Using Large-Scale Optimality Testing as a Tool for Analysis Tasks in Algorithm Courses

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

Abstract—**This "innovative practice" full paper proposes large-scale optimality testing as an alternative means in algorithm courses to address different educational goals. Testing is an engineering activity which is commonly conducted in an informal way in programming and algorithm courses, as well as in a more structured way in software engineering courses. However, the number of test cases commonly used is small. Large-scale testing refers to the inclusion of large numbers of test cases. We have presented in past works how to use it as a didactic tool for greedy and other optimization algorithms. We present here its use with more sophisticated aims: as a debugging tool, to call students' attention to problem preconditions, and to find out counterexamples of the optimality of inexact algorithms. The three innovations are illustrated by their use in the academic year 2019/20 in an algorithms course, with mixed results. We analyze the lessons learnt and outline future lines of instructional intervention.**

Keywords—algorithms, optimality testing, analysis, debugging, precondition, counterexamples

I. Introduction

Testing a program with a small, selected set of test cases is a common activity in programming courses. Instructors and students test programs routinely, usually in an informal way. More structured forms of testing can be found in program assessment systems [1]−[4]. The typical way of proceeding is as follows. The instructor delivers a problem statement, illustrated with one or a few examples. Students must construct a program intended to satisfy the problem statement and submit it to the automated assessment system. The assessment system runs students' programs, testing them against a small set of test cases and reporting students their score. There may be a number of reasons for failure of students' programs, e.g. not complying with input/output format, compiling or run-time error, or failure with any test case.

A number of issues vary in automated assessment systems [1]−[3]. For instance, an important part of the problem statement is the format of input and output. The system may be somewhat flexible with some elements of input and output (e.g. blanks) but, in essence, students must conform to them. Other educational issues have been addressed and investigated, such as user-friendliness [5], resubmission policies [6], coverage of different types of algorithms [7], etc.

A notable variation in the basic scheme outlined above is that students can be responsible to design their own test cases [8]. This variation is based on test-driven development, which is an instructional approach to programming, where test cases are designed as a part of the problem specification, and prior to program development. Consequently, students are encouraged to reflect on the problem to solve more deeply than if they only had to develop the program.

Testing can be applied in algorithm courses in any of the forms described above. In addition, we find other kinds of testing, intended to assess the quality of algorithms, usually with respect to performance [9], [10], but also with respect to optimality [10], [11]. We find several supporting systems, often under the names of workbenching [9] or benchmarking [10]. Some systems only support specific classes of algorithms [9], [11] while others provide generic support [10], [11].

From both the technical and the instructional points of view, design or generation of test cases is very different in benchmarking systems and in program assessment systems. In the former case, it is common to run algorithms many times for each test case to make sure that their running times are accurate [9], [12]. Furthermore, it is common to have a high number of test cases, with different sizes evenly spaced in a scale. In this paper, we use the term "large-scale testing" to refer to the inclusion of large numbers of test cases.

In previous works, we had used large-scale testing to find empirical evidence of the optimality or suboptimality of different optimization algorithms, most notably greedy algorithms [13]. This paper presents an experience report on three innovative instructional analysis tasks which also are based on large-scale testing. They provide opportunities to relate testing to other parts of the syllabus and reinforce them through instructional tasks. Firstly, supposedly exact algorithms can actually be inexact due to wrong design or implementation. Large-scale testing allows finding anomalous behaviors, thus easing the path to debug them. Secondly, we present the relation between preconditions in problem specifications and random generation of test cases. Making these tasks consistent would call attention to the importance of satisfying preconditions. Thirdly, we use experimentation to find counterexamples of the optimality (i.e. exactitude) of given optimization algorithms. The design of counterexamples is a skill which demands abstraction, thus large-scale testing is intended to assist with empirical evidence in this task.

The structure of the paper follows. In the next section, the basics of large-scale testing, its educational use and computer support are elaborated. Section III presents the three analysis tasks alluded above, jointly with our experience in the academic year 2019/20. Lessons learnt and implications are discussed in Section IV. Finally, we summarize our conclusions and identify future works.

This work was supported by research grants iProg of the MINECO (ref. TIN2015-66731-C2-1-R) and e-Madrid-CM of the CAM (ref. S2018/TCS-4307). The e-Madrid-CM grant is also co-financed by the Structural Funds FSE and FEDER

II. Large-Scale Optimality Testing

In this section, we overview the basics of optimality testing. We illustrate it with the scheduling problem [14, p. 313] which was used in the experience reported in this paper, although the method can be used with any other optimization problem.

A. An Optimization Problem

A team of computer scientists performs different tasks on a weekly basis. To facilitate their internal organization, they classify tasks into low-stress tasks and high-stress tasks. If they perform a low-stress task in the *i*-th week, they will have an income l_i>0, while performing a high-stress task will provide an income h_i>0. In general, high-stress tasks produce higher incomes than low-stress tasks, but it can also be the other way around.

In order to address satisfactorily any task, the team agrees to complete low-stress tasks at any week, but on completing a high-stress task only after one week of rest. The only exception to this constraint is the first week, i.e. any task can be performed in the first week.

A plan for *n* weeks consists in a sequence of *n* choices, where the choice for a given week can be either to perform a low-stress task, to perform a high-stress task, or to rest. A plan is valid if the constraint regarding high-stress tasks is satisfied. The revenue associated to a valid plan is computed as the sum of the corresponding incomes of those low- or high-stress tasks which are completed.

