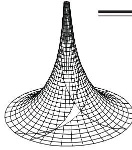

6/1 (2008), 139–152
tmcs@inf.unideb.hu
http://tmcs.math.klte.hu
Teaching Mathematics and Computer Science

"Frontier algorithms"

Zoltán Kátai

Abstract. In this paper we present a new method to compare algorithm design strategies. As in case of frontier towns the cultures blend, the so called "frontier algorithms" are a mixture of different programming techniques like greedy, backtracking, divide and conquer, dynamic programming. In case of some of them the frontier character is hidden, so it has to be discovered. There are algorithms that combine different techniques purposively. Furthermore, determining the programming technique the algorithm is using can be a matter of point of view. The frontier algorithms represent special opportunities to highlight particular characteristics of the algorithm design strategies. According to our experience the frontier algorithms fit best to the revision classes.

Key words and phrases: teaching methods, algorithm design strategies, programming techniques, case study.

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

Introduction

"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" [1]. How can we apply this principle stated by Comenius in case of algorithm design strategies (programming techniques) like greedy, backtracking, divide and conquer, dynamic programming?

The most of the teachers make this comparison analysis by a parallel review of the main characteristics of the studied techniques [2, 3]. A further, more concrete method is to solve the same problems with different techniques [4]. Kátai in his paper entitled "Upperview" algorithm design in teaching computer science in high

Copyright © 2008 by University of Debrecen

schools [5] presents a new way of comparing programming techniques. The goal of the "Upperview" method 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.

The "Upperview" method makes possible a uniform discussion of all the above mentioned techniques. The students can see each technique in the same time next to each other. This way it becomes possible to integrate all 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.

Katai notices that, in order to carry out an "upperview", a so called "abstract platform" might be necessary, where the entities (the techniques) 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. After a brief presentation of the core of the four strategies Katai concludes that all the techniques they are going to present are especially applied in the case of problems that have a hierarchic construction.

For example all techniques can deal with certain optimizing problems. Usually these problems consist of a target function which has to be optimized through a sequence of (optimal) decisions [5, 6]. So, for each optimizing problem a decision tree (tree structure) can be ordered. The root represents the starting state of the problem, the first level nodes represent the states the problem can reach after the first decision, the second level nodes those reached after the second decision etc. A node will have as many sons as the number of possible choices for the respective decision. Figure 1 presents a situation when the solution is obtained after four

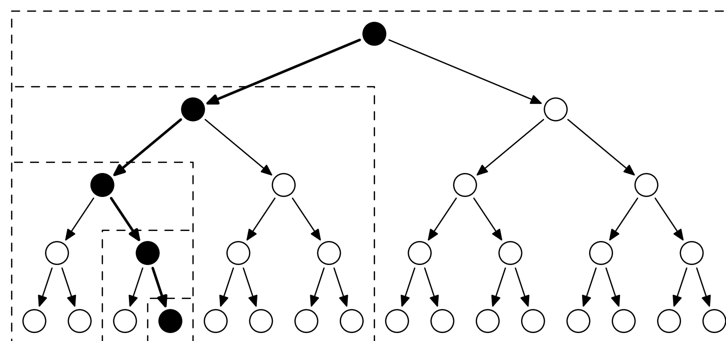

Figure 1. Decision tree

decisions. At each decision there is a choice of two possibilities. The labels of the nodes identify the corresponding states. By each decision the problem is reduced to a similar problem of smaller size, represented by one of the subtrees of the current node. In this situation the optimal solution will be represented by one of the root-leaf paths on the decision tree. The dotted rectangles show the way in which—in case of the bold arrows road is the optimal—the problem is reduced to its smaller and smaller subproblems.

In conclusion we can say that from the point of view of the techniques each one considers the problem as a tree structure. According to Katai this common tree structure is that abstract platform—necessary for the "upperview"—where the techniques can be laid next to each other

Katai establishes two criteria of comparison:

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

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

In this paper we are going to describe a fourth method to compare algorithm design strategies. As in case of frontier towns the cultures blend, the so called "frontier algorithms" are a mixture of different programming techniques. In case of some of them the frontier character is hidden, so it has to be discovered. There are algorithms that combine different techniques purposively. Furthermore, determining the programming technique the algorithm is using can be a matter of point of view. As a marriage sets off the masculinity respective the femininity of the spouses, the frontier algorithms represents special opportunities to highlight particular characteristics of the algorithm design strategies. According to our experience the frontier algorithms fit best the revision classes. Discovering hidden things, observing how matters complete each other and discussing different points of views are interesting tasks for the students. In the following we present four frontier algorithms.

