International Journal of Engineering Education Vol. 33, No. 2(B), pp. 908–917, 2017
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2017 TEMPUS Publications.

Using the SRec Visualization System to Construct Dynamic Programming Algorithms*

J. ÁNGEL VELÁZQUEZ-ITURBIDE and ANTONIO PÉREZ-CARRASCO
Departamento de Informática y Estadística, Escuela Técnica Superior de Ingeniería Informática, Universidad Rey Juan Carlos 28933 Móstoles, Madrid, España. E-mail: {angel.velazquez,antonio.perez.carrasco}@urjc.es

Dynamic programming is a demanding algorithm design technique. In this article, we introduce an extension of the recursion visualization system SRec, intended to support dynamic programming. The contributions of the chapter are threefold. Firstly, we present SRec support to several phases of the systematic development of dynamic programming algorithms: generation of recursion trees, checking recursion redundancy in a recursion tree, generation of the dependency graph associated to a recursion tree, and matching the graph to a table. These facilities require high degree of interactivity to be effective. The article illustrates these facilities with the construction of a dynamic programming algorithm for the 0/1 knapsack problem. Secondly, we address several pragmatic issues: usage in educational scenarios, our experience with dynamic programming algorithms, and limitations. Thirdly, the article reports on the results of an evaluation of the system usability. The results were very positive, providing evidence on the adequateness of extensions. Furthermore, they allowed identifying minor opportunities for improvements.

Keywords: algorithms; multiple recursion; dynamic programming; program visualization; human-compter interaction; SRec

1. Introduction

Visualization means creating a mental image of something not actually present to the sight. It is a process especially useful for abstract entities, such as software. There are many forms of software visualization. According to Price et al. [1], "program visualization is the visualization of actual program code or data structures in either static or dynamic form", whereas "algorithm visualization is the visualization of the higher-level abstractions which describe software". A large number of visualization and animation systems were developed in the last two decades [2]. A number of issues regarding the educational use of software visualization have also been addressed. Thus, the form of the learning activity in which visualizations are used has been acknowledged to be the most important feature for educational effectiveness [3] (although no definitive conclusions seem to exist on how to achieve such an effectiveness [4]).

Although visualization systems may be studied from many points of view, one distinctive issue is their scope [1]. Some systems are general purpose while others have a limited scope. Many systems or libraries were built to support different classes of algorithms. Some of them are algorithms operating on specific data structures, such as trees (e.g. [5]), graphs (see a partial review in [6]) or strings (e.g. [7]). Other systems allow illustrating either algorithms that solve a specific problem (such as sorting [8]) or a class of algorithms (such as geometric algorithms [9]). Finally, a few systems provide specific visualizations for an algorithm design technique, such as the greedy [10], branch-and-bound [11], or divide-and-conquer [12] techniques.

Algorithm design techniques are very important for algorithm instruction because they offer a criterion to structure algorithm courses that is more general than solving relevant but particular problems. A *de facto* consensus exists about the most important design techniques. If we browse well-known textbooks (e.g. [13–15, 16]), they unanimously include chapters devoted to three algorithm design techniques (namely, divide and conquer, greedy algorithms, and dynamic programming), as well as chapters on other techniques. Dynamic programming is a technique to solve optimization problems, being the most complex of the abovementioned techniques.

The article presents an extension of the recursion visualization system SRec [17], intended to support the construction of dynamic programming algorithms. The article is structured as follows. In section 2, we present background for the work here presented. Section 3 describes the extensions of SRec, illustrated with the development of a dynamic programming algorithm for the 0/1 knapsack problem. Section 4 presents several issues regarding the educational use of SRec for the dynamic programming technique. In Section 5 we describe an evaluation of usability conducted over SRec. Finally, we present our conclusions and identify lines for future work.

* Accepted 13 December 2016.

2. Background

We first introduce the system SRec and we then summarize how to design dynamic programming algorithms.

2.1 The SRec system

SRec is a system aimed at displaying recursive processes coded in Java [17]. It provides several graphical representations: traces, the control stack and, above all, recursion trees. Recursion is a feature found at the syntactic level of programming languages, therefore we may classify SRec as a program visualization system. However, the high abstraction level of a recursive invocation (compared to other statements, such as an assignment) results in that, for many algorithms, the visualization of their recursive behavior shows the most important events of the algorithm execution. Consequently, this kind of program visualization is often at the level of abstraction of algorithm animations.

From the point of view of a user of SRec, he/she may write a Java program and run it; as a side effect, SRec automatically generates a visualization of the recursive processes occurring during the program execution. The user typically uses SRec by iteratively performing the following process: load a file—select a method—launch an execution—interact with the visualizations generated.

Interaction with visualizations is a key element for students' engagement and for the usefulness of SRec as a tool for understanding and analysis. The simplest interaction is the (manual) animation of an algorithm execution, which displays how the algorithm visualization varies as the execution advances (forward or backward). SRec also provides other ways of interacting with a visualization [18]: change the graphical properties of the visualization components, filter the amount of data to display, change the relative order of data, browse a large visualization, look for specific data in a visualization, and give statistics about a visualization. In the rest of the paper, we include several figures that illustrate some of these interactions.