Given a sequence of *n* revenues for low- and high-stress tasks, the problem consists in determining a valid plan that provides the highest profit.

For example, consider incomes for five weeks, where the successive incomes for low-stress tasks are {10,10,10,10,10} and for high-stress tasks are {10,15,15,15,15}. A valid plan with the highest revenue consists in selecting the five low-stress tasks, with an income equal to 10+10+10+10+10=50. Alternatively, the high-stress task could have been performed in the first week. It is straightforward to check that any other valid plan provides lower incomes.

Any algorithm solving this problem must strictly conform to the following header:

```
public static int id (int[] h, int[] l)
```

with the only acceptable variation of the method identifier.

B. Large-Scale Optimality Testing

The basics of large-scale optimality testing lies in running several algorithms with a large number of test cases and comparing their optimality.

For instance, consider we have two heuristic algorithms (named H1 and H2) and a backtracking algorithm (named BK). For the current digression, it is not necessary to give full details of the algorithms. In particular, we do not elaborate on the backtracking algorithm but we know that it is an exact algorithm [15], i.e. an algorithm which always computes an optimal result. However, we sketch the heuristic algorithms for a better understanding of the paper.

H1 is based on comparing every two weeks the incomes obtained with either two consecutive low-stress tasks or a rest and a high-stress task. In other words, choices result from comparing values l_i+l_{i+1} and h_{i+1}.

Consider the previous example. Algorithm H1 chooses two low-stress tasks in weeks 0 and 1 (with income l_0+l_1=10+10=20, greater than h_1=15), makes the same selection in weeks 2 and 3 (again, with revenue 10+10=20) and, finally, it selects the low-stress task in the last week l_4=10. The total revenue amounts to 50.

H2 is a heuristic algorithm based on a different approach. It always selects high-stress tasks in alternative weeks. Its rationale is that, given that high-stress tasks usually provide higher incomes, we had better select them.

Consider again the previous example. H2 chooses three high-stress tasks in weeks 0, 2 and 4 (with incomes l_0, l_2 and l_4 equal to 10, 15 and 15, respectively). The total revenue amounts to 10+15+15=40.

If we compared the revenues computed by the three algorithms, we would note that H1 and the backtracking algorithm compute a value of 50 while H2 yields a value equal to 40.

Based on this single test case, we may claim that H2 is inexact [15]. However, we cannot claim that H1 is an exact algorithm based on a single test case. Actually, there are problems for which we may find inexact algorithms which compute optimal results in a high percentage of the test cases.

Large-scale testing may assist in finding test cases where a heuristic algorithm fails in computing an optimal result. For instance, we randomly generated 1,000 test cases with the two income arrays of length 5. In addition, the contents of the low-stress income array could vary between 5 and 20, while the contents of the high-stress income array could vary between 10 and 30. The result was not unexpected: H1 only computed a maximal value in 14.10% of the test cases, and H2, in 8.90%, whereas the backtracking algorithm computed a maximal value in 100% of the cases.

C. Educational Uses of Large-Scale Optimality Testing

The most obvious educational use of large-scale optimality testing is to determine the inexactitude of some algorithms, typically greedy, heuristic or approximation algorithms. The experimental method explained above can be adopted in a relatively straightforward way. For instance, it has been used to determine which greedy criteria do not always provide optimal results [13].

Furthermore, evidence can be gathered about the potential of some algorithms as exact algorithms. This complementary goal was also addressed in the context of greedy criteria [13]. Note, however, that high evidence of the exactitude of a greedy criterion does not guarantee its exactitude. A formal proof still is necessary.

A second use of large-scale testing is to check predictions of the theory. In particular, it can be checked whether greedy criteria, which have been formally proved to be exact, actually behave with exactitude. Predictions can also be checked for approximation algorithms, whose solutions can be formally proved to differ from optimal solutions in, at most, an absolute amount or a percentage [15].

A third usage of large-scale optimality testing is to explore the behavior of some inexact algorithms, as we do not know how often they are inexact or how much they deviate from optimal results [16].

D. Computer Support: The GreedEx and OptimEx Systems

The didactic activities presented above places a burden to students, as they must develop code for some operations: random generation of test cases, measuring and gathering algorithm outcomes, comparing the outcomes of algorithms, computation of descriptive statistics measures, etc. We developed the GreedEx system [11], [17] to support the educational activity described above for greedy algorithms. The system also provided additional facilities customized to each problem, such as visualizations. Consequently, the scope of the system was limited to 6 optimization problems.

Based on the experience gained with the GreedEx system, we later developed OptimEx [11]. It is a general-purpose system which supports large-scale optimality testing for any optimization problem. We briefly describe some features which will contribute to a better understanding of the educational activities presented in this paper.

