Reflections on Teaching Algorithm Courses

J. Ángel Velázquez-Iturbide
Department of Informatics and Statistics
Universidad Rey Juan Carlos
28933 Móstoles, Madrid, Spain
angel.velazquez@urjc.es

Abstract

There has been much less debate within the computing research community about teaching algorithms than about teaching introductory programming. However, it is advisable (or even necessary) to hold public discussions about different issues, independently of more focused research efforts. This position paper addresses two themes. Firstly, it advocates for an experiential approach to learning algorithms, as a complement to the more common formal and engineering approaches. We show how visualization and benchmarking can make algorithms more concrete to students and their learning more active and insightful. Secondly, we argue that some conceptual models present in algorithm textbooks are imprecise, or even implicit, making difficult to learn their corresponding topics. We elaborate on this concern by stressing that several algorithm design techniques address a specific class of problems (namely, optimization ones) and by visiting several aspects of three design techniques (greedy algorithms, dynamic programming and branch-and-bound).

CCS Concepts

•Theory of computation~Design and analysis of algorithms~Algorithm design techniques •Social and professional topics~Professional topics~Computing education~Computing education programs~Computer science education

Keywords

Algorithm education, optimization problems, visualization, experimental measures, conceptual models, algorithm design techniques

ACM Reference format:

J. Ángel Velázquez-Iturbide. 2025. Reflections on teaching algorithms. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025). February 26 - March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, *7 pages.* https://doi.org/10.1145/3641554.3701937

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA
© 2025 Copyright is held by the owner/author(s).
ACM ISBN 979-8-4007-0531-1/25/02.
https://doi.org/10.1145/3641554.3701937

1 Introduction

Computing education research has matured as a research field in the past decades [11]. The research has addressed both specific areas of informatics and traversal issues. Within the areas, introductory programming is the area that has received the most attention. This focus is due to the central role of programming in informatics and that it is the first or one of the first courses required to informatics majors.

Contributions to programming education have addressed a wide range of issues concerning psychology, pedagogy, organization and technology [21]. As a result, clear contributions were firmly established and had an influence on the field. For instance, a milestone was to confirm that students' difficulties are universalr [20][24]. This finding encouraged novel initiatives, such as instructional approaches to learners' incremental building of their programming knowledge (e.g., [19]).

Other informatics fields have not received that much attention, for instance algorithm education [22]. Indeed, there are numerous and varying research contributions, but they are not as informative or consolidated as in CS1. This position paper focuses on algorithms education and is based on our personal teaching and research experience in the area. We distill two broad issues that we consider highly relevant for algorithms instruction. We summarize them here and elaborate on them in the following two sections. As a position paper, we keep the discussion at an argumentation level.

Firstly, algorithm textbooks have a strong mathematical or engineering flavor. While these approaches are necessary, we argue that students would benefit from also being exposed to a more experiential approach. Experiencing constitutes a smoother approach to learning for some students and in general provides a more vivid and active learning experience. We elaborate on two specific experiential forms: visualization and benchmarking.

Secondly, conceptual models [27] of several algorithmic issues are not as mature as in other informatics areas. Some conceptual models are poorly elaborated, or are even not presented, in algorithm textbooks. These conceptual models should be made explicit, and consensus should ideally be achieved about them. We discuss this poor elaboration for aspects of three design techniques (namely, the greedy technique, dynamic programming, and branch-and-bound). We also note some curricular corollaries of the fact that these techniques always deal with optimization problems.

2 Experiencing Algorithms

An algorithm is typically explained in textbooks in abstract terms, illustrated with some examples and probably accompanied by correctness or efficiency analysis. Although some students feel comfortable with abstraction, other students prefer an experiential approach to learning [10]. Some education theoreticians emphasized experience to develop students' capability to solve problems. Bruner's constructivist theory [5] suggests that learning new material is effective when a progression is followed from enactive (action-based) to iconic (image-based) to symbolic (language-based) representation modes.

We present here two ways of enhancing students' experience with the behavior or performance of algorithms by making them more concrete, namely visualization and performance benchmarking. They are not new, but their relevance should not be diminished. These proposals also are consistent with Kolb's experiential learning cycle [16]. According to Kolb, there are two modes of grasping experience (concrete experience and abstract conceptualization), and two modes of transforming experience (reflective observation and active experimentation). He suggests that for a complete learning experience, students must go through all modes, which can be modeled as a learning cycle composed of four stages. The two approaches here presented contribute to concrete experience and active experimentation.

Both proposals can be supported with software tools. They provide "software oscilloscopes" [3] that allow actively exploring algorithm behavior. Quoting Jonassen [14], they are cognitive tools that support constructivist learning "because they actively engage learners in creation of knowledge that reflects their comprehension and conception of the information rather than focusing on the presentation of objective knowledge". There is evidence of the benefits of both proposals [38][40][42].

