The Education Column

by

Dennis Komm and Thomas Zeume

ETH Zurich, Switzerland and Ruhr-University Bochum, Germany

dennis.komm@inf.ethz.ch and thomas.zeume@rub.de

TEACHING DESIGN OF ALGORITHMS IN HIGH SCHOOLS BY CONSTRUCTIVE INDUCTION

Juraj Hromkovič
Department of Computer Science, ETH Zürich
juraj.hromkovic@inf.ethz.ch

Regula Lacher
Department of Computer Science, ETH Zürich
regula.lacher@inf.ethz.ch

Abstract

The design of algorithms is one of the hardest topics of high school computer science. This is mainly due to the universality of algorithms as solution methods that guarantee the calculation of a correct solution for potentially infinitely many instances of an algorithmic problem. The goal of this paper is to present a comprehensible and robust algorithms design strategy called "constructive induction" that enables high school students to solve a large variety of algorithmic problems. In general, this strengthens learners in problem solving and their ability to use and develop abstract representations.

1 Introduction or Why Teaching Algorithms in High Schools is Not Easy

"Why to teach algorithms?" Because it forces learners to strengthen their ability to abstract and to solve problems. Abstraction and problem solving are crucial dimensions of the human way of thinking and basic instruments for discovering and shaping the world. Therefore, abilities to abstract and solve problems should be a key issue in any educational system. Especially, the main focus of teaching mathematics and computer science has to be devoted to supporting the learner's ability to abstract and solve problems in their abstract representation.

"Why is teaching algorithms difficult?" To answer this question let us start with relating it to the question "Why is teaching the concept of variables in introductory programming courses the first big threshold?" Good programming courses for novices pay attention to the cognitive load of students and make sure that progress is made in very small steps. The starting point is to view programs as descriptions of activities in the programming language that are understandable for machines (computers, robots, etc.) with the goal to delegate the execution of these activities to technology. In this simplified scenario one program describes exactly one activity. If you take a good choice of your programming language and initial exercises, the first programs describe activities whose execution can be observed visually instruction by instruction (see the concept of the *notional machine* [2,4,6,8,10–12]). With this approach students can develop a program and immediately investigate its properties and functionality, and in this way verify the correctness of their program. This is the reason why even very small kids in primary school can master some basic programming with success and joy. This is also why the repeat-loop with the hidden control variable was introduced to Logo [13].

If you introduce variables, even as passive input parameters, the game changes heavily. Why is programming with variables much harder? One program is not responsible for only one activity anymore, but depending on the values of its parameters for potentially infinitely many different activities [1]. How now to check the functionality of your program? You cannot teach complete induction for proving correctness of programs or any sophisticated verification method of software engineering. For sure you can try to test your program for a few values of its parameters. Consequently, you can fail to believe in the correctness of an incorrect program, but the main problem is whether these few tests are really sufficient to get an intuition of why your program should work. And the minimal goal of any computer science teacher should be to offer at least some intuition of the program functionality.

An algorithmic problem consists of infinitely many concrete problem instances, and an algorithm is a solution method that works correctly and successfully for any one of these infinitely many instances. Hence you are asked to develop a strategy that behaves well in all infinitely many possible situations. This is a very nontrivial task, especially if you take into account that high school students have almost no experience with the universal quantifier. We cannot assume any experience with using mathematical induction for proving infinitely many parameterized claims, and so we cannot strive to prove the formal correctness of the algorithms developed. Of course we could omit verification proofs. But if our lessons have an educational value, then they have to offer some reasonable intuition, why the algorithm designed works properly. To reach this goal, teaching must be designed in such a way that students have enough freedom to design algorithms to a high extent on their own. This is a very nontrivial task we will try to approach in the subsequent sections.

Let us consider another dimension in answering our question "Why is teaching algorithms in high school not easy?" A teacher can aim to explain some famous algorithm and the goal could be to be able to execute the algorithm by hand. We question this goal. First of all, the value of teaching acting according to a given pattern is very low [9], because it contributes very little to the development of our thinking (creativity, improvisation, fantasy). Every procedure we can describe can be automized and so executed more reliably and quicker by technology than by humans. Secondly, most efficient algorithms are so specifically adjusted to the problems they solve, that a small change in the specification of the algorithmic problem can cause the algorithm to fail. We say that such algorithms are not robust and so their teaching does not guarantee an essential progress in developing and strengthening the ability of students to solve problems. We argue that we have to teach robust strategies for problem solving that can be successfully applied to a big variety of problems. The remaining question now is which of the algorithm design methods can be taught successfully in high schools. In the following we present one of these methods, called *constructive induction*, and show how a teacher can use it in schools to train problem solving and algorithm design.

