms* where the algoritms work as the original ones with the left and right elements reversed.

3. Applying a *wrong algorithm*. Some learners, for example, execute a preorder traversal in place of an inorder traversal.

4. Applying a *legal variant* of the algorithm. When working with a binary search tree we must be consistent in how we handle duplicate keys. Essentially we can have two different implementations of the algorithm and thus two different variants of which only one is recognized to be correct by the exercise.

5. *Nonrecognizable algorithms*, that is, algorithms that might work systematically, but do not aim for the same result as the original algorithm.

6. *Variants of the original algorithm*. These variants follow the overall idea of the algorithm, but make systematic changes in portions of the algorithm. These changes can for example simplify a part in the original algorithm. Such algorithms include omitting recursion, systematic off-by-one errors, looping errors etc.

7. *Whole new algorithms* that have the same ultimate (and verifiable) goal as the original algorithm. These have an air about them that the algorithms used have been inferred from examples rather than by misreading them from a textbook. We will look into such algorithms in detail in the next section.

The systematicity of the last two categories allowed us to automatically classify answers by implementing these "student-built algorithms" and then re-evaluating the students' answer sequences against each hand-implemented new algorithm.

In the following section, we will examine some of these algorithms for an exercise where the students were to execute the Build-Heap algorithm on a given data set.

3. CASE EXAMPLE: THE BUILD-HEAP EXERCISE

In the Build-Heap exercise, the students were asked to create a binary heap from a random set of values by manipulating a heap depicted as a binary tree. The method of manipulation is *swapping keys* initiated by dragging a key with the mouse onto another one.

The Build-Heap algorithm can be found in most textbooks [5, 12]. The data structure employed by the algorithm is an array, but as it is also an essentially complete binary tree, it can be illustrated and manipulated as such. The algorithm begins from the bottom right part of the tree and iterates through each key until the root of the tree is reached. If necessary, each key will be swapped recursively downwards with its smallest child until the heap property for that key is restored. The learning objectives of the exercise include the following items:

1. *Recursion*, *i.e.*, how the algorithm builds bigger heaps by combining smaller heaps together where the procedure is defined in terms of itself.

2. *Necessary swaps*, *i.e.*, the algorithm makes comparisons among the parent and its two children in order to do at most one swap in a recursive call.

3. *Order of traversal*, *i.e.*, the first parent for a subtree to be heapified is the middle element in the array and then the tree is traversed toward the first element (root of the tree).

The textbooks explain this algorithm in form of pseudocode (See Algorithms 1 and 2, adapted from [5]). Some students, however, have difficulties reading the notation. It is also worth to mention that textbooks use different pseudocode styles and have different approaches in terms of naming algorithms, ordering statements and dividing algorithms to subalgorithms.

Where [5] uses the names *Build-Heap* and *Heapify* for the algorithms, [12] calls them *BuildHeap* and *PercolateDown* and [7] gives a non-recursive algorithm by the name *HeapBottomUp*. Therefore even the textbook which should support solving the exercises can sometimes act as a source of misconceptions.

Algorithm 1 Build-Min-Heap(A)

```
for i ← ⌊heap-size[A]/2⌋ downto 1 do
    MIN-HEAPIFY (A, i)
end for
```

Algorithm 2 Min-Heapify(A, i)

```
l ← LEFT-CHILD-INDEX(i)
r ← RIGHT-CHILD-INDEX(i)
if l ≤ heap-size[A] and A[l] < A[i] then
    smallest ← l
else
    smallest ← i
end if
if r ≤ heap-size[A] and A[r] < A[smallest] then
    smallest ← r
end if
if smallest ≠ i then
    SWAP(A[i], A[smallest])
    MIN-HEAPIFY(A, smallest)
end if
```

3.1 Algorithm variants for Build-Heap

While we have explored almost half of our set of exercises the work is still in progress. The Build-Heap exercise was selected as an example because it displays most of the phenomena discussed earlier in the paper.

While roughly a third (34.4%) of the answers to this exercise simulate the correct algorithm without errors, there exist a number of relatively popular student-made algorithms. One fourth (25.5%) of all answers do not follow the original algorithm to the point, but still end with a state fulfilling the heap property.

In the following, we describe the versions that emerge from the data studied in this project. In the parenthesis, there is the percentage of occurrences from the overall submissions[4]. Discussion on possible reasons for existence of these variants is also given.

