УДК: 519.681.2, 51-8

SOLVING CONTEST PROBLEMS VIA FORMAL PROGRAM VERIFICATION[1]

N.V. Shilov, S.O. Shilova

Ershov Institute of Informatics Systems, 630090, Russia, Novosibirsk, Lavrentev av., 6,
tel. (+7) 383 330 52 63, fax. (+7) 383 332 34 94
E-mail: shilov@iis.nsk.su

The interface between mathematics and computer science is many-sided. In particular, E.W. Dijkstra promoted a special “computer science” approach to mathematics problem solving. The approach combines a heuristic algorithm design and rigorous mathematical proof of algorithm correctness (in style of A. Hoare and R. Floyd). The paper sketches two problems of this kind in a form of tutorials for undergraduate students that are interested in different programming contests (like ACM International Collegiate Programming Contests). These tutorials took place at Novosibirsk State University in years 2005–2008. The paper also dioceses some direction for further research that emerge from the problems.

Introduction

Science Olympiads and contests brings spirit of competitiveness to Science education. They benefit best students, engage them with research. Simultaneously Science Olympiads and contests challenge faculty to enhance teaching so that regular student can enjoy Olympiad and contest problems.

Programming contests become very popular with undergraduate students in recent years. Maybe the Association for Computing Machinery International Collegiate Programming Contest (ACM ICPC) is the most popular world-wide. The initiative was born at early 1970s in US, then evolved to North America Computer Science competition, and was formally inaugurated in 1977 at the first World Final ACM ICPC. Overall number of participants of annual multi-level contests (at local, sub-regional, regional and final levels) “has grown to several tens of thousands of the finest students and faculty in computing disciplines at almost 2,000 universities from over 80 countries on six continents” (http://cm.baylor.edu/welcome.icpc).

Basics of the adopted format of ACM ICPC follows. Every contest team consists of three undergraduates (to say nothing of a computer ;-). A team has to solve 8-10 “real-world” problems in five hours competition. Team-members jointly rank the difficulty of the problems, design a formal models of them and an algorithm that solves the formalized problems, implement the algorithms, test the resulting programs, and submit the programs to jury. Jury adopts a program being correct if the program successfully exercises a number of preliminary designed test data suites. All problem statements are provided with samples of test data and range of admissible data values. But teams have no access to jury test suites. Each incorrect submission is fined by a penalty. At the end of the contest, teams are ranked by number of correct submissions, and (if several teams have the same numbers) by value of penalties for incorrect submissions. (Please refer to http://cm.baylor.edu/welcome.icpc to learn more about the ICPC).

Unfortunately, ACM ICPC and many other Computer Science contests become much more about programming than about Science. They become more similar to a technical sport than to Science Olympiad due to limited role of research and innovative component in these contests. It seems that

- art of problem formal modeling,
- cooking book of algorithms in heads,
- rapid typing skills in hands

are three corner-stones of a success in these contests. Of course, all listed skills are related to Computer Science proficiency. The art of modeling is especially important since it is about research skills, not about technical skills. This research component puts Computer Science Contests in line with science Olympiad like Mathematics and Physics Olympiads.

But research component in Computer Science is not limited by art of modeling. In particular, it includes formal mathematical proofs of model properties and program correctness. Moreover, sometimes without these proofs, a utility of a program that “solves” a problem is very conventional in spite of successful and extensive program testing.

At this talk we present a number of particular problems that fit Mathematics Olympiad and Computer Science Contests format simultaneously. In the case of mathematics Olympiad, the problems are about existential proofs. In the case of Computer Science contests these problems are about algorithm design and implementation, but they can not be considered correct without formal proof. The proofs in these cases can be carried out in pure traditional mathematical style, or in Computer Science way, i.e. in a manner that is mathematically strict but Computer Science in nature. We do believe that these problems can help to overcome some alienation when gifted Computer Science students consider Mathematics being too pure, and talented Mathematics students consider Computer Science being too poor.

[1] The work is supported by RFBR grant 09-01-00361-a.

The paper presents material of two tutorials that author gave to undergraduate students who are interested in different programming contests like ACM International Collegiate Programming Contests (ACM ICPC). The tutorials were part of a special undergraduate courses presented for students of Novosibirsk State University during their preparation for regional ACM ICPC in years 2003-2008. Maybe this experience will help to engage students with theory of Formal Methods via Programming Contests [11].

Dijkstra approach

It is more than seven years since a laureate of the ACM Turing Award (1973), Prof. Edsger W. Dijkstra (May 11, 1930 – August 6, 2002) passed away. He always promoted an attitude to computer science as to a special branch of science like mathematics, physics, or biology [4]. This branch of science has its own objects of studies and methods of research. Of course, this particular branch can adopt ideas, approaches, and methods from other branches. The progress of quantum computing in the last decade is the most popular example of how another branch of science can benefit Computer Science. However E. Dijkstra believed that computer science in turn could nurture other branches of science. The advance of bioinformatics, for example, is perhaps the most recent evidence of the correctness of this belief of Dijkstra.

