122nd ASEE Annual Conference & Exposition
June 14 - 17, 2015
Seattle, WA
Paper ID #13846

Simulating the execution of algorithms using students as actors

Dr. Arturo Camacho, University of Costa Rica

Arturo Camacho is an Associate Professor in the Department of Computer Science and Informatics at the University of Costa Rica (UCR), where he works since 2009. He has taught the courses of Data Structures and Analysis of Algorithms, Sound Processing, and Probability and Statistics. He has done research in analysis of musical and biological audio signals at the university's Research Center on Information and Communication Technologies (CITIC). He received his Ph.D. and M.Sc. in Computer Engineering from the University of Florida in 2007 and 2003, respectively, and his B.Sc. in Computer Science and Informatics from the University of Costa Rica in 2001. He also studied Music at National University of Costa Rica from 1992 to 1995 and worked as a keyboardist in Latin-music bands from 1989 to 1999.

Simulating the execution of algorithms using students as actors

Introduction

Data structures and algorithms courses are usually taught by showing examples on the board or trough slides. This causes little stimulation in students, who often get bored in class. Besides, it is not uncommon that students understand the examples shown in class but are unable to generalize to new problems during homework or exams. In order to overcome these problems, we wanted to find a strategy to make learning more attractive for students. We found that simulations using students as actors met this purpose. In this paper we describe our experience with this strategy during six semesters. Rather than using class time to learn theory, it is used to illustrate the theory by running examples acted by the students, and the theory is learned by reading the textbook or watching lectures (previously recorded by the instructor). The proposed methodology facilitates so much the process of learning, that usually students do not need to study the underlying theory before class to understand the algorithms. This methodology is aimed to explode the abilities of those with kinesthetic learning (others with a preference for other style of learning can make use of the videos, textbook, or the instructor's lecture notes).[3] It has the beneficial side effect of letting students to socialize with their fellows by requiring them to get in touch and talk to each other, and giving them the opportunity to learn their fellow's names. It also avoids students to fall asleep in class, since they have to stand up and move to specific positions during the simulations. Even though similar proposals exist in the literature,[5, 6, 1, 4] some of them are not tailored to courses in data structures and graphs, are impersonal (students do not use their names in the process), require no verbal interaction between the participants, or require *dancing* abilities.

Sorting algorithms

In order to learn sorting algorithms, students are asked to get sorted by their name, following the strategy of the algorithms. The instructor introduce the basic idea behind the algorithm (as described in Cormen et al. textbook about algorithms; 2009)[2] and asks students to perform the comparisons and movements necessary to implement the idea.

A popular sorting algorithm is *insertion sort*. For simulating this algorithm, a group of some ten students are asked to form a line facing the class. Students are asked to introduce themselves – say their names – and the rest of the class take note of them and their order. They are told that the basic idea of the algorithm is to produce a sequence of subarrays, such that every time a new element is added, the subarray is guaranteed to be sorted. Hence, the first student in the line is asked to introduce herself to the class (tell her name) and check if she is sorted, which of course, she is. Then, the second student is asked to introduce himself and ask his fellow student her name (students are told they have short memory, so they should not be able to remember what happened in the previous stage nor the names they have learned earlier). He is given the responsibility to ensure that he and the previous fellow are sorted. If they are not, he is asked to request a swap. Next, the third student comes into play. She is asked to introduce herself and is given the same responsibility to conduct a

partial sort, as the previous fellow did. However, she has a harder task. She has to choose who to compare with from two choices. The best choice for her is to ask the second student. (If she does not do so and the instructor realizes that less comparisons would have been required if she had asked for the name of her neighbor first, after she finish her job, the instructor asks her what would have happened if she had compared herself with her neighbor first. Hopefully, at this point the class will have learned that it is smarter to compare with your neighbor first – the one with the currently "greatest" name – and then go down the list.) Another trick students may learn is that it is advantageous to move the pivot to a temporary area until its definite place in the subarray is found, rather than swapping every time a greater name is found.

Regarding the rest of the class, they are asked to have an active role during the simulation. Half of them are asked to count the number of comparisons performed by the simulation and the other half are asked to count the number of writings. These counts are saved to be used latter when comparing the performance of other algorithm with this one. Ideally, the class should learn in the process the strategy used by the algorithm (and they do). Students who discover the best strategy discuss with their fellows that strategy, and quickly the class learn the algorithm.

