International Journal of Engineering Education Vol. 32, No. 3(A), pp. 1097–1106, 2016
Printed in Great Britain

0949-149X/91 $3.00+0.00
© 2016 TEMPUS Publications.

GreedEx and OptimEx: Two Tools to Experiment with Optimization Algorithms*

J. ÁNGEL VELÁZQUEZ-ITURBIDE
Universidad Rey Juan Carlos, Escuela Técnica Superior de Ingeniería Informática, C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain.
E-mail: angel.velazquez@urjc.es

Experimentation is an important part of the education of computer engineers. In particular, it is a common educational practice to check algorithms correctness or efficiency. However, experimentation has seldom been used to check the optimality of (optimization) algorithms. The article presents two tools aimed at experimenting with this property. They have some common features, but differ in their degree of generality and scaffolding. GreedEx is a tool for novices, being aimed at the active learning of the foundations of greedy algorithms. It currently supports six optimization problems. OptimEx is a more advanced, general experimentation tool that can be used with any kind of optimization algorithms. If both systems are used in an algorithm course, they should be used at different stages. The paper presents two contributions. Firstly, we present the novel system OptimEx. Secondly, we give recommendations of use for both tools, based on the author's experience using and evaluating them. Of particular interest is a list of incorrect outcomes that may be produced by OptimEx, which are symptoms of students' misconceptions, as well as how to fix them.

Keywords: computer engineering education; optimization algorithms, experimentation, scaffolding, instructional recommendations

1. Introduction

Practicing concepts and methods is an essential part of education in many disciplines. Some terms coined for this approach to instruction are "learning through practice", "learning by doing" or "learning through experience" [1]. Computing is not an exception to the need of these activities.

The Denning report [2] identified three paradigms or cultures that permeate computing: mathematics, science, and engineering. These cultures each have different goals. Consequently, practice activities aimed at computing instruction have different features depending on the culture they are based on. Mathematics aims at proving theorems, science aims at experimenting with phenomena, and engineering aims at testing systems. The three cultures influence the instruction of any subfield of computing, but we focus in this article on algorithms. Depending on the inspiring culture, different practice activities may be conceived to learn algorithmics: properties of algorithms can be stated and proved, performance of algorithms can be measured, and the algorithms developed can be tested. They are not exclusive but complementary cultures. For instance, we may design and test an algorithm, analyze its properties and gather measures to obtain empirical evidence on them.

Algorithms can be analyzed with respect to several properties, correctness and efficiency being the most important. For an important class of algorithms, namely combinatorial optimization algorithms (optimization algorithms for short) [3], a third property can be characterized, namely optimality. We find in the computing education literature many contributions regarding activities to practice with correctness or efficiency, but we hardly find contributions regarding optimality [4, 5].

In this article, we present two tools designed to assist at experimenting with optimality. The materials for any experiment are a set of alternative algorithms for a given optimization problem. Both tools share some features whilst they differ in their degree of generality and scaffolding. GreedEx is a tool aimed at the active learning of the foundations of greedy algorithms. It is intended to assist novice students, thus they include explanations and visualizations, and is currently limited to six problems. OptimEx is a general experimentation tool that can be used with advantage by students knowing a range of design techniques for optimization problems (dynamic programming, branch and bound, etc.). Both systems can be used at different stages in algorithm courses.

The article presents two contributions. Firstly, we present the novel system OptimEx. Secondly, we give recommendations of use for both tools in algorithm courses. They are based on the author's experience using and evaluating both systems, although (for brevity) we do not describe the evaluations in the article. Of particular interest is a list of incorrect outcomes that may be produced by OptimEx, which are symptoms of students' misconceptions, as well as how to fix them.

The structure of the article follows. In the follow-

* Accepted 12 February 2016.

ing section, we introduce experimentation with different properties of algorithms, in particular with optimality. In the third section, we describe GreedEx and OptimEx, and in the fourth section, we present some recommendations for their use in algorithm courses. Finally, we include our conclusions and outline lines of future research.

2. Experimenting with optimization algorithms

In this section, we first overview the main features of experimentation with algorithms (see [6] for a comprehensive review). Then, we characterize the main features of optimality and describe ways of experimenting with it.

2.1 Experimenting with algorithms

Within computing education, most experiments have been conducted over algorithms. This is probably due to algorithms have a clear definition (amenable to formal specification), have well-defined properties, and are small-scale artifacts.

The underlying principles of experimentation with algorithms are in common with the natural sciences [7], but they are implemented differently because programming laboratories are virtual by definition. Therefore, we may speak of virtual or interactive experimentation. This kind of experiments is also related to scientific discovery [8].

The Denning Report [2] identified four steps in the experimental scientific method. Baldwin [9] further refined the elements in a scientific experiment:

- A hypothesis that the experiment must confirm or refute.
- An experimental system that will be observed to confirm or refute the effects predicted by the hypothesis.
- A quantitative measure of the results produced by the experimental system.
- The use of controls to assure that the experiment really proves the hypothesis.
- An analysis of data gathered to determine if they are consistent with the hypothesis.
- A report describing all the elements given above so that other researchers or practitioners can replicate the experiment.

Experiments on algorithms are not usually stated with the same terminology or even structure as scientific experiments, but this can be easily accomplished. For instance, consider that we want to check experimentally the performance of a given algorithm. The hypothesis is the expected behavior of the algorithm, as described by its order of complexity. The experimental system is the algorithm under execution, which must sometimes be modified to gather performance measures. We may use a number of quantitative measures, for instance the number of either relevant or atomic statements executed ([10], chapter 2). The controls depend on the monitoring software available, but it must be able to gather the quantitative measure for different input sizes. Finally, the data gathered will be analyzed to check whether they are compatible with the algorithm order of complexity.

