Educational Tasks, Fun, and Informatics

Lorenzo Repetto

Istituto di Istruzione Superiore "Italo Calvino", Genova, Italy
lorenzo.repetto@calvino.edu.it

Abstract. This contribution proposes some case studies and suggestions to introduce – at different levels of learning – several concepts of undoubted importance in informatics. The starting point is a set of problems faced by students in various competitions or while trying to solve classic puzzles, logical and algorithmic riddles, strategy games, and computer graphics amusements. I intend to provide teachers with few tips, hopefully useful to develop and strengthen computational thinking in students' minds, at differing stages of their courses.
The contents are concerned with: how to deal with knapsack problem and some hard problems on graphs (e.g. minimum vertex cover); comparisons between different greedy (efficient) algorithms, and exact (more complex) algorithms; examples of complementary problems or, more generally, problems that are equally demanding in terms of computational load; examples of puzzles (programmable by backtracking or by heuristic techniques or even as instances of exact cover set problem), the solutions of which consist either in a final state or in a sequence of moves to reach it; few hints to introduce strategy games for two competitors, impartial combinatorial ones as well; a didactic game, in which both players are dummy; finally, two notes about logical and algorithmic riddles (with some indicative examples) and graphic procedures related with computer science, both practical and theoretical.

Keywords: Games · Graphs · Time-complexity · Greedy / efficient algorithms · Hard / equivalent problems.

1 Introduction

Through more than thirty years of teaching informatics (both as the science and technique of *programming* in the broadest sense, and as *computer science*), I have often achieved unexpected benefits considering, analyzing, and solving – with the help of the computer – educational tasks of various kinds: questions (with a logical or mathematical background), puzzles, solitaires, or strategy games. As a matter of fact, students were usually more motivated both by the purely playful, amusing aspect, and by the sense of challenge that the computer implementation entailed, so that they learned more easily methods and techniques to represent information and to process data.

Another important aspect is that a lot of educational tasks like the ones proposed in Bebras competitions can offer the opportunity to explain key ideas and concepts at different levels of thinking skills: beginning with notions understandable by children and culminating in generalizations and formalization which can be covered in a university course, grasping the features of computational complexity, as well as power and limits of computing mechanisms.

The concept of *game* – in a broad sense – can be found in all cultures and in all historical periods, and there are many sectors of computer science benefiting from it: artificial intelligence, security, distributed computing, and so on. Only a few topics are proposed and discussed in this paper: suggestions and indications useful to find or create others are provided. Teachers will find countless ideas and hints in [1–5] and in the "canonical" fifteen books encompassing Martin Gardner's "Mathematical Games" columns from *Scientific American*, written by this prolific and unforgettable author from 1957 to 1980 and beyond. Italian teachers can also get inspiration from my book [6]; the short reports presented here are taken from it.

2 Efficient algorithms and hard problems

In Bebras competitions [7] instances of classic problems in informatics are often proposed in the form of educational tasks, in a realistic or fairytale scenario. Moreover, with a bit of imagination, you can always devise new questions or puzzles, in order to illustrate, for instance:

- "efficient" algorithms on unweighted (and possibly directed) graphs; e.g. to build a maximal tree, containing all the vertices reachable from a given starting vertex, either in depth (*Depth-First Search*), using the system stack, or in breadth (*Breadth-First Search*), using a queue;

- "efficient" algorithms on weighted (usually undirected) graphs; e.g. to find a "covering" tree of minimum weight (*Minimum Spanning Tree*, or MST). I would like to mention three *greedy algorithms* – following three different rules – due to O. Borůvka, J. B. Kruskal, and V. Jarník-R. C. Prim, respectively. The first is the oldest of all, dating back almost a century ago; the second one is the easiest to understand: the two "closest" vertices are connected at each step, making sure that no cycle is created; the third one inspired E. W. Dijkstra for the design of his famous algorithm to find the least cost paths from a given vertex to each of the others, more generally in a directed graph;

- "hard" problems on unweighted (and undirected) graphs; e.g. *Minimum Vertex Cover* and *Maximum Independent Vertex Set* (two "complementary" subset problems, which we will discuss later);

- "hard" problems on weighted graphs; e.g. *Travelling-Salesman Problem*, or TSP (a famous permutation problem);

- other "hard" problems; e.g. *Bin-Packing* (a partition problem), *Knapsack* (another subset problem; see below), *Multiprocessor Scheduling* (in which the possible precedence constraints between the processes – with different durations – are expressed by means of a directed acyclic graph).