Theoretical computer science is a common example of how mathematics has contributed to computer science. We can say that theoretical computer science is the study of the mathematical models of algorithms, data structures, programs, and so on with the use of mathematical logic, abstract algebra, topology, and so on. This leads to the advance of program and temporal logic, cryptography, domain theory, and so on. In contrast, few examples show how the ideas, approaches and methods of theoretical computer science advance mathematics. An importance of theoretical computer science ideas for better mathematics education has been acknowledged [7]. The heritage of Dijkstra, however, also includes a number of brilliant miniatures that illustrate how the ideas, approaches and methods of theoretical computer science can be used in fields such as elementary mathematics, combinatorics, and calculus. Although many of Dijkstra's miniatures were not published, they can be found in the electronic archive of Dijkstra manuscripts available on-line at the URL http://www.cs.utexas.edu/users/EWD/.

Unfortunately, some of Dijkstra's works are missing. To the best of our knowledge, the following lecture of E.Dijkstra at Marktoberdorf Summer School "Deductive Program Design" (1994) is not presented[2] in the archive. In this lecture he has addressed the following planimetry problem: There are N black and N white points on the plane. Every three of them are non-collinear[3]. Prove that it is possible to connect black and white points in 1-1 manner by intervals without intersections.

First of all, E. Dijkstra transformed the original planimetry problem into the following problem for algorithm design (that we will refer in the sequal as Dijkstra problem): Design an algorithm that couples in 1-1 manner N input black and N input white points on the plane so that intervals between coupled points do not intersect.

Next he proceeded the problem as follows. Let us fix positions of arbitrary given N black and N white points on the plane. Let COUPLING be a data type whose values are sets of intervals that couple these fixed black and white points in 1-1 manner. There are a constant ARBITRARY of COUPLING type and two operations that handle values of this type are absolutely natural for Computer Science[4]:

- GOOD : COUPLING $\rightarrow$ BOOLEAN,
- FLIP : COUPLING $\rightarrow$ COUPLING.

ARBITRARY returns some constant value of type COUPLING (i.e. a set of intervals). GOOD returns TRUE iff the value of its argument (i.e. a set of intervals) has no intersections. FLIP finds an intersection of a pair of intervals in the value of its argument (i.e. a set of intervals), flips it and returns this modified value (fig. 1).

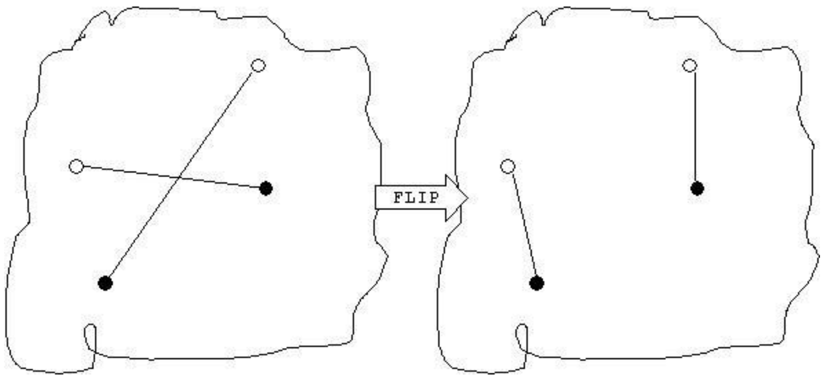

Figure 1. Semantics of FLIP

[2] We have discussed this lecture in our preliminary paper [12].

[3] Points are said to be collinear iff they belong to some straight line simultaneously.

[4] Computer Science notation F: $T_1 \times \ldots T_m \rightarrow T$ for operations has usual mathematical meaning when $m \geq 1$: a function from data types $T_1, \ldots T_m$ to T; a special case is notation F: $\rightarrow$ T with empty list of arguments: it is reserved for constants of type T.

From the very beginning, it is possible to suggest two approaches to design of an algorithm: exhaustive search and heuristic search. The exhaustive search approach is quite straightforward: search all *N!* possible couplings for a good one (if any). The heuristic search is also based on a simple idea: eliminate locally intersections of pairs of intervals in couplings, i.e. just flip them; if you are lucky, a heuristic search may find a good coupling!

In terms of the COUPLING data type the high-level design of the heuristic search algorithm can be presented as follows:

```
VAR X : COUPLING;
BEGIN
        X:= ARBITRARY;
        WHILE !GOOD(X) DO X:= FLIP(X)
END
```

E. Dijkstra observed that if the above heuristic search algorithm eventually terminates, it automatically terminates with a set of intervals without intersections. Then he hinted that a condition that may lead to termination of the algorithm is the precondition from the original planimetry problem: every three of the points are non-collinear. Hence, to solve the original planimetry problem, it is sufficient to prove that this precondition leads to termination of the heuristic search algorithm.