It is important to be aware of the limitations of experimentation. No matter how many experiments are performed, there is no guarantee that a given property of an algorithm has been proved. We must be aware that a given property is only established by means of its corresponding proof. Experimental methods only allow refuting a property or, at the best, accumulating evidence about the property. Consequently, experiments play a humble but important role, especially when the proof is difficult to build.

2.2 Characterizing optimality

An algorithmic problem is specified by means of the following elements:

- Input and output data, with their corresponding data types.
- A predicate to be satisfied by input data, i.e. its precondition.
- A predicate to be satisfied by input and output data, i.e. its postcondition.

Any valid solution must satisfy the pre- and the post-conditions. The pair formed by both predicates is often known as the problem restrictions.

A combinatorial problem aims at finding an object within a finite set (or, at least, a countable set) of potential solutions satisfying the problem restrictions. An optimization problem is a combinatorial problem with an additional element in its specification: an optimal solution is a valid solution that optimizes the measure stated by a target function, either a benefit to maximize or a cost to minimize. If there are several optimal solutions, any of them is acceptable. Without loss of generality, we assume in the rest of the article that the target function specifies a maximization goal.

It is not obvious how to find a valid or an optimal solution for any instantiation of most combinatorial or optimization problems. For this reason, textbooks often include immediately after the problem statement one or several examples of valid, optimal and even invalid or suboptimal solutions.

Combinatorial and optimization problems can always be solved using an algorithm that exhaustively traverse the set of potential solutions [3]. However, this naïve algorithm is extremely inefficient, with unpractical orders of complexity for most problems. Well-known design techniques [10] are aimed at designing efficient algorithms: the greedy technique, dynamic programming, etc.

2.3 Experimenting with algorithms and optimality

Experimentation has traditionally been used in algorithm courses to measure efficiency (correctness is more typically addressed in courses on programming methodology). We can find a number of publications reporting experiments with algorithm efficiency [4, 5, 9, 11, 12]. Common experiments are:

- To instrument execution time for different input data sizes. The goal consists in showing that actual measures of execution time are consistent with the theoretical order of complexity. Plotting is often used as a means to compare results.

  An alternative goal of this kind of instrumentation is to compare the efficiency of different algorithms that solve the same problem, or even to compare recursive and iterative versions of the same algorithm. Finally, comparison of execution times can be used for more subtle analyses, e.g. to determine a threshold in the problem size below which an iterative algorithm performs better than a divide-and-conquer one.
- To instrument execution time for cases or algorithms which are very difficult or impossible to derive analytically, and to try to infer the corresponding order of complexity.

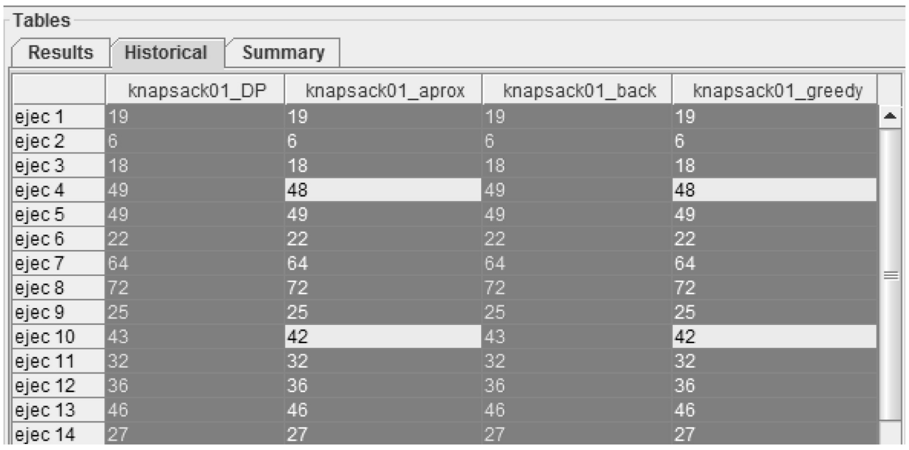
Performance experiments need not be restricted to execution time, but they may encompass any measure that can be gathered in run-time [9, 11–13], for instance: number of nodes generated by an algorithm that traverses a search space, space saving in a compression algorithm, etc.

Experimenting with optimality implies dealing with the actual outcomes of optimization algorithms. One form of experimentation consists in inferring (or checking), given several algorithms for a given optimization problem, which ones are optimal or suboptimal [14] based on the results of a number of algorithm runs.

For instance, consider the 0/1 knapsack problem ([10] chapter 20, [15] chapter 8, [16] chapter 6) and four algorithms that compute valid solutions: one greedy algorithm, one backtracking algorithm, one dynamic programming algorithm, and one approximation algorithm. Assume the greedy algorithm selects objects in non-increasing order of profit/weight (denoted P/W↓ for short) and the approximation algorithm is an enhancement of the greedy algorithm ([15] chapter 13).

Let us summarize an actual, short experiment carried out using OptimEx. We generated input data randomly, obtaining the values {5, 7, 10, 8, 7, 6, 2, 7} as weights, {5, 10, 5, 16, 16, 7, 19, 3} as profits, and a knapsack capacity equal to 6. The four algorithms computed a solution with associated profit 19. Therefore, we could not deduce which algorithms are optimal or suboptimal based on this single run. Randomly generating successive test cases, the four algorithms kept generating the same outcome for the two following test cases, but they computed different results for the fourth test case. For this run, the dynamic programming and the backtracking algorithms yielded 49 whereas the greedy and the approximate algorithms yielded 48. Therefore, we may claim that the two latter algorithms are not optimal, but we cannot claim anything conclusive about the two former algorithms.

