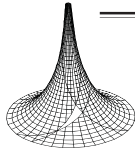


**3/2** (2005), 221–240
tmcs@math.klte.hu
http://tmcs.math.klte.hu

**Teaching**
**Mathematics and**
**Computer Science**

# "Upperview" algorithm design in teaching computer science in high schools

Zoltán Kátai

*Abstract.* In this paper we are going to present a teaching/learning method and suggest a syllabus that help the high school students look at the algorithm design strategies from a so called "upperview": greedy, backtracking, divide and conquer, dynamic programming. The goal of the suggested syllabus is, beyond the presentation of the techniques, to offer the students a view that reveals them the basic and even the slight principal differences and similarities between the strategies. In consensus with the Comenius principle this is essential, if we want to master this field of programming (*"To teach means scarcely anything more than to show how things differ from one another in their different purposes, forms, and origins. … Therefore, he who differentiates well teaches well."*).

*Key words and phrases:* teaching method, algorithm design strategies, algorithm design techniques.

*ZDM Subject Classification:* B20, B50, B70, C70, P00, P50, Q00.

## 1. Introduction

Comenius, considered the founder of modern teaching, made the following statement regarding teaching methods: *"To teach means scarcely anything more than to show how things differ from one another in their different purposes, forms, and origins. … Therefore, he who differentiates well teaches well."*

We are going to present a teaching/learning method and suggest a syllabus that help the high school students look at the algorithm design strategies from a so

Copyright © 2005 by University of Debrecen

called “upperview”: greedy, backtracking, divide and conquer, dynamic programming. We are in some places referring to these strategies as algorithm design techniques (short: techniques). The official Romanian curriculum does not include the strategy of dynamic programming, but in the programming contests high school students often have to deal with tasks which assume its use. The presented method offers the teacher the possibility to introduce the basic elements of this technique at the level of the whole class. The goal of the suggested syllabus is, beyond the presentation of the techniques, to offer the students a view that reveals them the basic and even the slight differences and similarities between the strategies. In consensus with the Comenius principle this is essential, if we want to master this field of programming.

Several issues of the bibliography compare the techniques. For example the authors Cormen, Leiserson and Rivest in their book “Introduction to Algorithm” [1] compare the dynamic programming and the greedy strategy. Some books (for example [2]) discuss how the backtracking and greedy techniques can complement each other. In other books (for example [3]) we can find a comparing analysis concerning the strategies of divide and conquer and dynamic programming. In the present paper we have developed this idea and we are presenting a method which makes it possible to discuss uniformly all the above mentioned techniques. We tried to establish such an “upperview” where each technique can be seen in the same time next to each other. By this means it becomes possible to integrate all the four techniques into a frame which forms a whole. If the students recognize the position of certain techniques related to the others, then the so called “more difficult” strategies become available for them.

Firstly we give a general description of the “upperview-method” and then we apply it in the field of algorithm design techniques. Thereafter we present a possible syllabus which offers the possibility to use the “upperview-method” in teaching. Eventually we report about an experiment in which we prove the efficiency of the method in an empirical way.

## General description of the “upperview-method”

What does it mean to see something from above?

Imagine the following situations: at the police they pin the evidence coming from different sources regarding a criminal case on the notice board. Why? The town planning department of the mayor’s hall produce a model and they stand

round it. Why? To get a general picture of the whole and also the similarities, differences and the connections between different parts to be more perceptible.

In the two situations the involved parties carried out the "upperview" in a different way. The policemen needed a "platform" (the notice board) where they could see the pieces of evidence displayed next to each other. The architects used reducation and abstraction in producing the model. The abstraction is important because it disregards from those aspects which are not important.

From the joint of the two methods it can be noticed that, in order to carry out an "upperview", a so called "abstract platform" might be necessary, where the entities being analyzed can be laid down next to each other in such a manner that the features and connections essential for the analysis become obvious.

We use the notion of "upperview" in our article in the following sense:

(1) We can see the entities being analyzed "next to each other".

(2) Only those elements can be seen which are essential for the analysis.

(3) The similarities and differences are obvious, the connections are striking.

The purpose of the method is to offer the student a position where to have such a view to the analyzed object. Of course several "upperviews" could be produced depending on how we make the mentioned steps. This is desirable, too, as each new "upperview" means a different point of view. One of the strengths of the method is that it enables the students to assess the "value" of the analyzed entities related to each other. For example in the case of the study of algorithm design strategies the strong and weak points of certain techniques become obvious, as well as the fact that in a given situation which is the most efficient to be used and why.

The basic principle illustrating the efficiency of the method can be checked with the following simple experiment: let's show a sheet of paper and ask the students to name as many of its characteristics as they can. Then – in an other classroom – we repeat the experiment, but this time we show them another shape together with the sheet of paper (for example, something made of wood, not a plane figure, not one single colour etc.).