There are many techniques for proving program termination. Another laureate of the ACM Turing Award (1978) Robert W. Floyd developed one of them. His method is based on mappings to well-founded sets. It can be briefly described as follows.

A well-founded set (WFS) is a partially ordered set *(D, ≤)* without infinite decreasing sequences $d_1 > d_2 > ...$ Assume that *f* is a total mapping from configurations of a program into a well-founded set. If some precondition guarantees that each iteration of the program decreases the value of the function *f*, then it guarantees program termination. (Please refer to a comprehensive textbook [1] for details.)

To apply the method to the heuristic search algorithm, one has to define a suitable WFS *(D, ≤)* and an appropriate function f : COUPLING $\rightarrow D$ that maps the values of the single algorithm variable X into elements of *D* and decreases after each iteration of the WHILE-loop in the algorithm. The loop body consists of the single operator X:= FLIP(X). Hence, if a suitable WPS and an appropriate mapping exit then the following must hold: if GOOD(X) is wrong then f(X) > f(FLIP(X)).

The most straightforward candidate for (*D*, ≤) is the set of natural numbers with the standard ordering *(Nat, ≤)*, and for f – the number of intersections in a set of intervals. Unfortunately, the number of intersections can increase after FLIP (fig. 2). In contrast, a value that really decreases every time after FLIP is the sum of lengths of intervals (fig. 3): that is, if no three points in B_1, B_2, W_1, and W_2 belong to a straight line, then $|B_1,W_1| + |B_2,W_2| > |B_1,W_2| + |B_2,W_1|$ due to the triangle inequality. Hence it is possible to adopt a finite set of positive real numbers D $=\{ r : r = \Sigma_{[Bi,Wj] \in X} |B_i,W_j|$ *where X is a coupling}* with the standard ordering $\leq$ as a WFS, and the function (λ X . $(\Sigma_{[Bi,Wj] \in X} |B_i,W_j|)$) on COUPLING as a mapping f : COUPLING $\rightarrow D$. This finishes E. Dijkstra's solution of the original planimetry problem.

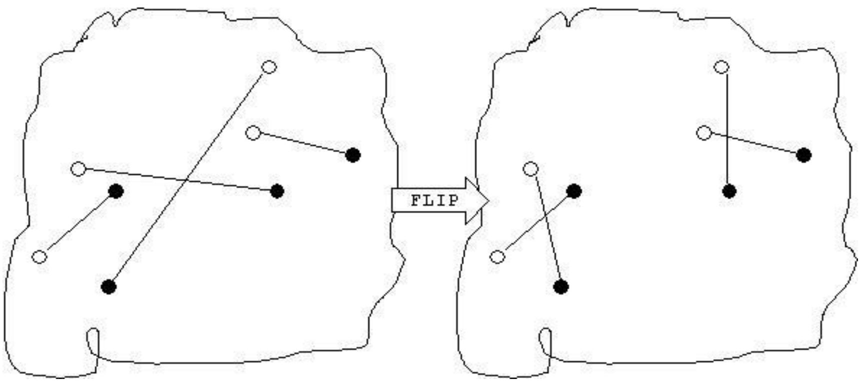

Figure 2. FLIP can increase the number of intersections

The above observation that FLIP always reduces value of the sum of the lengths of intervals implies the following conclusion: every coupling $X \in$ *argmin* (λ X . $(\Sigma_{[Bi,Wj] \in X} |B_i,W_j|)$) is intersection-free. Hence Dijkstra problem can be solved as a special case of assignment problem [3]. Really, let us define a complete weighted bipartite graph which consists of n black and n white vertices, let weight of an edge $[B_i, W_j]$ be equal to Euclidian distance $|B_i,W_j|$; then any maximal matching with minimal weight for this graph is a coupling without intersections. It is known that the assignment problem can be solved in time $O(n^3)$, for example, by so-called Hungarian algorithm [3].

However, Dijkstra's solution as well as reduction to assignment problem rise a natural question: how to solve good coupling generation problem efficiently in a pure geometric way? Fortunatly, an answer is known since 1990: a very efficient geometric algorithm for Dijkstra problem has asymptotic complexity is *O(n log n)* [8].

Surprisingly, but all three listed methods for Dijkstra problem have been successfully implemented at ACM ICPC 2007–2008, Northeastern European Regional Contest, by participants for solving problem A (Ants, at http://neerc.ifmo.ru/past/2007/problems.pdf) which is exactly Dijkstra problem. All these solutions were competitive in spite of lack of known "good" upper bound for Dijkstra heuristic search.