The data and results of the executions comprising an experiment must be displayed in a structured way, being tables a convenient format. Fig. 1 shows a part of the historic table of OptimEx, where the results of successive executions are displayed. Each row corresponds to a different test case. Each column corresponds to an algorithm. A cell is shadowed in grey when its associated algorithm (in its column) yields an optimal value for its



	knapsack01_DP	knapsack01_aprox	knapsack01_back	knapsack01_greedy
ejec 1	19	19	19	19
ejec 2	6	6	6	6
ejec 3	18	18	18	18
ejec 4	49	48	49	48
ejec 5	49	49	49	49
ejec 6	22	22	22	22
ejec 7	64	64	64	64
ejec 8	72	72	72	72
ejec 9	25	25	25	25
ejec 10	43	42	43	42
ejec 11	32	32	32	32
ejec 12	36	36	36	36
ejec 13	46	46	46	46
ejec 14	27	27	27	27

Fig. 1. A subset of runs of algorithms for the 0/1 knapsack problem, as displayed in the historic table.

corresponding test case (its row). Values in a column are written in blue font when the corresponding algorithm computed an optimal result for all the test cases (first and third algorithms from the left).

3. The GreedEx and OptimEx tools

We successively describe the simplest tool and the most complex tool, i.e. first GreedEx and then OptimEx.

3.1 The GreedEx tool

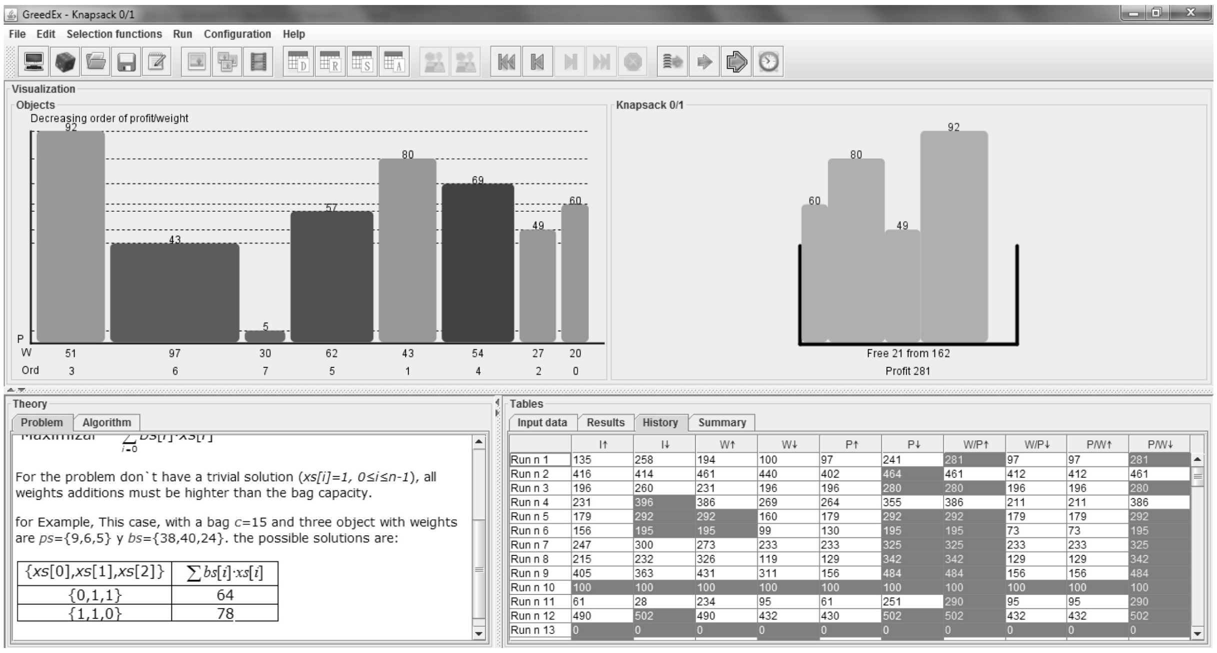
The GreedEx tool [17] was designed to support the active learning of greedy algorithms. Given an optimization problem, GreedEx offers a number of alternative selection functions to the student, who must experiment in a trial to determine which selection functions are optimal or suboptimal.

GreedEx is a restricted but extendible system. Currently, it supports six problems: the activity selection problem ([16] chapter 4, [18] chapter 16), the knapsack problem ([15] chapter 6), the 0/1 knapsack problem and three additional knapsack problems (maximizing the number of objects introduced into the knapsack, maximizing the weight, and maximizing the weight introduced into two knapsacks). Problems number one, two and four (in order of their above citation), can be solved optimally using greedy algorithms while, for the other three problems, any greedy algorithm is suboptimal.

Fig. 2 shows a screen capture of GreedEx in a session working with the 0/1 knapsack problem. The user interface of GreedEx consists of the main menu, the icon bar and three panels. The higher panel visualizes input and output data. Typically, input data are shown at the left (in this problem, the objects) and the results are displayed at the right (here, the knapsack with the objects introduced). Both the weight and the profit of each object are encoded as graphical attributes, i.e. weight and height of the object, respectively. Initially all the objects are colored in blue but their color is changed to grey as they are selected during the current execution. In addition, the objects are colored in different tones to suggest their ordering, according to the active selection function. Fig. 2 shows an intermediate state of execution (after inserting four objects) of the greedy algorithm based on the selection function P/W↓ using the first of the test cases.

The lower left area of the screen is the theory panel. It consists of two tabs: the problem tab, that hosts the problem statement (visible in the figure), and the algorithm tab, that contains a greedy algorithm that solves the problem, written in Java-based pseudocode. Finally, the lower right area is the table panel, with four tabs, each one containing one table: the data table, the result table, the history table (visible in the figure) and the summary table.