2.1 Visualization

Figures are a common resource in algorithm textbooks, mainly to illustrate the run-time behavior of an algorithm for a particular input, but also to show its behavior in a general case, to clarify the problem statement, or to analyze its efficiency [39]. Algorithm animation is a well-known technology to display algorithm behavior. It was an intensive area of research in the eighties [36], with an educational impulse in the 2000 decade [13]. Algorithm animations are usually expressive because they do not descend to the code level of the algorithm but provide higher-level displays. They can also be integrated into electronic resources [32]. Their design must provide mechanisms to avoid the risk that students engage in passive viewing, which does not lead to significant learning [26]. Algorithm animations also have the technical limitation that they are not generated automatically, thus they are a burden on either the instructor or the students who must instrument the algorithm to produce correct visualizations.

Program visualization [35] is an alternative technology that automatically generates displays from code. Program visualizations may be less expressive than algorithm animations if they display programming constructs with low abstraction level but can be similarly expressive if they display constructs that play a key role in the algorithm, e.g., method calls and recursion. These systems present the advantage of generating visualizations for any algorithm (within the scope of the system). Therefore, program visualization systems can be used by instructors and students to freely explore their own algorithms, in the former case for instruction, in the latter case for self-study or to solve assignments. If the program visualization system provides powerful interaction facilities, the user will be effortlessly engaged in the exploration and adaptation to her needs of the automatically generated displays.

We illustrate the power of program visualizations, generated automatically and interactively finetuned, with some figures generated by the visualization system for recursion SRec [43]. Consider the longest common subsequence (LCS) problem, common in textbook chapters on dynamic programming [1][7]. Figure 1 shows the recursion tree obtained by implementing a "forward" [12] recursive algorithm for the problem and running it for sequences *aaaa* and *bba*. The figure shows an overview+detail display [41], where the lower part shows the complete tree at a small scale, while the upper part shows a part (framed in the overview) at a larger scale. The recursion tree shows a computation state of the algorithm close to termination, where the active call is framed in green (at the right), pending calls are framed in cyan (also at the right) and completed calls are blurred.

The visualization allows illustrating the power of program visualization to make visible (i.e., concrete) several aspects:

- Algorithm behavior, the most common use of visualizations.
- Time complexity. Recursion trees may assist in making the time complexity order visible. For instance, the recursion tree in Figure 1 has 39 nodes. Simply increasing the parameter size by 1, the tree grows to 96 nodes, illustrating an exponential growth. (Note that these numbers may vary for alternative input data of the same size as the algorithm time complexity depends on both input size and input contents.)
- Space complexity. A key difference between time and space complexity is that the former is accumulative, while the latter is reused. Consequently, time complexity is proportional to the number of nodes of the recursion tree, whereas space complexity is proportional to its longest branch. This fact was stressed in the visualization by blurring completed calls. At any step of the recursive process, memory is only allocated for the calls from the root to the active node.
- Redundancy. Multiple recursive algorithms that solve optimization problems often are redundant, i.e., many calls are repeatedly computed. In the Figure, highlighting calls with the same parameter values (in this case, index values 3 and 1) allows identifying some redundant nodes (in brown).

Program visualization can also be used to enhance algorithms [42]. A well-known method for redundancy removal in multiple recursive algorithms [2] starts transforming a recursion tree into a dependency graph (i.e., a directed acyclic graph) by collapsing equal nodes into a single node, while preserving links. The nodes of the dependency graph must then be mapped into the cells of a table. Thus, we obtain a visual representation of the table to use in an iterative algorithm, as well as the computation dependencies it must respect.

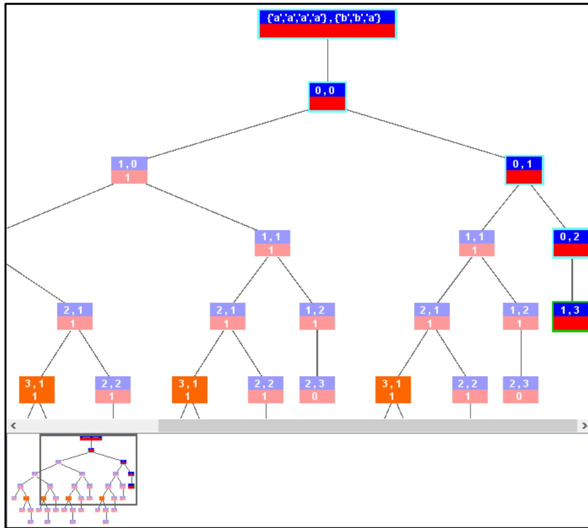

Figure 1: A recursion tree generated from the computation of LCS(*aaaa*,*baa*).