2 Constructive Induction

Everybody knows *mathematical induction* (also called *complete induction*) used to prove infinitely (countably) many parameterized claims. The word induction has its origin in Latin ("inductio" – "to lead into" in English). Probably the oldest induction proof was done by al-Karaji in the tenth century for proving the binomial theorem (Pascal's triangle) about the form of coefficients [3, 5]. Unfortunately, this manuscript was lost and we only have the reference in the book "The Brilliant in Algebra" by Al Samawal al-Maghribi (around 1150). In European culture induction as a proof method was used for the first time by Francesco Maurolico in 1575 [14] for proving that the sum of the first n odd integers is n^2. It took several years until the method was used again (Blaise Pascal, 1654; Jacob I Bernoulli, 1686). In 1888, Richard Dedekind started to call this proof method "complete induction." Giuseppe Peano presented induction as a part of his axiomatic system in 1889. Since then, this method belongs to the fundamental instruments of mathematics. Because reading, correcting, and writing proofs is not a mandatory subject of mathematics in high school in most countries, one can question whether it is reasonable to do it in computer science lessons. But we want to deal with constructive induction here, which is easier available to students.

Constructive induction for building sequences of objects or for solving problems is much older than complete induction as a proof method and was used already in ancient time. The simplest fundamental example is the construction of natural numbers. For each number n we can construct a larger number, for instance n+1. In this case we use the constructive induction to construct an infinite sequence of objects. Another antic example is the claim that there are infinitely many primes.

Take the n smallest primes $p_1, p_2, \dots, p_n$.

Now we develop a strategy for finding the next prime p_{n+1}. Take the number

$$m = p_1 \cdot p_2 \cdot \dots \cdot p_n + 1 \ ,$$

which is not divisible by any of the first n primes $p_1, \dots, p_n$. So there must be a prime in the interval between p_n and m. Now one has to check these finitely many candidates from the smallest one to the largest one to find the next prime. What we see in this example? For any n, we have a strategy for how to find the $(n+1)$-th prime if you have the first n primes, and you proved that this strategy works (i.e., that there are infinitely many primes).

In the examples above we saw how we can generate step by step the next object of an infinite sequence of objects if we know the previous ones. For sure this method works only if we have or can construct the starting object of this sequence.

We can extend the constructive induction presented above for solving problems and designing algorithms. The idea is as follows. First, we have to parameterize the problem. This means that we have to partition its infinitely many instances into infinitely many classes numbered by natural numbers. The classes can be finite or infinite. For instance, a parameter of a graph can be the number of its vertices, the number of its edges, the size of its adjacency matrix, the maximum degree, or the diameter. A parameter of an integer can be the integer itself or the length of its representation in some number system. A parameter for sorting or searching can be the number of elements or the length of the whole input representation. After parameterizing a problem, we say that a problem instance is of size k if it belongs to the k-th class.

The idea of using constructive induction to solve problems is now very similar to complete induction. First, one solves the problem for the smallest parameter, i.e., the smallest problem instances. We speak about the base of the induction. Then, for any positive integer n, one executes a general strategy, how to use solutions for instances smaller than n (with parameters smaller than n), in most cases even only with parameter $n-1$, to construct a solution for any instance of size n. The key issue here is searching for this general strategy.

In Sections 3 and 4 we show that there are plenty of problems for which such strategies are quite natural and can be discovered successfully. This is important, because after experiencing some examples the students can start searching for algorithmic solutions to similar problems on their own. In Section 3 we show how to use induction to solve problems, and in Section 4 we apply constructive induction to teach high school students to design efficient algorithms.

3 Solving Problems by Constructive Induction

We present a sequence of problems that can be transparently solved by constructive induction. A detailed description of the lessons for high school students can be found in a recent textbook [7].

The number of decimal numbers with at most *n* digits. The question is how many decimal numbers of length at most n exist. The length of a number is the number of digits in its representation, which is the parameter of this problem. This is a very simple introductory task.