Practical efficiency of Dijkstra's solution rises another question: what is time complexity of the heuristic search algorithm? An obvious asymptotic upper bound is $O(n!)$. Since we can not prove better bound, we have tested an efficiency of the heuristic search algorithm. A series of tests consisted of 9900 runs – 100 suites for every $n \in [2..100]$. In each test suite n black and n white randomly generated points (without collinear triples) were distributed uniformly in real square $[0..1] \times [0..1]$. In each test suite the following two numbers were counted:

- number of good couplings,
- number of iterations of FLIP.

Then for every $n \in [2..100]$ we have calculated arithmetic mean and found two empirical values:

- average number of good couplings;
- average number of iterations of FLIP.

The experiments have demonstrated that

- average fraction of good coupling is decreasing in geometric progression;
- average number of FLIP is growing up as quadratic function.

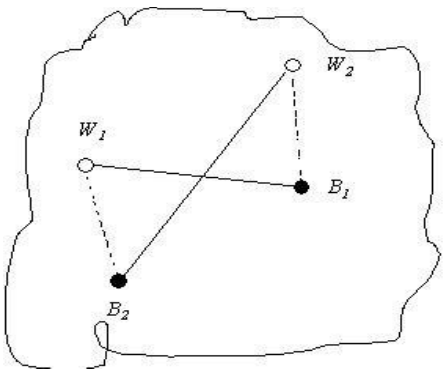

Figure 3. `FLIP` decreases the sum of lengths

The experimental study has lead us to two hypothesis that we would like to formulate as research problems:

- Prove that average rate of good couplings is decreasing in geometric progression.
- Prove that average time complexity of the heuristic search algorithm is quadratic.

But Dijkstra heuristic search has further research implications related to multiagent systems and algorithms. Multiagent system is a distributed system [16] that consists of agents. An agent is an autonomous rational reactive object (in OO-sense) with an internal state that can be characterized in terms of distinguishable sets of variables for agent's Believes (B), Desires (D), and Intentions (I). Variables from agent's Believes represent snapshots of its "knowledge" about itself and other agents (including an environment) that could be incomplete, inconsistent, and (even) incorrect[5]. Variables from agent's Desires represent its long-term aims, obligations and purposes (that also could be inconsistent). Variables from agent's Intensions are used for a short-term planning. Reactivity means that every agent can change its Believes (B), Desires (D), and Intentions (I) after communication with other agents(including environment), but every agent is autonomous, i.e. change of its Believes, Desires and Intensions depends on the agent and it is not decreed by other agents. Agents of the described kind are usually called BDI-agents [17]. Multiagent algorithm is a distributed algorithm [14] that solves some problem by means of cooperative work of agents in a multiagent system.

Dijkstra problem and FLIP operation in the heuristic search solution can easily be interpreted as a multiagent system and a result of an interaction of a pair of agents. This interpretation leads to the following problem (that is called Mars Robot Puzzle in [15]).

There are $n > 0$ robots and (the same) number n of shelters on a plane and shelters are assigned to robots (in 1-to-1 manner). Positions of all shelters are fixed and known to all robots, but non of robots does not know positions of the other robots. Robots can communicate with each other in pairs (in P2P manner) and we assume that communication is fair (i.e. if a robot would like to communicate with any other robot they will communicate eventually). At some moment all robots stop and have to select individual shelters to move at by a straight line. Definitely, robots should not collide. Hence every individual robot can move to its shelter only when it knows for sure that it will not collide with any other robot on the route. In every communication act of any two robots they can flip their current shelters iff their routes to these shelters intersect. However, all robots are autonomous and can not commend to each other. Problem: design a multiagent algorithm that guarantees that every robot will eventually get to some shelter and no robots collide.

Some preliminary results on multiagent algorithms for MRP has been published in [15].

[5] We do not assume that this "knowledge"' is "true belief"' (due to Plato) or knowledge in any formal sense [17, 14].

Sylvester problem

The next example of a problem[6] that can be solved by Computer Science approach is well-known problem of Sylvester (1814-1897) [10, 9]. The following story is citation from [9]: “Here is the question that Sylvester originally raised in 1893 in a journal known as the Educational Times: Prove that it is not possible to arrange any finite number of real points so that a right line through every two of them shall pass through a third, unless they all lie in the same right line. Sylvester used the term “right line” for straight line. Furthermore, two points which are the only two points on a line of a point/line configuration C have come to be known as ordinary lines. Today, we would state the result, in a way similar to the way Paul Erdös (1913-1996) did when he made the conjecture, about 40 years later: If a finite set of points in the plane are not all on one line then there is a line through exactly two of the points.”

Once again (as in the above) let us reformulate the problem in a computer science manner: Design an algorithm that inputs a set of points in general position[7] and outputs a straight line that contains exactly two points in this set.