3.1.1 Variant 1 — No-Recursion (6%)

Description

This variant is almost as the original, but the MIN-HEAPIFY method does not call itself recursively.

Discussion

Missing the recursive call in the MIN-HEAPIFY can result from a multitude of different causes. The simplest explanation is that the recursive call is just missed by mistake. Another possibility is that the recursive behavior was just not understood properly. Recursion is generally considered a hard topic for novices.

For a heap size of 15 items (our typical exercise size) the probability of randomly generating an exercise instance without any recursive calls is 1 to 27. This basically means that some students with no conception of recursion can pass the exercise. This also means that there exist model answers which also do not exhibit any recursive behavior. The exercise is to be modified to ensure that each student is tested for understanding of recursion.

[4] As the total number of examined submissions for this exercise was 880, a single student repeating his erroneous approach can have a significant effect on the numbers.

3.1.2 Variant 2 — Heapify-with-father (6%)

Description

In the original algorithm, the smaller of the two children is swapped with the parent if necessary. In this variant, both children are compared with the parent and swapped, if necessary, resulting in an algorithm that makes twice as many swap-operations as the original algorithm in worst case.

Discussion

This variant can easily be distinguished from the original algorithm due to the additional swaps made. While the algorithm is clearly incorrect, it will always create a valid heap if executed recursively. Students who have not studied the actual algorithm, but still know what kinds of operations the algorithm uses, can easily imagine up such an algorithm.

As with the non-recursive case, this variant performs identically with the original algorithm for some special cases of input data. For the difference to display, there must exist a state in the algorithm where both children of a node are smaller than their father node and the right child is greater than the left child. For randomly generated exercise with 15 elements, roughly 1 in 25 exercise instances never go through such a state when solved.

3.1.3 Variant 3 — Fix-Levelwise/Iteratively (0.8%)

Description

In this algorithm the recursive heapify-operations are not executed immediately, but only after a whole level has been gone through.

Discussion

This variant does every single swap done by the original algorithm. The order of the operations is just altered. For this variant to show up, both of the keys on the second highest level of the heap must be swapped down and the right one at least two levels.

3.1.4 Variant 4 — Left-To-Right (0.7%)

Description

In this algorithm, each level of the heap is traversed from left to right instead of right to left. In all other aspects, this resembles the original algorithm.

Discussion

While rare, the mere existence of this variant tells about mapping problems between the visualization and algorithm implementation. The binary heap is often depicted as an algorithm operating on a binary tree although the implementation works directly on an array, which can cause confusion.

In this case running the algorithm from left to right however does not affect the underlying algorithm, telling that the algorithm itself was correctly understood.

3.1.5 Variant 5 — Smallest-child-to-correct-place-instantly (<1%)

Description

This algorithm variant starts from the root node and traverses top-down and left-to-right. At each node, all its children are sought recursively for the smallest child, which replaces the node directly by a swap operation without swaps with the nodes in between.

Discussion

The most interesting feature of this variant is that while it has little to do with the original algorithm, they both create almost the exact same heap in the end. Not swapping with the immediate children of the node and beginning from the root both suggest that the student has no understanding of the original algorithm. The fact that the resulting heap is often almost identical to the heap created by the real algorithm indicates the possibility the student has inferred the version of the algorithm by using only the final state of the original algorithm sequence. This algorithm always creates a legal heap.

3.1.6 Variant 6 — Single-Skips (4.3%)

Description

This algorithm variant includes all algorithms where two or more swaps have been skipped. This tries to model carelessness errors although it might subsume other categories.

Discussion

Some students have complained about the exercise using alphabetic keys. They claim that working with alphabets is slower and more error-prone than working with numbers. Some of the skips could therefore be caused by problems with alphabetic order rather than not understanding the algorithm.

3.1.7 Other variants

Discussion

While examining the sequences we came by many sequences that while seemingly non-systematic, still make swaps that eventually lead to a legal heap. Such answers suggest that some students possibly just try out the exercise using only their knowledge that in a valid heap all nodes must fulfil the heap property or that there exist systematic algorithms we haven't yet deciphered.