What are we going to experience? In the second class the students can express considerably more characteristics of the sheet of paper. For example, it is not sure that in the first class they notice that the sheet is one-coloured, it is a plane figure, it can be crumpled, it is made of wood etc. This simple experiment proves a well-known truth: *The contradictions call our attention on the differences but also on the similarities.*

What was the role of the teacher in this experiment? To establish the "upperview" in the student's mind. It presumed to select the object laid next to the sheet of paper in such a manner to make those criteria striking, based on which he would like the students to make the comparison.

## "Upperviews" in teaching computer science

How can we carry out "upperviews" at the Computer Science classes? Most of the teachers are doing it – even if they might not call it like this – when teaching sorting algorithms. At the end of the chapter we take a sequence of numbers, on which we mime or have mimed all the taught sorting algorithms by the students, drawing their attention to the similarities and differences. Doing so, we form "upperviews" in the students, as:

- We lay the sorting algorithms "next to each other" by miming them on the same sequence of numbers.
- We draw the students' attention to what is essential and what is not from a certain "upperview". For example in the case of sorting algorithms based on comparison, the students can concentrate on the kinds of strategies used to compare the elements from a certain "upperview".
- We help the students to see the similarities and differences, the strong and weak points of the algorithms.

There are excellent computer simulations which make the production of the "upperview" easier concerning the sorting algorithms.

## Design of Algorithms from "upperview"

How can we carry out "upperviews" when teaching algorithm design strategies? The challenge consists in the fact that while the sorting algorithms solve the same problem, each algorithm design strategy has – more or less – its own territory. The picture below shows this (see Figure 1). The certain circles describe the set of those problems, which can be "solved" with the respective strategy, no matter whether they offer an optimal solution or not, is the algorithm efficient or not.

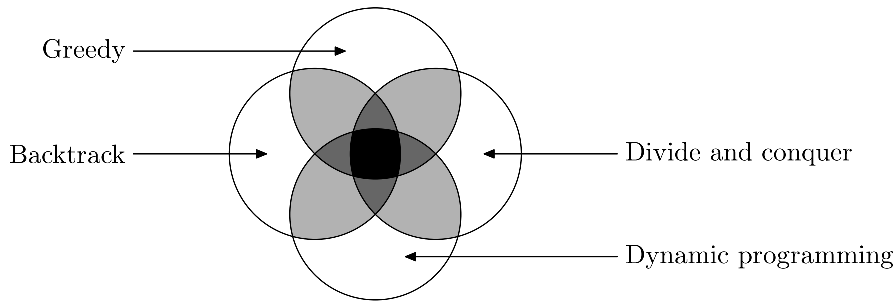



*Figure 1*

The fact that the circles intersect each other shows that there are problems which can be "solved" with several techniques, even more, some of them with all the techniques. Therefore we can conclude that there must be a "plane" where the sets of problems of certain techniques have basic similarities. Which is that "plane"?

## An "Abstract Platform"

The backtracking and the divide and conquer techniques usually approach the problem in a recursive way. The sequence of ideas in a recursion is the following: How can the problem be reduced to similar, simpler subproblems, then later again reduce these ones to similar, even simpler subproblems, until we get trivial subproblems? This type of break-down presumes that the problem – in its construction – should have the structure of a tree. The whole tree obviously represents the problem itself; the first level subtrees represent those subproblems the original problem can be reduced to in the first step, and so on. Eventually the leaves of the tree will represent the trivial subproblems resulted from the break-down.

The common characteristic of the greedy and dynamic programming techniques is that we usually apply them for problems that can be regarded as sequences of decisions. This leads again to a tree structure, where the root represents the original state of the problem, the first level nodes those states the problem might get into after the first decision, the second level nodes represent the ones resulted from the second decision etc. A node will have as many children as many choices we had when the selection was made at the respective decision.

Considering all this, we can say that we apply all the techniques we are going to present especially in the case of problems which in some consideration have

the structure of a tree. From the point of view of the techniques this means that each one considers the problem as a tree. Well, this common tree structure is that common plane or abstract platform – necessary for the "upperview" – where the techniques can be laid next to each other.

### Proposed Syllabus

The application of the "upperview" method according to the below presented syllabus requires basic programming skills from the students and a clear vision on such fields as recursion or tree structures. It is especially important that they know the tree structure's traverse modes. Should the curriculum plan the teaching of the rooted trees for later, it would be advisable to dedicate one-two hours for their presentation, without entering into details concerning the theory of graphs. We suggest the following syllabus:

(1) Revision of the recursion, presentation of the tree structures and their traversal.

(2) Through a demo problem solvable with each of the strategies, we offer a general and comprehensive image of the techniques. Without effectively solving the problem, we draft the certain strategies by having a conversation with the class.

(3) We present the backtracking technique.

(4) We go deeper into the backtracking strategy:
   - Practice specifically backtracking problems.

(5) We present the divide and conquer technique.