When a student launches GreedEx, nothing can be done but selecting a problem. Immediately, the theory panel becomes active so that the student may read the problem statement and a greedy algorithmic solution. Afterwards, the student may generate input data from one of three sources: the keyboard, a random generator or a file; current input data may also be modified interactively. Input data are dis-



Maximizar $\sum_{i=0} bs[i]\cdot xs[i]$

For the problem don`t have a trivial solution (*xs[i]=1, 0≤i≤n-1*), all weights additions must be highter than the bag capacity.

for Example, This case, with a bag *c*=15 and three object with weights are *ps*={9,6,5} y *bs*={38,40,24}. the possible solutions are:

$\{xs[0],xs[1],xs[2]\}$	$\sum bs[i]\cdot xs[i]$
{0,1,1}	64
{1,1,0}	78

	I↑	I↓	W↑	W↓	P↑	P↓	W/P↑	W/P↓	P/W↑	P/W↓
Run n 1	135	258	194	100	97	241	281	97	97	281
Run n 2	416	414	461	440	402	464	461	412	412	461
Run n 3	196	260	231	196	196	280	280	196	196	280
Run n 4	231	396	386	269	264	355	386	211	211	386
Run n 5	179	292	292	160	179	292	292	179	179	292
Run n 6	156	195	195	99	130	195	195	73	73	195
Run n 7	247	300	273	233	233	325	325	233	233	325
Run n 8	215	232	326	119	129	342	342	129	129	342
Run n 9	405	363	431	311	156	484	484	156	156	484
Run n 10	100	100	100	100	100	100	100	100	100	100
Run n 11	61	28	234	95	61	251	290	95	95	290
Run n 12	490	502	490	432	430	502	502	432	432	502
Run n 13	0	0	0	0	0	0	0	0	0	0

Fig. 2. A snapshot of the user interface of GreedEx, showing the visualization of an algorithm run, a part of the problem statement and a part of the history table.

played in the data table and in the visualization panel, and the student may select any of the selection functions supported by GreedEx to solve the problem.

When the student chooses a selection function, she may run it flexibly using four execution/animation controls: one step forward, one step backward, forward, and rewind. As animation controls are pressed, the visualization panel is consistently updated. The effect of each execution step can be analyzed to understand the selection function behavior. When a run is complete, its results are stored in the tables.

The four tables allow storing data of an experiment at different levels of abstraction:

- Data table: It contains the value of the parameters used as the current test case.
- Result table: It contains the value yielded by each selection function using the active test case, as well as information about the greedy decisions taken.
- History table: It contains the value yielded by each selection function in past runs (see Fig. 2).
- Summary table: It contains the global performance of each selection function, expressed as the percentage of runs that yielded an optimal value.

GreedEx provides some additional facilities for faster experimentation: executing all the selection functions of a problem for the current test case, executing a subset of them, and executing all the selection functions on a very high number of test cases randomly generated.

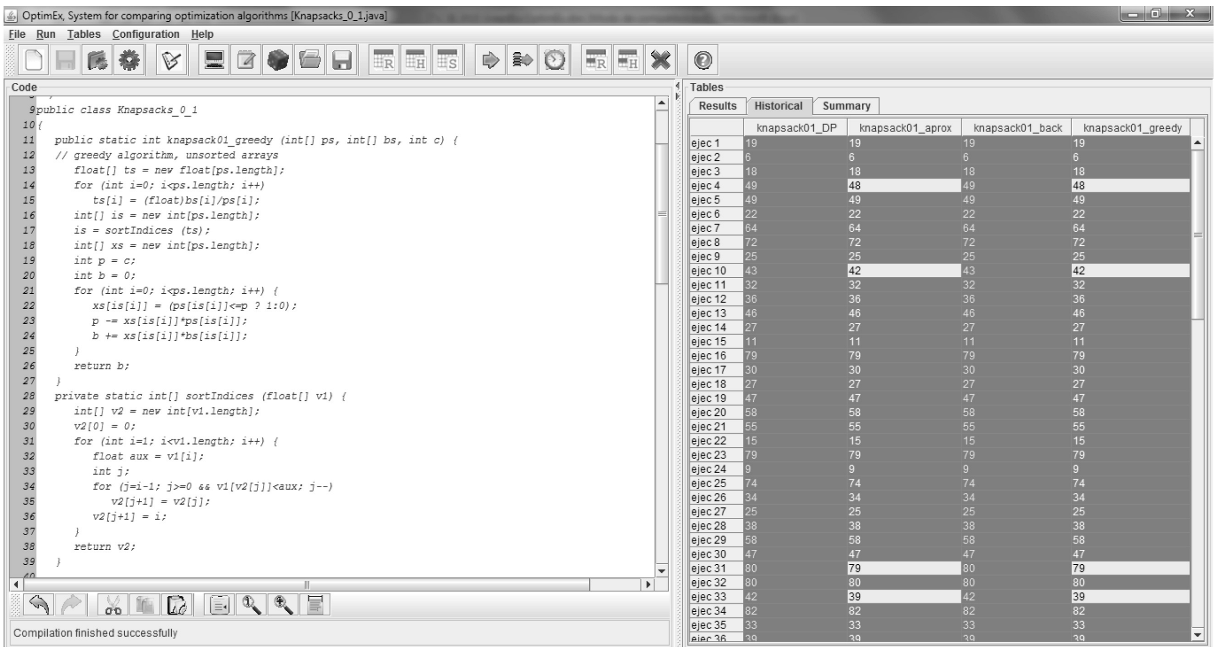
Finally, GreedEx provides some functions to support instruction: exporting tables and visualizations into graphical files, and changing the user interface language (currently, it supports English and Spanish).

