LEARNING MATHEMATICS SUPPORTED BY COMPUTATIONAL THINKING

Maciej M. Sysło[1,2], Anna Beata Kwiatkowska[1]
[1]Faculty of Mathematics and Informatics, Nicolaus Copernicus University
Chopin str. 12/18, 87-100 Torun, Poland
[2]Faculty of Mathematics and Informatics, University of Wroclaw
F. Joliot-Curie str. 15, 50-383 Wroclaw, Poland
syslo@ii.uni.wroc.pl; aba@mat.umk.pl

The purpose of computing is insight, not numbers
[R.W. Hemming, 1959]

Abstract
We focus on applying computational thinking mental tools to some topics in school mathematics. Our goal is twofold. We suggest how to extend and enrich traditional topics in school mathematics by applying computational thinking to obtain solutions which are supported by the power of computer science as a discipline and computers as computing tools. Moreover, our approach to deal with topics in mathematics with computational thinking and computing tools contributes to constructionist learning, to learning by doing and making meaningful objects in the real world – here computer solutions. Mental tools used here include: data representations, reductive thinking, approximation of numerical and intractable problems, recursive and logarithmic thinking, heuristics.

Keywords: computational thinking, mathematics, approximation, reduction, recursion, heuristics

1. Introduction

This paper is a continuation of our works [15, 17]. In [15] we demonstrate how informatics (here, computer science) education can contribute to mathematics education and in [17] we describe how computational thinking approach is implemented in students' activities managed by a textbook for informatics in which the project based learning is used to organize content and students' learning. Our interest in mathematics is driven by a strong belief that other domains can benefit from computational thinking. Although, all the topics described in [15] are included in the high school informatics textbook [5], they are still absent in teaching mathematics and in mathematics textbooks.

In this paper we focus on applying computational thinking mental tools to some topics in school mathematics. Our goal is twofold. From one hand, we suggest how to extend and enrich traditional topics in school mathematics by applying computational thinking to obtain solutions which are supported by the power of computer science as a discipline and computers as computing tools. On the other hand, our approach to topics in mathematics with computational thinking and computing tools contributes to constructionist learning, to learning by doing and making meaningful objects in the real world [10] – here computer solutions. As a very important by-product of constructing meaningful objects from the domain of mathematics but placed in computing environment students also learn and develop mental tools of computational thinking which are suitable for dealing with mathematical problems, and challenges, as well as with related to other subjects and disciplines.

Maurer has observed in [9] that ‘algorithmic mathematics has two radically different meanings’: traditional and contemporary, where ‘traditional algorithmic mathematics refers to performing algorithms and contemporary refers to creating them and … to thinking in terms of algorithms for solving problems …’. Today, 30 years after appearing [9], algorithms are still traditionally used in teaching and learning mathematics. We do hope that computational thinking mental tools used in learning by doing approach illustrated in this paper on a number of typical examples may change the role of computing in mathematics education. The contemporary meaning of algorithmics has been used in the books [13, 14] in which students are learning algorithms by developing them.

2. Computational thinking

Jeannette Wing [20] suggested that computational thinking is a fundamental skill for all, to be able to live in today’s world. It is a mode of thought that goes well beyond computing and provides a framework for reasoning about problems and methods of their solution. Computational thinking has a long tradition within computer science as *algorithmic thinking*, which is a competence to formulate a problem solution as an algorithm for transforming input to output and then to implement the algorithm on a computer. Computational thinking as an extension of this term includes thinking with many levels of abstraction as a problem solving approach inherently connected to computer science and addressed to all students in K-12 and in tertiary education to use computers and computing skills in solving problems coming from various scientific and applied areas.

There are several descriptions what computational thinking is however no agreement has been reached how to precisely define this way of thought. Denning claims in [4] that computational thinking is not an adequate characterization of computer science and he is right –it is a set of key practices originated in computing but addressed to all areas of human activities far beyond computer science. It is clear that basic computer science knowledge helps to systematically, correctly, and efficiently process information, perform tasks, and solve problems. Lu and Fletcher discuss in [8] the role of programming in practicing computational thinking when studying computer science and formulate the thesis that ‘programming is to computer science what proof construction is to mathematics’. In general, programming is a means which helps to communicate with a computer by using a programming language. To obtain a computer solution of a problem, for instance by applying some mental tools of computational thinking, one has to use a program to communicate the solution to a computer. There are a number of ways to do it, see [16]. Recently, there have been several open initiatives to introduce all students to programming. Moreover, there is a strong movement to early introduce students to programming, see [2] – we plan to do it also in the near future in Poland.