Assume that input points are fixed. (In the below we will refer then as “input points”.) An algorithm should output a line that contains exactly two points. Thus, it makes sense to introduce two data types POINT and LINE whose values are all input points and all line through any two of input points. What are operations that can use arguments of these data types or return values of these data types? The following two operations are absolutely natural for the problem:

- ORDININARY: LINE $\rightarrow$ BOOLEAN,
- DRAW_LINE: POINT×POINT $\rightarrow$ LINE.

ORDINARY returns TRUE iff its argument contains exactly two input points, whereas DRAW_LINE returns the line that is defined by its arguments (i.e. two input points). It is also natural to assume that two of input points are available as the following constants POINT_A and POINT_B.

Of course, given two points POINT_A, POINT_B, a line DRAW_LINE(POINT_A, POINT_B) can solve Sylvester problem by chance. However in general case we need more operations for our data types POINT and LINE.

What other operations can be useful? Have we used everything that is provided by problem statement? – No!

First, we haven’t used yet the fact that points altogether are non-collinear. We can use this information, for example, in the following manner: let

- OUT_POINT: LINE $\rightarrow$ POINT

be operation that returns an input point that is outside the argument (line).

Second, we have not used yet that every non-ordinary line contains three points at least. We can utilize this information, for example, in the following manner: let

- THIRD_POINT: LINE×POINT×POINT $\rightarrow$ POINT

be operation that returns an input point that belongs to the first argument (line) but is disjoint with both other arguments (points).

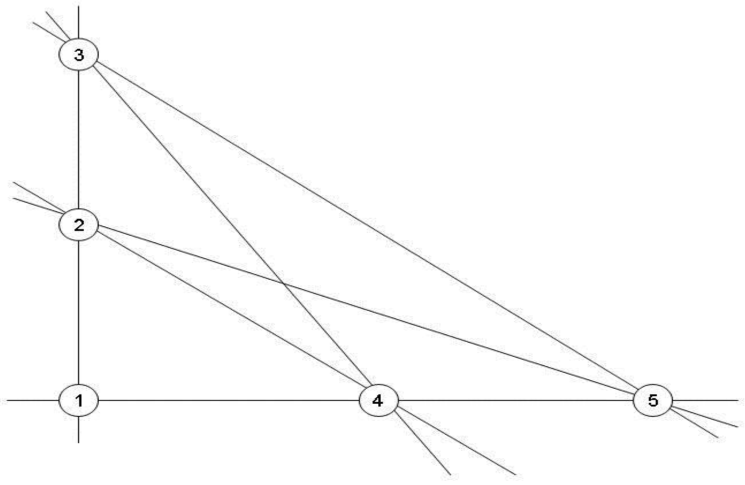

Figure 4. A sample input points and their lines

For example, for the positioning map depicted in fig.4 we can have the following:

- ORDINARY(2,3) = FALSE but ORDINARY(2,5) = TRUE;
- DRAW_LINE(1,2) = DRAW_LINE(1,3) = DRAW_LINE(2,3) is the line that comes through points 1, 2, and 3;
- POINT_A and POINT_B can be any disjoint input points, for example 1 and 2;
- OUT_POINT(DRAW_LINE(1,2)) = OUTSIDE_POINT(DRAW_LINE(1,3)) and it can any of points 4 or 5;
- THIRD_POINT(DRAW_LINE(1,2), 1, 2) = 3.

6 We have discussed this problem in another our preliminary paper [13].
7 i.e. they are not collinear simultaneously

The following preliminary version of a heuristic algorithm for the Sylvester problem implements an obvious idea: to iterate while the current line is non-ordinary drawing a new line using a point in the line and a point outside the line. It uses non-formalized operation "right partner" for it.

```
VAR L: LINE; VAR U,W,X,Y: POINT;
BEGIN
        X:= POINT_A; Y:= POINT_B;
        L:= DRAW_LINE(X, Y);
                WHILE !(Ordinary(L)) DO
                        U:= OUT_POINT(L);
                        W:= "right partner for U in {X, Y, THIRD_POINT(L, X, Y)}";
                        X:= U; Y:=W; L:= DRAW_LINE(X, Y)
                OD
END
```

Observe that if some computation of our preliminary algorithm terminates, then the final value of variable L is an ordinary line that contains exactly two input points (due to loop condition). Hence we should take care of algorithm termination only. The above description of method of R. Floyd again gives us a clue how to formalize "right partner" in the assignment W:= "right partner for U in {X, Y, THIRD_POINT(L, X, Y)}". – The right choice must decrease some value...

Observe also that WHILE-loop is executed iff the current line is not an ordinary one, i.e. it contains 3 input points at least. Let these input points be *1*, *2*, and *3* in fig.5. There are four opportunities for relative positions of an outside point *0* and points *1*, *2*, and 3. Observe again, that if *C* is a point in *{1,3}* such that point *2* lies in between *C* and the base of the perpendicular from outside point 0 to the line, then (fig.6) $d(2,line(0,C)) < d(0,line(1,3))$, where *d* is distance between a point and a line.