Fig. 1 contains a snapshot of OptimEx in a working session. The screen is divided into two vertical, movable panels. The left panel contains the text editor and a small area for compiler messages. The right panel hosts data of the large-scale testing in four tabs:

- Data table. It contains all the test cases.
- Runs table. It contains the outcomes computed by the algorithms in all runs. Each row corresponds to a different test case and each column, to a different algorithm. In Fig. 1, this table hosts the results of eight different algorithms which solve the task scheduling problem stated above. From left to right, they are two backtracking algorithms, one branch-and-bound algorithm, two dynamic programming algorithms, two heuristic algorithms, and a recursive algorithm. A coloring convention is used to highlight optimal vs. suboptimal results. In Fig. 1, a cell with olive green background means that the outcome of its corresponding algorithm applied to its corresponding test case is optimal. Likewise, cells hosting suboptimal results are colored in light green. Note in the figure that all the algorithms are exact, with the only exception of both heuristic algorithms.
- Numerical summary table. It contains a number of descriptive statistical measures computed from the results stored in the runs table (see Fig. 2). Some measures are percentages of suboptimal and optimal cases, while other measures are medium and extreme scattering measures with respect to optimal values.
- Graphical summary. It contains most of the values contained in the numeric summary table, displayed in two diagrams (see Fig. 3).

Other important elements of the OptimEx system are:

- Dialog to characterize the optimization problem. Outcomes of different algorithms that solve the same problem can only be compared by previously defining some issues, mainly the common header of the algorithms to compare and whether it is a maximization or minimization problem.
- Large-scale random generation of test cases. The number of test cases and their range of values are declared and consistent test cases are randomly generated.

III. Innovative Instructional Tasks Involving Large-Scale Optimality Testing

In this section, we present three innovative instructional tasks which involve large-scale optimality testing. Firstly, the educational context of the experience reported is introduced. Later, each instructional task and its results is successively presented.

A. Educational Context

The experience was held in the elective course "Advanced Algorithms", offered to senior students majored in either the computer science or the computer engineering degrees at Universidad Rey Juan Carlos. The course builds on the previous, mandatory course "Design and Analysis of Algorithms", offered to sophomore students.

The syllabus of the course "Advanced Algorithms" comprises a number of algorithm design techniques. It addresses advanced issues of some design techniques that students know superficially, namely the greedy and the backtracking techniques. It also introduces new design techniques, namely heuristic and approximation algorithms, branch-and-bound, dynamic programming and probabilistic algorithms.

Assignments had to be solved in teams, where a team could be either an individual or a pair of students, at students' choice. For each assignment, each group had to deliver in time a report elaborated according to an outline described in the assignment statement. Reports contained the algorithms constructed by students, as well as descriptions of key decisions of their design and the results of evaluating the algorithms with respect to either efficiency or optimality.

Students enrolled in the course had to solve six assignments about the different topics. For this paper, we focus on the following assignments:

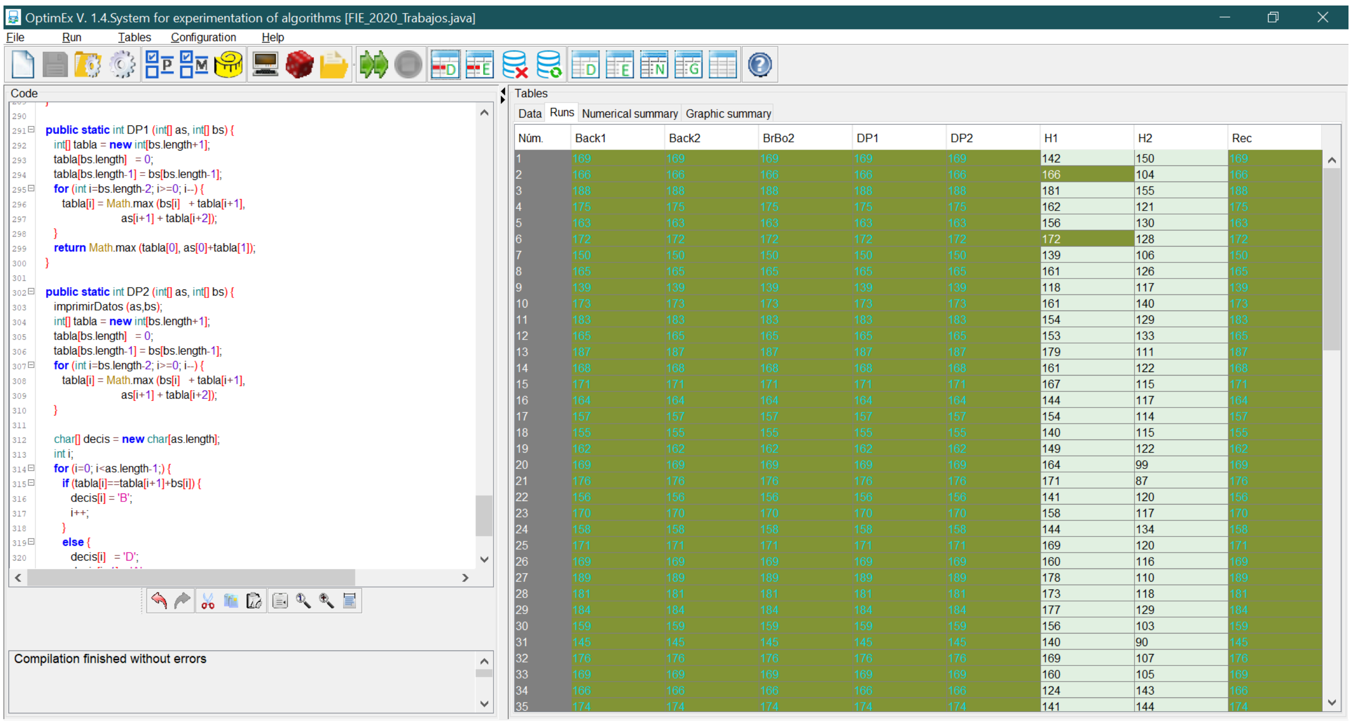
2. Design and evaluation of heuristic algorithms.
3. Design and evaluation of search-based algorithms. This assignment was delivered in two steps: (a) design and code of a backtracking algorithm, and (b) design and code of a branch-and-bound algorithm.
5. Design and evaluation of dynamic programming algorithms.

