Abstract—Algorithms are one of the core elements in computer science curricula. They involve design and analysis activities, mainly analysis of correctness and efficiency. A property which has received less attention is optimality. In order to gain insight and skills on optimization, and to promote active learning, we proposed an experimental method assisted by interactive assistants. In this paper we give a detailed account of how to use the experimental method in an algorithm course. Firstly, we show how to use it with greedy algorithms, with equivalent selection functions as the most interesting issue. Secondly, we use the method to demonstrate the need of other algorithm design techniques (e.g. dynamic programming) to solve other problems. Thirdly, nearly-optimal selection functions can be used as an introduction to approximation algorithms (i.e. heuristics).

Keywords—computer science education; optimization problems; algorithms; algorithm design techniques; experimentation

I. INTRODUCTION

Algorithms are one of the core elements in computer science curricula [1]. The main concern in algorithm courses is efficiency, especially time efficiency. Concern about correctness can also be found in courses on programming methodology. Finally, optimality is an important concern for many algorithms, i.e. optimization algorithms. Although a not so general concern, it is found in many of the most common algorithm design techniques, such as dynamic programming.

One common design technique is the greedy technique, usually named “greedy algorithms”. The greedy technique is perceived as simpler than other techniques. However, the experienced teacher notices that such simplicity is apparent. There is not much place for drill-and-practice exercises where to apply the foundations of the technique. As a consequence, the instructor commonly gives lectures in a passive way, hardly letting place for active learning. On the student side, most problems solved with greedy algorithms are learnt by rote.

If we let experimenting with optimality play a more central role in the learning of greedy algorithms, we make room for active learning. According to this idea, we proposed an experimental method in a previous work [2]. Students experiment with different alternatives for optimality with the assistance of interactive software. They must reason rigorously in order to determine plausible alternatives (called selection functions) and discard non-optimal ones.

In this paper, we give a detailed account of how to use the experimental method in algorithm courses. It is based on our experience of three years using the method in the mandatory third-year course “Design and Analysis of Algorithms” offered at our University to Computer Science students. In section 2, we introduce the fundamentals of greedy algorithms and we describe our experimental method. In section 3 we show the applicability of the experimental method to different design techniques. In particular, it is shown how to use it with greedy algorithms (we present the interesting issue of equivalent selection functions), approximation algorithms, and other techniques.

II. AN EXPERIMENTAL METHOD FOR GREEDY ALGORITHMS

A. Greedy Algorithms

Optimization problems seek a compound solution (typically, permutation, subset or sorting of a given set of items, called candidates) that must satisfy certain constraints and must optimize some measure. If several solutions yield the same measure value, any of them is valid. Some examples are shortest paths in graphs, scheduling problems, etc.

There are many design techniques that can be used to solve optimization problems. The most common techniques are [3,4,5] greedy algorithms, dynamic programming, and techniques based on tree state spaces (e.g. backtracking or branch-and-bound). Other techniques include approximation algorithms [3,4], evolutionary algorithms [6], etc.

A first step in solving optimization problems consists in designing a multistage process to incrementally construct an optimal solution from the candidates. This process takes different forms in the different design techniques. Greedy algorithms share the feature that only one candidate is taken into consideration at each step. The candidate is selected by a selection function which therefore is the key design element. In other design techniques, such as dynamic programming or backtracking, several candidates must be taken into consideration at each step.

Greedy algorithms are usually characterized in textbooks in abstract terms, but they are sometimes formalized as a code template, for instance [3]:

This work was supported by project TIN2008-04103 of Spanish Ministry of Science and Innovation
function greedy (C: set): set
{C is the set of candidates}
S ← ∅ {we build the solution in set S}
while C≠∅ and not solution(S) do
x ← select(C)
C ← C\{x} 
if feasible(S∪{x}) then S ← S∪{x}
if solution(S) then return S
else return «there are no solutions»

Surprisingly, the design process that leads to identifying an optimal selection function is hardly addressed explicitly in textbooks. We have only found two exceptions to this general situation. Firstly, this discussion is often found for the knapsack problem [3,4,5,7], where three strategies are introduced into the knapsack, a profit \(c\), and the capacity of the knapsack is not exceeded.

Secondly, a problem where several optimal selection functions are identified is the minimum-cost spanning tree. These selection functions lead to the well-known Prim’s and Kruskal’s algorithms [3,4,5,7], as well as Sollin’s [5] and Barůčka’s algorithms [7]. However, textbooks do not consider candidate, non-optimal selection functions.

B. The Experimental Method

A selection function is defined on the set of available candidates, returning at each step the most promising candidate according to some criterion. The selection function implicitly sorts candidates in increasing or decreasing order.

The set of candidate selection functions is not very large in general, as they are restricted to values of some parameters or values derived from them. Therefore, if we want to make explicit this design process, we must first identify a list of candidate selection functions, defined in increasing or decreasing order of some properties.