By *efficient algorithm* we mean a procedure which, in order to be executed, requires a time *at most polynomial* (that is to say, bounded above by a polynomial) in the length of the input data coded in bits. (Usually, then, for practical purposes, it is understood that the degree of this polynomial should not be too high.)

By *hard problem* we mean, basically, a problem (in general) for which no efficient algorithm, that exactly solve it, is known: either for sure no efficient algorithm exists (as happens for intrinsically exponential problems, e.g. the famous *towers of Hanoi*) or it is unknown whether such an algorithm exists. For the problems mentioned here, this latter meaning applies.

2.1 A first case study: *Knapsack*

In this subsection let's see how to gradually deal with the "backpack optimizing problem", starting from an instance that I leave to your imagination; it could be a mountaineer packing for a long hike, or a thief who is robbing a shop and wants to accumulate stolen goods of the greatest possible value.

In general, the input data (all positive integers) are: P, the maximum weight supported by the backpack; $(p_1, ..., p_n)$ and $(v_1, ..., v_n)$, respectively the lists of weights and values of n objects. The goal is to determine a set of objects whose overall weight is not greater than P and whose overall value is maximum.

The input data of an instance could be for example: $P = 12$, weights $=$ $(1, 2, 3, 3, 5, 6)$, and values $= (2, 4, 6, 6, 7, 9)$; so, $n = 6$. Generally speaking, the value does not necessarily increase with weight and the order in which the objects are arranged does not matter; moreover, the two objects with weight 3 and value 6 could be exactly the same or not.

The first idea that usually comes to mind (even to very young students) is to choose – from time to time – the most valuable object that can still fit in the backpack. It is expressed by a *greedy algorithm*, which in our case leads to the choice of objects of value $9, 7$, and 2 (18 in total) and weighing $6, 5$, and 1 (12 in total, leading to a full backpack). But can a higher overall value be achieved?

A side conversation can be started now, illustrating a problem for which there is a greedy algorithm that always leads to an optimal solution (e.g. the aforementioned Kruskal's algorithm to obtain an MST), and yet another classic problem: composing a given sum with the minimum number of coins, of certain denominations, since we dispose of them at will of each denomination. Suppose we choose the largest possible denomination at each step. Well, characterizing the coin systems (each one represented by its list of denominations), for which this greedy procedure leads to the minimum number of coins, is still an open question if there are more than five available denominations. It can be seen that this problem is actually a variant of *Knapsack*...

Let's start from another reflection: if we were handed over an already partially filled backpack, we will still get the greatest profit by maximizing the value of the objects that can still be added, choosing them among the remaining ones. Overturning the view: supposing that the capacity c of the backpack increases progressively from 1 to P, at each stage let's ask ourselves what is the maximum value, achievable by having only the first object, or the first two... or all the n objects (as we said, the order does not matter). When we have to decide whether to choose the k-th object or not, assuming that its weight does not exceed c, we will choose it only if its addition manages to improve the overall profit.

This idea leads to the well-known *dynamic programming algorithm*, which uses a support data structure consisting of a matrix M of integers, with $P+1$ rows and $n+1$ columns, where the first row and the first column (which we suppose have index 0) are filled with 0's (if the backpack cannot contain anything or there is no object, then its value is 0):

```
**for** $c$ **from** 1 **to** $P$
    **for** $k$ **from** 1 **to** $n$
        **if** $p_k > c$ **then** $\mathrm{M}[c,k] := \mathrm{M}[c,k-1]$;
        **else** $\mathrm{M}[c,k] := max\{\mathrm{M}[c-p_k,k-1]+v_k,\ \mathrm{M}[c,k-1]\}$;
```

After this, the element $\mathrm{M}[P,n]$ contains the value of an optimal solution – of course, the correctness of any algorithm should always be proved...

To find out which objects to choose, we proceed in the following way:

```
$c := P; k := n$;
**while** $c > 0$ **and** $k > 0$ {
    **if** $\mathrm{M}[c,k] \neq \mathrm{M}[c,k-1]$ **then**
        { choose the $k$-th object; $c := c - p_k$; }
    $k := k - 1$;
}
```

In our case, the application of this algorithm leads us to choose the objects in places $5, 4, 3$, and 1, with a total value of 21, and still a full backpack; students can understand the why by doing a hand simulation. Here are other equally good solutions yet: which ones? Which solution is found depends on the order in which the objects (with their respective p_k and v_k) are initially arranged!