The base of the induction is easy. We have 10 digits $0, 1, 2, \ldots, 9$, and so we have 10 integers of length 1. To discover the general step, we always encourage students to go from size 1 to size 2, from size 2 to size 3, etc. until they recognize a pattern in these concrete steps. Going from size 1 to size 2 here means to build a table of size 10×10. In the rows there are 10 digits $0, 1, 2, \ldots, 9$, and the columns contain the 10 decimal numbers of length 1. Combining the labels of the rows with the labels of the columns we get 100 sequences of 2 digits. Removing the leading zeros, we have all 100 decimal numbers of length at most 2. The generalization of this strategy is transparent. For the representation length at most n we take a $10 \times 10^{n-1}$ table. In the rows we have all 10 digits, and, in the columns, we have all sequences of $n - 1$ digits.

Note that this problem can be solved easier by asking which is the largest integer of length n. The answer is the number consisting of digits 9 only. But the advantage of using this task is to get a very simple introduction to constructive induction.

Lines in two-dimensional Euclidean space. Suppose we have n different lines in two-dimensional Euclidean space, and we are asked to place them in such a way that the number of crossing points is as large as possible (see Figure 1 for an example). Note that the minimal number of crossing points is 0 if the lines are parallel to each other. Obviously, the parameter n is the number of lines. For $n = 1$ there is no crossing point. For $n = 2$ there is exactly one crossing point if the lines are not parallel. The strategy is to place the n-th line in such a way that it crosses all $n - 1$ already placed lines in new $n - 1$ crossing points. This allows us to count the maximal number of available crossing points of n lines. It is

$$1 + 2 + 3 + \cdots + (n - 1) ,$$

and can be also derived by using the recurrence $L(n) = L(n - 1) + (n - 1)$.

Using the “small Gauss” one can get an explicit formula. But we can already use the sum above to develop an algorithm (program) counting the maximal number of crossing points of n lines.

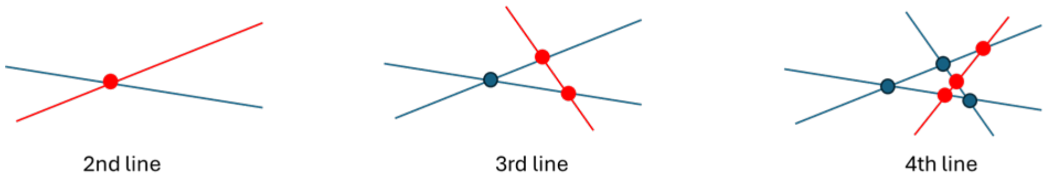


Figure 1: Crossing points when lines are added in the two-dimensional Euclidean space

The introductory example of Poya. The previous problem is only a preparation of the following problem used by Poya when introducing the power of induction. There are n lines in two-dimensional Euclidean place. What is the maximal number of areas in which the space can be partitioned in this way?

One can start with 1 line getting 2 areas. Taking 2 lines one can get 4. Taking 3 lines we will get 7 areas. Clearly, the solution cannot be 2^n, and the task starts to be challenging. For finding the solution, solving the previous task can be helpful. With the n-th line added to the previous $n-1$ lines we can get $n-1$ new crossing points. Each of the $n-2$ segments between two new crossing points partitions an area into two subareas. The segment "before" the first crossing point and after the "last" crossing point also partitions areas into two subareas. Hence, adding the n-th line to the already placed $n-1$ lines increases the number of areas by n. Therefore, the maximal number of areas obtained by placing n lines is

$$2 + 2 + 3 + \cdots + n \, .$$

To see this, denote the number of areas for n lines by $T(n)$ and consider $T(1) = 2$ and the derived recursion $T(n) = T(n-1) + n$.

If you are searching for a challenge, consider the number of subareas of three-dimensional Euclidean space obtained by placing two-dimensional planes. As an exercise for students, you can take circuits instead of lines (or other suitable geometric objects) and ask to solve the problem by constructive induction.