(6) We go deeper into the divide and conquer strategy:
   - Practice specifically divide and conquer problems.

(7) Divide and conquer and backtracking from "upperview":
   - We choose such problems which can be solved with both techniques.

(8) We present the greedy technique.

(9) We go deeper into the greedy strategy:
   - Practice specifically greedy problems.

(10) Backtracking and greedy from "upperview":
   - We choose such optimizing problems where the two techniques can be combined, completing each other.

(11) We present the technique of the dynamic programming.

(12) We go deeper into the technique of the dynamic programming:
- Practice specifically dynamic programming problems.

(13) Divide and conquer, greedy and dynamic programming from "upperview":
- We choose problems which make the comparison of the dynamic programming with divide and conquer as well as with the greedy possible.

(14) All the techniques from "upperview":
- With all the techniques, we solve the problem we used at the beginning to draft the basic characteristics of certain strategies.

At the classes presenting certain techniques we have to emphasise what it means from the point of view of the respective strategy to perceive a problem like a tree.

For the "upperview" classes we choose such problems which can be "solved" with each of the respective techniques. These are the classes where the similarities, differences and connections between the strategies are stressed.

Due to lack of space we cannot present in detail the concrete way these goals can be achieved, but a book with detailed material concerning the above syllabus is on its way. The following subchapter is giving an idea about it.

## Highlighting the similarities, differences and connections at the "upperview" classes

We are going to present a problem which can be used at the 2nd and 14th points of the above presented syllabus and which excellently illustrates the way the "upperview" method makes it possible to highlight the basic differences and similarities between the techniques. We are dealing with an optimizing problem, given in 1992 at the International Computer Sciences Contest in Sweden:

*Problem:*

On the main diagonal and in the triangle under the main diagonal of an `n`-row square matrix there are natural numbers. We presume that the matrix is stored in a two-dimensional array. Determine the "longest" path leading from peak (element `a[1,1]`) to the base (row `n`), taking into consideration the followings:

- On a certain path element `a[i,j]` can be followed by element `a[i+1,j]` (vertically down) or `a[i+1,j+1]` (diagonally to the right) where $1 \leq$ `i` $\leq$ `n` and $1 \leq$ `j` $\leq$ `i`.
- We consider the length of a path the sum of the elements found along the path.

For example, should for $\mathtt{n} = 5$ the matrix be the following:

| | | | | |
|---|---|---|---|---|
| 7 | | | | |
| 5 | 9 | | | |
| 10 | 1 | 4 | | |
| 2 | 7 | 3 | 1 | |
| 2 | 5 | 8 | 3 | 1 |

*Figure 2*

The "longest" path from the peak to the base is the shaded one, its length is 37.

Analyzing the problem we notice that we can reach the optimal solution by making $\mathtt{n} - 1$ decisions and for each decision we have 2 choices (which direction to go further, vertically down or diagonally to the right). With each decision the problem is reduced to a similar, but simpler subproblem. So the role of the abstract platform is fulfilled by the binary tree.

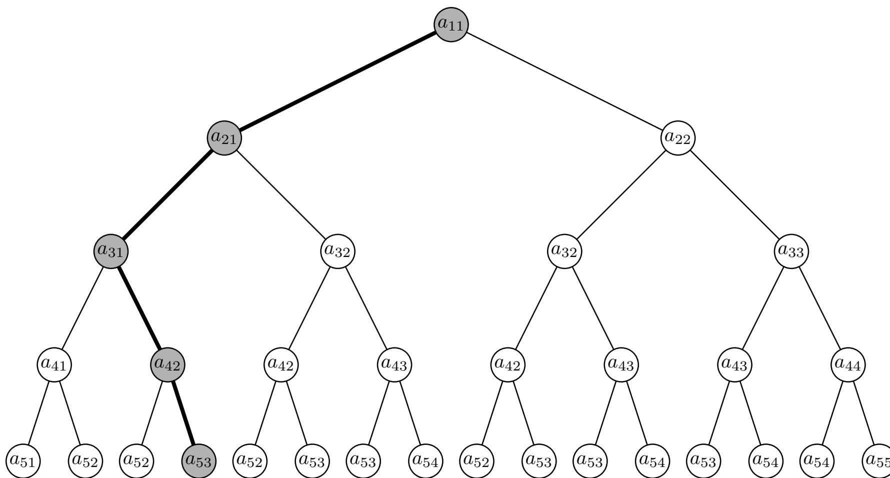



*Figure 3*

Determining the optimal solution means to find the optimal sequence of decisions. We can say that we have to find the best path out of the $2^{n-1}$ paths leading from root to leaf. In other words, we have to find the "best leaf" of the tree, to which "the best path" leads.

A more attentive traversal of the tree provides further observations:

(1) The number of nodes of the tree is $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$. This means that any algorithm which inspects the whole tree in order to find the optimal path will be of exponential complexity.