Although coming from computer science, computational thinking is not the study of computer science, it is not thinking about using computers either, though they play an essential role in designing problems’ solutions. It is reasoning about the problems in computational terms – students are supposed to gain the ability to model problems in a way that helps to obtain computer solutions; they learn how to distinguish between the results, and the processes by which the results are obtained. Computational thinking is very useful as an insight into what can and cannot be computed. Students are encouraged to study the possibility of designing computational models of problems that in general might be viewed as nothing to do with computing. Computational thinking involves concepts, skills and competences that lie at the heart of computing, such as abstraction, decomposition, generalization, approximation, heuristics, algorithm design, efficiency and complexity issues.

In computational thinking, abstraction and decomposition are used when dealing with large and complex tasks, such as problems on big data. An appropriate representation of a problem is chosen to make it tractable and the problem is reformulated as one which can be solved for instance by transformation, reduction, or simulation. Computational thinking is using mental tools specific for computer science such as recursive or logarithmic thinking, see [18, 19]. The most powerful tool of computational thinking is heuristic reasoning which, in the absence of complete domain knowledge, is used to discover and create a solution supporting creative thinking. Complexity issues play a very important role in computational thinking. Time and space efficiency is used in the process of designing solutions and then to estimate quality of final results. Computational thinking goes beyond only using ICT tools and information toward creating tools and information. It reminds our distinction between informatics (as creation of programs, tools, etc.) and ICT (as applying informatics tools) [16]. The creation of tools (e.g. programs) and new information requires thinking processes about how to use abstraction and manipulate data and many other computer science and computing concepts and ideas, as we describe here in relation to creating solutions of mathematical problems

The following definition of *computational thinking* [16] is widely accepted: it is the thought process involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by combinations of humans and machines. In applying computational thinking to solve a problem we use a framework based on its *operational definition* developed by ISTE [6] and CSTA [3], which reminds the algorithmic solving strategy [16]. Computational thinking can be characterized as a problem-solving process consisting of the following steps:

- Formulate a problem in a way that enables to use a computer for its solving.
- Organize and analyze data.
- Represent data through abstractions.
- Design a (computer) solution through algorithmic thinking.
- Identify, analyze, and implement a solution with the goal of achieving the most efficient (in terms of time and space) solution.
- Generalize the problem solved and its solving strategy to a wide variety of other problems.

3. Constructionist learning of mathematics

In this Section we present a number of topics, which are a part of informatics education in schools and we show how they can contribute to mathematics education. All the topics described in this paper are included in the textbook for informatics [5] for high school in Poland. However, most of these problems are still absent in mathematics textbooks used in schools.

3.1. Representations of numbers

Computational thinking is choosing an appropriate representation for a problem and its data [20]. Regarding numbers, there are two simple types of representations, exact (integer) and approximate (real). These types of number representations can be characterized by the properties of operations performed with the help of such representations and the results obtained. All results of operations on integer numbers are exact whereas operations on real numbers always produce approximate values. We deal with approximate calculations in Section 3.3 and here we discuss shortly the representation of integers in a computer memory. Students are usually familiar with the algorithm for finding the *binary representation* of a nonnegative integer *n*, which depends on divisions of *n* and the resulting quotients by 2 and taking the reminder as successive digits of the representation. However they find difficult to determine, how many bits does an integer *n* occupy in the computer memory. The answer to this question is important since it appears in solutions of many theoretical and algorithmic problems. In [18] we explain how the answer to this question is related to logarithm and conclude with the following algorithmic definition of logarithm:

logarithm $\log_2 n$ is equal to the number of steps in which, successive divisions of n and the resulting quotients by 2 lead to 1.

We demonstrate in [18] how important is this interpretation of logarithm and logarithmic thinking for designing and evaluating solutions to many problems, such as: searching in an ordered set, calculating the values of the exponential function, finding the greatest common divisor of two numbers, and applying a divide and conquer strategy.