At this point, students can be easily stimulated to make some further considerations, on slightly larger instances: what would be the value of the stolen goods if the strategy used by the thief was to always choose "the lightest object" that can fit in the backpack? Or, otherwise, "the object that has the best value/weight ratio"?

If the aim is to fill the backpack as much as possible, then just match the values of the objects with their respective weights. Considering only weights, a formulation of the *decision problem* (with answer yes or no) asked to establish if the backpack can be filled *exactly* with its maximum weight P, which is equivalent to ask if the equation

$$\Sigma_{i=1}^{n} p_i x_i = P \tag{1}$$

has solutions when each unknown value x_i is 0 or 1: this is the *Subset-Sum* problem, which is also found in cryptography. Incidentally, asking the same question for a linear system with integer coefficients does not change the complexity of the problem, which is called *integer (linear) programming 0-1*.

A significant variant of *Knapsack* assumes the unlimited availability of specimens of each type of object; the procedure requires less memory space (two arrays, each of $P + 1$ elements, instead of the matrix M) but its first part still consists of two nested **for** loops... Talking about computational complexity, the two nested loops would suggest a polynomial time (nP), but which is actually exponential in the size of the input data expressed in bits. Nevertheless, when the input is a list of numbers and the execution time is bounded by a polynomial in the greater of these numbers and in the length of the list, we speak of *pseudo-polynomial algorithms*. And this is our case, so *Knapsack* is *NP-hard* but not *strongly* (unlike, e.g., TSP and *Bin-Packing*).

How to explain in simple terms what an "NP-hard" problem is? "NP" stands for "Non-deterministic Polynomial" and means that the problem could be generally solved in polynomial time by a hypothetical algorithm such that, if a solution of the problem exists, at each step it makes the right choice to arrive at the *closer solution*, whereas, if the problem does not admit any solution, at each step it makes the choice leading to *further failure*. In practice, but for the purposes of our discussion: it is included in those problems to solve which we do not have a time-polynomial algorithm in the bit length of the input data, and we don't even know if such an algorithm there exists – probably not. "Hard" means that any NP problem can be reduced to it (i.e., any instance of any NP problem can be transformed in a suitable instance of it) in polynomial time.

Lastly, you could propose some instances of problems easily solvable by applying the idea of dynamic programming (referring to those processes in which it is required to find the best decisions one after the other, in particular for multistage ones, whose evolution can be described by a directed acyclic graph); it is one of the most general algorithmic techniques together with linear programming and backtracking. For instance: finding an optimal path from the first to the last column of a chessboard, in which each square has its own cost (or benefit) of visit, bearing in mind – as a general principle – that each terminal part of an optimal path is itself an optimal path, for a smaller sub-problem.

2.2 A second case study: some "hard" problems on graphs

Starting from a few simple tasks proposed in Bebras competitions, let's see how we can formulate and solve some "hard" problems on unweighted and undirected graphs. For instance, in 2016, the Swiss group produced the scenario sketched in Fig. 1 for a question, the starting point of which was given by a community of beavers who must decide in which of the ten places to build three medical stations for primary health care, so that from each of the other places it is possible to reach a station by swimming through only one water canal.

In general, when a set of vertices of minimum cardinality must be found in an undirected and connected graph, such that each of the other vertices is adjacent to (i.e. joined by an edge with) at least one of these, we have to solve a *Minimum Dominating Set* (or MDS) problem, which is NP-hard. (Incidentally, the proposed task has eight solutions of cardinality 3, which is the minimum.)

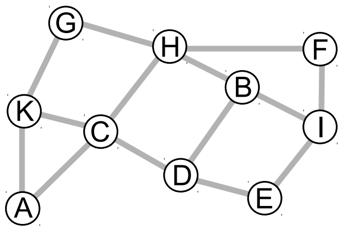

Fig. 1. The ten places of the beavers' community and the canals connecting them.

But, using the same scenario as base, let's ask ourselves another question: since from each place we can check all the water canals having an end at that place, what is the minimum number of places in which to install a watchtower, so that all canals are monitored? In general, this is called *Minimum Vertex Cover* (or MVC) problem, NP-hard too. If the graph, in addition to being unweighted and undirected, is simple (i.e. without parallel edges), connected and without "loops" (i.e. edges whose endpoints coincide), then it can be represented by a list of pairs (u, v) with $u < v$, denoting the n vertices with the numbers from 1 to n. The goal is to determine a set of vertices of minimum cardinality, such that every edge of the graph has at least one of the two endpoints (i.e., is incident to at least one vertex) in this set.