Number of triangles in a pyramid. One can build a pyramid from equilateral triangles. In Figure 2 we see the four smallest pyramids with the heights 1, 2, 3, and 4. The question is how many triangles are in the pyramid of height n for any positive integer n. After previous experience, the students can compute $PN(1) = 1$, $PN(2) = 4$, and finally the recurrence $PN(n) = PN(n-1) + 2n - 1$. To make the task easier the teacher may first ask for the number $A(n)$ of triangles in the lowest level of the pyramid of height n. Here one can easily establish $A(1) = 1$ and $A(n) = A(n-1) + 2$, and so $A(n) = 2n - 1$. Then the formula $PN(n) = PN(n-1) + A(n)$ follows. If you decide to search for an explicit formula,

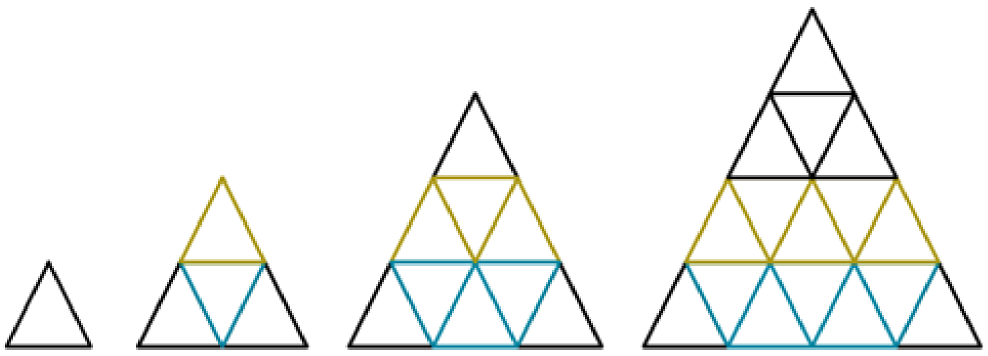


Figure 2: Pyramid built from equilateral triangles

you can derive $PN(n) = n^2$. We call attention to this because in this way you get the classical example of the sum of the first n odd numbers presented by Francesco Maurolico.

An interesting point here is that one can calculate the number of triangles in a pyramid by dividing the area of the pyramid by the area of the triangle (taking, for instance, 1 as the size of the equilateral triangle as the basic building stone). One can generate similar tasks by building pyramids from squares or hexagons.

Coloring regions of a map. The famous *four color theorem* states that 4 colors are always sufficient to color a two-dimensional map consisting of regions in such a way that no two neighboring regions have the same color. Two regions are considered to be neighbors if they have a common continuous border consisting of infinitely many points. One could ask whether fewer colors are sufficient if the partition of the map into regions is done in some regular way. Using our experience with the task from Poya, we can pose the following question: A map is partitioned into regions by n lines. How many colors are sufficient to color the map?

If you start with 1, 2, 3, or 4 lines (see Figure 3), after some attempts the students can discover that for small parameter values two colors are sufficient. So we have a hypothesis, but the question is how to prove it: The natural way is by constructive induction. First, observe that exchanging the two colors leads to a valid coloring. Secondly, add a new line to a map colored by two colors (see Figure 4). Again, we can view the new line as a sequence of segments between the crossing points with other lines. All these segments have the same colors on both sides now and all other boundaries between two regions have different colors. Keep the coloring on one side of the new line and flip the colors on the other side. In this way we get a valid coloring of the new map by two colors.

The strength of this task is that you can generate plenty of similar ones to train the class. You can take circles to partition the plane into regions, or triangles