3.2 The OptimEx tool

OptimEx is a more general system than GreedEx, intended to support experimentation with any optimization algorithm coded in Java. Those parts of the GreedEx user interface specific to a particular algorithm, namely visualizations and explanations, were removed. On the other hand, some functions were incorporated to support generality:

- Code editor. The algorithms to compare must be coded as public methods in a single Java class.
- Compilation and run support.
- Preparation of the experiment. The user must identify the Java algorithms to compare. As these algorithms are intended to solve the same problem, OptimEx requires that their corresponding methods have the same signature (i.e. data types of the parameters and of the result). In addition, the user must specify whether it is a maximization or a minimization problem. Finally, the user may select the specific methods to compare (among those sharing the signature selected) and may optionally mark one of them to note that she guesses it will be optimal.

Fig. 3 shows the structure of the OptimEx user interface. The left panel hosts the code editor, whist the right panel host the table panel. The main menu,


OptimEx, System for comparing optimization algorithms [Knapsacks_0_1.java]
File Run Tables Configuration Help
Code
Tables
Results Historical Summary

```
public class Knapsacks_0_1
{
  public static int knapsack01_greedy (int[] ps, int[] bs, int c) {
  // greedy algorithm, unsorted arrays
     float[] ts = new float[ps.length];
     for (int i=0; i<ps.length; i++)
        ts[i] = (float)bs[i]/ps[i];
     int[] is = new int[ps.length];
     is = sortIndices (ts);
     int[] xs = new int[ps.length];
     int p = c;
     int b = 0;
     for (int i=0; i<ps.length; i++) {
        xs[is[i]] = (ps[is[i]]<=p ? 1:0);
        p -= xs[is[i]]*ps[is[i]];
        b += xs[is[i]]*bs[is[i]];
     }
     return b;
  }
  private static int[] sortIndices (float[] v1) {
     int[] v2 = new int[v1.length];
     v2[0] = 0;
     for (int i=1; i<v1.length; i++) {
        float aux = v1[i];
        int j;
        for (j=i-1; j>=0 && v1[v2[j]]<aux; j--)
           v2[j+1] = v2[j];
        v2[j+1] = i;
     }
     return v2;
  }
```

Compilation finished successfully

	knapsack01_DP	knapsack01_aprox	knapsack01_back	knapsack01_greedy
ejec 1	19	19	19	19
ejec 2	6	6	6	6
ejec 3	18	18	18	18
ejec 4	49	48	49	48
ejec 5	49	49	49	49
ejec 6	22	22	22	22
ejec 7	64	64	64	64
ejec 8	72	72	72	72
ejec 9	25	25	25	25
ejec 10	43	42	43	42
ejec 11	32	32	32	32
ejec 12	36	36	36	36
ejec 13	46	46	46	46
ejec 14	27	27	27	27
ejec 15	11	11	11	11
ejec 16	79	79	79	79
ejec 17	30	30	30	30
ejec 18	27	27	27	27
ejec 19	47	47	47	47
ejec 20	58	58	58	58
ejec 21	55	55	55	55
ejec 22	15	15	15	15
ejec 23	79	79	79	79
ejec 24	9	9	9	9
ejec 25	74	74	74	74
ejec 26	34	34	34	34
ejec 27	25	25	25	25
ejec 28	38	38	38	38
ejec 29	58	58	58	58
ejec 30	47	47	47	47
ejec 31	80	79	80	79
ejec 32	80	80	80	80
ejec 33	42	39	42	39
ejec 34	82	82	82	82
ejec 35	33	33	33	33
ejec 36	39	39	39	39

Fig. 3. A snapshot of the user interface of OptimEx, showing the editor and the history table.

the icons bar and the table panel were designed to assure the highest resemblance with GreedEx (in the figure, the history table is visible).

The history and summary tables were slightly modified to shed more meaningful information about the results of the experimentation. The summary table displays the percentage of test cases for which each algorithm yielded optimal or suboptimal results, as well as the mean and maximum deviation of suboptimal results with respect to the optimal outcomes. In case the user marked a suboptimal method as optimal, the history table highlights those cells where the results of other algorithms are higher than the results of the marked algorithm. The percentages and deviations of these incorrect cases also are shown in the summary table. (More on these problematic situations is explained in Section 4.2.3.)

4. Educational use

The goal of this section is to give instructional advice on the use of GreedEx and OptimEx in a more clear and comprehensive way than in publications reporting evaluations. Our recommendations are based on 5 years of experience using GreedEx and 2 years using OptimEx in algorithm courses. GreedEx has been extensively evaluated with respect to its usability [17] and, more importantly, with respect to its educational efficiency [19], detecting statistically significant enhancements in students' comprehension of greedy algorithms. We have also analyzed the reports elaborated by students using GreedEx or OptimEx for assignments, having detected a number of students' difficulties and misunderstandings [14].

We first deal with GreedEx, which is intended for novices, and then with OptimEx, which should be used in later stages of algorithm courses. For each tool, we identify the most adequate algorithm design techniques (including algorithm examples) and we give instructional recommendations. We also give a list of incorrect outcomes that may be produced by OptimEx, which are symptoms of students' misconceptions, as well as how to fix them.

4.1 Use of GreedEx

GreedEx provides students with scaffolding regarding several issues: design of alternative selection functions, tracing greedy algorithms in detail, and experimenting with the optimality of alternative selection functions (i.e. greedy algorithms).

4.1.1 Design techniques supported

GreedEx was designed to enhance the learning of the foundations of greedy algorithms. It supports three problems that can optimally be solved using the greedy technique, namely two knapsack problems and the activity selection problem. Students become familiarized with proposing alternative selection functions for a given problem and with experimenting with them to determine their optimality.