(2) While the tree represents the whole problem, its subtrees represent those similar but simpler subproblems (the leaves are the trivial problems) it can be broken down to. Concretely: the subtree with root $a_{ij}$ describes the problem of determining the "longest path" leading from element `a[i,j]` to the base.

(3) The above presented picture also highlights that different sequences of decisions can lead to the same subproblems, which means that the tree has identical subtrees. It is not difficult to notice that the number of different subproblems is identical with the number of the elements of the matrix, namely $n(n+1)/2$. Hence, the algorithm that could avoid the repeated solving of the identical subproblems will have a $O(n^2)$ complexity.

*In what follows we summarize the strategy of the four techniques regarding this problem:*

*Greedy:* We start from the node and we always go further toward the element which seems the most promising at a certain moment (it has the biggest value).

Observation: In this situation it is not sure that the greedy strategy will take us to the optimal solution. On the case of the example-matrix the greedy path has the length of 31, and this is the following: `a[1,1]`, `a[2,2]`, `a[3,3]`, `a[4,3]`, `a[5,3]`.

*Backtracking:* We generate all he paths leading from the node to the base and we select 'the best one'.

*Divide and conquer:* We can notice that the definition of any "best path" starting from the element $a_{ij}$ ($i < n$) can be reduced to the definition of the "best paths" starting from the elements $a_{i+1,j}$ and $a_{i+1,j+1}$, since as long as they are available, we only have to insert the element $a_{ij}$ before the longest one. In other words first we reduce the problem to simpler and simpler subproblems through the mechanism of recurrence and then we build "the best path" on the "way back" – taking into consideration the previous observation.

*Dynamic programming:* Starting from the obvious solutions of the trivial subproblems, we build the solutions of the more and more difficult subproblems, until we get to the solution of the main problem. As we would like to avoid the repeated solving of the identical subproblems, we store the optimal subsolutions in a bidimensional `c` array (the element `c[i,j]` of the array will store the length of "the best path" leading from the element $\mathtt{a}_{\mathtt{ij}}$ to the base.) We fill the main diagonal and the triangle under the main diagonal of array `c` from bottom to top, row by row, taking into consideration that

$$\mathtt{c[i,j]} = \mathtt{a[i,j]} + \max(\mathtt{c[i+1,j]}, \mathtt{c[i+1,j+1]}), \quad \mathtt{i} < \mathtt{n}$$

Eventually the length of the optimal path will be in `c[1,1]`.

In order to have the path itself the filled array `c` contains enough information (in contrast to array `a`) to write it, advancing by way of optimal decisions.

| 37 | | | | |
|---|---|---|---|---|
| 30 | 25 | | | |
| 25 | 16 | 15 | | |
| 7 | 15 | 11 | 4 | |
| 2 | 5 | 8 | 3 | 1 |

*Figure 4*

## Criteria of comparison

(1) *How do the certain techniques traverse and "prune" the tree that can be associated to the problem?*

*Greedy* prunes the tree starting from the root and going toward the leaves, cutting whole subtrees from it. It traverses only one root-leaf path, giving thus the quickest algorithm (its complexity is linear), but unfortunately – at least in the present case – cannot give any guaranties of finding the optimal solution (see Figure 5).

The *backtracking* does not prune at all the tree in the present situation, as it sees a potential solution in any root-leaf path. In order to find them it traverses

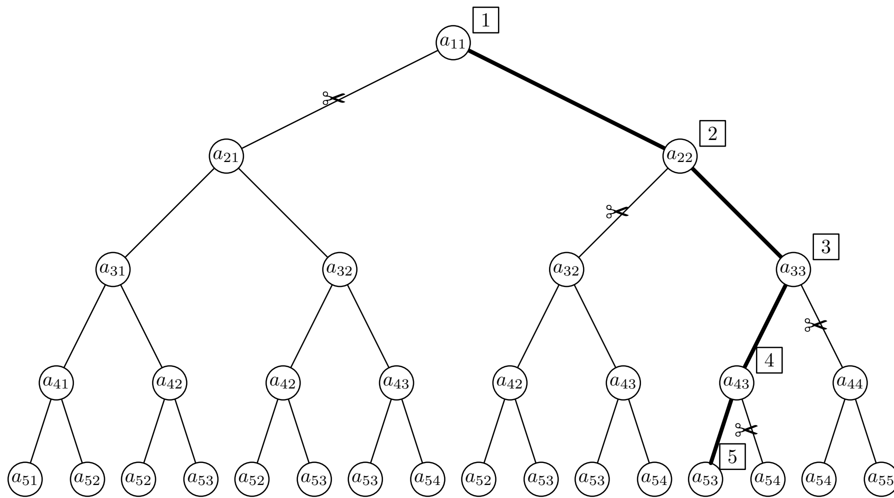



*Figure 5*

the tree in its depth, dealing with the nodes in a preorder sequence. As the number of the tree’s nodes depends exponentially on `n`, its algorithm will obviously have an exponential complexity (see Figure 6).