Fig. 1 shows the user interface of SRec, where the editor panel was collapsed to leave more room for the two visualization panels. The lower panel contains a collection of fourteen recursion trees, where the framed tree is displayed in the upper panel on a larger scale. The upper panel displays a recursion tree through a global+detail interface [21]. This interface is composed of two views, global and detail, which occupy the lower and the higher part of the panel, respectively. The contents of the global view of the tree are unreadable but its shape can be distinguished. A part of the tree displayed in the global view is framed and is displayed in the detail view with larger resolution. The resulting interface allows both navigating the whole visualization and examining selected parts in detail.

SRec also provides several educational facilities, including the exportation of a visualization to a graphical file. About half the figures contained in the paper were obtained using the export function.

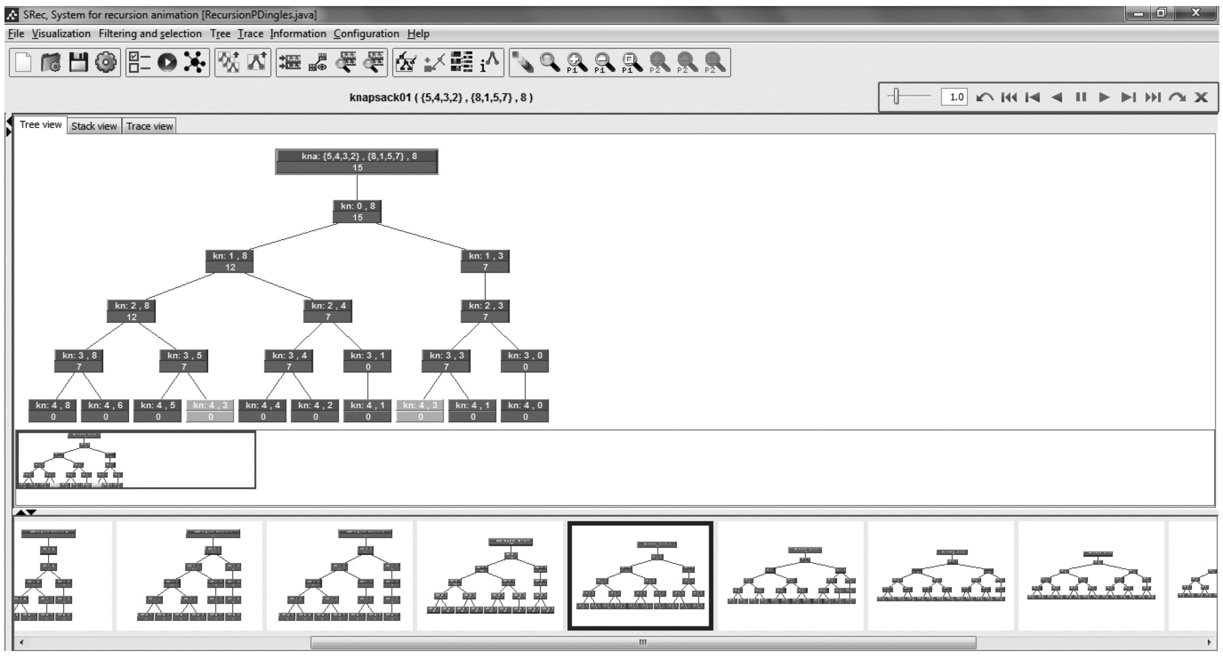


Fig. 1. Capture of the SRec user interface, showing a subset of 14 recursion trees for the 0/1 knapsack problem.

2.2 Dynamic programming

Dynamic programming algorithms are obscure algorithms, very difficult to understand. They are not too complex, as they are iterative algorithms that compute values and store them in tables. However, their rationale is difficult to grasp. Their development can be simplified by decomposing it into a series of phases. In particular, we may follow a methodology formed by four phases [14, chap. 15] [16, chap. 20]:

(1) Characterize the structure of an optimal solution.
(2) Develop a recursive algorithm that computes an optimal value in a top-down fashion.
  (a) Check the redundancy of the recursive algorithm using a recursion tree.
(3) Develop an equivalent, iterative algorithm that computes an optimal value in a bottom-up fashion. In turn, this phase may be decomposed into several steps:
  (a) Analyze the redundancy pattern after converting the recursion tree into a dependency graph.
  (b) Design a table capable to store the value of all the subproblems (that is, the results of the different recursive calls).
  (c) Design an iterative algorithm that computes all the subproblems without redundancy, preserving their dependencies and using a table to store their results.
(4) Extend the iterative algorithm to determine the decisions associated to the optimal value computed.

Steps of phase 3 are explained in full detail in technical publications [19, 20]. We focus here on phases 2(a) and 3(a–b), where a redundant, recursive algorithm is handled.