The binary representation of integers, and, in general, the representation of integers in a positional system to any base is an opportunity to deal in school mathematics with *polynomials* of higher degrees, which in the school algebra are restricted to degree 1 or 2. The representation of an integer n in the system to base p is the polynomial in p with coefficients from $\{0, 1, \ldots, p-1\}$. With such representation of integers, operations can be designed and analyzed by means of algorithms on polynomials, such as Horner's rule, see [18, 19] where a fast algorithm for calculating the exponential function is derived using the polynomial interpretation of the binary representation of exponents.

3.2. Reduction and decomposition

Reduction is a problem-solving approach, yet another way of thinking – *reductive thinking*. Solving problems by reduction is a very important issue in science education. Reduction is one of the skills characterizing computational thinking [20]. Reductive thinking however is not an easy topic and a (mental) tool to cope with in high school, college, and university [1]. In algorithmics, reduction can be characterized as follows: identify connections between the problem to be solved (P) and another problem (S) which we know how to solve, translate an input to P to an input to S, and then translate an output to S to an output to P. In general, problem P can be decomposed into a number of problems S, which are used to obtain a solution of P. Moreover, as S we may choose the same problem P for a subset of input and obtain a recursive algorithm, see Section 3.4. In academic computer science the concept of reduction in known as a polynomial-time transformation, which is used in advanced courses on computability and complexity theory. Reduction strategy is strongly related to abstraction – first, identifying a relation between two problems P and S requires ignoring irrelevant characteristics of these problems and focusing on their similarities, and second, in the solution of S the details of the solution to P are ignored. Then, in constructing a reduction-based solution to P we only need to know that a correct solution to S exists and can be efficiently obtained. The other details of such a solution can be ignored – this means working on a higher level of abstraction.

In the rest of this section, using three problems, we illustrate how reductive thinking can be used in designing algorithms for these problems. We use the same problem S, described first.

Finding a smallest or a largest element in a set

Finding a smallest or a largest (or best) element in a set of elements of a given type is one of the most frequently used 'activity' in everyday life. Therefore, it is interesting to know how to perform this operation in the most effective way. Dealing with such problems we first use abstraction and think of elements as numbers[1], then we can use comparison as the basic operation. To find the largest number we usually go through all elements keeping at hand the largest element found so far. Therefore, if there are n elements, we begin with the first element at hand and perform $n - 1$ com-

[1] There are some exceptions to this assumption, for instance when the best player or a team is to be found in a tournament. In such a case, a match between two players or teams is a comparison.

parisons and sometimes exchanging the largest element found so far with a larger current element. This is a *linear search*. Some students argue, that if a tournament is played then the number of comparisons (matches) is smaller than that in a linear search, so we ask them to play tournaments with different numbers of players – they find that there is the same number of matches played in a tournament as in a linear search. At this point we try to convince our students that in fact $n – 1$ is the smallest possible number of comparisons needed to find the best item among n items. To this end we use a beautiful argument given by Hugo Steinhaus in his excellent book [12]: if John wins a tournament then each of the other players lost at least once (to John or to other players). Therefore at least $n – 1$ matches were played in the tournament (we assume that there are no ties). Thus, a linear and a tournament search for the best element are the *optimal algorithms* since they perform exactly $n – 1$ comparisons. As an extension, we ask our students, whether a looser in the final of a tournament is the second best player of the tournament. The answer to this question is not obvious.

Finding simultaneously a smallest and a largest element in a set

The problem in the title can be solved by, finding a smallest element, then – a largest element among the remaining elements, giving rise to the algorithm which performs $2n – 3$ comparisons on a set of n numbers. However this is not a method of simultaneous finding both numbers. To design such an algorithm, we first ask the following question: when finding both a smallest and a largest element, how we can interpret the inequality $x \le y$ satisfied by two numbers. It is clear that x is a candidate for a smallest element, and y – is a candidate for a largest number in the set. Therefore, we first split the set into two subsets of equal size (initially, we assume that the set is of even size) of candidates for a smallest element and candidates for a largest element. Thus the problem reduces to two subproblems: find a smallest element in the former set and find a largest element in the latter set. Hence, splitting needs $n/2$ comparisons, and two subproblems can be solved with $n/2 – 1$ comparisons each. The total complexity equals $3n/2 – 2$. When n is odd, the complexity equals $\lceil 3n/2 \rceil – 2$. This algorithm is also optimal, as the algorithms used to solve its subproblems.