Inquiring about the optimality of different selection functions for a given problem allows students to find out results that are unknown in advance. An interesting issue that is not addressed in textbooks is the existence of several kinds of selection functions [20]:

- Equivalent selection functions. Textbooks identify non-decreasing order of finish time as an optimal selection function for the activity selection problem ([16] chapter 4, [18] chapter 16). However, it is not unique as it can be symmetrically restated as non-increasing order of start time. The same applies to the knapsack problem, where P/W$\downarrow$ (non-increasing order of profit/weight) can be restated as W/P$\uparrow$.
- Nearly optimal selection functions. An intuitive selection function for the activity selection problem consisting in selecting activities in non-decreasing order of duration. Although it is not optimal, it yields the optimal result in about 95% of the cases.

GreedEx supports three additional knapsack problems that cannot be solved optimally with the greedy technique, remarkably the 0/1 knapsack problem. These problems can be used to motivate the need of applying other design techniques. Here, GreedEx can be used to show or to find counter-examples of the optimality of these algorithms.

Suboptimal algorithms also provide opportunities for interesting inquiries, such as determining the percentage of cases where a given suboptimal algorithm yields an optimal solution. For instance, selecting objects in non-increasing order of profit/weight (P/W$\downarrow$) for the 0/1 knapsack problem yields an optimal solution in about 81% of the cases. Even more surprising is that this selection function has worse performance for this problem than selecting objects in non-increasing order of profit (P$\downarrow$), since the latter yields an optimal solution in about 85% of the cases.

4.1.2 Recommended use

GreedEx can be used by the instructor in the classroom to illustrate the basic structure of greedy algorithms and to check the optimality of a number of selection functions. Furthermore, GreedEx can be used for inquiry-based assignments on any of the supported problems.

After using GreedEx for several years, we refined its use as a complete instructional method [14]. It is important not to use GreedEx isolated from the rest of the course, but to integrate it with the schedule of classroom and lab sessions, and to provide students with instructional materials. In brief, we suggest paying attention to the following issues:

- Educational materials. The basic concepts and skills for reasoning about optimality may seem trivial. However, this is not the case and this background must be given to students. As algorithm textbooks hardly contain background on optimality, it is useful to elaborate and make them available to students.
- Syllabus contents. Selection functions must not be given for granted but students must be aware that they are a design product and that they must be verified for optimality. Consequently, an emphasis must be placed in the first classes on proposing alternative selection functions for any problem.
- Organization of labs. A single lab session is not the most adequate organization. A preliminary, short session should be devoted to let students become familiarized with GreedEx (e.g. by experimenting with the knapsack problem). In a second lab session, students are given an assignment statement based on a more difficult problem (e.g. the activity selection problem). The goal of the assignment is to identify which selection functions offered by GreedEx are optimal (if any). Optionally, a third, short session may be scheduled for students who underperformed the assignment. A statement may identify the optimal selection functions of the second session and urge students to investigate why either they did not identify them or they proposed other (suboptimal) selection functions.

4.2 *Use of OptimEx*

OptimEx was designed by removing the scaffolding and problem-specific features of GreedEx, resulting in a general-purpose experimentation tool. Therefore, it should be used after students are familiar with GreedEx.

4.2.1 *Design techniques supported*

OptimEx can be used to conduct different experiments with optimization algorithms not supported by GreedEx. An interesting experience with greedy algorithms consists in showing the suboptimality of some promising selection functions. Two problems where intuition often fails are:

- The scheduling problem with fixed deadline ([15] chapter 6) has an optimal selection function, namely selecting tasks in non-decreasing order of deadline. However, we may wonder whether tasks could be selected in non-increasing order of benefit.
- The single-source shortest-paths problem ([10] chapter 18, [15] chapter 6, [16] chapter 4, [18] chapter 24) has an optimal selection function consisting in selecting nodes in non-decreasing order of path length (i.e. Dijkstra's algorithm), but we may wonder whether nodes could be selected in non-decreasing order of arch length.

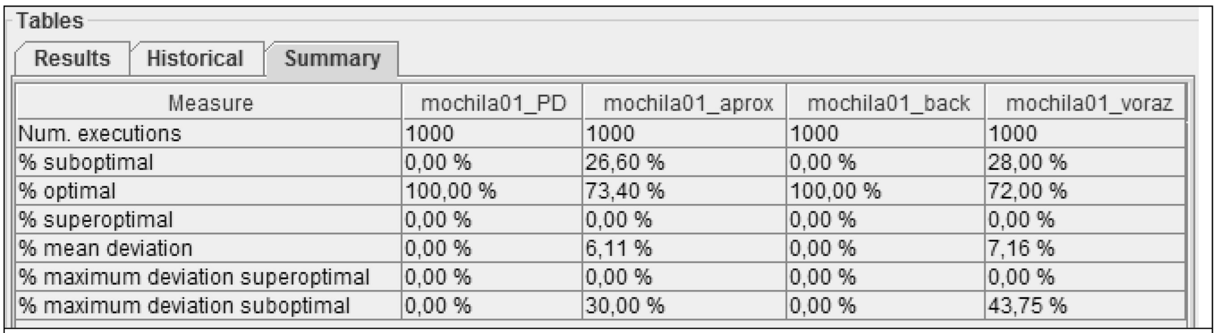
Another formative use of OptimEx is to illustrate that some optimal selection functions become suboptimal if the problem statement is slightly modified. For instance, this effect happens if we modify the statement of the fractional knapsack problem to give place to the 0/1 knapsack problem. Minor changes in the statements of the coin change problem ([15] chapter 6) and of the single-source shortest-paths problem also produce this effect. These problems can be used to motivate the need, for many optimization problems, of design techniques different from the greedy technique.

Finally, OptimEx is especially adequate to experiment with heuristic or approximation algorithms. We may compare these suboptimal algorithms with algorithms developed using exact design techniques (e.g. dynamic programming or branch and bound). Experimental results should reinforce the concepts taught in the classroom. These experiments also allow making students' misconceptions emerge when unexpected results are obtained (see Section 4.2.3 below).