Figure 2 shows the result of mapping the nodes of the dependency graph generated from Figure 1 into a bi-dimensional table, with the values of the first parameter associated to columns, and of the second parameter to rows. Note that there are only 18 different nodes for the 39 nodes of the recursion tree in Figure 1. The method is mainly used to develop dynamic programming algorithms (see Section 3.3). In this case, several equivalent, iterative algorithms with two nested loops may be derived.

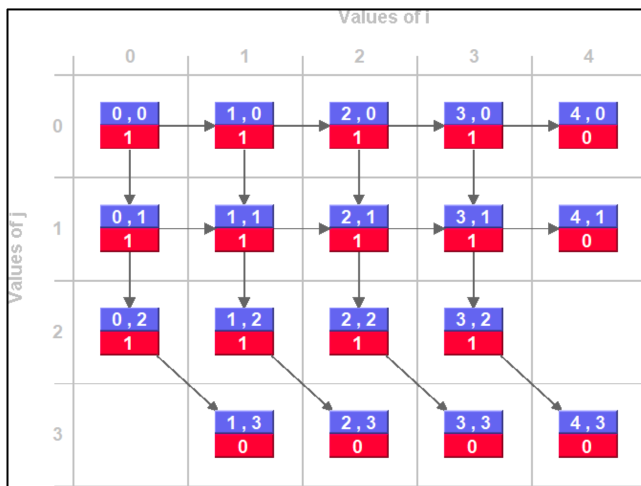

Figure 2: The dependency graph derived from the recursion tree of Figure 1, mapped into a bi-dimensional table.

2.2 Benchmarking

Another way of vividly experiencing an algorithm consists in experimenting with it by gathering performance measures. Furthermore, these measures can be gathered for alternative algorithms that solve the same problem, comparing against each other and making clear their performance differences. These activities can be facilitated by an interactive system that supports the auxiliary tasks of generating test cases, executing algorithms, and storing, comparing and displaying their outcomes [40]. The system may also remove or mitigate the difficulties of run-time instrumentation of algorithms [25][31].

Measurement gathering has often been proposed to check run-time performance [23][30], typically of sorting algorithms. Benchmarking alternative algorithms that solve the same problem can be conducted with respect to their running times or other measures or criteria [23]. In particular, solution quality allows assessing algorithms for optimization problems [6][33][40]. Let us remind that an exact algorithm always yields an optimal solution for any valid input data, while an inexact algorithm may yield suboptimal solutions. Dynamic programming and branch-and-bound algorithms are exact, while heuristic and approximate algorithms are inexact.

Benchmarking exact and inexact algorithms provides a vivid insight on the quality of their solutions. If we focus on heuristic algorithms, we know that their solutions are not bounded with respect to optimal solutions. But, how often do these algorithms compute suboptimal outcomes?, or what is the mean deviation of suboptimal from optimal outcomes? Even for approximate algorithms, we only know the maximum deviation from optimal outcomes. Jointly benchmarking approximate and exact algorithms provides empirical evidence of the quality of approximate algorithms, more vivid than just a formal proof of a bound on the quality of their solutions.

For instance, consider the minimization version of the bin packing problem [1][4][12][31], and four algorithms that solve it, for instance two approximation algorithms (first-fit and next-fit), a branch-and-bound and a backtracking algorithm. Assume we use the AlgorEx experimentation system [40], randomly generating 100 test cases with 12 objects of weights ranging between 1 and 7, and bins of capacity ranging between 7 and 20. Figure 3 summarizes the result of running the four algorithms on the test cases and of comparing their outcomes quality (shown left to right in the same order as mentioned above).

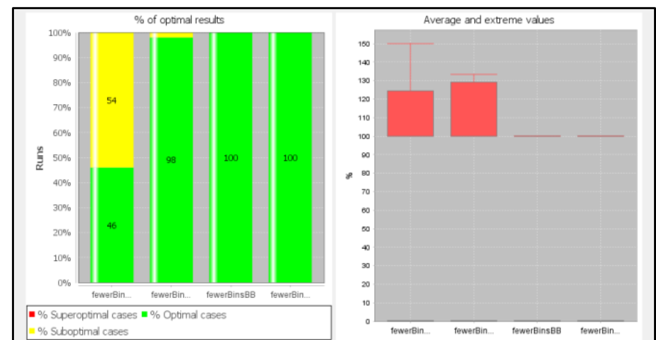

Figure 3: Two charts that summarize: (a) the percentages of cases that given algorithms yield optimal outcomes for given test cases, and (b) deviation of suboptimal solutions with respect to optimal ones.