We illustrate the features of SRec with the 0/1 knapsack problem [13, chap. 8] [15, cap. 6] [16, chap. 20]. Consider a set of objects and a knapsack. Each object is characterized by a weight w_i and a profit p_i. The knapsack has limited capacity c. Each object i can be either introduced (reducing the spare capacity s of the knapsack, $0 \leq s \leq c$, in w_i) or not introduced into the knapsack. In the former case, the object must have weight w_i less or equal to the spare capacity s of the knapsack and, as a consequence, an associated profit p_i is gained. The problem is to determine a subset of the objects that produces maximum-profit while it does not overload the knapsack capacity.

Assume the n objects are numbered from 0 to n--1. We define $kn(i,s)$ as the subproblem consisting in determining a maximum-profit subset of objects $i..n$-1 provided the spare knapsack capacity is s. Obviously, the complete problem can be solved by the initial call $kn(0,c)$.

Given this modeling of the problem, the following recursive algorithm solves it:

$$kn(n,s) = 0 \qquad \textit{for } 0 \leq s \leq c$$

$$m(i,s) = \begin{cases} kn(i+1,s) & \textit{if } s < w_i \\ \max(kn(i+1,s), kn(i+1,s-w_i)+p_i) & \textit{if } s \geq w_i \end{cases} \quad \textit{for } \; 0 \leq i \leq n-1, \;\; 0 \leq s \leq c$$

Assuming that weights and profits have integer values, it can be coded in Java as follows:

```
public static int knapsack01 (int[]weights, int[] profits, int capacity){
  return knapsackAux(weights,profits,capacity,0,capacity);
}
private static int knapsackAux (int[] weights, int[] profits, int capacity,
                                int i, int spare) {
  if (i==weights.length)
    return 0;
  else if (spare<weights[i])
    return knapsackAux (weights,profits,capacity,i+1,spare);
  else
    return
      Math.max (knapsackAux (weights,profits,capacity,i+1,spare),
                knapsackAux (weights,profits,capacity,i+1,spare-weights[i])+profits[i]);
}
```

where *knapsack01* is the main method and *knapsackAux* is an auxiliary, recursive method.

3. Development of a dynamic programming algorithm

In this section, we present SRec support to four steps of the development methodology described in section 2.2: generation of an adequate recursion tree, redundancy analysis of the recursion tree, transformation of the recursion tree into a dependency graph, and laying the graph nodes out in the cells of a table.

3.1 Generation of recursion trees

A previous step to checking redundancy in a recursive algorithm is to generate an adequate recursion tree. Note that each recursion tree is bound to a different test case. Some test cases, especially for small input data, do not exhibit redundancy. Other cases may be too large to be easily understood and analyzed. A good approach would be to generate simultaneously a set of recursion trees and selecting a "good" one. SRec allows generating such a set in an atomic operation. The SRec dialog to launch an execution was modified so that several values can be specified as input data. As a consequence, several executions of the algorithm are launched and a visualization is displayed for each test case.

There are two ways of giving several values for a parameter:

- Specifying several values, separated by commas.
- Specifying a range of values, using the syntax 'lower value .. higher value. This syntax is only allowed for integer values.

If several values are given to several parameters, the Cartesian product of all the values is computed, launching as many algorithm executions as different cases result.

Consider again the recursive algorithm proposed in Section 2.2 for the 0/1 knapsack problem. The parameters that control the recursive process are the index and weight of objects, and the spare knapsack capacity. The resulting recursion trees will probably not be too large with four objects. We may generate a number of similar but differing recursion trees as follows. Profit values are irrelevant for recursion, so they can be given arbitrary values (e.g. {8,1,5,7}). Weights of objects may differ in just one unit and may be arranged in decreasing order (e.g. {5,4,3,2}). If these parameters have the same value in a number of executions but the knapsack capacity successively varies by one unit, their corresponding trees will show incremental changes. The minimum value for the knapsack capacity that makes sense is the highest value for which no object can be introduced (for the weight values given above, capacity 1); the maximum sensible value will be equal to the sum of the object weights (i.e. 14). Therefore, fourteen cases can be used to represent a complete range of situations for this instance of the 0/1 knapsack problem.

Fig. 1 shows the result of invoking *knapsack01* with the 14 test cases designed in the previous paragraph, where the tree displayed in the upper panel corresponds to the case of the knapsack capacity equal to 8. Each node of a recursion tree hosts input values in its higher half and output values in its lower half. Actually, such a visualization is the result of first generating automatically a visualization and then the user performing several operations on the visualization. The first operation was to filter some parameters. In particular, the three parameters of the main method were filtered in the auxiliary method because their value does not change from call to call. Therefore, they are only displayed in the call to the main method *knapsack01*, yielding a more compact and readable visualization. A second interaction was to zoom the resulting visualization to an exact fit of the panel. Finally, the third operation performed was looking for occurrences of redundant calls. This operation is intended for the step described in the following subsection.