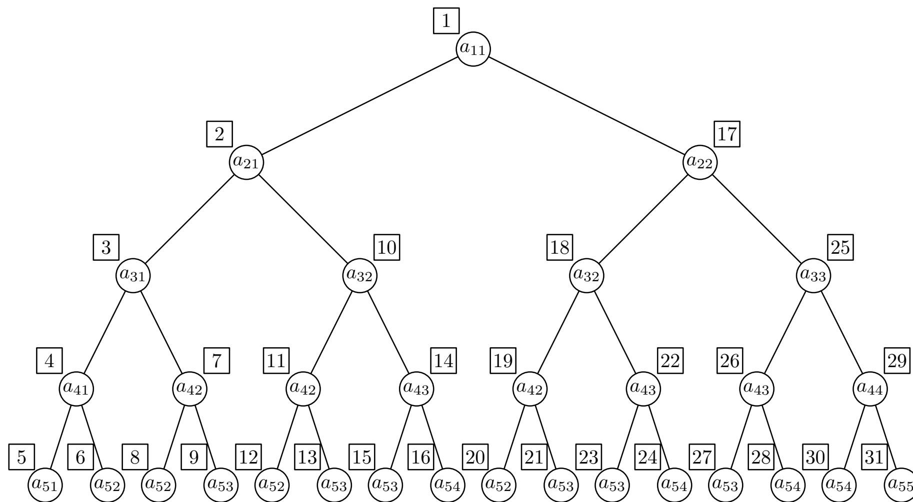



*Figure 6*

In certain cases it might be advisable to combine the backtracking and greedy techniques: before the backtracking algorithm should continue searching for the optimal solution by traversing the current subtree, with a greedy-like algorithm we assess the possibility that the respective subtree contains the optimal leaf.

The *divide and conquer* prunes every node, in the order they appear according to the postorder traverse. It leaves exactly one branch at each node, the one that represents the optimal solution of the respective subproblem. As it traverses the whole tree, obviously its algorithm will also have an exponential complexity (see Figure 7).

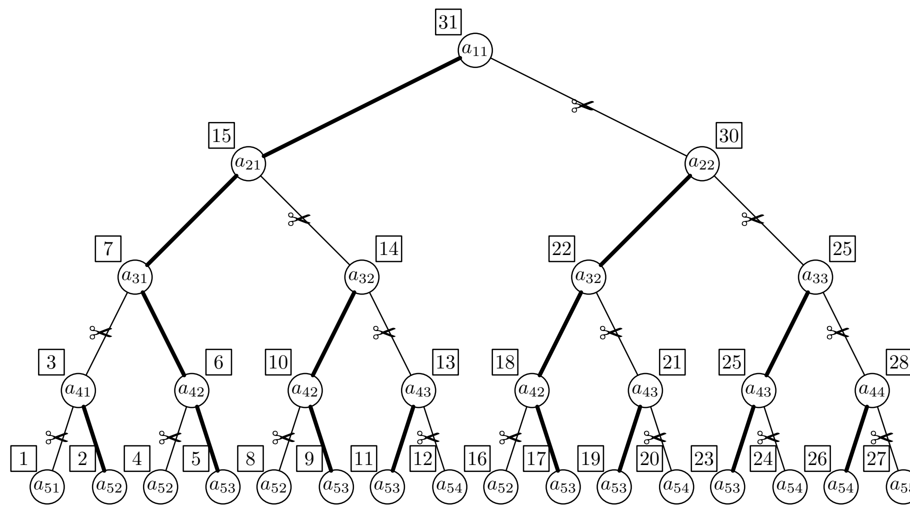



*Figure 7*

The *dynamic programming* first “lays one subtree over the other one” in case of identical ones, thus creating a “contracted-tree” (strictly it is no more a tree), which only contains the nodes representing the different subproblems (therefore the dynamic programming can really make use of its strengths when there are a lot of subproblems). Following this it prunes the dry branches from each node, the same as divide and conquer (see Figure 8).

Which order does the dynamic programming prune the nodes of the contracted tree? In the classical version it goes through the tree from bottom to top, from level to level. The recurrent realization deals with the nodes in the same order as divide and conquer, but it leaves out the repeated ones. Regarding that

the number of the nodes of the contracted tree is $\mathtt{n}(\mathtt{n}+1)/2$, the complexity of its algorithm is $O(\mathtt{n}^2)$.