Selection sort

Sorting appears in many real-world situations and may be treated from different points of view: practical applications, algorithmic techniques, complexity issues. Once our students know how to find a smallest element in a sequence, they quite easy come up with an idea how to reduce sorting of n numbers to iteration of a procedure for finding a smallest element among k non-sorted elements, for $k = n, n – 1, \dots, 2$. Since in each step, $k – 1$ comparisons are performed to find a smallest element, the total complexity of this sorting procedure equals exactly $n(n – 1)/2$. Two issues are in order regarding the complexity of this algorithm. First, students may verify using some educational software that this complexity figure is the same for any n numbers, regardless of the initial order of elements, even for sorted numbers. Second, there exist sorting algorithms, which are practically and theoretically faster. In conclusion, students should observe that reducing a problem to use an optimal algorithm for a subproblem not always results in an optimal algorithm for the original problem.

Triangles

The following problem in [15] illustrates a difference between two its solutions, one given in mathematics class and another developed during informatics lesson**.** Let set A contain a large amount of integers. Verify if each triple of numbers in A can be the lengths of sides of a triangle. It is obvious that each triple $a, b, c \in A$ has to satisfy the triangle condition.

A 'mathematical' solution. Test all possible triples of elements from A if they satisfy the triangle condition. If A contains n integers, then Cn^3 operations, where C is a constant, are performed in this case. When A contains a very large amount of numbers, this algorithm is very impractical.

A solution using reduction. We direct our students to obtain a practical solution. First, they should observe that if $a \leq b \leq c$, then a, b, c satisfy the triangle condition if and only if $a + b > c$. Hence, they conclude that if all triples in A have to satisfy the triangle condition, then we have to test this condition only for the two smallest integers and the largest integer in A. They find also that we do not need to sort all numbers in A to find these tree numbers – a single run of a linear search through A is sufficient. The time complexity of this solution is proportional to the number of elements in A.

This example shows that developing a computer solution to a mathematical problem may lead to an elegant and efficient solution, from computational point of view and as a mathematical solution.

3.3. Approximation

Approximation has many facets in both, in mathematics and in computer science. In mathematics, it is mainly a subject of theoretical considerations and in computer science – there are at least three main reasons to focus on approximation:

- approximate representation of real numbers has influence on accuracy of computations and in some cases may lead to propagation of errors;
- processors perform only 4 basic arithmetic operations and therefore calculation of any other operation (roots, trigonometric functions) must be performed as a sequence of these 4 operations;
- there are a number of optimization problems, which are intractable, for which only approximate solutions are practical since optimal solutions are beyond the reach of computers.

In this section we address some problems from the first two categories and in Section 3.5 we discuss heuristics for intractable problems, which generate their approximate solutions.

Rounding errors

Solving a quadratic equation $ax^2 + bx + c = 0$ is one of the traditional topics in school mathematics. Usually, the Δ-algorithm is applied, where $\Delta = b^2 - 4ac$, and when $\Delta > 0$, then: $x_1 = (-b - \sqrt{\Delta})/(2a)$ and $x_2 = (-b + \sqrt{\Delta})/(2a)$. However, when b and $\sqrt{\Delta}$ agree on a number of consecutive significant digits, then one of the roots loses significant digits. In this case, the Viete'a formula can be used: $x_1x_2 = c/a$. Assume, that b and $\sqrt{\Delta}$ are close to each other, then if b and $\sqrt{\Delta}$ are of opposite signs then calculate x_2 as above and $x_1 = c/(a\, x_2)$, otherwise – calculate x_1 as above and $x_2 = c/(a\, x_1)$. We haven't seen a textbook for school mathematics in which a quadratic equation is solved in such a way.