The binary search algorithm

The binary search algorithm solves, for example, the following problem: Develop a function for locating *efficiently* an element of a particular value within a sequence of sorted elements. [7] Assume that the ascending sequence of numbers is stored in the array `a[1..n]`, and the number we are looking for is x.

The basic idea behind the binary search algorithm is to compare x with the middle element of the array `a` (`a[n/2]`). If the two elements are equal, the binary search function returns the index of the middle element. Otherwise, if the value we are searching for is less or greater than the middle element the searching process is continued (in the same way) either in the subarray `a[1..n/2-1]` or subarray `a[n/2+1..n]`, respectively. The function returns 0, if the current array section has become empty.

Since at each step the current interval is divided into two halves, this algorithm is traditionally considered a divide and conquer technique and consequently implemented as a recursive function. Despite of this fact we will show that this algorithm contains several greedy elements, and misses key elements of the divide and conquer technique.

The greedy technique is usually used in case of optimizing problems. Can the binary search algorithm be seen as optimization? Yes it can on account of the term "efficient" in the description of the problem. Contrary to the linear searching that needs n comparisons in the worst case, the binary search algorithm solves the problem after only $\log(n)$ decisions even in case the number we are searching do not exist in the array. The linear algorithm can also be considered as the one that divides the current interval (`a[i..n]`) into two parts: `a[i..(i-1)]` (an empty interval) and `a[(i+1)..n]`. In the worst case, at each step, the algorithms have to continue the searching process, and they have to do this in the greater part of the divided interval. If the length of the current interval is m, then the length of this "greater part" (after the comparing operation) varies between $((m-1)/2)$ (the optimal value) and $(m-1)$. Since the binary search algorithm at each step eliminates the maximum possible search space (exploiting the fact that the searching process takes place in a sorted array) we can state that it takes optimal decisions regarding the time complexity of the algorithm.

A binary tree can be attached to the problem. The whole array is ordered to the root node and the intervals (gained through the dividing process) to the other nodes. The leaf nodes represent the empty intervals, excepting the one that is attached to the interval having as middle element the number we are looking for (in case the searched value exists in the array).

A divide and conquer algorithm works by *recursively breaking down a problem into two or more* sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. In case of the tree attached to the problem the dividing process takes place in preorder and the building one in postorder.

Consequently, the divide and conquer algorithms “announce the result” of the problem solving process *getting back to the root*. A greedy algorithm builds the globally optimal solution through a sequence of locally optimal decisions. At each step the current problem *is reduced to one* similar subproblem. In other words it traverses only one root-leaf path of the decision tree, and provides the solution *in the “optimal leaf”* of the decision tree. Therefore the greedy strategies are usually implemented as *iterative* algorithms.

Analyzing the binary search algorithm from this point of view the following statements can be concluded:

- Although at each step the current interval is divided into two sub-intervals, the searching process is continued only in one of them. Because of this fact all steps can be seen as “greedy decisions” that *reduce the current problem to one subproblem* (to find the searched element in half of the current interval).
- The binary search algorithm can provide the solution of the problem in the “optimal leaf” of the decision tree, which is again typical of greedy algorithms. This fact explains why the iterative implementation of this so called divide and conquer algorithm is so obvious.
- According to the statements above in case of the binary search algorithm the solution building momentums of the divide and conquer strategy (at the postorder visits of the subproblems) are missing.

The Dijkstra algorithm

Dijkstra’s algorithm, named after its discoverer, is a greedy algorithm that solves the single-source (s) shortest path problem for a directed graph with non negative edge weights. [7] Despite of this fact we will show that beside its greedy characteristics this algorithm contains some dynamic programming elements, too.