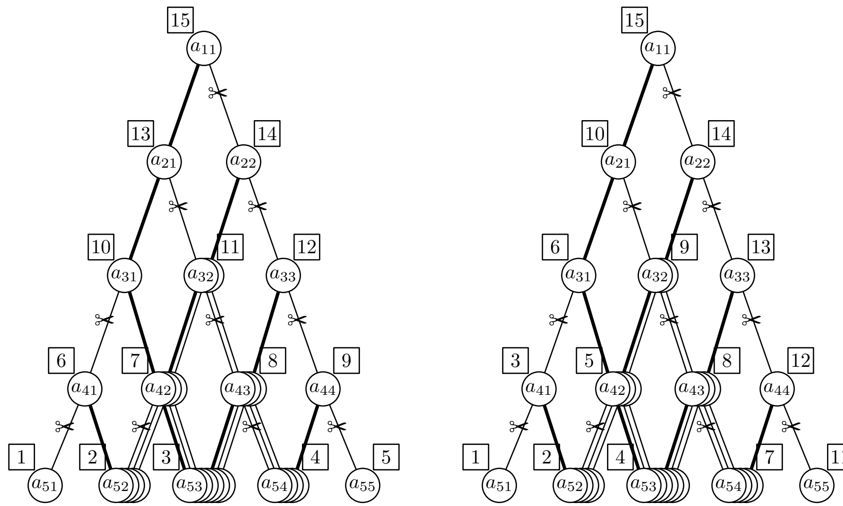



*Figure 8*

(2) *How do the four techniques "build up" the solution of the problem?*

The *greedy* and *backtracking* techniques built up the solution from top to bottom, announcing the result in leaves.

The *divide and conquer* and the *dynamic programming* built up the optimal solution exactly in an opposite way, from bottom to top. Both techniques announce the result in root, the classical version of the dynamic programming arriving to it from below, divide and conquer getting back to it. The difference between divide and conquer and the recurrent version of the dynamic programming consists only in the fact that the first one solves the subproblems independently from each other, there-fore not noticing if they are repeated, as long as the second one stores the solution each time it solves a subproblem and hence if it meets again the same subproblem its solution is already available.

The *backtracking* and *divide and conquer* have the common characteristic that both of them traverse the tree in its depth. But as they built the solution in opposite directions, one of them deals with the nodes in preorder, the other one in postorder sequence. This explains why backtracking generates several potential solutions (it has to select the optimal solution out of them), as long as divide and

conquer builds only one, the optimal one: the tree grows in size downwards but becomes narrower upwards, it has many leaves but only one root.

What is the other basic difference between *greedy* and *dynamic programming* techniques? The first one tries to build up the optimal solution through optimal decisions, the second from through optimal subsolutions. Therefore the greedy approach is satisfying only when we can prove that the optimal decisions really lead to the optimal solution for the given problem (the global optimum presumes local optimums). The condition for using the dynamic programming is the possibility to build up the optimal solution of the problem from the optimal solutions of the subproblems (the basic theorem of the optimality to be valid).

## The assessment of teaching the "upperview-method"

In the following we are going to relate about an experiment where we assessed how the presented method and the suggested syllabus contributed to the more efficient teaching of the algorithm design strategies. We conducted the experiment at Bolyai Farkas Highschool from Targu Mures (Romania), in the school year 2003–2004. At this school there are at the moment for each grade (IX–XII) three classes of Computer sciences functioning in parallel. The official curriculum foresees the teaching of the techniques (backtracking, divide and conquer, greedy) in the Xth grade. We involved all three classes in the experiment: X.G., X.H. and X.I. We could consider the general training of the classes identical as they had not been formed in the IX. grade according to their marks from the entrance examination, but according to the foreign language they were studying. Classes H and I had the same teacher, but class G a different one. Both teachers involved in the experiment were different from the author of the present article, who developed this method. We chose classes G and I as experimental classes and class H for the control group. The teachers in the experimental classes were teaching using the "upperview"-method, according to the suggested syllabus, and using the classical method in the control class (we treat the techniques as separate units).

As the goal of the method is that the students can see the basic, principal similarities and differences between the strategies, we reckoned that they must leave a durable trace in the students. There-fore, and in order to be able to measure it, we did not carry out the experimental measurement at the end of the school year, but we postponed it for almost a year, for the second term of the school year 2004–2005. Prior to the test written by the students, we had in all three classes a single class of an hour, where we refreshed their knowledge.

The material of the test had been chosen in such a way to offer the possibility to assess with it how clearly the students can see and how well they can apply it in practice. 26 students from the control group and 18 students in the first experiment group, 26 in the second participated at the test. In the following we present the test the students had to solve in 50 minutes:

(1) Which zeros of the enclosed `a[1..7,1..7]` array and in what order does the recurrent backtracking procedure from below changes into twos at the command fill $(3, 3)$? (3 points)

```
procedure fill(i,j:integer);
  begin
    a[i,j]:=2;
    if a[i-1,j]=0 then fill(i-1,j);
    if a[i,j+1]=0 then fill(i,j+1);
    if a[i+1,j]=0 then fill(i+1,j);
    if a[i,j-1]=0 then fill(i,j-1);
  end;
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(2) A man would like to get out from the enclosed maze (1–wall, 0–free) from the square with the coordinates $(4, 3)$ on the shortest route. The divide and conquer technique traces the determination of the shortest path starting from the man back to the definition of the neighbouring (up, to the right, down and to the left) shortest paths starting from free positions. Which order will the subproblems belonging to certain positions be solved? What would be a greedy approach of the problem and which path would that one find the shortest? (4 points)

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(3) What techniques would you use to solve the following problem?

(a) A shelf with the width `H` is given, as well as `n` books with the thickness `v1`, `v2`, ..., `vn`. Exactly how many ways can the shelf be filled with these books, if we can tear out pages from only one book? Which are these arrangement possibilities? (0.5 point)

(b) A shelf with the width `H` is given, as well as `n` books with the thickness `v1`, `v2`, ..., `vn`. Which is that exact way of filling the shelf with these books that uses the most books? We can tear out pages from only one book. (0.5 point)

(4) A square and a circle are given. Let's determine the area of their intersection surface with 3 decimals accuracy. Which of the techniques would approach this problem in the following way: I cut the square in four and I determine the area where each small square intersects the circle and I add them. I proceed in a similar way with each small square, until I get squares small enough. (1 point)

(5) Let's cut a bar with the length of `n` meters into smaller 1 and 2 meter bars in all the possible ways. What solutions does the following procedure offer and what order does it write them? Which of the techniques would you include this algorithm in? (The procedure `printout(A,k)` writes the first `k` element of array `A`. We call the procedure `bar(4,1)`). (3 points)

```
procedure bar(n,k:integer);
  begin
    if n=0 then printout(A,k-1);
    else
      begin
        if n>=1 then
          begin
            A[k]:=1;
            bar(n-1,k+1);
          end;
```