Calculating a square root

Almost all real-world problems deal with real numbers and their solutions are usually approximations of exact solutions. A method which finds an approximate solution usually consists of a sequence of steps which brings an approximate solution closer to the exact solution. In school mathematics, the calculation of a square root is quite often used do demonstrate such a method. Let $x = \sqrt{a}$, where $a > 0$. Then starting with an initial approximation x_0 of x, its successive approximations $x_1, x_2, \ldots$, are calculated: $x_i = (x_{i-1} + a/x_{i-1})/2$. We found that in the majority of mathematics textbooks this formula is presented as a 'black box', with no explanation of its origin. However, we can model calculation of $\sqrt{a}$ using a simple geometric argument. If $x = \sqrt{a}$, then $x^2 = a$. Therefore, finding x means finding the side length x of a square, whose area is a. If our guess of x is not correct and we want to keep the area size equal to a, then the length of the other side of the rectangle should be equal a/x. If x is too small then a/x is too big, and if x is too big then a/x is too small. Hence students easily conclude that the correct answer for x lies somewhere between these two numbers, and as a next approximation we may take the average of the lengths of these two sides, as in the formula. Calculations according to the iterative formula may be performed using a spreadsheet even by students in middle school (see [15]). They can make experiments with different initial approximations and observe that only a few iterations are needed to obtain a good approximation.

3.4. Recursion

Recursion is not a separate topic, it is a method and a tool, the way of thinking, used for problem solving by decomposition of a problem into subproblems. To solve a problem by this approach one has to specify: decomposition of a problem into subproblems and composition of the solutions of the subproblems into the solution of the problem to be solved. Recursion appears in implementations of a divide and conquer method. In some cases, recursion is used to reduce a problem to the same problem but with a reduced size, e.g. in a binary search. Recursion may be viewed as an alternative to iteration, but our main focus is on its properties as a concept which has a computational power in designing solutions. As a tool for problem solving, it is especially popular in designing computer solutions, since in a program which uses recursion, much of the work is done by the computer itself. Algorithms and programs which use recursion are usually shorter and more 'readable' than their iterative counterparts, although in general their computer execution takes longer.

Recursion remains a challenge – it is one of the most difficult topics in discrete mathematics to master for students. In [19] we present various facets of recursion together with tools which can help teachers to explain and students to understand how to think recursively. There are several methods for introducing and explaining recursion: induction, runtime stack, the trace, and the recursion tree, which is the most popular way to visualize the structure and the process of recursive calls. Knowing the difficulty in introducing, and using recursion, we differentiate our approach, tools, and methods. Recursion can be also introduced as a 'real-life topic' and then software for visualization of recursive computations can be very helpful to overcome some difficulties by novices.

There exists a close relation between: induction and recursion, although problem-solving using recursion is more popular in computer science than in mathematics. The authors of [7] claim: 'recursion is an executable version of induction'. To the question 'which of the two topics is simpler? which should come first in the learning sequence?' they answer: 'recursion is initially more accessible than induction … it is not recursion *per se* that is easier than induction. Rather, it is recursive activities with the computer that are easier than inductive proofs with pencil and paper.' Our experiences confirm these findings. Recursion and a recursive way of thinking and solving problems seem to be the most solid and concrete bridge between informatics and mathematics education. We are convinced that recursion introduced while developing computer solutions in informatics classes may contribute also to mathematics education, in particular to solving pure mathematical problems.

3.5. Heuristic thinking

A *heuristics* is an experience-based technique for solving problems when other methods are too slow or fail to find an exact solution. This is achieved by trading optimality and accuracy for speed. The goal of a heuristics is to produce a solution in a reasonable time that is good enough in terms of a given objective. Heuristic solutions in most cases only approximate the exact solutions but they are accepted since finding them do not require a long time. Heuristic algorithms are the only practical methods for solving complex optimization problems, which – on one hand – are theoretically intractable (NP-hard), and – on the other hand – are routinely solved in real-world situations. For such problems, the exhaustive search is impractical and the heuristic methods are used to speed up the process of finding a solution which is accepted as satisfactory.

The most natural heuristics is *trial and error*, which can be applied to solve almost any problem in science. In mathematics and in computer science very popular is a heuristics called a *greedy algorithm*, which locally makes the best choice at each step with the hope that this strategy will lead to a global best solution. For many problems, a greedy approach does not in general produce an optimal solution, but may yield solutions that approximate a global optimal solution in a reasonable time. In some cases, quality of greedy solutions (i.e. distance form an optimal solution) can be estimated.

In what follows we describe shortly (for more details see [14, 16]) two problems which theoretically are very hard and a greedy approach naturally generates quite good solutions.

A greedy strategy seems very natural in searching, for instance when we want to reach a destination through a shortest path or in a shortest time in a network. In one of the simplest form of such situation a greedy algorithm produces an optimal solution (see below), but in general such an approach fails, for instance to produce a shortest closed walk (TSP).