Another popular algorithm is *merge sort*. It is asymptotically faster than insertion sort (i.e., sorts faster for large arrays). As before, students are asked to form a line in front of the class. For better results (i.e., comparing the results with the ones of the previous algorithm) it is convenient to use the same students as before, and place them in the same order. Students are told that the algorithm follows a *divide and conquer* approach: The array is split in two halves, then every half is split in two halves and so on, until reaching arrays with one element (which are sorted); then, halves of arrays are combined to form larger and larger sorted arrays. The simulation starts by splitting the array in two halves. For this, students in the left part are asked to move slightly to the left and students in the right part are asked to move slightly to the right. Then, the left subarray is again split in two subarrays, and so on, until reaching arrays of size one. Individuals of the last split are then asked to combine in a sorted pair. Next, they are asked to combine with another pair (or individual, depending on how the splitting proceeded). It is up to them to find the best way to combine themselves in a single array (after some thinking and discussion they do). As with the previous algorithm, the rest of the class count the number of comparisons and writings performed, to compare them with those of insertion sort. Hopefully, merge sort requires less comparisons and writings than insertion sort to "confirm" its efficiency. Whether or not it is true, students are reminded that the theoretical advantage of merge sort is guaranteed for large arrays, not for a small ones like the one used.

The next algorithm in the list is *heapsort*. This algorithm uses a heap to sort the elements. Even though in practice the heap is implemented by an array, it is conceptually better to use a *complete* binary tree. To form the tree, students are given a rope, which they must use as the pointer to their "parent". Students are asked to put the rope around their waist and give the other extreme to their parent, forming a complete binary tree (every parent should hold up to two ropes, one on each hand). Then, the mechanics of the algorithm of comparing parents with children and swapping positions is explained, and students are asked to

simulate it. (They do have fun doing it!) As usual, the rest of the class is asked to count the number of comparisons and writings (two per swap).

Other sorting algorithms can be simulated using the same dynamics (or similar) presented here.

Data structures

Stacks, queues and linked lists are so simple that we deem it unnecessary to use simulations to learn them. Conversely, binary search trees are more complex, so simulations can come into help. Simulations can be helpful to illustrate how insertion, search and deletion works. It is recommended to switch to a different group of students from the ones used for sorting algorithms to learn the name of more students. Ask them to form a line close to the scenario (just to be sure there enough "volunteers" before starting). Students are explained the fundamental property of binary search trees: given a node *X*, keys in its left sub-tree are smaller than the key in *X*, and keys in its right sub-tree are larger than the key in *X*. Then, the first student is asked get inserted in the (currently empty) tree (it is trivial!). Next, the second student is asked to join the tree. She is told that the entry to the tree is always at the root (it is the only known node to newcomers). Her work is easy: he has to ask the root his name and place herself either to his "southwest" (if her name is lesser) or to his "southeast" (if her name is greater). Next students follow the same approach, comparing themselves with every node they found on their way down. Once everyone is in the tree, searching is illustrated by some examples (search is similar to insertion). Finally, deletions are shown. We suggest to start with the easiest case: deleting a node with no child. This is as simple as removing the node. Then, a node with one child is deleted: the node is deleted and its child is given in adoption to the grandparent. The deletion of a node with two children is more elaborated and usually needs explanation. It is important at the end to take note of the height of the tree produced and discuss why it is not a good idea to have a tall tree.

A more interesting data structure is the Red-black tree. Nodes in red-black trees are either red or black. Students are given red hats to wear if they are *red*. No hat is worn if they are *black*. Since the color of the nodes may change during operations, students may need to put on or take off their hats during the simulation. Unlike previous data structures, red-black trees' operations are not straightforward, so prior learning of them is required.

Graphs

Because the number of adjacent vertices of a vertex can be large, the use of ropes for graphs is unfeasible. Therefore, we rely on the board. Students are asked to draw on the board a vertex with their name. To make the graph, they are asked to choose as adjacent vertices those ones with names that match the initial of their last name.

For the depth-first search algorithm (DFS), the instructor first explains the purpose of the algorithm, and then its mechanics (briefly… we want students to learn it through the simulation). Then, an arbitrary vertex is chosen (usually the vertex with the "lesser" name, in a lexicographic order – just to avoid asking for a volunteer). Its owner is asked to write its

discovery time[2] next to the vertex, and pass the control to one of its adjacent vertices (usually the vertex with the "lesser" name in a lexicographic order – just to save students from having to choose) not before marking the corresponding edge as a *tree edge*. The process is repeated until a vertex with no *undiscovered* adjacent vertices is reached. At that time, its owner writes its *finishing time* next to the discovery time, and she is asked to return the control to her parent (whoever passed the control to her earlier) who transfers it to the next undiscovered adjacent vertex, if any. Once all his adjacent vertices are finished, the parent return the control to his own parent, and so on. Once the whole tree is finished, a new undiscovered vertex is chosen (usually the next undiscovered vertex, in a lexicographic order) and another tree is processed. The process continues until the whole graph is processed. After that, students are informed that *discovery* and *finishing* times are used to solve some problems that use DFS as a subroutine, such as topological sort and strongly connected components. Then, *forward*, *backward*, and *crossed* edges are introduced, and students are asked to review the simulation. In the process each student is asked to classify each of its non-tree outgoing edges as one of those. (Alternatively, they can be asked to do this while running the simulation, but experience has shown that it is easier for students to mark discovery and finishing times and classify edges in separate steps).