The first greedy idea usually coming to mind to any good student is the following: to construct the vertex cover X (initially empty), at each step choose a vertex v with maximum degree (the degree of v is the number of edges incident to v), insert v into X, and delete both v and the edges incident to it (and any vertices that remain isolated, too) from the graph. In our case, at the first step we have two alternatives: C or H, both with degree 4; if we choose H, any subsequent choice, with the same maximum degree, finally leads to one of the two minimum covers {H, K, I, D, A} and {H, K, I, D, C}, both of 5 vertices. But if we initially chose C and, at the second step, B (with degree 3), then we would not find a minimum vertex cover. Therefore, this procedure does not always work; in fact, it is not even a constant-factor approximation algorithm: in bad cases, the ratio between the number of selected vertices and the number of vertices in a minimum cover tends to grow (although logarithmically) as the number of vertices in the graph increases.

A second greedy idea arises from observing that, for each edge, at least one of the two endpoints must be in a minimum vertex cover. So, to build the set X, at each step we choose (at random) an edge, let's say the one between the two vertices u and v, we insert both u and v into X, and we delete u and v, as well as each edge incident to u or v, from the graph. In the proposed example, no choice leads to an optimal solution; if we indeed are unlucky, we will even take all the vertices! In general, in the worst case, the cardinality of the resulting vertex cover is twice the minimum: this is therefore an *approximation algorithm with a constant factor* of 2. For instance, in the case of $K_{n,n}$ (i.e. the bipartite graph with n vertices in each of the two parts, and each vertex of one part joined by an edge to each vertex of the other part) all $2n$ vertices are taken, whereas each of the two minimum covers contains only n vertices. (It should be noted that, however, on any bipartite graph the problem can be solved efficiently.)

One could think of combining the two criteria, choosing an edge not at random, but with the maximum sum of the degrees of its two endpoints. In the case of $K_{n,n}$ there would be no benefit; in our example, we would get a vertex cover of 8 vertices: it doesn't work! In truth, as we said, MVC is an NP-hard problem.

Nevertheless, the second greedy algorithm suggests an easy way for a procedure that gives an *optimal solution*: we *also* calculate what is obtained by inserting only u or only v into X, bearing in mind that, when adding only one of the two, we must necessarily add at the same time also all vertices adjacent to the other. For instance, in Fig. 1, if we consider the edge between C and H, choosing H and excluding C, we need to include D, A, and K. We can finally formulate an algorithm to achieve an optimal solution, given the graph G.

a. If G has no edges, then return the empty set (of vertices). End.
b. Otherwise, let (u, v) be an arbitrarily chosen edge of G; recursively, and if possible in parallel, solve the following three problems:

1. $X := \{w | (w, u) \text{ is an edge of } G\}$ and let G_1 be the graph obtained by removing from G the vertices belonging to X and the edges incident to them; let X_1 be the solution found when G_1 is given; finally, $X_1 := X_1 \cup X$.
2. $X := \{w | (w, v) \text{ is an edge of } G\}$ and let G_2 be the graph obtained by removing from G the vertices belonging to X and the edges incident to them; let X_2 be the solution found when G_2 is given; finally, $X_2 := X_2 \cup X$.
3. Let G_3 be the graph obtained by removing from G the vertices u and v and the edges incident to them; let X_3 be the solution found when G_3 is given; finally, $X_3 := X_3 \cup \{u, v\}$.

c. Return the set with the least cardinality, between X_1, X_2, and X_3. End.

If k is the number of vertices belonging to a minimum cover, you can prove that the depth reached by the algorithm just described is at most k, and therefore its time-complexity has a factor of 3^k.

MVC can be seen as a particular case of *Minimum Set Cover* (or MSC), a problem that consists in choosing the least number of sets (from a given family of sets) whose union coincides with a given "universe" (often, the family union). To be convinced of this fact, just consider, for each vertex, the set of edges incident to it and, as "universe", the set of all the edges of the graph.

MSC is *equivalent* to MDS. In fact, if we consider the time required to solve them exactly, we can say that they are equally difficult; more precisely, any instance of one can be transformed, in polynomial time, into a corresponding instance of the other. Let's reduce MDS to MSC: with each vertex v we associate the set consisting of v and its neighbors; once covered with the least number of such sets the set of all the vertices, it will be enough to consider – as a solution of MDS – the vertices with which such sets are associated. You can let mature students prove the other way around – a little more challenging.

In computer science, it may sometimes be convenient to transform our problem into another equivalent to it (or more general), to solve which we already have special software; but then we also need to know how to interpret (easily) the results, to get the solution of the original problem.