3.2 Redundancy analysis

Some multiple recursive algorithms are asymptotically very efficient, such as divide-and-conquer sorting algorithms (i.e. mergesort and quicksort) [13, chap. 7] [14, chaps. 4 and 7] [15, chap. 5] [16, chap. 19]. However, other multiple recursive algorithms are very inefficient. This happens when recursive calls do not represent independent but overlapping subproblems. Consequently, many calls are invoked more than once, recomputing their value in each invocation. Recursive algorithms designed for dynamic programming correspond to the latter class of redundant algorithms.

SRec makes the analysis of redundancy easier with a function that allows searching recursive calls. The user specifies input values in a dialog and SRec highlights all the nodes in the recursion tree that match the search criterion. The search function is very flexible, as it allows specifying the value of only some parameters or even of output values. A complementary function restores the original colors to nodes highlighted. As explained in the last paragraph of the previous section, Fig. 1 includes the results of searching occurrences of *knapsackAux* (4,3). These nodes are highlighted in both the global view and the detailed one of the top panel (in the figure, in light tone). Alternatively, the user may select a node of the tree and, using the mouse right button, command to highlight nodes that contain the same values as the selected node.

3.3 Dependency graphs

After checking that a recursive algorithm is computed redundantly, redundancy must be removed. A number of techniques exist for this goal, being tabulation [19] the most common in dynamic programming algorithms.

The resulting algorithms are iterative algorithms that store the value of the different subproblems in a table. The iterative algorithm solves subproblems in a sequential order that preserves the dependencies of the original recursive algorithm.

Dependencies among subproblems can be determined by first generating a dependency graph [19, 20], i.e. an acyclic directed graph. The dependency graph associated to a recursion tree is built by joining all the occurrences of each call in a single node, preserving arcs between calls.

Fig. 2(a) shows the dependency graph generated automatically by SRec from the recursion tree displayed in the upper panel in Fig. 1. Note that nodes are distributed in an apparently arbitrary way.

SRec allows the programmer to rearrange the nodes and try to identify some redundancy pattern. Fig. 2(b) shows the result of relocating the nodes so that nodes sharing the value of the first parameter are placed in the same column. It can be noticed that each node always depends on one or two nodes placed at its right.

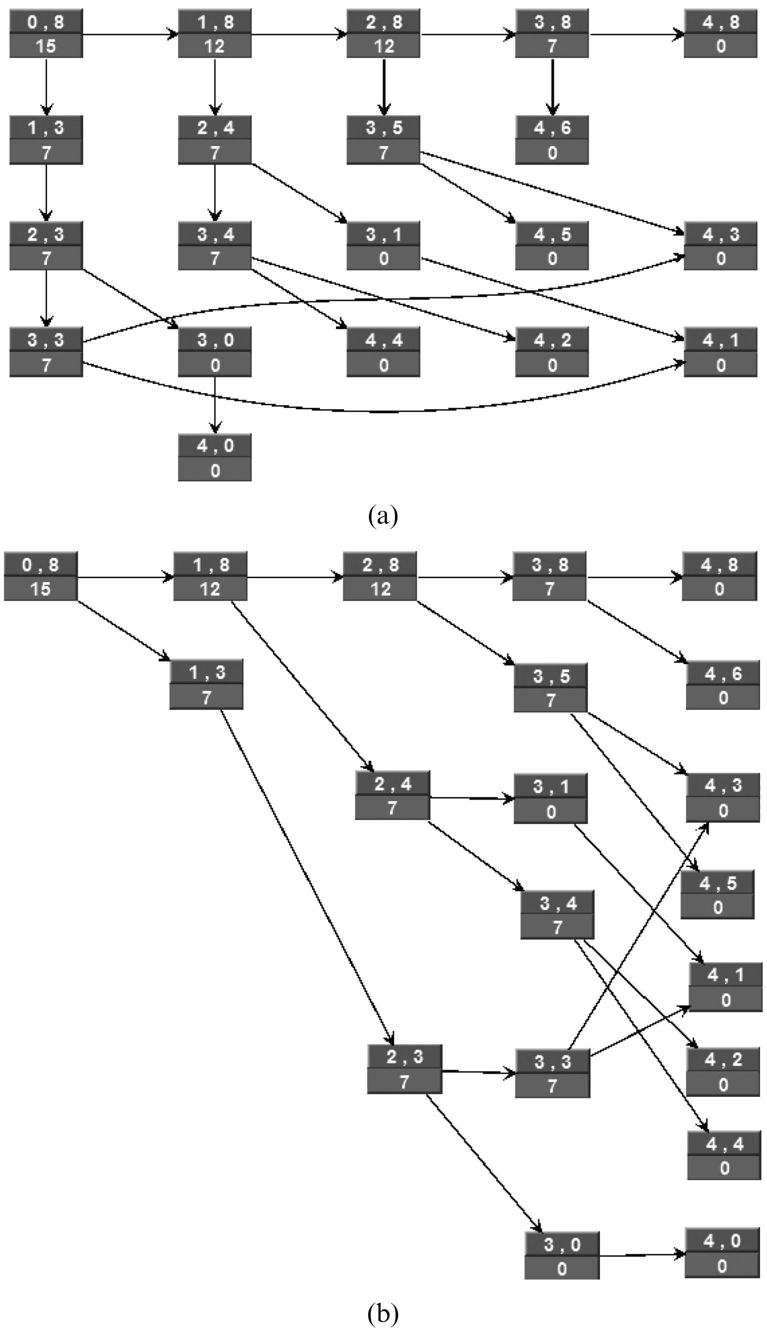
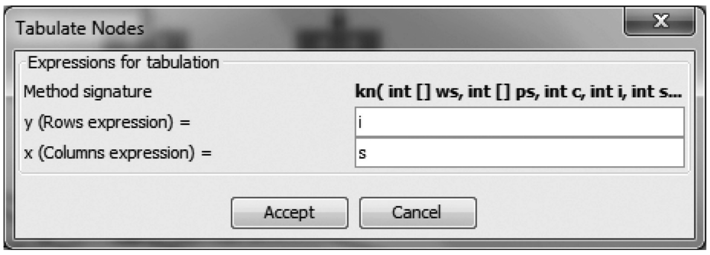