It is worth to mention that in the last 20-25 years there have been a number of heuristics proposed, called *metaheuristics*, which are based on some natural principles in science and in the nature, for instance: tabu search, simulated annealing, genetic algorithms, ant colony optimization. No theoretical evaluations of these metaheuristics are known, however they behave quite well in practice generating solutions which are 2-5% from optimum.

The Change-Making Problem

The following problem is presented to students in middle school: given a change V, find the least number of notes and coins which constitute the value V. We assume that each type of notes and coins is available in unlimited quantity. In the beginning, on the way to develop and use abstraction, students discuss how they usually get their change – quite often they get many small coins. The discussion leads to better understanding the problem, its formulation and usually ends up with a greedy strategy: in each step choose a largest possible note or coin which can be used to form the remaining change V. To make experiments with this strategy students code their greedy algorithm in a spreadsheet and test it on different values of V. This problem serves also as an example of *algorithmics with a spreadsheet*.

Then we usually post some extra questions for students, for instance: do their programs really find the smallest possible number of notes and coins for each value V? Most of the students answer YES but they do not know how to convince each other and the teacher. To illustrate that the greedy approach is not always optimal we ask students to extend our currency system (consisting of coins: 1, 2, 5, 10, 20, 50, 100, 200, 500 and some notes) by a new coin of 21 and use the greedy algorithm to make a change for $V = 63$ using the new system – instead of 3 coins (21 + 21 + 21) the greedy algorithm finds 4 coins (50 + 10 + 2 + 1). It happens quite often that some coins are not available in a cash desk. We ask students to find subsets of coins and values of V for which the greedy algorithm does not guarantee optimal solutions and moreover for some values of V may not be possible to form *V* from the available coins. It is interesting to know that for most currency systems, including the Polish Zloty, the Euro and US Dollar, the greedy strategy does find optimum solutions.

The Knapsack Problem

Packing a knapsack is another simple problem, which can be solved by using a greedy approach: a knapsack of a given capacity *W* is to be packed with some items, each having a value and weight. The goal is to take items of the total weight at most *W* and the highest possible total value. Students usually come up with one of the three greedy strategies: first take the most valuable item, first take the lightest item, first take the item with the largest value to weight ratio. Then, they try to find a problem instance for which none of these strategies produces an optimal solution. Greedy solutions can be also easily coded in a spreadsheet. We also encourage some students to develop a dynamic programming algorithm for this problem (see [13, 14] for details) which could be viewed as a recursive algorithm based on the Bellman's optimization principle applied to a two-dimensional array.

The shortest path problem

A greedy strategy applied to some optimization problems generates optimal solutions to every instance of the problem. The shortest path problem represents a family of such problems which are defined on a net consisting of points (e.g. cities or crossings of roads) and roads between the points. Each single road between two points has a length (or travel time) assigned. The shortest path problem in its simplest form is to find a path between two points (origin and destination) which consists of a sequence of consecutive simple roads and has the minimum length. Such problems/questions are usually illustrated with real-world pictures, such as in Fig. 1, which help students to make a correct abstraction.

It is natural in solving this type of problems to propose a greedy approach. Students usually come up with a strategy called the *nearest-neighbor algorithm*: start from the origin and at each step, before you reach destination, extend the path constructed so far by choosing a shortest single road. The road map in Fig. 1 constitutes a counterexample for this strategy – the shortest path has length 20 and such a path cannot be generated by this approach. A slightly modified greedy strategy always generates a shortest path between two given points. The *optimal greedy strategy* (the Dijkstra's algorithm) is defined as follows: start from the origin and at each step, before you reach destination, add to the set of points that have been reached a point which is not in this set and has a shortest path from the origin.