For instance, consider the knapsack problem [3,4,5,7]. It can be stated as follows. We have \(n\) objects, with weight \(w_i\) and a knapsack of capacity \(c\). If a fraction \(x_i\), \(0 ≤ x_i ≤ 1\), of object \(i\) is introduced into the knapsack, a profit \(p_i x_i\) is gained. The problem consists in maximizing the total profit, provided the capacity of the knapsack is not exceeded.

We may think of the following selection functions:
- Increasing order of weight (W↑).
- Decreasing order of weight (W↓).
- Increasing order of benefit (B↑).
- Decreasing order of benefit (B↓).
- Increasing order of rate benefit/weight (B/W↑).
- Decreasing order of rate benefit/weight (B/W↓).
- Increasing order of rate weight/benefit (W/B↑).
- Decreasing order of rate weight/benefit (W/B↓).

As a consequence of this set of executions, the four selections functions that did not yield the greatest value can be discarded. If we now use different input data, the four remaining candidate selection functions can be reduced to two. Further experimentation always produces optimal values for these two selection functions (see Table II). Consequently, we conclude that there is empirical evidence about the plausibility of B/W↓ and W/B↑ as optimal selection functions. (Their optimality should be formally proved.)

This experimental process can be supported by computer. We have developed several interactive assistants, the latest being called GreedEx. It is available at the following URL:

http://www.lite.etsii.urjc.es/greedex/

In the remaining subsections, we elaborate on particular cases of this analysis and their use with design techniques.

Given a set of candidate selection functions, the optimal ones must be identified. Experimentation is a good way of filtering the plausible ones. Non-optimal selection functions can be identified by finding counterexamples. The remaining selection functions are plausible candidates, but they should be proved for optimality. More details about how to conduct this discarding process, supported by interactive assistants, can be found elsewhere [2]. It is not a difficult task, with similarities to testing and, in general, to scientific experiments [2]. For the sake of completeness, we sketch it here.

For instance, consider the knapsack problem with the following input data: \(w=\{31,46,81,61,13,14,63\}\), \(b=\{97,86,42,91,53,52,92\}\) and \(c=246\). If we solve the problem by the selection functions given above, we obtain the results shown in Table I.

<table>
<thead>
<tr>
<th>Selection function</th>
<th>Objects introduced</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>W↑</td>
<td>{1, 0.22, 1, 1, 1}</td>
<td>480.33</td>
</tr>
<tr>
<td>W↓</td>
<td>{0, 0.89, 1, 0, 0, 1}</td>
<td>301.65</td>
</tr>
<tr>
<td>B↑</td>
<td>{0, 1, 1, 1, 1, 1}</td>
<td>369.27</td>
</tr>
<tr>
<td>B↓</td>
<td>{0, 0.89, 1, 0, 0, 1}</td>
<td>301.65</td>
</tr>
</tbody>
</table>

TABLE I. RESULTS OF APPLYING DIFFERENT SELECTION FUNCTIONS IN THE KNAPSACK PROBLEM

TABLE II. RESULTS OVER SEVERAL INPUT DATA FOR THE KNAPSACK PROBLEM

<table>
<thead>
<tr>
<th>Selection function</th>
<th>1st run</th>
<th>2nd run</th>
<th>3rd run</th>
</tr>
</thead>
<tbody>
<tr>
<td>W↑</td>
<td>480.33</td>
<td>217.21</td>
<td>–</td>
</tr>
<tr>
<td>W↓</td>
<td>301.65</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B↑</td>
<td>369.27</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B↓</td>
<td>480.33</td>
<td>260.80</td>
<td>–</td>
</tr>
<tr>
<td>B/W↑</td>
<td>301.65</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B/W↓</td>
<td>480.33</td>
<td>262.21</td>
<td>277.69</td>
</tr>
<tr>
<td>W/B↑</td>
<td>480.33</td>
<td>262.21</td>
<td>277.69</td>
</tr>
<tr>
<td>W/B↓</td>
<td>301.65</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
III. USING THE EXPERIMENTAL METHOD WITH DIFFERENT ALGORITHM DESIGN TECHNIQUES

A. Equivalent Selection Functions

A consequence of proposing a set of alternative selection functions is that unexpected, equivalent functions may be found. These discoveries enrich the discussion on optimal selection functions. In particular, they illustrate that a given problem may be optimally solved with several, equivalent selection functions. As we show in the following paragraphs, such equivalences can even be discovered in simple problems.

A first example can be found in the knapsack problem. Typically, \( B/W \) is proposed as the optimal selection function. However, notice in Table II that \( W/B \) is equivalent (with a less intuitive definition).

A proof of their equivalence is straightforward, as reversing quotients \( B/W \) sorted in decreasing order produces quotients \( W/B \) sorted in increasing order. Therefore, both criteria yield the same order of selection of objects or parts of objects. Consequently, if the first selection function is optimal, the second one is optimal too.