Fig. 2. Dependency graph for the 0/1 knapsack problem obtained from the recursion tree displayed in Fig. 2 (a) automatically (b) after manually rearranging the nodes.



Expressions for tabulation

Method signature kn(int [] ws, int [] ps, int c, int i, int s...

y (Rows expression) = i

x (Columns expression) = s

Fig. 3. Dialog to specify by means of expressions the mapping between the method parameters and the table cells for the 0/1 knapsack problem.

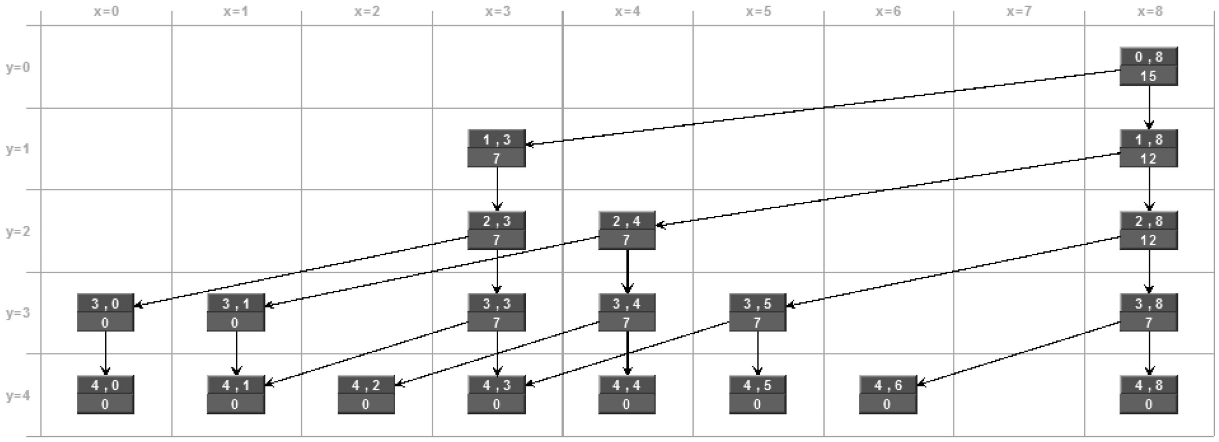


Fig. 4. Table generated for the 0/1 knapsack problem by means of mapping $y = i$, $x = s$.

3.4 Tabulation

Once the dependency pattern of recursive calls is known, the programmer must determine a sequential order of computation of the subproblems that preserves dependencies, and must design a table adequate to store the values of all the subproblems. SRec also gives support to this task.

The programmer may make SRec to lay the dependency graph out on a one- or bi-dimensional table, where he/she must only specify its dimensions. However, the result is often poorly constructive.

An alternative function allows the user to specify with expressions a matching between recursive calls and table cells. Fig. 3 shows the dialog for this function, where parameter *i* is associated to each row *y* of the table, and parameter *s* is associated to each column *x*. In general, the dialog allows specifying how to compute the table cell corresponding to each recursive call. SRec determines automatically the dimensions of the table necessary to store all the calls.

Figure 4 shows the result of the previous dialog. Note that we have generated a dependency graph that is rotated 90^o right that that displayed in Fig. 2(b).