B. Large-Scale Testing and Debugging

In this subsection and the following ones, we describe an innovative use of large-scale optimality testing. In each subsection, we first describe the intended goal of an activity and we then present the results obtained in the experience.

Description of the Task

Optimality testing not only provides an opportunity to explore unknown behaviors or to check expected behaviors but also to detect anomalous behaviors. Some of these behaviors are compiler or run-time errors. Furthermore, the following results are not acceptable:

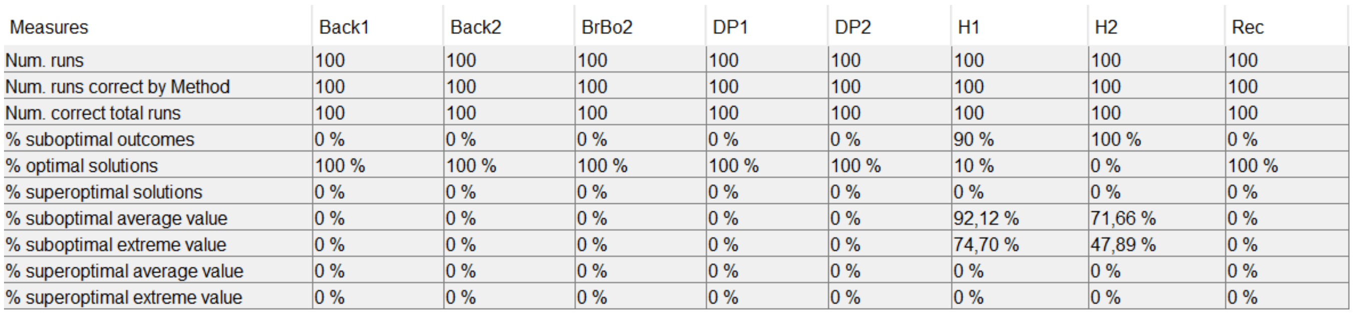

OptimEx V. 1.4.System for experimentation of algorithms [FIE_2020_Trabajos.java]

```
public static int DP1 (int[] as, int[] bs) {
  int[] tabla = new int[bs.length+1];
  tabla[bs.length]   = 0;
  tabla[bs.length-1] = bs[bs.length-1];
  for (int i=bs.length-2; i>=0; i--) {
    tabla[i] = Math.max (bs[i]   + tabla[i+1],
                         as[i+1] + tabla[i+2]);
  }
  return Math.max (tabla[0], as[0]+tabla[1]);
}

public static int DP2 (int[] as, int[] bs) {
  imprimirDatos (as,bs);
  int[] tabla = new int[bs.length+1];
  tabla[bs.length]   = 0;
  tabla[bs.length-1] = bs[bs.length-1];
  for (int i=bs.length-2; i>=0; i--) {
    tabla[i] = Math.max (bs[i]   + tabla[i+1],
                         as[i+1] + tabla[i+2]);
  }

  char[] decis = new char[as.length];
  int i;
  for (i=0; i<as.length-1;) {
    if (tabla[i]==tabla[i+1]+bs[i]) {
      decis[i] = 'B';
      i++;
    }
    else {
      decis[i]   = 'D';
```

```
Compilation finished without errors
```

Núm.	Back1	Back2	BrBo2	DP1	DP2	H1	H2	Rec
1	169	169	169	169	169	142	150	169
2	166	166	166	166	166	166	104	166
3	188	188	188	188	188	181	155	188
4	175	175	175	175	175	162	121	175
5	163	163	163	163	163	156	130	163
6	172	172	172	172	172	172	128	172
7	150	150	150	150	150	139	106	150
8	165	165	165	165	165	161	126	165
9	139	139	139	139	139	118	117	139
10	173	173	173	173	173	161	140	173
11	183	183	183	183	183	154	129	183
12	165	165	165	165	165	153	133	165
13	187	187	187	187	187	179	111	187
14	168	168	168	168	168	161	122	168
15	171	171	171	171	171	167	115	171
16	164	164	164	164	164	144	117	164
17	157	157	157	157	157	154	114	157
18	155	155	155	155	155	140	115	155
19	162	162	162	162	162	149	122	162
20	169	169	169	169	169	164	99	169
21	176	176	176	176	176	171	87	176
22	156	156	156	156	156	141	120	156
23	170	170	170	170	170	158	117	170
24	158	158	158	158	158	144	134	158
25	171	171	171	171	171	169	120	171
26	169	169	169	169	169	160	116	169
27	189	189	189	189	189	178	110	189
28	181	181	181	181	181	173	118	181
29	184	184	184	184	184	177	129	184
30	159	159	159	159	159	156	103	159
31	145	145	145	145	145	140	90	145
32	176	176	176	176	176	169	107	176
33	169	169	169	169	169	160	105	169
34	166	166	166	166	166	124	143	166
35	174	174	174	174	174	141	144	174

Fig. 1. A snapshot of the user interface of OptimEx, showing the text editor and a part of the runs table.