Another problem where equivalent selection functions can be found is the activity selection problem [4], which can be stated as follows. Given a set of \( n \) activities, each one with a start time \( s_i \) and a finish time \( f_i \), we seek to select a maximum-size subset of non-overlapping activities. For instance, given the set of activities in Table III, the subset composed of activities \( \{2,6\} \) is a valid solution, while subset \( \{8,9,3\} \) is a maximum-size solution.

Cormen et al. propose increasing order of finish time as the optimal selection function (we denote it \( F^{↑} \)). An equivalent selection function is decreasing order of start time (denoted \( S^{↓} \)). In this case, both criteria are equally intuitive.

Their equivalence is again easy to prove. Consider an optimal solution using the \( F^{↑} \) selection function. For instance, consider the set of five activities displayed in Fig. 1 and indexed top-down from 0 to 4. The application of this selection function results in successively selecting activities 2 and 1, discarding activity 3, selecting activity 4, and discarding activity 0. (Selected activities are displayed in green and discarded activities, in red.)

B. Non-Optimal Selection Functions

This process can also be applied to problems for which no selection function is optimal. In this case, the experimentation process serves to discard greedy algorithms as a design technique adequate to optimally solve the problem. Consequently, other design techniques must be considered, such as dynamic programming or backtracking (or its refinement branch-and-bound).

For instance, consider the 0/1 knapsack problem [7]. It has the same data and an optimization function similar to the (fractional) knapsack problem, so the same set of candidate selection functions can be proposed. However, no selection function always yields an optimal result (see Table III, where the runs with optimal values are shaded).

C. Nearly-Optimal Selection Functions

In subsection 2.1, we proposed to identify a list of candidate selection functions, some of which are optimal and others, non-optimal. Moreover, for some problems, we find non-optimal selection functions which, however, are worth to be considered. For instance, we may find selection functions which are optimal for a high percentage of the cases. Or, we may find selection functions which are bounded in the difference of their solution and an optimal solution.
For instance, consider again the activity selection problem. A selection function which is an intuitive, good candidate for most students is increasing order of duration (we denote it $D^\uparrow$). However, it is non-optimal as shown in Table IV, where results over 100 input data randomly generated are displayed. Notice that $D^\uparrow$ produces an optimal solution in only 96% of the cases.

### Table V. Results over 100 Input Data

<table>
<thead>
<tr>
<th>Selection function</th>
<th># optimal results</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^\uparrow$</td>
<td>26</td>
</tr>
<tr>
<td>$S^\downarrow$</td>
<td>100</td>
</tr>
<tr>
<td>$F^\uparrow$</td>
<td>100</td>
</tr>
<tr>
<td>$F^\downarrow$</td>
<td>30</td>
</tr>
<tr>
<td>$D^\uparrow$</td>
<td>96</td>
</tr>
<tr>
<td>$D^\downarrow$</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 3 shows a counterexample composed of three activities. It is neither difficult to build nor trivial.

![Figure 3. Three activities selected in order $D^\uparrow$](image)

This selection function allows illustrating two issues:

- A class of approximation algorithms that compute the optimal solution in a high percentage of the input data.
- The design of a counterexample often requires detailed analysis, which gives further insight on the problem.

### IV. Discussion

Experimentation is an integral part of the scientific method [8,9]. Similar concerns apply to our approach, provided they are adapted to optimality. In fact, our experimental method is complemented with some materials. Students are explained in the classroom the principles of the scientific method and experiments (namely, hypothesis, experiment planning, empirical evidence, counterexample, refutation, inductive reasoning). Furthermore, they are provided with notes about the scientific method, the experimental method, greedy algorithms, and the interactive assistants. Finally, they must fill a template to report and justify their experimental findings.

Our approach not only promotes rigor, but also creativity (see section III.A for the discovery of optimal selection functions). David Ginat has dealt with the creative design of algorithms [10], reporting the student misconceptions he found, including the abuse of the greedy technique [11]. His approach is very different from ours, but it shares some elements (use of counterexamples) and conclusions (students’ lack of rigor).

Ginat reports [10] that he devotes only 20% of the course time to problem solving activities. Our contributions are complementary, and can be used as a part of the remaining 80% of the time, where basic concepts are given and closed laboratories are conducted.

An unexpected consequence of the use of our experimental method was the discovery of students’ misconceptions about optimization algorithms [12]. This gave us the opportunity to further improve the teaching of optimization algorithms by taking actions to minimize the number of misconceptions. We are currently analyzing the results of these corrective actions, so we cannot make strong claims (although, according to our preliminary results, we consider that the number of students with misconceptions decreased).

### V. Conclusion

We have argued the importance of dealing with optimality in algorithm courses. Such concern has been fostered with an experimental method. Although the experimental method was conceived to deal with greedy algorithms, we have shown that it can also be applied with profit with other algorithm design techniques: (a) other basic design techniques for optimization problems (e.g. dynamic programming, backtracking, or branch-and-bound), and (b) approximation algorithms.

### References