If we consider the complement of a minimum vertex cover w.r.t. the set of all the vertices in the graph, we get a solution of another interesting optimization problem: finding the *Maximum Independent Vertex Set* (or MIVS), a set of vertices of maximum cardinality which, two by two, are *not* joined by an edge.

If X is a set of vertices, the following statements are equivalent:
- X is an independent set
- every edge is incident to at most one vertex belonging to X
- every edge is incident to at least one vertex not belonging to X
- the complement of X is a vertex cover.

A maximum independent set complements therefore a minimum vertex cover.

The *complement graph* of a graph G has the same vertices as G, none of the edges of G, and all edges (between two different vertices) that are *not* in G. An independent set is made up of the vertices belonging to a complete subgraph (*clique*) in the complement graph, and therefore each solution of a MIVS problem is also a solution of the *Maximum Clique* problem on the complement graph, and vice versa: they are therefore computationally equivalent problems. The example in Fig. 2 clarifies what has just been said.

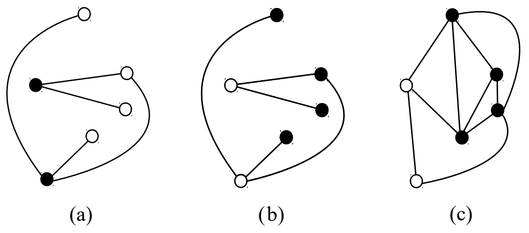

Fig. 2. In black, examples of *Minimum Vertex Cover* (a) and *Maximum Independent Vertex Set* (b); *Maximum Clique* in the complement graph (c).

If X is a maximum independent set, for each vertex v of the graph either v or at least one of its neighbors belongs to X. To solve exactly an MIVS problem, it is therefore possible to solve several of its reduced size instances, taking then the best; it is hoped to contain in this case the number of such instances, choosing – from time to time – a vertex with minimum degree. The detailed design of an appropriate procedure can be left to mature students.

To conclude, for all the aforementioned "hard" problems on graphs, it can be proposed to solve, recursively and if possible in parallel, two or more sub-problems of reduced size, and then to package an exact solution using one of those of the sub-problems that has certain characteristics. Doing so, time grows however exponentially with the number of vertices. The best algorithms for MIVS so far conceived have indeed a time-complexity of the form c^n, where n is the number of vertices in the graph and c is a constant slightly greater than 1.2.

3 Puzzles

There are many fun puzzles that can be solved, more or less laboriously, with the help of a computer. Some of them are easily "programmable" using the *backtracking* technique, actually based on a "trial and error" procedure. If we get to an impasse, the idea is to go back to the last previous crossroads and choose another path, and if there are no other possibilities, then we backtrack one more step back, and so on. Sooner or later, either a solution is reached or all possible paths have been tried in vain and fruitlessly; in the second case, then, the particular instance of the problem does not admit any solution.

If a solution really exists, the one we will find might be entirely contained in the "final state", and it doesn't matter *how* we got there. To avoid running into loops, if there is this danger, we will have to leave a trace of the path – which will eventually indicate the solution found – in the current state. Examples of this type of puzzle are: Sudoku; classic puzzles with interlocking tiles, without knowing the final image or the final arrangement of the tiles; the drawing of a tour (open or closed) of the knight on a chessboard (square or rectangular or other shape), or the design of a path to get out of a labyrinth: the unknown final state, which must be reached, is precisely the solution to the problem. We could also force the continuation of the process after finding a solution and perform an exhaustive search for *all* solutions, providing that the time required does not become prohibitive.

The completion of a Sudoku scheme can also be seen as an instance of *Exact Cover Set* (or ECS; an NP-hard problem which, unlike MSC, requires that each element of the universal set be covered once and only once, i.e., the sets chosen are two by two disjoint, but not necessarily as few as possible) or dealt – at least for the most part – exploiting *ad hoc* techniques.

One of the first programs that used the backtracking technique is due to the eminent logician Dana S. Scott, who created it in 1958, with the help of Hale F. Trotter, just to get all the essentially different solutions of a combinatorial puzzle with the 12 *pentominoes* [8]: it was a question of forming an 8×8 square, with a 2×2 hole in the center. This puzzle can be traced back to three instances of ECS. Incidentally, there are 65 different solutions, found by Scott-Trotter's program in about 3.5 hours.