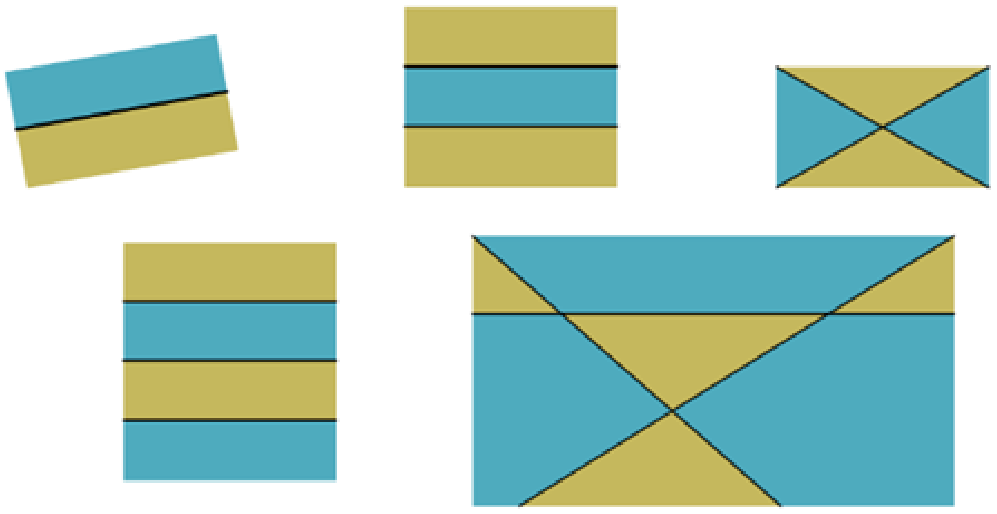


Figure 3: Two colors are sufficient to color all areas created by 1, 2, or 3 lines

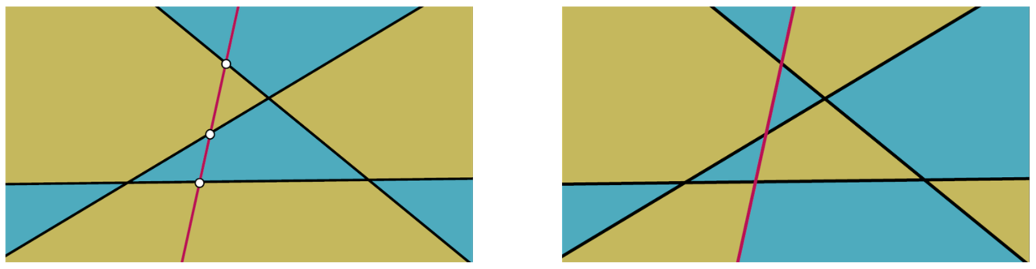


Figure 4: Adding a 4-th line and changing the color of all areas on one side of the new line

(rectangles, etc.) with and without the request that these geometrical objects intersect in a finite number of points only. If the request is satisfied, two colors are always sufficient. The strategy is to put the next object into the plane, and then flip the colors either inside of the new object or outside. If the constraint is not satisfied, one can ask students to construct examples that need at least 4 colors to be properly colored. Then the students can be asked to find the reason why the previous strategy with flipping colors inside of the new object does not work.

4 Designing Algorithms by Constructive Induction

Here we can start with the last task of the previous chapter, which is a good example for moving from problem solving to algorithm design.

Coloring regions of a map. The approach to solve the problem of dividing a plane by lines (circles, etc.) into regions offers an algorithm for two-coloring. Place the first line and color the plane with 2 colors. Then add one line after another and always exchange the colors on one side of the line.

Discovering the multiplication algorithm. Only few people are aware of the fact that the multiplication algorithm currently used in schools was designed by constructive induction. If you want to compute the product $a \cdot b$ you can parameterize by the length of the representation of b. The length of a does not matter. Suppose b consists of $n + 1$ digits and assume we can multiply by b with n digits. Let

$$b = b_n b_{n-1} \dots b_1 b_0 \ .$$

We can then write

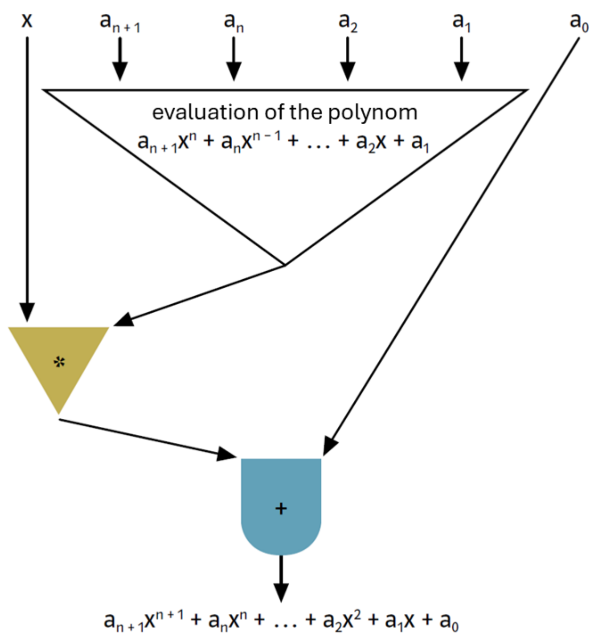
$$a \cdot b = a \cdot (b_n b_{n-1} \dots b_1 b_0) = a \cdot b_n b_{n-1} \dots b_1 \cdot 10 + a \cdot b_0 \ .$$

So we see that one multiplication by a number consisting of n digits, one shift by one position, one multiplication by one digit, and one addition are sufficient to compute the multiplication by a number consisting of $n + 1$ digits. This is exactly the base of our school multiplication algorithm.