Hence, it makes sense to formalize "right partner" by a new operation

- BEST: POINT× POINT× POINT× POINT → POINT

that is defined if the last three arguments (three points *1*, *2*, and *3*) are colinear, the first argument (point *0*) is not colinear with them, and that returns the utmost point in *1*, *2*, and *3* that is separated from the base of the perpendicular from *0* by another point in *1, 2,* and *3*. For example, in fig.6, BEST(*0,1,2,3*) = *3* in the first row and BEST(*0,1,2,3*) =*1* in the second row.

Observe also, that we must use point *2* as the next outside point for new line instead of any other. Let the corresponding operation be

- NEXT_OUT: POINT× POINT× POINT× POINT → POINT

The final design of a heuristic algorithm for Sylvester problem is represented below.

```
VAR L: LINE; VAR U,W,X,Y,Z: POINT;
BEGIN
        X:= POINT_A; Y:= POINT_B;
        L:= DRAW_LINE(X, Y);
        Z:= OUTSIDE_POINT(L);
        WHILE !(Ordinary(L)) DO
                U:= Z;
                W:= BEST(U,X,Y, THIRD_POINT(L, X, Y));
                X:= U; Y:=W;
                Z:= NEXT_OUT(U,X,Y, THIRD_POINT(L, X, Y));
                L:= DRAW_LINE(X, Y)
        OD
END
```

Although the final design is not a formal refinement of the preliminary design, we hope our informal development from the preliminary design to the final design is convincing. The last thing we have to do in this section is to prove that the resulting algorithm solves the Sylvester problem. That is, if the input points are not simultaneously colinear, then the algorithm always terminates with a value of L that is a line that contains exactly two input points.

Really, it is sufficient to apply Floyd method for proving algorithm termination. Let *D* be *{ d(z,line(x,y)) : x, y and z are input points, line(x,y) is a line that go through x and y, and* $z \notin line(x,y)\}$ where *d* is (as above) distance between point and line. Observe that *D* is a finite (since set of input points is finite) subset of $\boldsymbol{R}^+ = \{ r \in \boldsymbol{R} : r \geq 0\}$. Set *D* provided with natural order for real numbers ≤ is a WFS. Let *f* : POINT×LINE → *D* be the following mapping λ L,Z. *d*(z,L). Then after every iteration of the WHILE-loop, *f*(X,L) is always smaller than the previous iteration's *f*(X,L) as follows from the definition of operations BEST and NEXT-OUT and is illustrated in fig.6. Hence in accordance with method of R. Floyd, the algorithm always terminates. A final value of the variable L must be a line that connects exactly two input points, since the condition of WHILE-loop is !ORDINARY(L). It also proves Sylvester problem.

Finally we would like to discuss utility of the above algorithm for Sylvester problem. Unfortunately, complexity is not a strong point of the algorithm. It is explicit that the upper bound for the number of iterations of the WHILE-loop is $n \times (n-1)/2$. A straightforward complexity for THIRD_POINT is $O(n)$. It implies an overall asymptotic algorithm complexity $O(n^3)$ that is not better than an exhaustive search algorithm. Nevertheless our algorithm drives proof of Sylvester problem, while exhaustive search proves nothing. This is a strong point of the above algorithm.