Measures	fewerBinsAprox1	fewerBinsAprox2	fewerBinsBB	fewerBinsBack
Num. correct total runs	100	100	100	100
Maximum time	0.072 ms	0.025 ms	2.676 ms	168.287 ms
Medium time	0.005 ms	0.004 ms	0.453 ms	72.520 ms
Minimum time	0.001 ms	0.001 ms	0.008 ms	0.022 ms

Figure 4: A table that summarizes the running times of given algorithms gathered from given test cases.

Figure 3(a) shows that the first-fit approximate algorithm computes minimal outcomes in 46% of the test cases, the next-fit algorithm in 98% of the cases, and the branch-and-bound and backtracking algorithms in all the runs (as expected). Figure 3(b) shows the deviation of suboptimal solutions. The first-fit algorithm exhibits less deviation that the next-fit in most cases, but its maximum deviation is greater (50% versus 33.33% above the minimal solution). Obviously, the results differ for different sizes and ranges of values of input data, but with a similar trend.

Benchmarking is more complete by also instrumenting run-times. Using the same random test cases, we obtain the results shown in Figure 4 (we include the results in tabular format rather than in charts because the run-times measured have different orders of magnitude). The results are presented with the algorithms in the same order as in Figure 3. Note that approximate algorithms are the fastest, but also that branch-and-bound improves dramatically the efficiency of backtracking.

3 Conceptual Models

Mental and conceptual models are complementary concepts in education [27]. A mental model is the representation that a student builds of a conceptual model, i.e., her understanding of it. Consequently, a mental model often is partial and ambiguous. However, a conceptual model is a representation of knowledge, built by the instructor to transmit it to the students. Consequently, a conceptual model must be precise, complete, and consistent.

Many works have contributed to understanding students' difficulties and misconceptions [18][28]. However, little discussion can be found about the conceptual models used in algorithm courses. We ignore the reasons of this situation, but we may hypothesize that research on students' difficulties is valued within the computing education research community, while discussion of adequate conceptual models is not.

In this section, we discuss conceptual models of an outstanding type of computational problems and of three algorithm design techniques. We have found the conceptual models here presented useful in our algorithm courses either to keep consistency on the treatment given to certain issues along the course or to provide students with useful guidelines to apply the design techniques.

The discussion is checked against a selection of algorithm textbooks [1][4][7][12][15][31]. They do not constitute an exhaustive list of algorithm textbooks, but they are reputed and they vary with respect to several issues (e.g., emphasis on algorithm design techniques vs. data structures, either mathematical or engineering orientation, use of pseudocode vs. a programming language), thus we consider them representative.

3.1 Optimization Problems

As noted in the introduction, many common design techniques deal with optimization problems. This observation leads to several educational corollaries. Lack of sufficient consideration of these issues could be related to students' misconceptions on optimization problems and algorithms [38][44].

Firstly, the distinct features of optimization problems should be further elaborated. In particular, their post-condition consists of two differentiated parts: the validity condition and the target function. A solution is valid if it satisfies the validity condition, and it is optimal if it optimizes the value declared by the target function. In general, there are multiple valid or optimal solutions, being the latter a subset of the former. This fact is remarked by Sahni [31] by devoting a specific section to optimization problems (see section 18.1). However, other textbooks present these concepts passing by in the context of some design technique where it is necessary to explain the difference between valid and optimal solutions, such as greedy algorithms [4][12] or approximation algorithms [1].

A second, related observation is that the notion of correctness in optimization problems is subtle. Most problems addressed in introductory programming or algorithm courses, such as sorting an array or determining whether a natural number is prime, define one single valid solution. This situation is relaxed in combinatorial problems, such as the *n*-queens problem [4], where several valid solutions typically exist. The situation becomes even more complex in optimization problems, where we must distinguish between valid and optimal solutions, and there typically are multiple solutions of both types. Cormen *et al.* [7] stress the multiplicity of optimal solutions in their dynamic programming chapter (see p. 359).

Thirdly, optimization problems have associated some mathematical preliminaries, such as the maximum and minimum operations and their properties, order relations, and optimal vs. optimum values in an ordered set. They are not too many concepts, thus it is surprising that they were traditionally not included in the mathematical prerequisites in Computing Curricula. Fortunately, CS2023 [17] (partially) include optimization in the calculus knowledge unit, but it is not considered Core CS. Some textbooks include an introduction or an appendix with mathematical preliminaries for algorithmics [4][7], with topics such as series, summations and limits; sets, relations and functions; counting and probability; or matrices. However, order relations and the related concepts listed above are not addressed at all. Only Cormen *et al.* [7] address the related concept of bounding summations (see Appendix A.2).