Finally, the programmer will determine a sequential computation order that preserves the dependencies shown in the table. In this example, subproblems must be computed by rows bottom-up. However, cells in a row can be indistinctly computed either from left to right or right to left. If we choose to proceed right to left within each row, the following iterative algorithm can be coded:

```
public static int knapsack01 (int[] weights, int[] profits, int capacity) {
    int[][] table = new int[ws.length+1][c+1];
    for (int s=0; s<=c; s++)
      table[ws.length][s] = 0;
    for (int i=ws.length-1; i>=0; i--) {
      for (int s=c; s>=ws[i]; s--)
        table[i][s] = Math.max(table[i+1][s], table[i+1][s-ws[i]]+ps[i]);
      for (int s=Math.min(c,ws[i]-1); s>=0; s--)
        table[i][s] = table[i+1][s];
    }
    return table[0][c];
  }
```

4. Experience

We have used the new version of SRec since autumn of the academic year 2015/16 in the optional fourth-year course "Advanced Algorithms" offered at our University to Computer Science students. SRec was used by the instructor to prepare materials and in the classroom. In addition, students could use the tool for their assignments, either to document their designs with figures or as a development tool.

The facilities described in this article were used for the dynamic programming technique. Actually, it was incrementally addressed in two steps:

1. Removal of redundant recursion. The methodology outlined in Section 2.2 was presented in detail and exercised. It was adapted to derive either tabulated or memoized algorithms. We focused on numeric algorithms, including:
   - Fibonacci series
   - Recursive definition of combinatorial numbers.
   - Problem of the sporting competition [13, chap. 8].
2. The dynamic programming technique. Emphasis was given to the design of recursive algorithms for optimization algorithms, a non-trivial task. Recursion removal was exclusively accomplished using tabulation. A number of algorithms were handled, including the most common in algorithm textbooks:
   - 0/1 knapsack, which is the problem used to illustrate this paper.
   - Coin change [13, chap. 8].
   - Sequence alignment [15, chap. 6].
   - Longest common subsequence [14, chap. 15].
   - Matrix-chain multiplication [13, chap. 8] [14, chap. 15] [16, chap. 20].
   - Multistage graph [22, chap. 5].
   - Single-source shortest path (Bellman-Ford's algorithm [14, chap. 24] [15, chap. 6] [16, chap. 20]).

We detected a limitation of SRec for generating multiple recursion trees for graph problems. The amount of memory necessary to store a detailed description of each execution step exhausts the memory available in a laptop. Therefore, one single animation can be launched each time.

We must also note that the conventional, bi-dimensional representation of tables restrict SRec to algorithms with one- or bi-dimensional tables. Consequently, we cannot handle the recursive algorithm for the all-pairs shortest paths problem (Floyd's algorithm [13, chap. 8] [14, chap. 25] [16, chap. 20]), which has three varying parameters and therefore needs a tri-dimensional table in a first approach. The bi-dimensional matrix used by Floyd's algorithm can only be deduced in a later step of the methodology, after analyzing dependencies and invariant computations in the tri-dimensional table.

5. Usability evaluation

In our course "Advanced algorithms", students must solve one assignment per chapter of the syllabus. We made use of the assignment on recursion removal to evaluate SRec usability in the academic year 2015/16. Students attended at a laboratory session, which was two hours long. Given a redundant algorithm, students had to convert it into two efficient algorithms (a tabulated algorithm and a memoized one). Students were given a report outline, and had one week to complete the work and submit it to the instructor through the virtual campus. However, they also had to submit a (partial) report at the end of the laboratory session.

The instrument to measure usability was a questionnaire, which students had to fill at the end of the laboratory session. The questionnaire consisted of three parts:

- Multiple-choice questions on general properties. They assessed six general claims about SRec.
- Multiple-choice questions on specific elements. They assessed the quality of fourteen elements of SRec.
- Open questions on general issues. It comprises six open questions about positive and negative features of SRec.

Answers to multiple-choice questions were in a Likert scale ranged from 1 (very bad) to 5 (very good). We gathered 13 questionnaires.

The first part of the questionnaire asked students to give their opinion in six multiple-choice questions, directly related to ease of use, satisfaction, and perceived utility and quality. Table 1 shows the questions and the mean and standard deviation of their scores. (The median is a most adequate measure for ordinal values, but we show the mean for a more intuitive perception of students' opinions.)

Notice that the marks obtained for all the properties are very high. The highest rated property is ease of use;

Table 1. Results to multiple-choice questions on general properties

Property	Mean	Std. deviation
Easy to use	4.77	0.44
If the user liked the tool	4.46	0.66
Useful to understand the behavior of the recursive algorithm	4.46	0.52
Useful to analyze redundancy in the recursive algorithm	4.31	0.85
Useful to design a table adequate to remove redundancy	4.08	1.04
General quality	4.08	0.95

Table 2. Results to multiple-choice questions on the quality of specific elements of SRec

Element	Mean	Std. deviation
Generation process of animations	4.56	0.73
Animation controls	4.45	0.69
Generation and handling of dependency graphs	4.38	0.65
Export visualizations	4.38	0.77
Recursion tree view	4.31	0.85
Tabulation of nodes	4.14	0.69
Control of the amount of information to display	4.09	0.70
Organization of menus	3.91	0.70
Control of the visualization graphical format (colors, etc.)	3.82	0.98
Organization and design of icons	3.64	0.50
Interaction with panels	3.50	1.35
Zoom control	3.50	0.97
Search and highlight of nodes	3.36	1.43
Explanation of compilation/run-time errors	3.14	1.21

actually, this property was rated 4 or 5 by all the students. They were also rated high: satisfaction (4.46) and usefulness to understand or analyze recursive algorithms (4.46 and 4.31). The lowest grade was obtained for its usefulness to design tables and for its overall quality. These two factors also obtain the highest standard deviation.