To train this concept one can multiply in other number systems or take other arithmetic operations [7].

Horner's method. The development of Horner's schema for evaluating a polynomial is a show case of applying constructive induction. Better than by any formula, the algorithm is explained by Figure 5.

Who is the agent? We have n people and want to discover who of them is an agent. We know that there is an agent among those n people. How to recognize her or him? The agent is the person who knows everybody (all other $n - 1$ people), but nobody knows her or him. We are allowed to pose the following questions: "Person A, do you know person B?" and will always get the correct answer. The task is to find the agent with as few questions as possible.



x
a_{n+1}
a_n
a_2
a_1
a_0
evaluation of the polynom
$a_{n+1}x^n + a_n x^{n-1} + \ldots + a_2 x + a_1$
*
+
$a_{n+1}x^{n+1} + a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0$

Figure 5: Horner's schema for the evaluation of a polynomial

First we observe that there can be at most one agent in the group of people. If there would be two, then each of them would know the other, and so none of them could be an agent. The induction base for the group size of one person is simple, no question is needed. For the induction step, the point is to recognize that one question is sufficient to reduce the number of candidates by 1. If we ask whether A knows B, then the answer "yes" excludes B as an agent candidate. If the answer is "no," then A cannot be the agent. In this way we see that $n - 1$ questions are sufficient to find the agent if one knows in advance that there is an agent in the given group of people.

There is a wonderful extension of this task by allowing some restricted cheating (one or more wrong answers). This extension leads to the development of self-verifying codes for the corresponding numbers of errors.

Sorting and searching. Also in this fundamental area of algorithm design the concept of constructive induction is very fruitful. One can start with searching the minimum or maximum of n elements and design an algorithm with $n - 1$ comparisons by constructive induction.

The next step could be to do *binary search*. One can take the length of the number representation of n (approximately the discrete logarithm of n) as the parameter for the size of sorted sequences of n elements. The induction step then shows that, if one can find an element in a sorted sequence of m elements, then one more comparison is sufficient to find an element in a sorted sequence of $2m$ elements.

For sorting algorithms one can consider *insertion sort*. If one has a sorted sequence of n elements and takes a new element, then one has to find its position in the sorted sequence. One can get different algorithms depending on the strategy used. If one uses *bubble sort* for placing the new element, the resulting algorithm has a quadratic number of comparisons. If one uses *binary search*, the complexity is in $O(n \log n)$.

Another sorting algorithm can search for the maximum (as bubble sort does) and then sort the remaining $n - 1$ elements.

There are plenty of possibilities to create further exercises here. For instance, one can search for the k largest elements.

Winning strategies for games. Constructive induction is a genius strategy for searching for winning strategies for finite games. You start with your winning configurations, and in each constructive step you label the winning configurations until all winning configurations are labeled. You do the same for the losing configurations. The labeling rules are very simple: Start with your winning configurations and losing configurations defined by the game, and in each constructive step you label further your winning or losing configurations.

You continue until all configurations of the game are labelled. You label a configuration as your winning configuration if you can move from this configuration in one step to a configuration that is already labeled as your winning configuration (if the turn is yours). If its your opponent's turn, you label a configuration as your winning configuration if she or he can only move to one of your winning configurations. When it is your turn, you label a configuration as your losing configuration (winning configuration of your contrary) if all your possible moves end in an already marked winning configuration of your opponent. Also, you label a configuration as your losing configuration, if your opponent can reach in one step a winning configuration for her or him.

There are many simple games that can be completely analyzed by this strategy [7].

Dijkstra's algorithm. This is an advanced example. Especially since the number of different paths between two vertices in a network can be exponential in the size of the network, and we want to avoid looking at all possible paths in order to find the shortest one.