One of the main differences between these programming techniques is that the greedy algorithms are top-down strategies whereas dynamic programming applies a bottom-up approach. The greedy approach with each decision reduces the problem to one of its subproblems. The dynamic programming algorithms start from the optimal solutions of the trivial subproblems and build the optimal solutions of the more and more complex subproblems and finally of the original problem.

The Dijkstra algorithm works by keeping, for each vertex v, the cost $d[v]$ of the shortest path found so far between s and v. Initially, this value is 0 for the

source vertex s ($d[s] = 0$), and infinity for all other vertices, representing the fact that we do not know any path leading to those vertices. When the algorithm finishes, $d[v]$ will be the cost of the shortest path from s to v—or infinity if no such path exists.

The algorithm maintains two sets of vertices S and Q. Set S contains all vertices for which it is known that the value $d[v]$ is already the cost of the shortest path and set Q contains all the other vertices. Set S is initially empty, and in each step one vertex is moved from Q to S. This vertex is chosen as the one with the lowest value of $d[u]$. This is the greedy part of the algorithm. When a vertex u is moved to S, the algorithm relaxes every outgoing edge (u, v). That is, for each neighbour v of u, the algorithm checks to see if it can reduce the weight of the shortest known path to v by first following the shortest path from the source to u, and then traversing the edge (u, v). If this new path is better, the algorithm updates $d[v]$ with the new smaller value.

On the one hand, from the point of view of set Q, this algorithm is a veritable greedy strategy. At each step, after a greedy choice, set Q (representing the remained subproblem that has to be solved) is reduced to a smaller one. On the other hand, from the point of view of set S, at each step the next shortest path is determined and a new vertex is added to this set. According to the principle of optimality the cost of the current shortest path is calculated on the score of the costs of the already determined shortest paths. This approach is characteristic of dynamic programming strategies. This duality can be explained by the fact that the principle of optimality is true for this problem in two forms: if the optimal (greedy) succession of the vertices is $(u_1, u_2, \ldots, u_i, \ldots, u_n)$, then both $(u_1, u_2, \ldots, u_i)$ and $(u_i, u_{i+1}, \ldots, u_n)$ are optimal sub-successions ($i = \overline{1, n}$). The first form is exploited by the dynamic programming elements, and the second one by the greedy part of the strategy. In other words, the greedy technique establishes the order the dynamic programming has to calculate the shortest paths, and the dynamic programming readies\approves of the greedy choices.

The knapsack problem

The knapsack problem is a problem in the field of combinatorial optimization. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a cost and a value, determine the number of each item to include in a

collection so that the total cost is less than a given limit and the total value is as large as possible. [7, 8]

In the fractional version of a problem the items can be broken into smaller pieces, so that the hiker may decide to carry only a fraction x_i of object i, where $0 \leq x_i \leq 1$. This problem can be solved by a greedy algorithm in polynomial time. Then again the greedy approach cannot warrant the optimal solution for the 0/1 version of the problem ($x_i \in \{0, 1\}$). An option would be to use backtracking strategy that generates all possible collections and selects the optimal one of them. Unfortunately this algorithm has exponential time complexity. Although the best solution for this problem is a pseudo-polynomial dynamic programming algorithm, an interesting frontier algorithm can be developed by improving the backtracking approach with greedy elements.

Since in connection with each item there are two possibilities (to put it into the bag or not), an n-level binary tree can be attached to the 0/1 knapsack problem. The optimal solution is represented by the optimal root-leaf path. The backtracking technique in its primitive form traverses the entire tree in its depth. In a first phase this algorithm can be improved by generating only collections that fit into the bag. A second optimization would be to keep a record of the current cost (the sum of the costs of the items that have already been put into the bag). If the sum of the costs of the remained items is less than or equal with the remained free space in the bag then all remained items are packed into the bag without traversing the current sub-tree. How can this strategy be enhanced more by using greedy elements? Let the items be considered in greedy order, and for each item let the "put it into" variant be analyzed first (see Figure 2). This strategy provides the potential solutions in a specific order (first the greedy solution of the 0/1 knapsack problem). The key idea of this third optimization is to investigate first if there are any chances of finding the optimal leaf in the current right sub-tree. In order to do this the algorithm switches provisionally to the fractional knapsack problem, and the searching process is continued by a "fractional greedy" procedure. If this "overestimated" optimum value (the "0/1 optimum" is less or equal then the "fractional one") is not better than the best solution the backtracking algorithm has found up to the current node, then the current sub-tree does certainly not contain the optimal leaf and its traverse can be abandoned.