Students were also asked about the quality of specific elements of the system. The results are show in Table 2, in decreasing order of mean.

Notice that most elements rated above 4 are related to visualizations, thus they are elements that were refined after past usability evaluations [23]: generation of animations, animation controls, exporting visualizations, the recursion tree view, and control of the amount of information to display. It was a nice surprise that two of the features included in this new version of SRec were also rated high: generation and handling of dependency graphs, and tabulation of nodes.

Medium-high rates were obtained for three elements of the user interface (menus, icons, and panels) and two functions to interact with visualizations (graphical format and the zoom). Finally, two medium rates are obtained for two features: search and highlight of redundant nodes, and explanation of errors. All of these elements should be reviewed.

A third part of the questionnaire contained six open questions about positive and negative aspects. We analyzed them using a qualitative methodology starting with no a priori categories, as grounded theory advocates [24]. We followed three steps. First, we prepared answers for their analysis. Afterwards, we performed two rounds of analysis and identification of categories.

Answers given were not directly usable for analysis. Some answers had no informative value (e.g. "I did not use the tool enough to find any negative feature"), other answers were composed of simpler suggestions or comments, and others were inadequate for the corresponding question. Therefore, in a first step we re-catalogued the answers as "simple" answers to only four questions. The results are shown in Table 3.

Table 3. Results to open questions, after being analyzed and re-catalogued

Question	Number of informative simple answers
Positive features of SRec	13
Useless features that you would discard	3
Negative features of SRec	14
Useful features to be included	5

Thirteen answers were gathered that identified positive features of SRec. They were of three classes:

- Usefulness to understand recursive algorithms (5 answers). Two representative quotations follow: "It makes much more definite the way of building the tree and the recursion produced in it", "The behavior of algorithms is watched in a visual way and that makes their understanding easier".
- Ease of use (3 answers). One quotation: "Very definite and intuitive. Easy".
- Other features (5 different answers).

There only were three answers that identified elements that could be suppressed or simplified (zooming, animation control, and configuration of visualizations). We do not consider these answers relevant as such elements provide comprehensive and useful functions.

The third category of answers are about negative features of SRec, which must be considered suggestions for improvement. From the fourteen answers gathered, the following ones were given by more than one student:

- Poor performance with large recursion trees (3 answers). One representative answer follows: "Slow for relatively large trees. It sometimes gets hanged".
- User interface (2 answers).
- Icons (2 answers).
- Configuration of visualizations (2 answers).
- Redundancy analysis (2 answers).

The first comment has no remedy, as poor asymptotic behavior of an algorithm is independent from any particular implementation. However, the other suggestions deserve closer examination.

Finally, five comments were gathered demanding including new features. The only one suggested by several (two) students was to generate the iterative algorithm from the table.

6. Conclusions

The article presents several extensions of the system SRec intended to support the development of dynamic programing algorithms. We may highlight several issues. Firstly, we do not know of any program visualization system that visually supports the construction of algorithms. Secondly, the extensions included dealt with different graphical representations. Some extensions provided additional operations on recursion trees, whereas others introduced new graphical representations (namely dependency graphs and tables). Thirdly, a key issue in the adequate support of SRec to the different tasks is interactivity. In the article, we have shown the need for dynamic programming algorithms of, at least, operations to filter data, zoom visualizations, navigate very large visualizations, search nodes, and rearrange nodes. Fourthly, we conducted a usability evaluation that proved that the approach used to implement these extensions was adequate. The usability evaluation conducted yielded very high results to the extensions. Different suggestions for improvement are being analyzed to build a new version of the system.

There still is room for additional extensions of SRec regarding dynamic programming algorithms. In particular, it would be very useful to support the (probably semiautomatic) generation of iterative algorithms from tables.

Acknowledgments—This work was supported by research grants TIN2011-29542-C02-01 and TIN2015-66731-C2-1-R of the Ministry of Economy and Competitiveness, S2013/ICE-2715 of the Regional Government of Madrid, and 30VCPIGI15 of the Universidad Rey Juan Carlos.