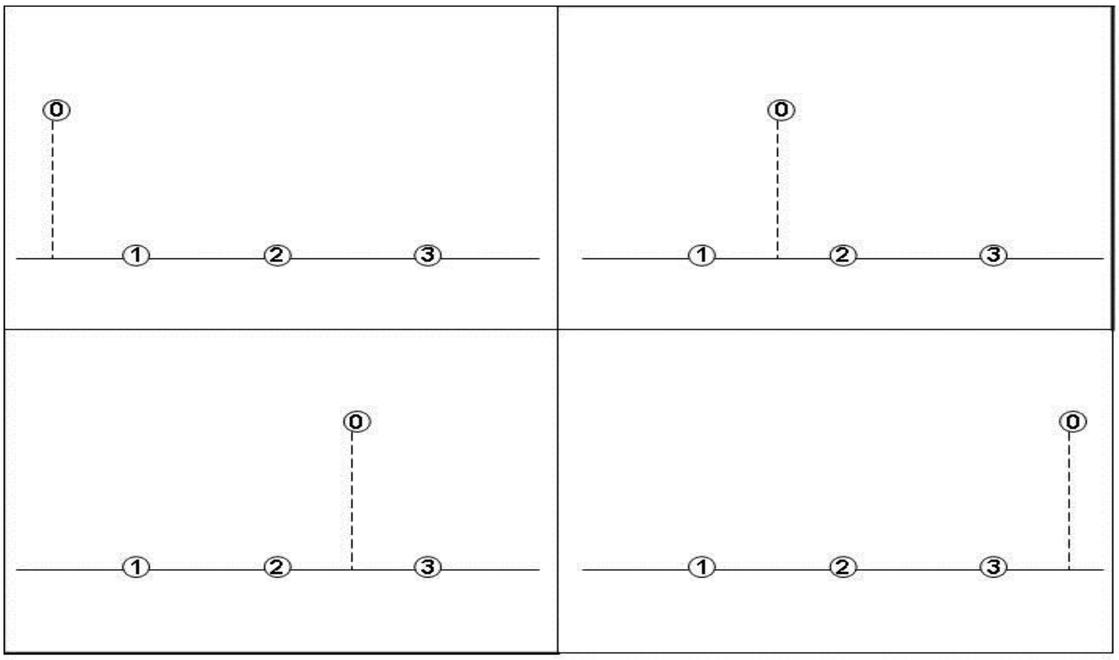

Figure 5. Relative positions between input points and a non-ordinary line

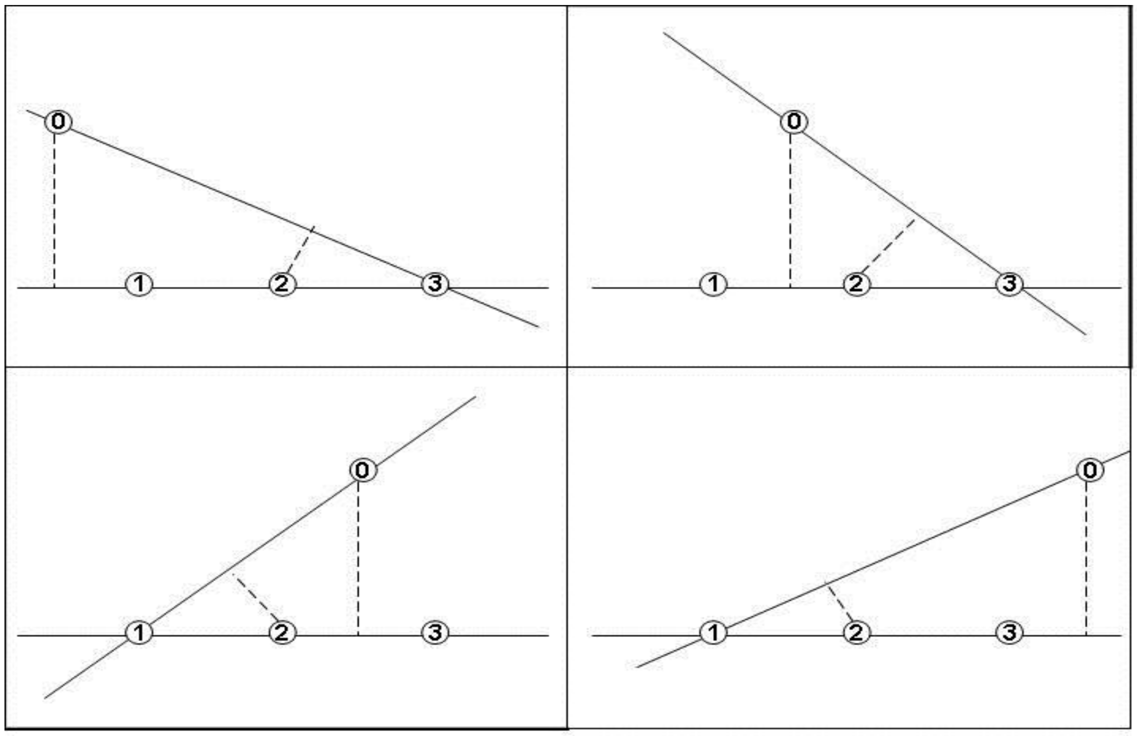

Figure 6. How to select next line and next outside point

Conclusion

We have seen two examples in which Dijkstra’s approach helps to solve some mathematical (geometrical) problems. According to Dijkstra's approach, we solve mathematical problems by encoding them in some algorithms, and by verifying (i.e. proving the correctness and termination) of these algorithms in style of A. Hoare and R. Floyd. This formal program verification style is a mathematically rigorous proof of program correctness [5, 1]. Further research and development formal program verification is one of the most important problems in the science of programming, a “Grand Challenge of Computer Science” [6].