Measures	Back1	Back2	BrBo2	DP1	DP2	H1	H2	Rec
Num. runs	100	100	100	100	100	100	100	100
Num. runs correct by Method	100	100	100	100	100	100	100	100
Num. correct total runs	100	100	100	100	100	100	100	100
% suboptimal outcomes	0 %	0 %	0 %	0 %	0 %	90 %	100 %	0 %
% optimal solutions	100 %	100 %	100 %	100 %	100 %	10 %	0 %	100 %
% superoptimal solutions	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
% suboptimal average value	0 %	0 %	0 %	0 %	0 %	92,12 %	71,66 %	0 %
% suboptimal extreme value	0 %	0 %	0 %	0 %	0 %	74,70 %	47,89 %	0 %
% superoptimal average value	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %
% superoptimal extreme value	0 %	0 %	0 %	0 %	0 %	0 %	0 %	0 %

Fig. 2. A snapshot of the numeric summary table.

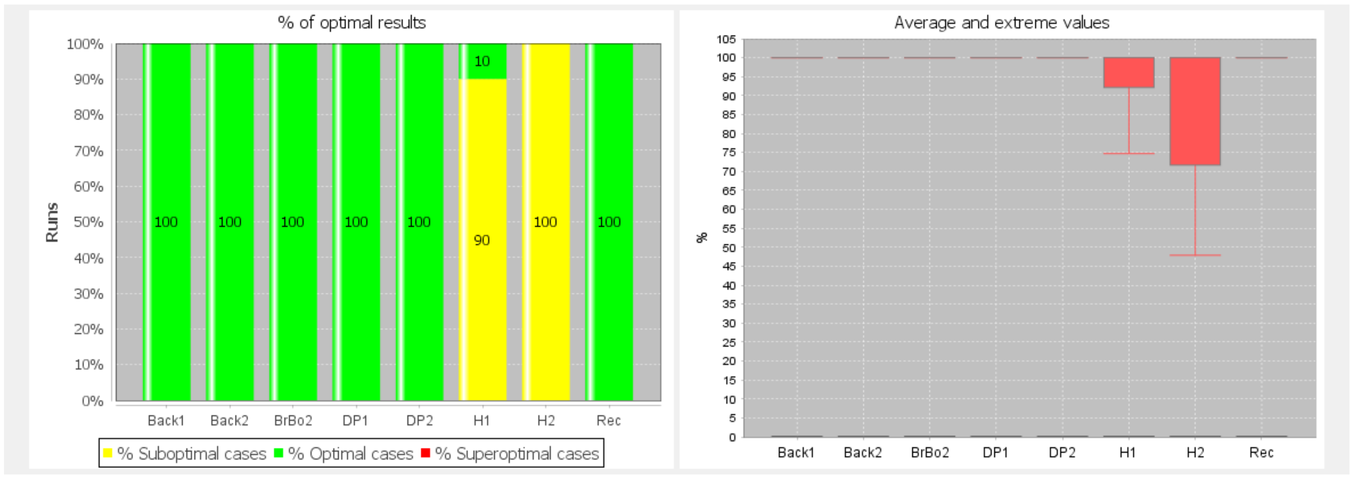


Fig. 3. A snapshot of the graphic summary.

- An exact algorithm does not compute optimal outcomes in 100% of the cases. This behavior is expected in algorithms designed according to the following design techniques: greedy, backtracking, branch-and-bound, and dynamic programming.
- An inexact algorithm computes an optimal outcome for a given test case, while an exact algorithm computes a suboptimal result.

Both situations are a symptom of the existence of, at least, one bug in one algorithm. It may be that an exact algorithm computes a suboptimal result or that an inexact algorithm computes a result which "is better" than a valid, optimal result. In any case, those tests cases where an anomalous situation is detected can be used to analyze the behavior of the algorithms involved and ultimately debug those algorithms which are wrong.

This situation is even more evident for certain classes of related algorithms. For example, tabulated dynamic programming algorithms are derived by transformation from multiple recursive algorithms. The transformation of the latter algorithm into the former has the goal of improving efficiency, but both algorithms must compute the same outcomes. A similar situation holds for branch-and-bound algorithms when they are derived from equivalent, backtracking algorithms.

Results

In assignments 3 and 5, we asked students to report on any "incident" they had on conducting their optimality testing. By "incident" we meant that the students discovered problems after running their optimality tests and modified the algorithms developed in either previous or the current assignment to fix those problems. A high number of groups reported on such incidents, due to unacceptable experimental results.

Table I shows the number of teams who reported to have modified any algorithm. The information is presented classified by assignment and algorithm design technique. Cells containing a hyphen correspond to design techniques which had not been studied at the moment of the corresponding assignment. Remember that assignment 3 was solved in two steps, which are shown in two different columns of the table. One group reported in assignment 3b to have modified algorithms based on two different design techniques.

TABLE I. ALGORITHMS DEBUGGED

Algorithm design technique	**Assign. 3a (N=32)**	**Assign. 3b (N=31)**	**Assign. 5 (N=30)**
Heuristic	6 (19%)	0 (0%)	1 (3%)
Backtracking	6 (19%)	3 (10%)	0 (0%)
Branch-and-bound	–	7 (23%)	1 (3%)
Backtracking, and branch-and-bound	–	1 (3%)	0 (0%)
Dynamic programming – recursive version	–	–	2 (6%)
Dynamic programming – tabulated version	–	–	1 (3%)
Total	12 (38%)	11 (35%)	5 (17%)

Note that it is probable that not all groups reported incidents. However, over one third of the groups did so in the two partial submissions of assignment 3. In some cases, the algorithm which was corrected had been developed and submitted in a past assignment. This is the case of heuristic algorithms in any of these assignments, of backtracking in assignment 3b, or in general of search-based algorithms in assignment 5. We also note modifications due to lack of equivalence in related algorithms, as explained in the paragraph immediately preceding this 'Results' header.