```
          if n>=2 then
            begin
              A[k]:=2;
              bar(n-2,k+1);
            end;
        end;
    end;
```

(6) In case of an optimization problem which techniques "would think" according to the following principles? Give a reason for it! ($4 \times 0.25$ points)

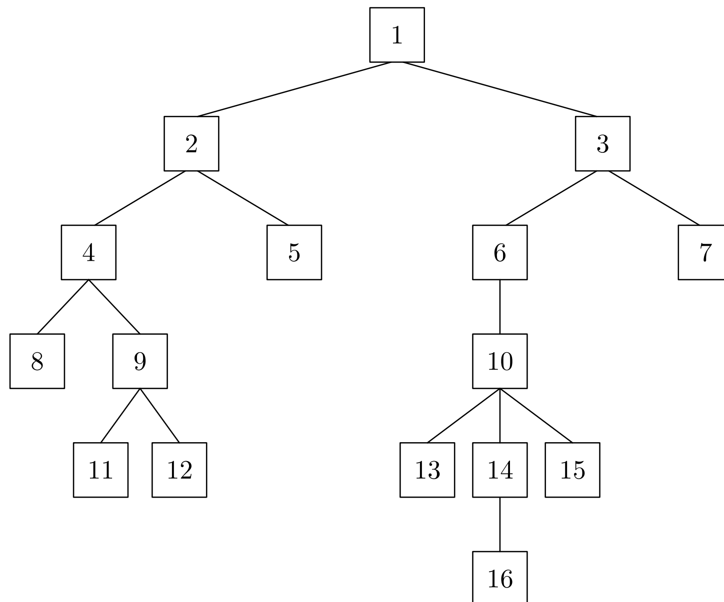
(a) What is sure is sure!

(b) Live the day!

(c) Don't postpone for tomorrow what you can do today!

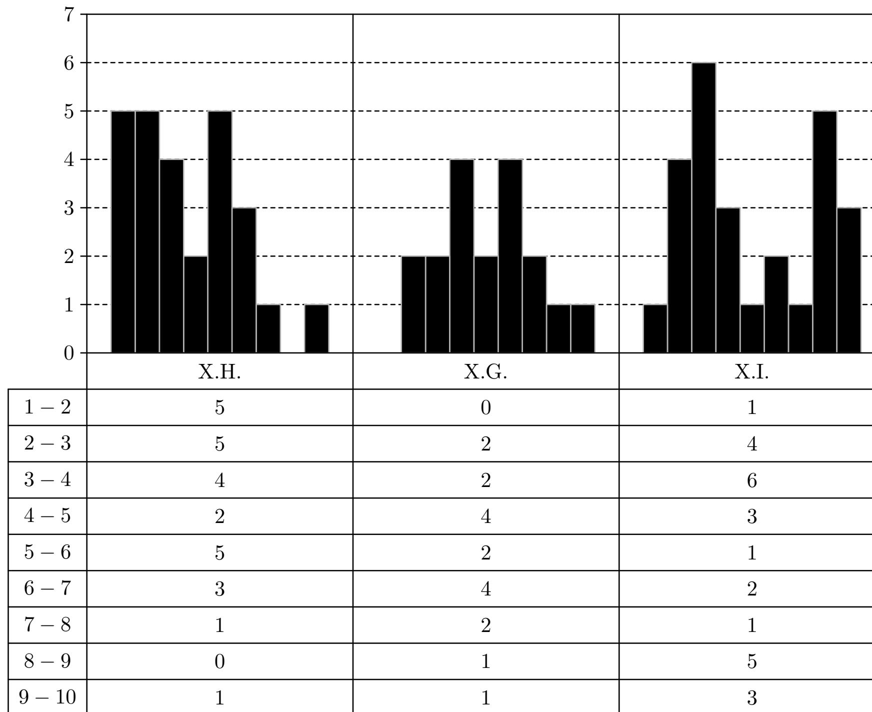
(d) The right person to the right place!

(7) Let's find the longest root-leaf path in the enclosed tree. How do the three techniques approach this problem? (2 points).




We converted the obtained number of points into marks, according to the 1–10 scale used in Romania. The following charts present the results obtained after this experiment. The horizontal axe represents the interval of marks (in

consonance with the row-indexes of the table) and the vertical axe the number of students from the respective interval of marks.




| | X.H. | X.G. | X.I. |
|---|---|---|---|
| 1 – 2 | 5 | 0 | 1 |
| 2 – 3 | 5 | 2 | 4 |
| 3 – 4 | 4 | 2 | 6 |
| 4 – 5 | 2 | 4 | 3 |
| 5 – 6 | 5 | 2 | 1 |
| 6 – 7 | 3 | 4 | 2 |
| 7 – 8 | 1 | 2 | 1 |
| 8 – 9 | 0 | 1 | 5 |
| 9 – 10 | 1 | 1 | 3 |

*Figure 9*

The average of the three classes:

- Control group (class H): 4.05
- First experimental group (class G): 5.48
- Second experimental group (class I): 5.52

As we can notice, the average marks in both experimental classes (although the methods were applied by different teachers), are approximately 1.5 marks higher (on the 1–10 scale), which represents a significant difference.

*Observation* [4]: Considering the size of the samples we have used the two-sample $t$-test. We checked the conditions of the applicability of this test:

(1) The samples obviously come from a population which can be considered of normal dispersion.

(2) We checked with an $F$-test if the unknown population dispersions can be regarded as equal (for example in case of comparing classes H and I the population dispersion can be considered equal with a level of significance of 0.9999).

Applying the two-sample $t$-test for classes H and I we found the difference of their average – to the advantage of the experimental class – significant with a probability of 0.9877.

## Conclusions

The above described experiment empirically proves the efficiency of the "upperview-method" in teaching algorithm design strategies. The didactic value of the method is already obvious from the way we defined it:

(1) We can see the entities being analyzed "next to each other".

(2) Only those elements can be seen which are essential for the analysis.

(3) The similarities and differences are obvious, the connections are striking.

We are convinced that if the teacher can develop such a point of view in his students, it will considerably contribute to the improvement of the problem solving skills of the whole class in this field of computer sciences. Even more, it can improve the general problem solving skills of the students.

Of course teaching algorithm design strategies according to this method can be efficient not only in high school. At the moment an experiment is being conducted regarding the effectiveness of the "upperview-method" in education at universities. The flexibility of the method makes it possible to include further techniques which are part of the university curriculum into the frame (for example branch and bound strategy).

We are planning further experiment for an additional assortment of the method. The students get four problems, each of them solvable with the discussed techniques (not necessary optimal). In the case of the first problem we give students a clue to algorithms to which the application of certain strategies lead and let them to choose which one to implement. In the case of the second problem we give students a clue to algorithms too, but they are told which one to implement. In the last two step of experiment they are not helped in any way. In the case of the third problem they have to decide on their own which techniques to use. Finally, in the case of the fourth problem they are given exactly what technique to use.

This experiment hopefully will reveal more about the impact of "upperview-method" on the students. The results of the experiment will be presented in a future paper.

## References

[1] T. H. Cormen, C. E. Leirserson and R. L. Rivest, *Introduction to Algorithms*, Massachusetts Institute of Technology, 1990, 266–270, 287–289.

[2] I. Odagescu, C. Copos, D. Luca, F. Furtuna, I. Smeureanu, *Programming Methods and Techniques*, Intact Press, Bucuresti, 1994, 95–108 (in Romanian).

[3] R. Andonie, I. Garbacea, *Fundamental Algorithms a C++ Perspective*, Libris Press, Cluj-Napoca, 1995, 185–187, 219–221 (in Romanian).

[4] Korpás Attiláné, Sándorné Kriszt Éva, Varga Edit, Veitzné Kenyeres Erika, *General Statistics*, National Press for Coursebooks, Budapest, 1997, 97–104 (in Hungarian).

ZOLTÁN KÁTAI
SAPIENTIA
ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELY
ROMANIA

*E-mail:* katai_zoltan@ms.sapientia.ro

*(Received May, 2005)*