4.2.2 *Recommended use*

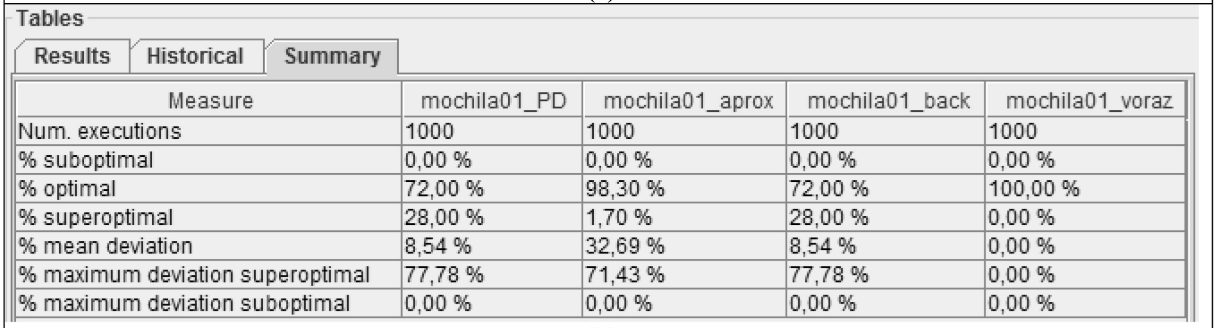
OptimEx is a generic tool for experimentation with optimality. The user may develop alternative algorithms for a given problem and compare their outcomes. However, the instructor must be careful on using OptimEx, because the mere comparison of outcomes is not always exciting to students. In addition, developing several algorithms to solve a given problem demands an effort that the student must perceive as useful.

We have successfully used OptimEx with a particular organization of assignments. As a part of the first assignment, students develop one greedy algorithm for the activity selection problem. For other algorithm design techniques (backtracking, branch and bound, and dynamic programming), an assignment is proposed where students must develop a new algorithm for a related problem, namely the weighted activity selection problem ([16] chapter 4). In a final assignment on approximation algorithms, students are urged to put together the different algorithms in a single, shared class and compare their outcomes. (The greedy algorithm must be adapted to the new



Measure	mochila01_PD	mochila01_aprox	mochila01_back	mochila01_voraz
Num. executions	1000	1000	1000	1000
% suboptimal	0,00 %	26,60 %	0,00 %	28,00 %
% optimal	100,00 %	73,40 %	100,00 %	72,00 %
% superoptimal	0,00 %	0,00 %	0,00 %	0,00 %
% mean deviation	0,00 %	6,11 %	0,00 %	7,16 %
% maximum deviation superoptimal	0,00 %	0,00 %	0,00 %	0,00 %
% maximum deviation suboptimal	0,00 %	30,00 %	0,00 %	43,75 %

(a)



Measure	mochila01_PD	mochila01_aprox	mochila01_back	mochila01_voraz
Num. executions	1000	1000	1000	1000
% suboptimal	0,00 %	0,00 %	0,00 %	0,00 %
% optimal	72,00 %	98,30 %	72,00 %	100,00 %
% superoptimal	28,00 %	1,70 %	28,00 %	0,00 %
% mean deviation	8,54 %	32,69 %	8,54 %	0,00 %
% maximum deviation superoptimal	77,78 %	71,43 %	77,78 %	0,00 %
% maximum deviation suboptimal	0,00 %	0,00 %	0,00 %	0,00 %

(b)

Fig. 4. A summary table obtained for the 0/1 knapsack problem when (a) no algorithm is marked as optimal, and (b) the greedy algorithm is marked.

problem, but this is an easy task.) If they notice any unexpected outcome of any algorithm, they also must discover the cause and fix it.

A problem with the previous organization of assignments is the short period of time devoted to compare algorithms. A longer period allows students to internalize the different situations (and their consequences) that can be found on experimenting with optimality, as well as fixing erroneous situations. OptimEx can be used differently without a need to change much the organization of assignments. Every new algorithm developed in an assignment (but the first one) must be put together in a shared class and compared with the previously constructed algorithms. If any unexpected result is obtained, the student must discover the cause and fix it. This Java class plays in the course a role similar to a "portfolio".

The same considerations given for GreedEx on the use of tool in the classroom and on the importance of integrating it with the course also are critical for OptimEx.

4.2.3 Difficulties and incorrect results

Experimenting with algorithms is an activity that often makes surprise. Experimentation is a human activity and the algorithms themselves are the result of human activity, therefore they are subject to errors. This is especially probable when students are their main actors. For instance, an algorithm may trigger a runtime error for some test cases, or test cases can be generated without taking into account all the constraints stated in the problem specification. The first case is easy to detect with OptimEx because runtime errors are captured in the history table. The second case is potentially more difficult to note because it may lead to a number of behaviors, depending on the algorithms and the test cases: runtime errors, unexpected outcomes, etc.

Let us illustrate unexpected results with an example. Fig. 4(a) shows the results of an experiment conducted under correct conditions. The experiment compares the outcomes of the four algorithms for the 0/1 knapsack problem referred in Fig. 1 by using one thousand test cases. The algorithms designed using the backtracking and the dynamic programming techniques are exact, therefore they yield maximal results in 100% of the cases. Fig. 4(b) displays a summary of the same results when the user has wrongly marked the greedy algorithm as optimal. This choice would be correct for the fractional knapsack problem, but not for the 0/1 knapsack problem. Consequently, the other three algorithms produce (in different percentages) profits that are greater than the "optimal" ones! This contradictory situation can be identified in the summary table by noticing the presence of non-zero percentages for (non-sense) "superoptimal" results.