The key issue when developing Dijkstra's algorithm by constructive induction is the choice of the parameter. First, one defines the problem of finding the shortest paths in a network from the source to the k closest vertices. Then k is the parameter, not the size of the network. The base for $k = 1$ is easy, one takes that neighbor of the source that is connected to it by the cheapest edge. The induction step takes the tree with the k shortest paths to the k closest vertices and argues that the shortest path to the $(k+1)$-th closest vertex must go via the edges of the tree of the k shortest paths. Then one can efficiently estimate the next closest vertex by considering only edges of the tree and the edges leading from the tree to the vertices not belonging to the k closest ones.

If one wants to start teaching or discovering Dijkstra's algorithm with an easier task, then one can search for shortest paths from a source to all other vertices in a network whose edges all have the same value and so develop the *breadth-first-search* algorithm.

5 Conclusion

If one wants to strengthen students in algorithmic (computational) thinking, one should not try to correctly execute presented algorithms only. Instead, one has to push the students to use and develop abstractions and problem solving competencies. This requires developing robust strategies for designing algorithms for a large variety of problems. Teaching in high school asks for natural and well understandable strategies.

Constructive induction is the oldest robust strategy for solving problems and it is very transparent and comprehensible to students. But the key point is that constructive induction has a very high educational value with respect to the development of the student's way of thinking. Except the potential of introducing induction as a proof method, constructive induction offers a special case of recursion and so can be used as an easy introduction to the recursive algorithm design method "divide et impera" (divide and conquer). The induction step results in a recurrence that corresponds to the reduction of solving problem instances of size n to solving problem instances of size smaller than n, and so to the concept of "divide et impera." Also note that using constructive induction in problem solving consequently from smaller instance sizes to larger ones is also the base of "dynamic programming," another fundamental algorithm design technique.

In this paper we only outlined what kind of problems can be approached by constructive induction in high schools. A careful implementation of teaching solving problems by constructive induction is presented in our textbook [7], which shows how to guide students such that they discover solutions and develop algorithms to a large extend on their own.

References

[1] J. Arnold, C. Donner, U. Hauser, M. Hauswirth, J. Hromkovič, T. Kohn, D. Komm, D. Maletinsky, and N. Roth (2019). *Programmieren und Robotik*. Klett und Balmer AG, Baar, Switzerland.

[2] B. du Boulay (1986). Some difficulties of learning to program. *Journal of Educational Computing Research* 2(1):57–73.

[3] W. H. Bussey (1917). The origin of mathematical induction. *The American Mathematical Monthly* 24(5):199–207.

[4] A. Cypher (1993). *Watch What I Do: Programming by Demonstration*. MIT Press. Cambridge, MA, USA and London, UK.

[5] Encyclopedia of Mathematics (2021). *Mathematical Induction*. Online: http://encyclopediaofmath.org/index.php?title=Mathematical_induction, last accessed September 20, 2024.

[6] S. Fincher, J. Jeuring, C. S. Miller, P. Donaldson, B. du Boulay, M. Hauswirth, A. Hellas, F. Hermans, C. Lewis, A. Mühling, J. L. Pearce, and A. Petersen (2020). Notional machines in computing education: the education of attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2020)*. ACM, New York, NY, USA, pp. 21–50.

[7] J. Gallenbacher, J. Hromkovič, R. Lacher, D. Komm, and H. Pierhöfer (2023). *Algorithmen und Künstliche Intelligenz*. Klett and Balmer AG, Baar, Switzerland.

[8] J. Hromkovič, T. Kohn, D. Komm, and G. Serafini (2016). Combining the power of Python with the simplicity of Logo for a sustainable computer science education. In *Proceedings of the 9th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2016)*. LNCS 9973, pp. 155–166. Springer.

[9] J. Hromkovič and R. Lacher (2023). How teaching informatics can contribute to improving education in general. *Bulletin of EATCS* 139.

[10] T. Kohn (2017). *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. Dissertation 24076, ETH Zürich, Switzerland.

[11] T. Kohn and D. Komm (2018). Teaching programming and algorithmic complexity with tangible machines. In *Proceedings of Informatics in Schools: Fundamentals of Computer Science and Software Engineering (ISSEP 2018)*. LNCS 11169, pp. 68–83. Springer.

[12] H. Lieberman (2001). *Your Wish is My Command: Programming By Example*. Morgan Kaufmann, San Francisco, CA, USA.

[13] S. Papert (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

[14] G. Vacca (1909). Maurolycus, the first discoverer of the principle of mathematical induction. *Bulletin of the American Mathematical Society* 16(2):70–73.