C. Large-Scale Testing and Problem Specification

Both this subsection and the following one refer to assignment 2. A total of 34 assignment reports were submitted.

Description of the Task

An algorithm is a recipe to automatically solve a given problem. Problems are usually stated informally in natural language, perhaps complemented with one or several examples. Formally, a problem is described by means of its specification, which consists of several parts:

- Signature. It identifies input and output data of the problem. Input data consists of parameters and their data types, and output is characterized by the data type of the result.
- Precondition. It declares any additional restrictions on valid input values.
- Postcondition. It declares the relationship that must hold between valid input data and the output. In the case of optimization problems, the postcondition can be further decomposed into a validity condition and a target function. The former is similar to other postconditions, while the latter declares the function to optimize.

Senior students know specifications and their constituent elements. However, their knowledge is fragile and they do not perform as expected when they are faced with specifying a given problem. In previous academic years, we had noticed [18] students' difficulties in correctly specifying algorithmic problems. However, large-scale problem solving potentially give the opportunity of reinforcing some of these elements, as they are implicitly handled in the testing process. In particular:

- Signature. Given that the algorithms to compare with respect to optimality solve the same problem, they must also share the same signature (method head in Java terminology). Thus, a student using OptimEx must identify the head of those methods to compare, in case there is more than one head in the Java class.
- Target function. The student must also select the choice of comparing outcomes for either maximization or minimization.
- Precondition. A large number of test cases must be randomly generated before running the optimization algorithms on them and comparing their outcomes. Test cases must be valid for the problem. Therefore, the specification of ranges for random generation must be compatible with the problem specification, i.e. they must define a subset of valid input values.

For instance, consider the task scheduling problem described in Section II.A. The problem precondition declares that the two arrays containing incomes for high- and low-stress tasks must have the same length, and incomes must be strictly positive.

Fig. 4 shows the OptimEx dialog used to specify the random generation of test cases for this problem. Note that the user has specified that the two arrays will always have length twelve. In addition, the values contained in the first array (i.e. revenues for high-stress tasks) will vary between 10 and 30, while the values to generate for the second array (i.e. revenues for low-stress tasks) will vary between 5 and 20. Note that these restrictions define a subset of the set of valid input values for this problem.

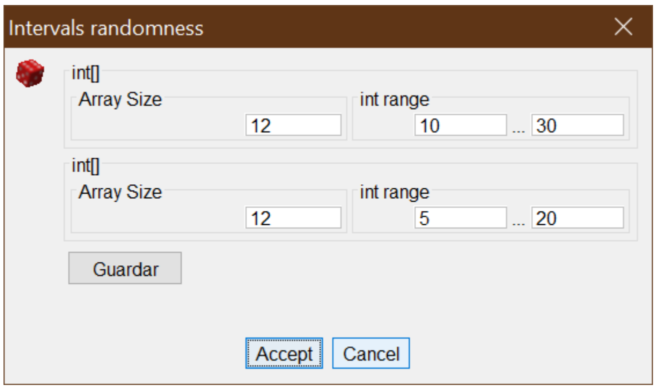


Fig. 4. A snapshot of the graphic summary tab.

We hypothesized that the specification of ranges of values necessary for random data generation was a chance of reinforcing the relevance and the identification of preconditions. As a part of the second assignment, students had to declare the different parts of the problem specification (in particular, the precondition) and had to explicitly describe the conditions of their optimality experiment (in particular, the ranges of values for random data generation).

Results

An analysis of the specifications of the problem allows classifying their preconditions into three categories: complete, partial and null. The definitions of the first and the third categories are obvious. A precondition is partially declared if it only contains one of its two parts: either that both arrays must have the same length or that their contents must be strictly greater than zero. The results are summarized in Table II.

TABLE II. PRECONDITION SPECIFICATIONS (N=34)

Category	# groups	% groups
Complete precondition	11	32%
Partial precondition	17	50%
Null precondition	6	18%

Note that only one third of the preconditions were complete. Furthermore, about one fifth of the preconditions were null because either the precondition was empty or it contained alien elements.

We also wondered whether students' preconditions and the dialogs used for random data generation were consistent. With respect to the length of both arrays, all students generated arrays of the same size. Therefore, we only lack knowing whether they were consistent in the expected values of incomes.

The analysis of their random generation conditions allows classifying the contributions into three categories:

- Coherent. A contribution is coherent when incomes constraints are coherent between the problem precondition and the random data generation dialog. We include in this category those students who developed a partial precondition which included the constraint on incomes.
- Partially coherent. A contribution is partially coherent when the precondition declares non-null incomes but they are allowed in the generation dialog.
- Unknown. We do not know whether a contribution is coherent if it did not declare the range of values used for data generation.

The results are summarized in Table III.