For Dijkstra's algorithm, we ask students to augment the graph used in DFS by weighting the edges as the difference in age between the people vertices represent (in months). After that, the instructor explains the strategy of the algorithm briefly, and one vertex is chosen as the *source* (usually the one with the "lesser" name). Its owner is asked to write the distance from the source (itself) to each of its adjacent vertices, mark itself as their predecessor, and leave the stage (she has no more work to do). Then, the owner of the next vertex with the current shortest distance from the source is asked to come up and update the shortest known distance from the source to its adjacent vertices, and then leave the stage. The simulation goes on until the last reachable vertex from the source is used as pivot.

Results

Before applying the proposed methodology, it was common for a significant proportion of students to fail in exams, which consisted mostly of simulating the execution of the algorithms for a given input. Usually, students made two types of errors: simple involuntary mistakes and errors that reflect a lack of understanding of the algorithm. After applying the methodology, the former were less common, and the latter were infrequent. We started using the methodology in the II semester of 2010 (in our college, I semester goes from March to June and II semester from August to November). Tables 1 and 2 show the average grades for Test 1 applied during the II semester of 2009-2010 and the I semester of 2010-2011 (before and after applying the methodology, in each case). This test is about analysis of algorithms and sorting algorithms. Results are shown separately for each semester because there is a tendency for students in the II semester to do better than in students in the I semester. (In our curricula, the course is scheduled to be taken in the II semester. Most students who take it in the I semester are students who have lost some course, and in general make lower grades). The results show a statistical difference in the grades at a 90% confidence level, but not at a 95% level ($p = 0.07$) in the II semester, and no statistical difference in the I semester. However, simulations took only a small fraction of the exam (32/130 pts. in II-2010 and

40/138 pts. in I-2011). Conversely, Test 2, which is about data structures, has a larger fraction dedicated to simulations (62/117 pts. in II-2010 and 97/120 pts. in I-2011), and the effect of the methodology is more clear. The difference in grades is significant in both the I and II semesters ($p < 0.01$). Graphs are evaluated in Test 4, but results are not shown for this test because simulations for graphs were introduced at a slower pace, taking several semesters for a complete treatment of the topic using simulations. Besides, it started much later, in 2012, and records of the test grades were lost after the course ended. (Test 3 is about topics for which the methodology has not been applied yet: backtracking, dynamic programming, and greedy algorithms.)

Table 1. Average grades for Test 1 applied in the II semester of 2009 and 2010.

Year	Method	Pts. simulation	*N*	Average grade	Diff. w.r.t. lecture	P-value
2009	Lecture	76/147	16	61		
2010	Simulation	32/130	26	73	12	0.07

Table 2. Average grades for Test 1 applied in the I semester of 2010 and 2011.

Year	Method	Pts. simulation	*N*	Average grade	Diff. w.r.t. lecture	P-value
2010	Lecture	37/166	17	66		
2011	Simulation	40/138	21	72	6	0.41

Table 3. Average grades for Test 2 applied in the II semester of 2009 and 2010.

Year	Method	Pts. simulation	*N*	Average grade	Diff. w.r.t. lecture	P-value
2009	Lecture	40/100	16	73		
2010	Simulation	62/117	26	102	29	< 0.01

Table 4. Average grades for Test 2 applied in the I semester of 2010 and 2011.

Year	Method	Pts. simulation	*N*	Average grade	Diff. w.r.t. lecture	P-value
2010	Lecture	40/125	17	63		
2011	Simulation	97/120	20	107	44	< 0.01

The proposed methodology could have an impact not only in the simulations required in the tests, but also in the whole course. For that reason, we present also final grades statistics. Figure 1 shows the average final grades of the course for the I semester between 2010 and 2013, and Figure 2 for the II semester between 2009 and 2012. The figures show a big jump in the grades after applying the methodology. There is, however, a big drop in the grades of the I semester of 2012. This can be attributed to a particularly large class that semester (40 students vs. an average of 20; for administrative reasons, two sections were merged into one that semester). It is known that students like to take classes with this instructor. Since the registration system gives priority to students with high GPA, this instructor usually receives only the best students (other students enroll with other instructors). However, that semester he had to receive all kind of students: good and bad. Overall, a small decrease in the grades from the first time the methodology was used can be observed year after year. This is attributed to an attempt of the instructor of always "rising the bar" (i.e., trying to do the exams harder than before). The low grades in the II semester of 2009 and I semester of 2010 can also be attributed to the inexperience of the instructor (those were the first two semesters the instructor taught the course, actually, any Computer Science course).