3.2 Greedy Algorithms

In algorithm textbooks, the emphasis to design greedy algorithms is obviously given to greedy criteria. However, coding also is relevant, and it is usually left apart. The greedy technique is usually described in natural language. The only textbook that proposes a code template is Brassard and Bratley [4]:

```
function greedy (C: set): set
   {C is the set of candidates}
   S ← ∅ {We build the solution in set S}
   while C≠∅ and not solution(S) do
      x ← select(C)
      C ← C\{x}
      if feasible(S∪{x}) then S ← S∪{x}
   if solution(S) then return S
   else return «there are no solutions»
```

This template is very clear and is extremely useful for ensuring that the elements of greedy algorithms are understood. However, it presents two problems:

- It is only adequate for problems where the candidates and their values are known in advance. Consequently, it is adequate for some problems (e.g., the knapsack problem), but it does not generalize algorithms where the candidates vary (e.g., nodes in Prim's algorithm) or their values change (e.g., path lengths in Dijkstra's) along the algorithm execution.
- An analysis of the greedy code presented in textbooks shows that they do not fit the template given above.

The first problem can be solved by adopting a more general template. Fortunately, it can be generated from Brassard and Bratley's template. We need to generalize it so that candidates are not parameters, but they can be extracted from these. Similarly, their values may be updated at the end of each iteration. A more general template [37] follows (changes underlined), where these two operations are performed by functions *extract* and *update*.

```
function greedy (D: data): set
   C ← extract(D)
   S ← ∅ {We build the solution in set S}
   while C≠∅ and not solution(S) do
      x ← select(C)
      C ← C\{x}
      if feasible(S∪{x}) then
         S ← S∪{x}
         C ← update(D,x,S)
   if solution(S) then return S
   else return «there are no solutions»
```

The second problem demands guidelines to implement different greedy algorithms, and an explanation of the relation of different implementations with the common template. These explanations usually are scattered in the problems solved in the textbooks, but the different coding alternatives and the guidelines to use them should be explained in a relevant part of the chapter. Thus, greedy algorithms with candidates known in advance and with fixed values can be implemented more efficiently with an alternative template (not given here for lack of room), composed of a first phase where the candidates are sorted and a second phase with the greedy loop. For other algorithms, the above template can be either used straightforwardly or optimized by using specific data structures (usually, priority queues).

3.3 Dynamic Programming

Dynamic programming is often tagged as one of the most difficult design techniques, among those included in textbooks (see [31], p. 797). Actually, a number of studies addressed students' difficulties [9] and misconceptions [8][34][45]. However, it is surprising that the method contained in textbooks to develop dynamic programming algorithms often is extremely vague. Thus, it is opportune to wonder whether students' misconceptions are partially due to imprecise conceptual models.

The dynamic programming technique is best characterized as a development method rather than by a code template. Some textbooks provide several worked examples in the chapter, but do not provide a general description of the technique [4]. Others do, but all of them give insufficient detail to students on some steps of the method. Some textbooks even acknowledge the lack of precision of their own description of the method. Kleinberg and Tardos devote a short subsection to a "basic outline" of the technique ([15], p. 260), which contains "informal guidelines". Horowitz *et al.* ([12], p. 257) sketch general guidelines, but they trust "these examples should help you understand the method better".

We reproduce here the description given by Cormen *et al.*, which probably is the most complete one ([7], p. 359):

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Obviously, each textbook makes a better job in some steps of the method. For instance, Horowitz *et al.* [12] emphasize problem characterization and the design of recursive algorithms, even providing two alternative recursive algorithms for some problems, a "backward" one and a "forward" one.

However, most textbooks quickly go over step 3, in spite of being clearly established in the literature [2]. Thus, we may read in Brassard and Bratley [4], page 259: "The underlying idea of dynamic programming is thus quite simple: avoid calculating the same thing twice, usually by keeping a table of known results that fills up as subinstances are solved". The chapter contains several problems which are solved using dynamic programming, but it is never made explicit why the table has certain shape and size, and why the table is filled in a particular way. Sahni and colleagues describe the recursion removal phase as follows ([12], p. 256, [31], p. 800): "solve the dynamic-programming recurrence equations". However, removal of redundant recursion is not a common topic in computing courses, thus students must perform this phase intuitively. Alternatively, Bird's tabulation technique [2] makes step 3 relatively easy, especially with the aid of adequate graphical representations (see Section 2.1).

Let us consider the longest common subsequence (LCS) problem again. A recursion tree allows confirming the redundancy of the recursive algorithm (see Figure 1) and is later converted into a dependency graph. Its nodes can be laid out on a

bi-dimensional table (see Figure 2) by associating one parameter to columns and another one to rows. The table size is easily derived from this figure. In addition, the figure shows subproblems and their dependencies, thus a valid reverse topological order is easy to derive. For instance, we could follow two alternative orders:

- Bottom-up by rows, filling each row right to left.
- Right to left by columns, filling each column bottom-up.