TABLE III. CONSISTENCY BETWEEN PRECONDITIONS AND RANDOM DATA GENERATION (N=34)

Category	# groups	% groups
Coherent	16	47%
Partially coherent	2	6%
Unknown	16	47%

Note that we ignore for almost one half of groups whether they were consistent in their treatment of both issues. However, for those groups whose coherence is known, almost all of them were consistent. Note also that there are partially (in)coherent contributions but not completely (in)coherent contributions.

D. Large-Scale Testing and Counterexamples

Description of the Task

A counterexample is a specific example which illustrates that a given claim does not hold. In algorithmics, it is common to present counterexamples of an algorithm satisfying a property, for instance a counterexample of the exactitude of a greedy criterion.

In general, constructing a counterexample is an easier task than formally proving that an algorithm satisfies a given property. Nevertheless, it demands abstraction capability to analyze the problem domain and to identify properties which may lead to counterexamples. Consequently, it is a difficult task for many students.

We propose an alternative way of constructing counterexamples, based on large-scale optimality testing. In order to show the inexactitude of a given algorithm, we may conduct an optimality testing experiment which include that algorithm and other algorithms which solve the same problem. If any counterexample is experimentally found, it can be analyzed to identify those conditions that lead to the computation of a suboptimal outcome by the algorithm. From those properties, it is usually an easy task to construct a simpler counterexample, i.e. a counterexample of smaller size or containing smaller values.

Results

Remember that assignment 2 asked students to develop several heuristic algorithms for the optimization problem stated above. It also asked students to contribute with experimental counterexamples, in case of obtaining less than 100% optimal cases. In addition, they had to analyze those counterexamples found and construct a simpler counterexample for each inexact heuristic algorithm. Finally, they had to explain the reasons of the inexactitude of each heuristic algorithm.

As explained in the previous subsection, 34 assignment reports were gathered. An analysis of the contributions regarding counterexamples can again be classified into three categories:

- Complete. A complete contribution includes a counterexample for each inexact algorithm. Note that any heuristic algorithm for this problem should be inexact. However, the experiment conducted could fail in finding a counterexample due to different reasons: small number of test cases, small size of the arrays, ranges of income values, etc. In case a heuristic algorithm seemed to be exact according to the experiment, for consistency we did not require the contribution of a counterexample.
- Partial. A partial contribution only includes counter-examples for some inexact algorithms or it includes incomplete counterexamples (typically, without providing its corresponding revenue).
- Null. No counterexample was contributed.

Table IV presents the results. The second column shows the number and the percentage of contributions classified into the corresponding category for counterexamples found experimentally. The third column is similar for simpler counterexamples, designed from the experimental counterexamples.

TABLE IV. Experimental and Designed Counterexamples, First Submission (N=34)

Category	Experimental # (%)	Designed # (%)
Complete	13 (38%)	11 (32%)
Incomplete	13 (38%)	11 (32%)
Null	8 (24%)	12 (35%)

The experiment was conducted by 33 out of 34 groups. However, about one fourth of the groups did not contribute with any counterexample found experimentally. The results are even worse for designed counterexamples, with about one third of groups with no own contribution. Furthermore, note, as expected, that the percentages of complete or incomplete counterexamples designed were smaller than the corresponding percentages of experimentally found counterexamples.

Let us remind that the groups were allowed to submit a second, enhanced version of the assignment report, based on the instructor's comments. Five out of nine groups changed their contribution of counterexamples in a second submission. The results after considering the second submission are shown in Table V.

Four groups who had not contributed with any experimental counterexample in their first submission did contribute in their second submission, although with incomplete counterexamples. One group who contributed in their first submission with counterexamples did not contribute in their second submission (we count it as null, although we guess that it was a mistake or oblivion). Overall, the number of complete experimental counterexamples was not increased, but the number of incomplete counterexamples was.

TABLE V. Experimental and Designed Counterexamples, Final Submission (N=34)

Category	Experimental # (%)	Designed # (%)
Complete	13 (38%)	14 (41%)
Incomplete	17 (51%)	12 (35%)
Null	4 (12%)	8 (24%)

With respect to designed counterexamples, the number of groups without any designed counterexample decreased. Three groups who previously had not contributed with any counterexample provided complete counterexamples, and one group, contributed with incomplete counterexamples.

Students were also asked to explain the reasons why counterexamples found or designed for each heuristic algorithm made it fail in computing an optimal solution.

Table VI shows three categories of groups, according to their rationale:

- Reasoning. These groups give a reasoned explanation. Most groups in this category contributed with both experimental and designed counterexamples, but we also find some groups who only contributed with either class of counterexamples.
- No reasoning. Groups in this category either contributed with both experimental and designed counterexamples, or contributed with experimental counterexamples. However, they did not explain the effect of the counterexamples on the suboptimal behavior of algorithms.
- No counterexamples. These groups did not find any kind of counterexample.

TABLE VI. Reasoning Provided on Counterexamples (N=34)

Category	First submission # (%)	Second submission # (%)
Reasoning	21 (62%)	25 (74%)
No reasoning	6 (18%)	7 (21%)
No counterexamples	7 (21%)	2 (6%)

Note that most students were successful in explaining, at least, why some heuristic algorithms failed with the experimental or designed counterexamples contributed. This percentage was 62% in the first submission, and it increased to three quarters in the second submission. Most groups belonging to this category contributed with both classes of counterexamples (19 or 20 groups after the first or the second submission, respectively).