Then there are puzzles a little more complicated to "program", where a solution (if any) consists in the sequence of transitions (or moves), not known, which led from the initial state to the final state (of success), perhaps already known (that is, you know where you want to go, but you don't know how). Sometimes there is no risk of falling into loops (e.g., when the pieces cannot retreat or one of them is removed with each move), but often, on the contrary, it is necessary to keep a list of the states visited along the path traveled so far. In some cases, it is interesting to look for the shortest solution, or one of the shortest, or even all the shortest ones: e.g., in the “English 16” puzzle (first published by W. W. Rouse Ball in 1892) or in the “15” puzzle (perhaps the most famous of the sliding-block puzzles). Of course, if you want to find all the shortest solutions, in addition to forcing backtracking, you need to keep in computer memory a set of solutions (and each solution is a list of states): when a solution of the same length as the others is found, it is added to them; if a shorter solution is found, then it replaces all the others. In the “English 16” puzzle, also known as “Fore and aft”, the same position cannot be repeated, and therefore it also makes sense to ask ourselves what the longest solutions are.

In many cases, we can usefully apply a *heuristic algorithm* that performs, for example, a *best-first search* (that gives precedence to the most promising moves, which give more hope to reach near the target, according to an appropriate evaluation function) in which the “cost” of a move is calculated as the sum of the number of moves already made starting from the initial state and the minimum estimated number of moves that remain to be made to reach the final state. If the estimate never exceeds the true number of moves needed to get a solution, then the procedure finds one of the shortest solutions (if at least one solution exists) in a hopefully reasonable amount of time. For instance, in the “15” puzzle, where each move involves the movement of a single tile, an estimate can be made by adding the “Manhattan distances” between each tile and its final position. In this way a lower bound is certainly achieved for the number of remaining moves, since each tile must move at least as many times as the places (both horizontally and vertically) that separate it from its final position.

4 Strategy games

Let’s consider now strategy games between two competitors, especially zero-sum games with perfect information, in which a computer can be programmed to play (if possible at best) against a human or another software. The popular *tic-tac-toe*, possibly dating back to Ancient Egypt, was the subject of the first video and graphical computer game, for EDSAC at the University of Cambridge, in 1952. As is well known, it is a “futile” game: if neither player makes mistakes, the game ends in a draw. However, it can be used as a base to understand some fundamental ideas for solving (in a strong sense) a game, such as the *minimax* algorithm (maybe in its simplest *negamax* version) and *alpha-beta pruning* (or even more modern pruning techniques). You can explain then how to design, instead, a program stopping the analysis at a certain depth; consider, e.g., some of the simplest variants of the *Mancala* family, where a non-final state can be heuristically evaluated by a subtraction of pieces (or material).

Within the reach of younger students there are the procedures to determine which of the two players has a winning strategy in some *impartial combinatorial games* (characterized by no distinction of material, i.e. the same moves are available to each player, and no possibility of a draw), such as:

- (normal) *NIM*, the most classic game with matches, paradigmatic of the whole class of impartial combinatorial games: for its analysis, binary numbering and *exclusive or* operation are needed;
- *Euclid's game* [9], where, instead, the remainder of integer division, the golden ratio, and also the Fibonacci sequence have a decisive role;
- simple instances of other games on graphs; see, e.g., [10].

Attention should be paid both to *infinite games*, where if nobody makes mistakes the game doesn't end (e.g. *Pong Hau K'i* and *Picaria*), and to those *dummy games* in which it *seems* that the two opponents are playing a game, whereas – in reality – the outcome of each game is established *a priori*, whatever the choice made by the turn player from time to time. For instance, I propose the following didactic one: let's place n identical coins on the tabletop; the two players take turns stacking two stacks of the same height (each player's first move is forced, if $n > 3$); and when there are no two stacks of the same height, the player in turn loses. Once n is fixed, all games end after the same number of moves and with the same winner (either the first or the second player). At the end of each game with n coins, on the table there will be indeed a stack of k coins for each k in the representation of n as sum of powers of 2. Discarding therefore the least significant bit of the binary number n and then counting the 1's, if these bits are odd then the first player wins, otherwise the second wins. The sequence that associates the winning player with each n is related to the famous non-periodic Prouhet-Thue-Morse sequence [11].

Suppose, instead, of starting with a single stack of n coins; now a turn consists of splitting a single stack into two stacks of different heights, and a player loses when he cannot move because all stacks have height 1 or 2. This game, proposed by P. M. Grundy in 1939, is not a dummy game; however, the characteristics of the winner's sequence are still unknown: it is conjectured that it eventually does become periodic [1], but the question is an unsolved problem.