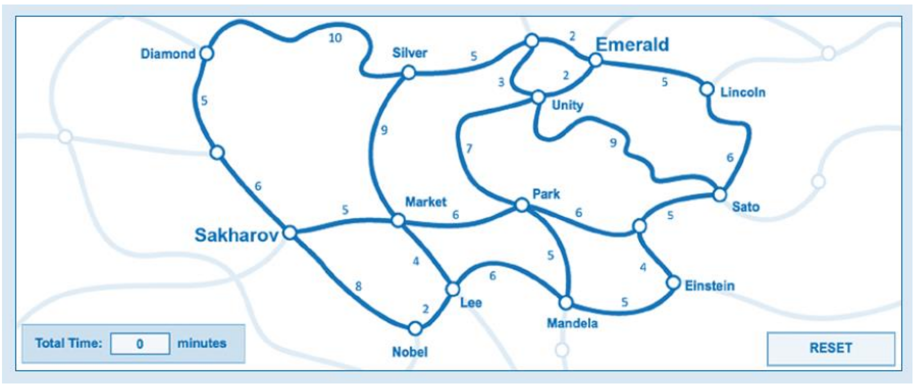

Figure 1. Finding a shortest path from Sakharov to Emerald (PISA 2012 Test, OECD [11])

4. Conclusions

In this paper we focus on applying computational thinking mental tools to some topics from school mathematics. Our goal is twofold: (1) to extend and enrich traditional topics in school mathematics by applying to them computational thinking and as a result to obtain solutions which use and are supported by the power of computer science as a discipline and computers as computing tools; (2) and to contribute to constructionist learning of mathematics, learning by doing and making meaningful objects in the real world.

References

[1] M. Armoni, "Reduction in CS: A (Mostly) quantitative analysis of reductive solutions to algorithmic problem", *ACM J. Educational Resources in Computing*, vol.8, Article 11, 2009

[2] *Computer science: A curriculum for schools*, Mar. 2012: http://www.computingatschool.org.uk/data/uploads/ComputingCurric.pdf

[3] CSTA: Computational Thinking Task Force: http://csta.acm.org/Curriculum/sub/CompThinking.html

[4] P. J. Denning, "The profession of IT beyond computational thinking", *Comm. ACM*, vol. 52, no. 6, pp. 28-30, June 2009.

[5] E. Gurbiel, G. Hard-Olejniczak, E. Kołczyk, H. Krupicka, M. M. Sysło, *Informatics. Textbook for high school*, (In Polish), vols. 1 and 2, Warszawa: WSiP, 2002-2003.

[6] ISTE, http://www.iste.org/learn/computational-thinking

[7] U. Leron, R. Zaskis, "Computational recursion and mathematical induction", *For the Learning of Mathematics* vol. 6, no. 2, pp. 25-28, 1986.

[8] J. J. Lu, G. H. L. Fletcher, "Thinking about computational thinking", in *SIGCSE*, 2009, pp. 260-264.

[9] S. B. Maurer, "Two meanings of algorithmic mathematics", *The Mathematics Teacher*, vol. 77, pp. 430-435, 1984.

[10] S. Papert, I. Harel, *Situating constructionism*, in *Constructionism*, Ablex, 1991.

[11] PISA 2012 Results (OECD): http://www.oecd.org/pisa/keyfindings/pisa-2012-results.htm

[12] H. Steinhaus, *Mathematical Snapshots*, 3rd ed. New York: Oxford University Press, 1969.

[13] M. M. Sysło, *Algorithms* (in Polish), Warszawa: WSiP, 1997.

[14] M. M. Sysło, *Pyramids, Cones and Other Algorithmic Constructions* (in Polish), Warszawa: WSiP, 1998.

[15] M. M. Sysło, A. B. Kwiatkowska, "Contribution of informatics education to mathematics education in schools, in *ISSEP 2006*, 2006, pp. 209-219.

[16] M. M. Sysło, A. B. Kwiatkowska, "The challenging face of informatics education in Poland", in *ISSEP 2008*, 2008, pp. 1-18.

[17] M. M. Sysło, A. B. Kwiatkowska, "Informatics for all high school students, A computational thinking approach, in *ISSEP 2013*, 2013, pp. 43-56.

[18] M. M. Sysło, A. B. Kwiatkowska, "Think logarithmically!", *KEYCIT*, July 2014.

[19] M. M. Sysło, A. B. Kwiatkowska, "Introducing students to recursion: a multi-facet and multi-tool approach", *ISSEP 2014*, September 2014.

[20] J. M. Wing, "Computational thinking", *Comm. ACM*, vol. 49, no. 3, pp. 33-35, Mar. 2006.

[21] J. M. Wing, "Research notebook: computational thinking – what and why?", http://link.cs.cmu.edu/article.php?a=600