Some additional, unexpected results that can be obtained in experiments follow:

- No algorithm gives optimal outcomes in 100% of the test cases. This result may be correct if all the algorithms under experimentation are suboptimal, but it is wrong if any of them is optimal. In this case, the algorithms must be revised to find out the cause of these results and fix the corresponding algorithms: either exact algorithms give lower outcomes than expected or any suboptimal algorithm gives high, wrong outcomes.
- An algorithm yields outcomes that are better than the outcomes of some "optimal" algorithm. There are several plausible explanations for this phenomenon. Firstly, it could be that the user marked a suboptimal algorithm as optimal, as Fig. 4 illustrated. In this case, the experiment must be replicated under different conditions: either no marking that algorithm or marking an exact algorithm. Secondly, it could be that the user marked an exact algorithm (e.g. a dynamic programming or a backtracking algorithm) as optimal, but the results do not confirm this choice. This case is similar to the case in the previous paragraph, where the cause may be that the "optimal" algorithm yields suboptimal results for some test cases or that a "suboptimal" algorithm computes higher values than it should.

These problems are in common with discovery learning activities in any discipline. Students have problems with explaining phenomena and with interpretation of data, among others [8]. The instructor must be aware of them and address them explicitly in the lectures, educational materials and assignments.

5. Conclusions

We have presented GreedEx and OptimEx, two tools aimed at experimenting with the optimality property of optimization algorithms. GreedEx is a friendlier but more limited tool than OptimEx. GreedEx supports six optimization problems and is aimed at the active learning of the foundations of greedy algorithms. OptimEx is a more general experimentation tool that can be used by students knowing other design techniques to solve optimization problems. Both tools have some common features so that the transition from the former to the latter is relatively easy. Consequently, both systems can be used at different stages of algorithm courses. We have also included in the article some recommendations of use, based on the author's experience using and evaluating both systems. Finally, we have included a list of problematic outcomes, common on experimenting with OptimEx, and an explanation of their causes and how to fix them.

We have plans to carry out immediately or in the near future. First, in the current academic year we have distributed the use of OptimEx in several assignments, as explained in section 4.2.2. Although experiences experimenting in a final assignment were positive, students did not always perform as expected. When they obtained unexpected results, they often neither revised the experiments conditions nor corrected the potentially wrong algorithms. In addition, students did not have much time to exercise their analysis skills experimenting with optimality. We hope to obtain more encouraging results by spacing the use of OptimEx along the course. Second, we found in these years that optimization concepts are not as simple for students as we expected. Therefore, we would like to develop a catalog of optimality concepts to avoid students' difficulties and misconceptions. We argue that optimality concepts should be incorporated into the background of mathematical concepts of algorithm courses.

Acknowledgments—This work was supported by the Ministry of Economy and Competitiveness under research grants TIN2011-29542-C02-01 and TIN2015-66731-C2-1-R, the Region of Madrid under research grant S2013/ICE-2715, and the Universidad Rey Juan Carlos under research grant 30VCPIGI05.

References

1. D. Laurillard, *Teaching as a Design Science*, Routledge, New York, NY, 2012.
2. P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner and P. R. Young, Computing as a discipline, *Communications of the ACM*, **32**(1), 1989, pp. 9–23.
3. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, Dover Publications, Mineola, NY, 1982.
4. 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, *Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2006)*, ACM, 2006, pp. 123–127.
5. J. W. Coffey, Integrating theoretical and empirical computer science in a data structures course, *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2013)*, ACM, 2013, pp. 23–27.
6. J. Á. Velázquez-Iturbide, C. Pareja-Flores, O. Debdi and M. Paredes-Velasco. Interactive experimentation with algorithms, *Computers in Education—Volume 2*, Sergei Abramovich (ed.). Nova Science Publishers, 2012, pp. 47–70.
7. C. G. Hempel, *Philosophy of Natural Science*, Prentice-Hall, Englewood Cliffs, NJ, 1966.
8. T. de Jong and W. R. van Joolingen, Scientific discovery learning with computer simulations of conceptual domains, *Review of Educational Research*, **68**(2), 1998, pp. 179–201.
9. D. Baldwin, Using scientific experiments in early computer science laboratories, *Proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1992)*, ACM, 1992, pp. 102–106.
10. S. Sahni, *Data Structures, Algorithms and Applications in Java*, 2nd ed. Silicon Press, Summit, NJ, 2005.
11. J. Matocha, Laboratory experiments in an algorithms course: technical writing and the scientific method, *Proceedings of the 22nd ASEE/IEEE Frontiers in Education Conference (FIE 1992)*, Stipes Publishing, 1992, pp-T1G 9-13.

12. D. McCraken, Three "lab assignments" for an algorithms course, *ACM SIGCSE Bulletin*, **2**(2), 1989, pp. 61–64.
13. T. K. Moore, A. G. Rich and M. R. Vich, Scientific investigation in a breadth-first approach to introductory computer science, *Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1993)*, ACM, 1993, pp. 63–67.
14. J. Á. Velázquez-Iturbide, An experimental method for the active learning of greedy algorithms, *ACM Transactions on Computing Education*, **13**(4), 2013, article 18.
15. G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996.
16. J. Kleinberg and É. Tardos, *Algorithm Design*, Pearson Addison-Wesley, 2006.
17. J. Á. 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, pp. 130–143.
18. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, Cambridge, MA, 2009.
19. N. Esteban-Sánchez, C. Pizarro and J. Á. Velázquez-Iturbide, Evaluation of a didactic method for the active learning of greedy algorithms, *IEEE Transactions on Education*, **57**(2), 2014, pp. 83–91.
20. J. Á. Velázquez-Iturbide and O. Debdi, Experimentation with optimization problems in algorithm courses. *Proceedings of the International Conference on Computer as a Tool (EUROCON'11)*, IEEE, 2011, pp. 1–4.

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 an affiliate 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).