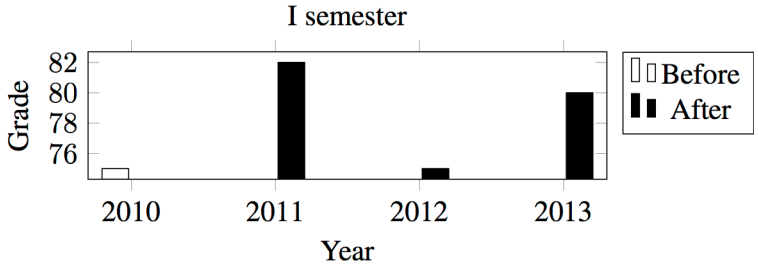

Figure 1. Average student's grades for the I semester

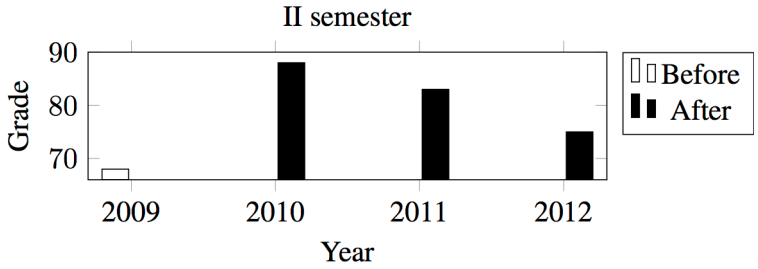

Figure 2. Average student's grades for the II semester

Regarding the students' assessment, most of them are eager to participate in the simulations (only about 1 out of 5 show reluctance to participate). Surveys conducted at the end of each semester confirm this. Besides, every semester an election is performed among students to choose what methodology to use in class: lecturing or simulation of the algorithms using them as actors. Before voting, students are asked to get informed about the method by reading the course reports written by the instructor at the end of each semester and asking former students. The last time such election was performed, the result was 14 vs. 7 votes in favor of the simulations. Results of previous elections also resulted in favor of simulations, but with a smaller difference, which shows that over time students are becoming more interested in this kind of learning. We have not asked students the reasons to prefer one kind of learning over the other. Anyway, students who prefer more traditional lectures, have always the option to watch them online.

Conclusions

We have presented a methodology to learn how algorithms on data structures and graphs work. The methodology exploits kinesthetic learning by requiring students to stand up and play the role of data in an array, nodes in a tree, and vertices in a graph, while simulating the execution of algorithms. Social interaction of students is promoted by the need to communicate to each other, either to ask names or to explain a specific step of an algorithm. The significant improvement in students when asked to simulate data structures and graph algorithms in exams, and the raise in final grades suggest that the methodology helped students to learn better the material.

Future work

In the future, we plan to explore the suitability of the proposed methodology to other Computer Science courses. We have considered to use this methodology in two more courses taught by the author: Audio Signal Processing (elective course) and Probability & Statistics (mandatory course), but we have not been able to find course contents that lend themselves to use kinesthetic learning. Nevertheless, we still believe that other Computer Science courses could benefit from the use of this methodology, particularly, Computer Architecture & Organization, Distributed Systems, and Parallel Computing.

References

1. Begel, A., Garcia, Daniel. D., Wolfman, Steven A. Kinesthetic Learning in the Classroom. SIGCSE Bull., 36(1):183–184, March 2004.

2. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, C. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill (2009).

3. Fleming, N. D. *Teaching and learning styles: VARK strategies*. Christchurch, New Zealand: N.D. Fleming (2001).

4. Katai, Z. and Toth, L. Technologically and artistically enhanced multi-sensory computer-programming education. Teaching and Teacher Education, 26(2):244-251 (2010)

5. Goldweber, M. Two kinesthetic learning activities: Turing machines and basic computer organization. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ITiCSE '11, pp. 335–335, New York, NY, USA, 2011. ACM.

6. Sivilotti, P. A. G. and Pike, S. M. The suitability of kinesthetic learning activities for teaching distributed algorithms. SIGCSE Bull., 39(1):362–366, March 2007.