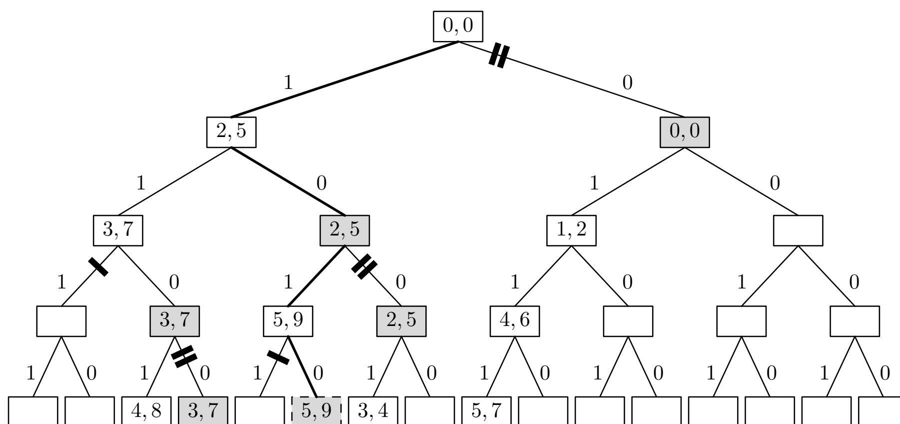

Figure 2. The binary decision tree attached to 0/1 knapsack problem in case of 4 items, the costs of them are $\{2, 1, 3, 1\}$, the values of them are $\{5, 2, 4, 1\}$ and the total cost capacity of the bag is 5. The bolded root-leaf path represents the optimal solution. The visited nodes contain the current cost and value. The "fractional greedy" procedure is called in the grey right-son nodes. The first and third optimizations use single and double line "shears", respectively.

Mouse in a Maze

Problem Specification: Design a program to calculate the length of the optimum path in a maze from the mouse to the cheese. The maze is represented by the binary array `a[1..n][1..m]` (1—wall, 0—free space), and the positions of the mouse and cheese are (`xm`, `ym`) and (`xc`, `yc`), respectively. The mouse can move in left, right, up and down directions. [6]

The teacher presents the following "cousin algorithms" and the students have to determine (and justify) the applied programming techniques: backtracking or divide and conquer (it is worthy to note that the most efficient algorithm for this problem is also provided by the dynamic programming technique). Parameters `x` and `y` localize the current position of the mouse. Parameter `k` indicates the number of steps that has been made by the mouse on the current path from its initial position to its current position. Parameter `kmin` stores the length of the optimal path. The "bold parameters" are transmitted by address and the others by value. `h1`, `h2`, `h3`, `h4` are local variables.

	1	2	3	4	5	6
1	**m**	0	0	0	0	0
2	0	1	1	1	0	0
3	0	1	0	0	0	1
4	0	0	0	1	0	**c**

Figure 3. A maze ($n = 4$, $m = 6$). The starting position of the mouse: $(1, 1)$. The position of the cheese: $(4, 6)$.

```
Procedure P1(a[][],kmin,xc,yc,x,y,k)
  if x == xc and y == yc then
    if k < kmin then
       kmin = k
    endif
  else
    a[x][y] = 1
    if a[x-1][y] == 0 then
      P1(a,kmin,xc,yc,x-1,y,k+1)
    endif
    if a[x][y+1] == 0 then
      P1(a,kmin,xc,yc,x,y+1,k+1)
    endif
    if a[x+1][y] == 0 then
      P1(a,kmin,xc,yc,x+1,y,k+1)
    endif
    if a[x][y-1] == 0 then
      P1(a,kmin,xc,yc,x,y-1,k+1)
    endif
    a[x][y] = 0
  endif
end P1
```

```
Procedure P2(a[][],xc,yc,x,y)
  if x == xc and y == yc then
    return 0
  endif
  h1 = n*m
  h2 = n*m
  h3 = n*m
  h4 = n*m
  a[x][y] = 1
  if a[x-1][y] == 0 then
    h1 = P2(a,xc,yc,x-1,y)
  endif
  if a[x][y+1] == 0 then
    h2 = P2(a,xc,yc,x,y+1)
  endif
  if a[x+1][y] == 0 then
    h3 = P2(a,xc,yc,x+1,y)
  endif
  if a[x][y-1] == 0 then
    h4 = P2(a,xc,yc,x,y-1)
  endif
  a[x][y] = 0
  return minim(h1,h2,h3,h4) +1
end P2
```