In the two examples we have presented, these algorithms are “engineered” with only purely computer scientific concepts, i.e. abstract data types (not geometric continuous space) and “computable” operations on the data types, and are designed so that their outputs solve the geometric problems. In other words, Dijkstra's approach witnesses an example that we can solve mathematical problems only with computer scientific concepts. Because Dijkstra's approach suggests a methodology of mathematical problem-solving based on the most elementary and seemingly less directly mathematics-related ideas of computer science, we believe that the approach might be the most beautiful case among the influences of computer science on mathematics, including from the classical example of the four-color problem to the examples of theoretical computer science which progresses tightly-coupled with mathematics such as non-classical logics, proof theory and category theory. (At the same time we acknowledge that Computer Science owes theoretical concepts to Mathematics and it comes from Mathematics historically.) Computer science is a branch of science with its own objects of studies and methods of research, and like other sciences such as physics or economics it has progressed exchanging influences with mathematics. Dijkstra's approach shows that such influences on mathematics may arise casually in the most elementary parts of computer science, not only in the special areas of theoretical computer science with extremely biased emphasis on theories.

Finally let us encourage readers to try Dijkstra’s approach themselves. In particular try to solve in this way the following geometry problem. A finite set L of straight lines on the plane is said to be in general position if no two lines in L are parallel, and no three lines in L go through a common point. A piecewise straight curve is said to be simple if it does not intersect itself. If we are given a piecewise straight curve P and draw a straight line through each segment of P, we obtain a finite set of lines A(P), which we call the induced arrangement of P. Prove that for any finite set of lines L in general position, there is always a simple piecewise straight curve P that L = A(P).

Acknowledgements: We would like to thanks Eugene Bodin, Lidiya Gorodnjay, Seunghwan O, and Kwangeun Yi for many suggestions and discussions. It was very hard for us to formulate our ideas without their assistance and moral support.

1. *K.R. Apt, F.S. de Boer and E.R. Olderog.* Verification of Sequential and Concurrent Programs, 3rd ed., Springer-Verlag, 2009.
2. *E.V. Bodin, L.V. Gorodnjay, N.V. Shilov*. What is the subject of the computer science contest? Preprint #126, A.P. Ershov Institute of Informatics Systems, Novosibirsk, 2005 (in Russian).
3. *R. Burkard, M. Dell'Amico, S. Martello*. Assignment Problems. SIAM, 2009.
4. *E.W. Dijkstra*. On a cultural gap. The Mathematical Intelligencer, 8(1), 1986. – P. 48–52.
5. *R.W. Floyd.* Assigning Meanings to Programs, in Mathematical Aspects of Computer Science (Proc. of Symp. in Applied Mathematics), 1967, vol. 19. – P. 19–32.
6. *C.A.R. Hoare*. The Verifying Compiler: A Grand Challenge for Computing Research, in Lecture Notes in Computer Science (Proc. of Conf. “Perspectives of System Informatics” (PSI 2003)), 2003, vol. 2890. – P. 103–111.
7. *O. Hazzan*. Application of computer science ideas to the presentation of mathematical theorems and proofs. ACM SIGCSE Bull., 35(2), 2003. – P. 38–42.
8. *J. Hershberger and S. Suri*. Applications of a semi-dynamic convex hull algorithm. Proc. of the 2nd Scandinavian Workshop on Algorithm Theory SWAT’90, Lecture Notes in computer Science, v. 447, Springer-Verlag, 1990. – P. 380–392.
9. *J. Malkevitch.* A Discrete Geometrical Gem. AMS Feature Column Monthly Essays on Mathematical Topic. July-August, 2003, http://www.ams.org/featurecolumn/archive/index.html.
10. *J. Pach, P. Agarwal*. Combinatorial Geometry. Wiley-Interscience 1995.
11. *N. Shilov and K. Yi.* Engaging Students with Theory through ACM Collegiate Programming Contests. Communications of ACM. – v. 45, N 9. – 2002.
12. *N.V. Shilov, S.O. Shilova.* Etude on theme of Dijkstra. ACM SIGACT News, 35(3), 2004. – P. 102–108.
13. *N.V. Shilov, S.O. Shilova*. On Mathematical Contents of Computer Science Contests. Proceedings of the 1st KAIST International Symposium on Enhancing University Mathematics Teaching, 12–16 May, Daejeon, Korea. – 2005. – P. 223–233.
14. *N.V. Shilov, N.O. Garanina.* Modal Logics for reasoning about Multiagent Systems. In Encyclopedia of Artificial Intelligence. J.R. Rabucal, J. Dorado, A.P. Sierra, editors. Information Science Reference. 2008. – P. 1089–1094.
15. *N.O. Garanina, N.V. Shilov and L.E. Konyaev*. Can Robots Solve the Assignment Problem? Proceedings of the 2009 Workshop on Concurrency, Specification, and Programming. Kraków–Przegorzały, Poland, 28–30 September 2009, v. 1. – 2009. – P. 154–163.
16. G. Tel *Introduction to Distributed Algorithms.* Cambridge University Press, 2nd Edition, 2000.
17. *M. Wooldridge*. An Introduction to Multiagent Systems. John Willey & Sons Ltd, 2002.