In certain cases, the same player always has a winning strategy: e.g. the first player in (normal) *Kayles*, for any row of $n > 0$ pins.

5 Logical and algorithmic riddles

I'd like to recommend the logical riddles concerning with informatics and computers: above all, those designed by the eclectic and brilliant Raymond Smullyan, and not only in the style of the classic questions involving logical deductions. In his book *The Lady or the Tiger?* [12] (chapters 9, 16, and 17) he proposes formidable questions that highlight – and magically show! – both the undecidability of Turing's halting problem and Gödel's first incompleteness theorem. These themes are then explored in the subsequent *Satan, Cantor, and Infinity* [13]; in its riddles, some languages that have few essential rules are involved too.

Puzzles with marbles of two or more colours (that combine in sequences or stack in cylinders) can be invented to understand certain properties of *string rewriting systems* (see Fig. 3 and [6], pp. 349–351) or *Post production systems* (see [6], pp. 356–357), and their relationship with Turing machines.

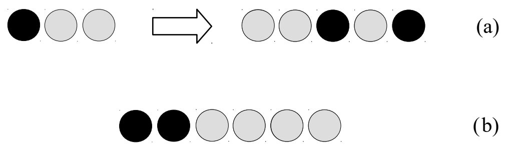

Fig. 3. An example of string rewriting system with only one rule (a). Starting from any sequence of marbles, for instance (b), as long as possible, replace a single occurrence (at a time) of the subsequence on the left in (a) with the one on the right. In this case, the process always ends, whatever the initial sequence; furthermore, when multiple substitutions are allowed in a rewrite step, all alternatives lead to the same final sequence (called *normal form* of the initial sequence), i.e. the system is *confluent.* If in rule (a) we add a grey marble at the head of the sequence on the right, then we get a system that does not always terminate.
As a general result, if the system is finitely terminating, then the confluence is decidable; and if it is confluent too, then each sequence has one and only one normal form.

You can moreover exploit instances of *Post's correspondence problem*, in general only semi-decidable (see Fig. 4 and [14]).

In the form of educational tasks, you can formulate new problems; some have appeared in past Bebras competitions, as well as several tasks useful for introducing different classes of *formal grammars* to define artificial languages.

6 Graphic amusements

Among the many instructive graphic entertainments, let me cite the *Photomaton* [15] (an image processing that repeats a peculiar redistribution of the same pixels and, after surprising effects, finally returns to the initial image) and the *Persian recursion* [16] (to draw a Persian carpet by assigning a colour to only the four pixels at the corners and then triggering a recursive procedure that applies itself four times, if both sides of the carpet are greater than 2 pixels).

Starting from *Voronoi's diagrams*, spectacular images – but not cumbersome to get with the computer aid – are shown and explained in two papers written by C. S. Kaplan and available on the net [17, 18].

Cellular automata in one or two dimensions (from S. M. Ulam and J. von Neumann to J. H. Conway and S. Wolfram; visit the truly inexhaustible mine [19]) offer the vision of amazing evolutions in their "digital world", together with their ability to emulate the behavior of complex systems and even with their universal computing power.

Finally, I would like to recall the *turmites* (C. G. Langton and others; see [20, 19]), i.e. *Turing machines in two dimensions*, automata able to move on a plane, partitioned into square cells of various colours; these colours, in finite numbers,

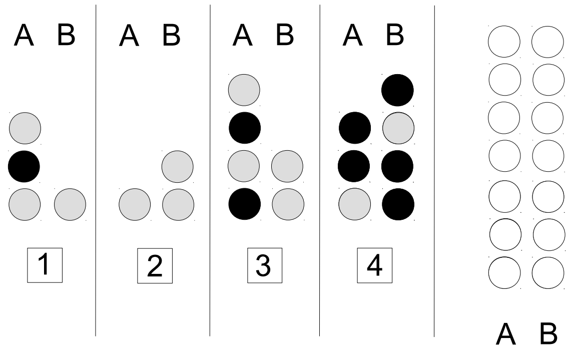

Fig. 4. Whenever you click a numbered button, the corresponding pair of marble stacks falls, from above, into the two cylinders A and B on the right. In these cylinders, initially empty and as high as you want, you have to build two identical nonempty stacks. In this case, apart from repetitions, the only solution is to click 2, 4 and 1 in this order; so button 3 is useless. But, without the two marbles at the base of button 4 (a grey marble at the bottom of stack A and a black one at the bottom of stack B), the task would be impossible.
In general, there is no algorithm able to decide whether two identical stacks can be built or not, given an arbitrary set of $n > 4$ pairs of marble stacks (i.e., in our riddles, n buttons); when $n = 2$ the problem is decidable, but with $n = 3$ or $n = 4$ we don't know (see [6], pp. 352–355).

play the role of the symbols of the alphabet in a classic Turing machine. One of the cells is occupied by an automaton that can move (orthogonally) only one cell at a time, maybe after changing the colour of the cell it just left, thus composing who knows what fascinating patterns, perhaps related to known mathematical constructions, like a Fibonacci spiral!