IV. Discussion

The results obtained in the three tasks are different. We may consider that large-scale optimality testing was successfully used as an auxiliary tool to debug new or past algorithms. We found that at least one third of the groups made use of optimality testing to debug their algorithms and were aware of this fact. We may consider that this positive result is due to students' maturity as programmers, who are familiar with debugging in algorithm development.

However, the results for the other two instructional tasks were not as positive. In general, students did not have problems in specifying valid ranges for random data generation. However, a relevant percentage of groups failed in making explicit the more general precondition associated to the problem. We may conclude that testing did not assist them in this process and probably they were not aware of the relation between precondition and ranges of values.

With respect to counterexamples, a relevant percentage of students failed in proposing them, even though they had found them in their testing. More reflection is necessary on the experimentation process and more attention should be paid to analyzing the results obtained, without giving for granted that students will be able to do it by themselves. The fact that the percentage of groups contributing with designed counterexamples is smaller than the percentage of groups contributing with experimental counterexamples is a symptom that some groups have difficulties in analyzing counterexamples to determine the weaknesses of each heuristic algorithm. However, we found that most groups were successful in analyzing counterexamples and explaining why the heuristic algorithms failed in computing optimal solutions.

V. Conclusions

We have presented the use of large-scale optimality testing with more sophisticated aims than usually: as a debugging tool for optimization algorithms, to call students' attention to the problem precondition, and to find out counterexamples of the optimality of inexact algorithms. The three innovations have been illustrated by their use in the academic course 2019/20 in an algorithms course. The results were mixed, with better results for debugging than for the other two tasks. We have also analyzed these results, identifying some lessons learnt.

For next academic year, we plan to refine the two less successful tasks. We should make explicit the relationship between preconditions and random data generation. It could even be suggested that constraints given for random data generation could be used as a basis to specify general preconditions. Furthermore, we should make explicit in the classroom the process of identifying experimental counterexamples, analyzing their features and using them to design simpler counterexamples.

References

[1] K. M. Ala-Mutka. A survey of automatic assessment approaches for programming assignments. *Computer Science Education*, 15(2): 83-102, 2005. DOI 10.1080/08993400500150747.

[2] C. Douce, D. Livingstone and J. Orwell. Automatic test-based assessment of programming: A review. *Journal of Educational Resources in Computing*, 5(3), article 4, 2005. DOI 10.1145/1163405.1163409.

[3] P. Ihantola, T. Ahoniemi, V. Karavirta and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling 2010)*, ACM DL, 2010, pp. 86-93. DOI 10.1145/1930464.1930480.

[4] S. Wasik, M. Antczak, J. Badura, A. Laskowski and T. Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys*, 51(1): article 3, 2018. DOI 10.1145/3143560.

[5] L. Llana, E. Martín-Martín, C. Pareja-Flores and J. Á. Velázquez-Iturbide. FLOP: A user-friendly system for automated program assessment. *Journal of Universal Computer Science*, 20(9):1.304-1.326, 2014.

[6] V. Karavirta, A. Korhonen and L. Malmi. On the use of resubmissions in automatic assessment systems. *Computer Science Education*, 16:3, 229-240, 2006. DOI 10.1080/08993400600912426.

[7] O. Giménez, J. Petit and S. Roura. Jutge.org: An educational programming judge. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, ACM DL, 2012, 445-450. DOI 10.1145/2157136.2157267.

[8] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*, ACM DL, 2004, 26-30. DOI 10.1145/971300.971312.

[9] D. Berque, J. Bogda, B. Fisher, T. Harrison and N. Rahn. The KLYDE workbench to study experimental algorithm analysis. In *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'94)*, ACM DL, 1994, 83-87. DOI 10.1145/191033.191065.

[10] M.-Y. Chen, J.-D. Wei, J.-H. Huang and D.T. Lee. 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 2006)*, ACM DL, 2006, 123-127. DOI 10.1145/1140123.1140159.

[11] J. Á. Velázquez-Iturbide. GreedEx and OptimEx: Two tools to experiment with optimization algorithms. *International Journal of Engineering Education*, 32(3A): 1,097-1,106, 2016.

[12] S. Sahni. *Data Structures, Algorithms, and Applications in Java*. Silicon Press, 2nd ed., 2004.

[13] J. Á. Velázquez-Iturbide. An experimental method for the active learning of greedy algorithms. *ACM Transactions on Computing Education*, 13(4): article 18, 2013. DOI 10.1145/2534972.

[14] J. Kleinberg and É. Tardos, Algorithm Design. Boston, MA: Pearson Addison-Wesley, 2006.

[15] G. Brassard and P. Bratley, Fundamentals of Algorithmics. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[16] J.Á. Velázquez-Iturbide, and O. Debdi, "Experimentation with optimization problems in algorithm courses," in Proceedings of the IEEE Internacional Conference on Computer as a Tool, EUROCON 2011, IEEE, 2011. DOI 10.1109/EUROCON.2011.5929294.

[17] O. Debdi, M. Paredes-Velasco and J. Á. Velázquez-Iturbide. GreedExCol, a CSCL tool for experimenting with greedy algorithms. *Computer Applications in Engineering Education*, 23(5): 790-804, 2015. DOI 10.1002/cae.21655.

[18] J. Á. Velázquez-Iturbide. Students' misconceptions of optimization problems. In *Proceedings of the 24th Annual Conf. Innovation and Technology in Computer Science Education (ITiCSE 2019)*, ACM DL, 2019, 464-470, DOI 10.1145/3304221.3319749.