4. ORIGINS OF MISCONCEPTIONS

While there is no single theory that explains all the erroneous behavior observed and all the origins of these misconceptions, two theories, namely the Imitative Problem Solving [10] by Robertson the Repair Theory [3] by Brown and VanLehn both explain much of the observed phenomena.

4.1 Imitative problem solving

Analogical Problem Solving (APS) is a paradigm that tries to explain how solutions of previously solved problems are used when solving new problems (and possibly in new domains). According to Robertson [10] APS however has a number of built-in assumptions about the knowledge of the solver. Robertson tries to account for cases where these assumptions are not met by another paradigm, Imitative Problem Solving (IPS). Robertson claims that novices mostly solve problems by imitation.

If the student tries to solve a problem through imitation, he might look for a problem example in the textbook and then map the *surface features* onto the target problem. In our case, finding a correct example is easy and straightforward as such examples are available not only in books, but also in the assessment tool itself. The part where the student faces difficulty is in applying the procedure. When imitating, the student tries to perform the same operations on the new data that were done in the example.

Some algorithms, such as No-Recursion and Heapify-with-Father are explicable with imitation. If the level of extracting information from the example only goes as far as recognizing that smaller keys should be swapped upwards with their parents, the student can proceed to select either of these variants.

4.2 Repair Theory

The Repair Theory[3] by Brown and VanLehn is a theory that tries to explain the causes of bugs and why only certain bugs do occur. Their key idea is that bugs result from *repairing impasses*. Assume that a learner is for some reason missing a fragment of a procedure. When applying this procedure it leads him to a situation where he believes some step cannot be carried out. In such a case, the learner will attempt a repair that allows him to continue applying the procedure. As Brown and VanLehn shortly put it, they believe *that many bugs can best be explained as "patches"*.

In the Heap-Exercise we can easily figure out the potential impasse - executing a non-recursive variant to the end and then realizing that the resulting structure still violates the heap property. One possible fix is of course the original algorithm, but different iterative fixing schemes can also be applied to reach a final state.

5. CONCLUSIONS

We have described a procedure for finding incorrect systematic answers to algorithm simulation exercises. Preliminary results for one exercise, Build-Heap were also presented. These were collected using the procedure described. We also discussed possible reasons for the erroneous algorithms and misconceptions found.

While this paper describes a method which only covers systematic errors through hand-implemented algorithms, we plan on extending the approach to help find slips and erroneous algorithms through automatic means.

The results also pointed out some new guidelines for selecting input for the simulation exercises.

1. Only inputs that result in a sequence showing all the essential features of the algorithm are to qualify. For the Build-Heap exercise such a requirement would be about the use of recursion in the exercise.

2. None of the previously known erroneous variants should create a sequence identical with the correct sequence.

This is particularly important as receiving points with an incorrect algorithm might reinforce the misconception.

3. Two erroneous variants should not create identical sequences to make it possible to identify the underlying misconceptions.

It is still important to point out that the existence of slips will affect recognition of the misconceptions and can never be fully eliminated. One of the aims of this research is to provide the student with better automatic feedback, when human assistance is not available. Studying the quality of this feedback on a real course will give us more insight on how accurate our methodology is.

6. ACKNOWLEDGMENTS

This work was supported by the Academy of Finland under grant number 210947.

7. REFERENCES

[1] M. Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[2] J. S. Brown and R. B. Burton. Diagnostic models for procedural bugs in mathematical skills. *Cognitive Science*, 2:155–192, 1978.

[3] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

[4] R. B. Burton. Debuggy: Diagnosis of errors in basic mathematical skills. In *Intelligent Tutoring Systems*. Academic Press, 1981.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[6] A. Korhonen. *Visual Algorithm Simulation*. Doctoral thesis, Helsinki University of Technology, 2003.

[7] A. Levitin. *the Design and Analysis of Algorithms*. Addison Wesley, 2003.

[8] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[9] D. A. Norman. Some observations on mental models. In D. Gentner and A. Stevens, editors, *Mental Models*, pages 7–14. Lawrence Erlbaum Associates, 1983.

[10] S. I. Robertson. Is analogical problem solving always analogical?: The case for imitation. HCRL Technical Report 97. Technical report, HCRL, The Open University, 1993.

[11] R. Sison and M. Shimura. Student modeling and machine learning. *International Journal of Artificial Intelligence in Education*, 9:128–158, 1998.

[12] M. A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison Wesley, 1997.