Coding any of these reverse topological orders as two nested loops is straightforward, obtaining a dynamic programming algorithm. Alternatively, we might fill the table by diagonals, but those alternatives lead to more complex code.

We find descriptions of step 3 in Baase and van Gelder [1] and Cormen *et al.* [7]. These authors make clear that dependency graphs (or "subproblem graphs") are a good tool to analyze the number of total subproblems and their relations, but they only use explicitly dependency graphs for Fibonacci numbers [2, Section 10.2] and the rod cutting problem [6, Section 15.1]. Both textbooks also provide figures with the tables filled in, but without transiting through a dependency graph which justifies the format, size and order followed to fill the table. The discussion is mostly mediated by text or symbols rather than visual representations.

With respect to the development of the final, dynamic programming algorithm, both textbooks point out that it must traverse the dependency graph by following some reverse topological order. However, alternative, feasible orders are not discussed. Baase and van Gelder [1] even privilege memoization over tabulation. They also suggest the use of an algorithm to extract a reverse topological order rather than giving students a simpler method [2] to visually determine the order by themselves.

Similarly to Elström and Kann' conclusions [9], in our experience it is useful to remark the separate and detailed treatment of the different phases identified by Cormen *et al.* [7]. In particular, this implies explaining in full detail the conversion of the recursive algorithm into the iterative one. Using this approach, we have found that students' difficulties were mostly focused on phases 1 and 2, coinciding with prior work [34][45]. The use of visualizations (as illustrated in Figures 1 and 2) also contributes to make tangible the conversion process, resulting in students' detailed reports of their development process for dynamic programming assignments [42].

Prior studies on dynamic programming difficulties and misconceptions are not clear about this step. They either do not separate phases in their study [8], omit this step [9], are ambiguous about the procedure used to remove redundancy [34], or consider it jointly with step 1 [45]. In any case, our claim is that students would benefit from an explicit method for tabulation rather than just relying on their intuition. There exists experience on the feasibility of this approach and evidence on the benefits obtained by supporting it with programming visualization tools [42].

3.4 Branch-and-Bound

Search-based techniques, such as backtracking and branch-and-bound, are not included in some algorithm textbooks [1][7][15] or they are dealt with less depth than other techniques [4]. Probably, Sahni and colleagues are the authors who have presented these techniques in more detail [12][31].

The definition of branch-and-bound can be characterized by several elements, namely the strategy adopted for searching the state space and the use of bounds on the solutions in construction. For maximization problems, an upper bound is used, while a lower bound is used for minimization problems. (Minimization problems admit both bounds [12], but we do not address this issue here.)

Although Sahni et al. deal in detail with search strategies, they hardly give guidelines to define bounding functions. Fortunately, guidelines can be found in artificial intelligence textbooks because they are used in search techniques, such as the A* algorithm. A bounding function $f(j)$ frequently has the following format [29]:

$$f(j) = g(j) + h(j)$$

where $g(j)$ computes the value associated to the first j decisions adopted for the solution in construction (i.e., it is the value of the target function limited to those j decisions), and $h(j)$ makes an optimistic guess on the remaining, pending decisions.

For instance, consider the 0/1 knapsack problem [4][7][12][15][31]. The target function is equal to the sum of profits provided by those objects introduced into the knapsack. As a formula, it is declared as follows: $\max \sum_{j=1}^{n} p_j \cdot x_j$, where x_j represents the decision adopted on object j, 0 denoting its exclusion from the knapsack and 1 denoting its inclusion. It is a maximization problem; thus, the bounding function must define an upper bound on the optimal solution found so far. Therefore, a feasible bounding function for this problem is as follows:

$$\boldsymbol{f(j) = \sum_{i=1}^{j} p_i\, x_i + \sum_{i=j+1}^{n} p_i}$$

where the optimistic guess is that all the remaining objects might be introduced into the knapsack.

4 Conclusion

We have presented a reflection on several concerns for algorithm courses based on our experience of many years as instructors. Its purpose is to contribute to debating the contents and instruction of algorithm courses, and ultimately serve as an aid or as an inspiration to instructors of these courses. The paper has not addressed how to integrate these issues in a course. Some suggestions may be addressed in introductory algorithm courses, such as those on program visualization or greedy algorithms. Other suggestions are more adequate for advanced, elective courses, such as those on dynamic programming or branch-and-bound. Conceptual models specific of some algorithm design techniques can easily be integrated into their corresponding chapters. However, the experiential approaches and the treatment of optimization algorithms should permeate the whole course.