7 Conclusions

Here I presented two case studies that develop different topics, starting from examples of simple tasks up to the discussion of complex notions step by step. I then gave suggestions or ideas for fun tasks or games, which can be used very well to introduce fundamental algorithmic and programming techniques. Examples like these have been successfully tested over several years in teaching students aged between 16 and 19, but more recently the basic ideas have also been passed on to primary school children. This step has been encouraged by engaging and challenging games designed specifically for younger pupils, which inspired a compelling series of books and teaching materials to pleasantly introduce informatics from nursery school up to the age of 15 [21].

The firm belief remains that the learning of a large part of informatics can be considerably facilitated by this kind of approach, which is always stimulating for students and appreciated by them, and which often brings fruitful didactic repercussions even in other branches of knowledge.

References

1. Berlekamp, E. R., Conway, J. H., Guy, R. K.: Winning Ways for your Mathematical Plays. 2nd edn. 4 vols. A. K. Peters Ltd., Wellesley, MA (2001–2004).
2. Bewersdorff, J.: Luck, Logic, and White Lies. The Mathematics of Games. A. K. Peters Ltd., Wellesley, MA (2005).
3. Shasha, D. E.: Puzzles for Programmers and Pros. Wrox Press/Wiley, New York, NY (2007).
4. Knuth, D. E.: Selected Papers on Fun and Games. Lecture Notes, Vol. 192. CSLI Publications, Stanford University, Stanford, CA (2011).
5. Levitin, A., Levitin, M.: Algorithmic Puzzles. Oxford University Press, New York, NY (2011).
6. Repetto, L.: Dai giochi agli algoritmi. Un'introduzione non convenzionale all'informatica. 2nd edn. Edizioni Kangourou Italia, Monza, Italy (2019). https://drive.google.com/file/d/1tMllvuRregMCwwYb-c25DlhJ7FxsPaM8/view?usp=sharing
7. Bebras Homepage. https://www.bebras.org (for resources by the Italian group: https://bebras.it/materiali.html)
8. Pentomino. https://en.m.wikipedia.org/wiki/Pentomino
9. Cole, A. J., Davie, A. J. T.: A game based on the Euclidean algorithm and a winning strategy for it. Mathematical Gazette 53(386), 354–357 (1969).
10. Edmonds, J. R., Gurvich, V. A.: Games of no return. RUTCOR Research Report, RRR-4-2010. Rutgers University, New Jersey (2010).
11. The On-Line Encyclopedia of Integer Sequences. https://oeis.org/A010060
12. Smullyan, R. M.: The Lady or the Tiger? And other logic puzzles including a mathematical novel that features Gödel's great discovery. A. A. Knopf Inc., New York, NY (1982).
13. Smullyan, R. M.: Satan, Cantor, and Infinity: and other mind-boggling puzzles. A. A. Knopf Inc., New York, NY (1992).
14. Post's Correspondence Problem. http://webdocs.cs.ualberta.ca/%7Egames/PCP
15. Delahaye, J.-P., Mathieu, P.: Images brouillées, Images retrouvées. Pour la Science 242, 102–106 (1997).
16. Burns, A. M.: "Persian" Recursion. Mathematics Magazine 70(3), 196–199 (1997).
17. Voronoi Diagrams and Ornamental Design. http://www.cgl.uwaterloo.ca/%7Ecsk/papers/kaplan%5Fisama1999.pdf
18. Aliasing Artifacts and Accidental Algorithmic Art. http://www.cgl.uwaterloo.ca/%7Ecsk/papers/kaplan%5Fbridges2005a.pdf
19. Stephen Wolfram's A New Kind of Science. https://www.wolframscience.com/nks/
20. 2D Turing Machines. http://www.mathpuzzle.com/MAA/21-2D%20Turing%20Machines/mathgames%5F06%5F07%5F04.html
21. Hromkovič, J., *et alii*: Einfach Informatik (Zyklus 1, 5/6, 7–9). Klett und Balmer Verlag, Baar, Switzerland (2018–2020).