Figure 4 shows the decision tree that can be attached to the problem. The grey nodes represent the dead ends and the bold ones the solution leaves (the broken line rectangle indicates the optimal leaf). The optimal path of the mouse is marked by a bold line. The identical subtrees are circled.

The similarities between the procedures presented above can be ascribed to the fact that both techniques (backtracking and divide and conquer) traverse the tree behind the problem in its depth. The differences are due to the opposite directions in which the two techniques build up the solution (backtracking—from root

Figure 4. The decision tree attached to the “Mouse in a maze” problem.

to leaves, divide and conquer—from leaves to root), and the different order they deal with the nodes (backtracking—preorder, divide and conquer—postorder).

These strategic differences explain why backtracking selects the optimal solution out of several potential ones, as long as divide and conquer builds only one solution, the optimal one (a tree has many leaves but only one root).

An unusual "summit"

How can teachers of computer sciences make use of the above presented material during revision classes? In order to make these classes funnier they should organize them as summits. At a previous class the teacher divides the students into four groups, each group representing a "country". The countries are Backtracking, Greedy, Divide and conquer and Dynamic programming. All groups get the agenda of the summit. The agenda contains, for example, the following items:

(1) Petition sent up by the Greedy delegate wherein they contest the exclusive divide and conquer statute of the binary search algorithm.
(2) Petition sent up by the Dynamic programming delegate wherein they claim recognition of their contribution to the Dijkstra algorithm.
(3) The Backtracking and Greedy delegates present a common project for the "Knapsack problem". The Dynamic programming delegate criticizes their proposal.
(4) The Backtracking and Divide and conquer delegates present the results of a common research about two cousin frontier algorithms (`P1` and `P2`).

The summit is chaired by the teacher. In case of the first point the petitioners start the discussion by presenting their arguments. The Greedy delegate identifies the greedy elements in the binary search algorithm emphasizing their importance. Parallel with this, they have to try to diminish the significance of the divide and conquer attributes of the algorithm. After that the Divide and conquer side has the opportunity to answer them.

At the second point, the Dynamic programming delegate highlights the importance of the principle of optimality in the solution building process of the Dijkstra algorithm. The Greedy side has to defend their primary role in this algorithm.

In connection with the "Knapsack problem" the sides have to stress how their collaboration enhanced their previous individual attempts to solve this problem. They also have to explain how the similarities and differences between these techniques made such an efficient cooperation possible. The Dynamic programming delegate reasons for the superiority of their solution in the case of this problem.

The two delegates have to localize the elements that endow procedures `P1` and `P2` with backtracking and divide and conquer character, respectively. They also have to identify and explain the similarities and differences between these procedures.

Didactical aspects of the presented revision method

The above presented revision method is on intimate terms with the case study. The case approach to teaching is something common in social sciences and in the teaching of business studies but it is rarely used in computer sciences. [7] The frontier algorithms can be seen as "cases" in the comparative analysis of the programming techniques. Our "summit" provides an interesting real life organizational context to develop students' problem solving and analytical skills in ways that traditional repetitive exercises cannot. Students need to learn how to apply technical skills and knowledge in case of real life problems. These "real life algorithms" are rarely applications of just one certain technique. They often include elements of different strategies. To develop such algorithms efficiently, a good command of the "philosophies and politics" behind the algorithm design strategies is presumed. "Cases" also teach students that there is rarely one correct answer but merely several possible alternatives with different consequences, and the decision making process often requires significant value judgments. [9]

The participative approach to learning, central to the case study method of teaching, is also an important tool to improve students' communication and analytical skills. "... by being forced to actively participate in a discussion a student better internalizes his or her own ideas while preparing to communicate with others. Also such communication assists other students to develop their own understanding of the problem, alternatives and solution ..." [10]

The success of the case teaching method presumes adequate preparation (individual and in groups) on the part of students for the following reasons:

- the frontier algorithms subject fits best revision classes,
- the students have to receive the agenda of the summit in good time,
- the teacher should be available for students to help them in their preparation for the summit.