7. References

1. B. Price, R. Baecker and I. Small, An introduction to software visualization, in J. Stasko, J. Domingue, M. H. Brown and B. A. Blaine (eds), *Software Visualization*, MIT Press, 1998, pp. 3–27.
2. C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce and S. H. Edwards, Algorithm Visualization: The State of the Field, *ACM Transactions on Computing Education*, **10**(3), 2010, article 9.
3. C. Hundhausen, S. Douglas and J. Stasko, A meta-study of algorithm visualization effectiveness, *Journal of Visual Languages and Computing*, **13**(3), 2002, pp. 259–290.
4. J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide, A survey of successful evaluations of program visualization and algorithm animation systems, *ACM Transactions on Computing Education*, **9**(2), 2009, article 9.
5. G. Roessling and S. Schneider, An integrated and "engaging" package for tree animations, *Proceedings of the Third Program Visualization Workshop*, A. Korhonen (ed.), University of Warwick, Department of Computer Science, Research Report CS-RR-707, 2004, pp. 23–28.
6. J. Á. Velázquez-Iturbide, O. Debdi and M. Paredes-Velasco, A review of teaching and learning through practice of optimization algorithms, in R. Queirós (ed.), *Innovative Teaching Strategies and New Learning Paradigms in Computer Programming*, IGI Global, 2015, pp. 65–87.
7. R. Baeza-Yates and L. O. Fuentes, A framework to animate string algorithms, *Information Processing Letters*, **59**, 1996, pp. 241–244.

8. D. Furcy, T. Naps and J. Wentworth, Sorting Out Sorting—The sequel, in *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008*, pp. 174–178.
9. A. Tal and D. Dobkin, Visualization of geometric algorithms, *IEEE Transactions on Visualization and Computer Graphics*, **1**(2)**,** 1995, pp. 194–204.
10. J. A. Velázquez-Iturbide, O. Debdi, N. Esteban-Sánchez and C. Pizarro, GreedEx: A visualization tool for experimentation and discovery learning of greedy algorithms, *IEEE Transactions on Learning Technologies*, **6**(2), 2013, 130–143.
11. K. V. Ramani and T. P. Rama Rao, A graphics based computer-aided learning package for integer programming: The branch and bound algorithm, *Computers & Education*, **23**(4), 1994, 261–268.
12. J. Á. Velázquez-Iturbide, A. Pérez-Carrasco and J. Urquiza-Fuentes, A design of automatic visualizations for divide-and-conquer algorithms, *Electronic Notes in Theoretical Computer Science*, **224**, 2009, 159–167.
13. G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
14. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009.
15. J. Kleinberg and É. Tardos, *Algorithm Design*, Pearson Addison-Wesley, 2006.
16. S. Sahni, *Data Structures, Algorithms and Applications in Java*, Silicon Press, Summit, NJ, 2005.
17. J. Á. Velázquez-Iturbide, A. Pérez-Carrasco and J. Urquiza-Fuentes, "SRec: An animation system of recursion for algorithm courses, in *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008*, pp. 225–229.
18. J. Á. Velázquez-Iturbide and A. Pérez-Carrasco, InfoVis interaction techniques in animation of recursive programs, *Algorithms*, **3**(1), 2010, pp. 76–91.
19. R. S. Bird, Tabulation techniques for recursive programs, *ACM Computing Surveys*, **12**(4), 1980, pp. 403–417.
20. A. Pettorossi, A powerful strategy for deriving efficient programs by transformation, in *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1984, pp. 273-281.
21. A. Cockburn, A. Karlson and B. B. Bederson, A review of overview+detail, zooming, and focus+context interfaces, *ACM Computing Surveys*, **41**(1), 2008, article 2.
22. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Pitman, 1978.
23. J. Á. Velázquez-Iturbide, A. Pérez-Carrasco and O. Debdi, Experiences in usability evaluation of educational programming tools, in C. González (ed), *Student Usability in Educational Software and Games*, IGI Global, 2013, pp. 241–260.
24. B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine, Chicago, 1967.

J. Ángel Velázquez-Iturbide received the Computer Science degree and the Ph.D. degree in Computer Science from the Universidad Politécnica de Madrid, Spain, in 1985 and 1990, respectively. In 1985 he joined the Facultad de Informática, Universidad Politécnica de Madrid. In 1997 he joined the Universidad Rey Juan Carlos, where he is currently a Professor, as well as the leader of the Laboratory of Information Technologies in Education (LITE) research group. His research areas are software and educational innovation for programming education, and software visualization. Prof. Velázquez is a member of IEEE Computer Society and IEEE Education Society, and a member of ACM and ACM SIGCSE. He is the Chair of the Spanish Association for the Advancement of Computers in Education (ADIE).

Antonio Pérez-Carrasco received the Computer Science degree and the Ph.D. degree in Computer Science from the Universidad Rey Juan Carlos (URJC), Madrid, Spain, in 2008 and 2011, respectively. He was an Assistant Professor in the Universidad Rey Juan Carlos from 2008 to 2013, where he was a member of the Laboratory of Information Technologies in Education (LITE) research group. He is an Assistant Professor in the Universidad Internacional de La Rioja (UNIR) since 2012. That year, he also joined SICE, an ACS company, where he is currently working as an IT Project Manager on Intelligent Transport System projects.