Acknowledgements

This work was supported by the project PROGRAMA, funded by MICIU/AEI/10.13039/501100011033 and by ERDF, EU (ref. PID2022-137849OB-I00).

References

[1] Sarah Baase and Allen van Gelder. 2000. *Computer Algorithms* (3rd ed.). Addison Wesley Longman, Reading, MA.

[2] Richard S. Bird. 1980. Tabulation techniques for recursive programs. *ACM Computing Surveys* 12, 4 (Dec. 1980), 403-417. https://doi.org/10.1145/356827.356831.

[3] Heinz-Dieter Böcker, Gerhard Fisher and Helga Nieper. 1986. The enhancement of understanding through visual representations. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing (CHI '86)*, Boston, MA, USA, April 1986; pp. 44-50. https://doi.org/10.1145/22339.22347.

[4] Gilles Brassard and Paul Bratley. 1996. *Fundamentals of Algorithmics*. Prentice-Hall, Hertfordshire, UK.

[5] Jerome S. Bruner. 1960. *The process of education*, Harvard University Press.

[6] Ming-Yu Chen, Jyh-Da Wei, Jeng-Hung Huang, and D. T. Lee, 2006. Design and applications of an algorithm benchmark system in a computational problem-solving environment. In *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '06)*, ACM Press, pp. 123-127. https://doi.org/10.1145/1140123.1140159.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, Cambridge, MA.

[8] Holger Danielsiek, Wolfgang, Paul and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, 21-26. https://doi.org/10.1145/2157136.2157148.

[9] Emma Enström and Viggo Kann. 2017. Iteratively intervening with the "most difficult" topics of an algorithms and complexity course. *ACM Transactions on Computing Education* 17, 1, article 4 (Jan. 2017), 38 pages. http://doi.org/10.1145/3018109.

[10] Richard M. Felder and Linda K. Silverman. 1988. Learning and teaching styles in engineering education. *Engineering Education, 78*(7), 674-681.

[11] Sally A. Fincher and Anthony V. Robins, Eds. 2019. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press.

[12] Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran. 1997. *Computer Algorithms*. Computer Science Press, New York, NY.

[13] Christopher D. Hundhausen, Sarah A. Douglas and John T. Stasko. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13, 3, 259-290. https://doi.org/10.1006/jvlc.2002.0237.

[14] David H. Jonassen. 1992. What are cognitive tools? In *Cognitive Tools for Learning*, Piet A.M. Kommers, David H. Jonassen and J. Terry Mayes (Eds.), Springer-Verlag, NATO Series F18, pp. 1-6.

[15] Jon Kleinberg and Éva Tardos. 2006. *Algorithm Design*. Pearson Addison-Wesley, Boston, MA.

[16] David A. Kolb. 1984. *Experimental Learning: Experience as the Source of Learning and Development*. Prentice Hall.

[17] Amruth N. Kumar, Rajendra K. Raj, Sherif G. Aly, Monica D. Anderson, Brett A. Becker, Richard L. Blumenthal, Eric Eaton, Susan L. Epstein, Michael Goldweber, Pankaj Jalote, Douglas Lea, Michael Oudshoorn, Marcelo Pias, Susan Reiser, Christian Servin, Rahul Simha, Titus Winters, and Qiao Xiang. 2023. *Computer Science Curricula 2023*. ACM Press, IEEE Computer Society Press and AAAI Press. https://doi.org/10.1145/3664191.

[18] Colleen M. Lewis, Michael J. Clancy and Jan Vahrenhold. 2019. Student knowledge and misconceptions. In *The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Robins, eds. Cambridge University Press, pp. 773-800.

[19] Raymond Lister. 2016. Toward a developmental epistemology of computer programming. In *Proceedings of the 11th Workshop Primary and Secondary Computing Education (WiPSCE '16)*. ACM Press, New York, NY, 5-16. https://doi.org/10.1145/2978249.2978251.

[20] Raymond Lister, Elisabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morter Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119-150. https://doi.org/10.1145/1041624.1041673.

[21] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: A systematic literature review. In *Proceedings Companion of the 23th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '18 Companion)*. ACM Press, New York, NY, 55-106. https://doi.org/10.1145/3293881.3295779.

[22] Michael Luu, Matthew Ferland, Varun Nagaraj Rao, Arushi Arora, Randy Huynh, Frederick Reiber and Jennifer Wong-Ma. 2023. What is an algorithms course? Survey results of introductory undergraduate algorithms courses in the U.S. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE' 23)*. ACM Press, New York, NY, 284-290. https://doi.org/10.1145/3545945.3569820.

[23] Jeff Matocha. 2002. Laboratory experiments in an algorithms course: Technical writing and the scientific method. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference (FIE '02)*, pp. T1G 9-13.