The Harvard Business School [11] identifies three key rules for teachers: careful preparation, good control of discussion, concern for the students. Most educators would probably see these as essential ingredients of any good teaching,

but case study proponents would stress that they are even more critical for the effective use of case studies.

One of the major characteristics of the presented revision method is that students have to work in groups. Students may have different learning styles. Teachers have to take account of this diversity. Students may be introverts or extroverts and sensing or intuitive.

Introverts are concentrators and reflective thinkers. For the introvert, there is no impression without reflection. Extroverts prefer interaction with others, and are action oriented. For the extravert, there is no impression without expression. Extroverted students learn by explaining to others. They enjoy working in groups. Introverted students want to develop frameworks that integrate or connect the subject matter. To an introvert knowledge means interconnecting material and seeing the "whole picture". Introverted students especially appreciate the frontier algorithms subject, extroverted students do it less (but they like to work in groups). According to cognitive psychologists the compare/contrast analyses and building concept maps contribute to the effectiveness of the teaching-learning process.

Sensing students are detail-oriented, want facts, and trust them. Intuitive students seek out patterns and relationships among the facts they have gathered. They trust hunches and their intuition and look for the "whole picture". Sensing students prefer organized, linear, and structured lectures. Intuitive students prefer the discovery learning. The discovery method appeals to intuitive students and teaches sensing students how to uncover general principles. The intuitive student can help the sensing student to discover the theory; the sensing student can help, identify and marshal the facts of the task. Since the "summit method" implies discovering, the group of delegates should consist of both sensing and intuitive students. [12]

Conclusions

The presented so called "summit method" is a special application of the case study method in teaching computer sciences. The "Upperview" method applies analysis hand by hand with synthesis. [5, 6] In [6] the chapters that present the techniques make use of analysis and the so called "upperview chapters" apply synthesis. In addition, the frontier algorithms subject completes the progressive synthesis of the "upperview" method by a thorough analysis of the algorithms on the agenda of the "summit". The effectiveness of the applied didactic methods and

the active and participative learning environment the summit creates warrant the efficiency of the presented revision method. Organizing the revision as a summit teachers bring computer sciences education (in this case) closer to Comenius' dream that study should be "entirely practical, entirely pleasurable, and such as to make school a real game, i.e., a pleasant prelude to our whole life".

References

[1] Comenius, *Orbis sensualium pictus*, 1653.

[2] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The Massachusetts Institute of Technology, 1990.

[3] R. Andone, I. Garbacea, *Fundamental Algorithms a C++ Perspective*, Libris Press, Cluj-Napoca, 1995 (in Romanian).

[4] `http://www.cis.upenn.edu/~matuszek/cit594-2004/Lectures/44-dynamic-programming.ppt`

[5] Z. Kátai, "Upperview" algorithm design in teaching computer science in high schools, *Teaching Mathematics and Computer Science* **3**, no. 2 (2005), 221–240.

[6] Kátai Z., *Algoritmusok felülnézetből*, Scientia, Cluj-Napoca, 2007 (in Hungarian).

[7] `http://ro.wikipedia.org/`

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

[9] `http://lsn.curtin.edu.au/tlf/tlf1995/sims.html`

[10] W. R. Knechel, Using the Case Method in Accounting Instruction, *Issues in Accounting Education* (Fall 1992), 205–217.

[11] Harvard Business School, *Hints for Case Teaching*, Harvard Business School Publishing division, Boston, 1984.

[12] `http://www2.gsu.edu/~dschjb/wwwmbti.html`

ZOLTÁN KÁTAI
SAPIENTIA UNIVERSITY
MATHEMATICS AND INFORMATICS DEPARTMENT
TÎRGU-MUREŞ
ROMANIA

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

(Received November, 2007)