[24] Michael McCracken, Yifat B.-D. Kolikant, Vicki Almstrum, Cary Laxer, Danny Diaz, Linda Thomas, Mark Guzdial, Ian Utting, Dianne Hagan, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33, 4, 125-140. https://doi.org/10.1145/572133.572137.

[25] Catherine C. McGeoch. 2012. *A Guide to Experimental Algorithmics*. Cambridge University Press.

[26] Thomas L. Naps, Guido Roessling, Vicki Almstrum, Wand Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger and J. Ángel Velázquez-Iturbide. 2003. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin* 35, 2, 131-152. https://doi.org/10.1145/960568.782998.

[27] Donald Norman. 1983. Some observations on mental models. In *Mental Models*. D. Gentner and A. Stevens (eds), pp. 7-14. Erlbaum, Hillsdale, NJ.

[28] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education* 18, 1, article 1 (Oct. 2017), 24 pages.

[29] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education, 2010.

[30] Ian Sanders. 2002. Teaching empirical analysis of algorithms. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*, pp. 321-325. https://doi.org/10.1145/563517.563468.

[31] Sartaj Sahni. 2004. *Data Structures, Algorithms, and Applications in Java* (2nd ed.). Silicon Press, Summit, NJ.

[32] Clifford A. Shaffer. 2016. OpenDSA: An interactive e-textbook for computer science courses. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*, 5.

[33] Michael Shindler, Michael T. Goodrich, Ofek Gila, and Michael Dillencourt. 2022. Beyond big O: Teaching experimental algorithmics. *Journal of Computing Sciences in Colleges*, 37(10):23-36

[34] Michael Shindler, Natalia Pinpin, Mia Markovic, Frederick Reiber, Jee Hoon Kim, Giles Pierre Nunez Carlos, Mine Dogucu, Mark Hong, Michael Luu, Brian Anderson, Aaron Cote, Matthew Ferland, Palak Jain, Tyler LaBonte, Leena Mathur, Ryan Moreno & Ryan Sakuma. 2022. Student misconceptions of dynamic programming: a replication study. *Computer Science Education*, 32, 3, 288-312. https://doi.org/10.1080/08993408.2022.2079865.

[35] Juha Sorva, Ville Karavirta Lauri and Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13, 4, article 15 (Nov. 2013), 64 pages. http://doi.org/10.1145/2490822.

[36] John Stasko, John Domingue, Marc H. Brown and Blaine A. Price, eds. 1998. *Software Visualization*. MIT Press, Cambridge, Massachusetts, MA.

[37] J. Ángel Velázquez-Iturbide, 2011. The design and coding of greedy algorithms revisited. In *Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM Press, New York, NY, 8-12. https://doi.org/10.1145/1999747.1999753.

[38] J. Ángel Velázquez-Iturbide. 2013. An experimental method for the active learning of greedy algorithms. *ACM Transactions on Computing Education*, 13, 4, article 18 (Oct. 2013), 23 pages. https://doi.org/10.1145/2534972.

[39] J. Ángel Velázquez-Iturbide. 2013. Using textbook illustrations to extract design principles for algorithm visualizations. In *Handbook of Human Centric Visualization*, W. Huang (ed.), pp. 227-249. Springer Science+Business Media.

[40] J. Ángel Velázquez-Iturbide. 2021. A unified framework to experiment with algorithm optimality and efficiency. *Computer Applications in Engineering Education*, 29, 6, 1,793-1,810. https://doi.org/10.1002/cae.22423.

[41] J. Ángel Velázquez-Iturbide and Antonio Pérez-Carrasco. 2010. InfoVis interaction techniques in animation of recursive programs. *Algorithms*, 3, 1, 76-91. https://doi.org/10.3390/a3010076.

[42] J. Ángel Velázquez-Iturbide and Antonio Pérez-Carrasco. 2016. Systematic development of dynamic programming algorithms assisted by interactive visualization. In *Proceedings of the 21st Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'16)*. ACM Press, New York, NY, 71-76. 10.1145/ 2899415.2899450.

[43] J. Ángel Velázquez-Iturbide and Antonio Pérez-Carrasco. 2016. How to use the SRec visualization system in programming and algorithm courses. *ACM Inroads*, 7, 3, 42-49. https://doi.org/10.1145/2948070.

[44] J. Ángel Velázquez-Iturbide. 2019. Students' misconceptions of optimization problems. In *Proceedings of the 24th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM Press, New York, NY, 464-470. https://doi.org/10.1145/3304221.3319749.

[45] Shamama Zehra, Aishwarya Ramanathan, Larry Yueli Zhang, and Daniel Zingaro. 2018. Student misconceptions of dynamic programming. In *Proceedings of the 49th Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, 21-26. https://doi.org/10.1145/3159450.3159528.