

# GEOMBINATORICS, COMPUTATIONAL COMPEXITY AND SAVING ENVIRONMENT: LET'S START

Olga Kosheleva, Vladik Kreinovich

*Computer Science Department, University of Texas at El Paso, El Paso, TX 79968*

**Abstract.** We show that a certain recycling problem can be expressed in terms of equidecomposability of polygons (in 3-D case, polyhedra). In 2-D case, this problem was solved more than 150 years ago by F. Bolyai, and the known solution is actually algorithmic. How fast is this algorithm? If this particular algorithm is too slow to be of practical use, can we find another one that is faster? In this paper, we show that in some cases, all possible algorithms require exponential computation time and are thus not feasible.

From geometrical viewpoint, our result is very simple. We view it not as a ready-to-use result, but as a start of the new class of problems: environmentally motivated computational problems in geombinatorics.

## 1. FORMULATION OF THE GEOMETRICAL RECYCLING PROBLEM

**The description of a manufacturing situation.** In many manufacturing situations, we encounter the following problem. Initially, we thought that a piece of plastic (or metal, or whatever) of a certain shape will be useful. So, we made one or several pieces of that shape. Later on, it turned out that we initially erred, and that we need plastic that is shaped differently.

**The necessity and possibility of recycling.** In principle, we could simply throw the initially shaped piece away, but financial and environmental concerns both prompt us to try to reuse the original piece by reshaping it. One possibility of reusing it is to melt the original piece, and then mold it again into the new shape. This is potentially possible if both the old and the new shape have the same volume (and hence, the same mass). This way, however, we waste energy on melting. It would be thus better if we could simply cut the original object into pieces, and then glue them together into a new shape (if it is plastic, then instead of glueing, we can warm the bordering surfaces so that they melt and stick together).

**How to describe it geometrically?** From the geometrical viewpoint, “cutting” means cutting by a plane. In many cases, both the original and the new shapes could in principle be obtained by cutting only. This means that the borders of these shapes are formed by planes, and thus, these shapes are polyhedra (or, in 2-D case, polygons). When we cut the original shape, the resulting small pieces are also polyhedra. So, for these shapes, our recycling problem can be reformulated in geometrical terms as follows:

We have two polygons (or polyhedra)  $P, Q$  of equal area (volume). It is necessary to divide  $P$  into finitely many polygonal (polyhedral) pieces  $P_1, \dots, P_m$ , so that after applying motions  $M_1, \dots, M_m$ , we will get new polygons (polyhedra)  $Q_i = M_i(P_i)$  that together form  $Q$ .

**This is a well known problem in geombinatorics.** If such pieces exist for  $P$  and  $Q$ , then  $P$  and  $Q$  are called *equidecomposable*. The first general result in equidecomposability was obtained by F. Bolyai and P. Gervin who proved that arbitrary two polygons

of equal area are equidecomposable (see, e.g., [Boltianskii 1978, Chapter 2]). In 3-D case, the existence of non-equidecomposable polyhedra of equal volume (actually, a cube and a tetrahedron) was conjectured by D. Hilbert as his 3rd problem [Hilbert 1902], and solved as early as 1900 [Dehn 1900] (for recent developments, see, e.g., [Boltianskii 1978] and [Sah 1979]).

**We are interested not only in finding out whether  $P$  and  $Q$  are equidecomposable, but also in actually designing the pieces  $P_1, \dots, P_m$ .** So, we are interested in finding an *algorithm* that would generate  $P_i$ .

Such algorithms are already known. The 2-D construction of Bolyai and Gerwin is actually algorithmic. Some algorithmic problems related to the 3-D case are solved in [Kosheleva 1980], [Kosheleva et al 1989], and [Kosheleva et al 1994].

**Not all algorithms are feasible.** In theory of computations, it is well known that not all algorithms are feasible (see, e.g., [Garey et al 1979], [Lewis et al 1981], [Martin 1991]): it depends on how many computational steps they need. If for input of length  $n$ , the algorithm requires computational time  $2^n$ , then for the input of a reasonable length  $n \approx 300$ , we would need more computational steps than there can be during the lifetime of our Universe. So, such algorithms (called *exponential-time*) are usually considered not feasible.

The fact that an algorithm is not feasible, does not mean that it can never be applied: it simply means that there are cases when its running time will be too large for this algorithm to be practical.

**What we are planning to do.** In this paper, we prove that even for a 2-D case, every algorithm that finds the polygons  $P_i$  is (in the worst case) exponential-time.

Therefore, there exist cases, when no feasible algorithm will find the way to cut and paste  $P$  into  $Q$ . In such cases, we cannot cut and paste, so we will have to melt.

## **2. DEFINITION AND THE MAIN RESULT: IN 2-D CASE, EVERY ALGORITHM THAT ENABLES US TO TRANSFORM A POLYGON INTO ANOTHER ONE REQUIRES (IN THE WORST CASE) EXPONENTIAL TIME**

**Definition 1.** By a (*computer representation of a*) *point*, we mean a pair of numbers  $(x, y)$ . Points will be denoted by capital letters  $(A, B, C, \text{etc})$ .

*Comment.* When we describe computer representations, we have to assume that a coordinate system is given, so a point is described by its coordinates. A number here can mean an integer, a rational number, or a real number. An integer is represented by us humans as its decimal representation, and inside the computer an integer is represented by a binary representation.

**Definition 2.** By a (*computer representation of a*) *polygon*, we mean a sequence of points  $A_1, \dots, A_p$ . These points will be called its *vertices*.

*Comment.* This polygon is obtained by connecting the vertices  $A_i$  consequently, i.e.,  $A_1$  with  $A_2$ ,  $A_2$  with  $A_3$ , ..., and finally,  $A_p$  with  $A_1$ .

**Definition 3.** A *motion* is an arbitrary composition of rotations and shifts. In Cartesian coordinates, a motion is a linear transformation  $x \rightarrow a_{11}x + a_{12}y + a_1$ ,  $y \rightarrow a_{21}x + a_{22}y + a_2$ . By a (*computer representation of a*) *motion*, we mean a list of 6 real numbers  $a_{ij}$ ,  $a_i$ .

**Definition 4.** We say that an algorithm  $U$  *enables to transform a polygon into another one*, if it takes two polygons  $P$ ,  $Q$  as inputs, and in case they have the same area, returns as an output the two lists: a list of polygons  $P_1, \dots, P_m$ , and a list of motions  $M_1, \dots, M_n$  so that different  $P_i$  have no common interior points,  $P = \cup P_i$ , and  $Q = \cup M_i(P_i)$ .

*Comment.* We are going to prove the negative result: that every algorithm that transforms a polygon into another one takes (in the worst case) exponential time. Since we are talking about the worst case, it is sufficient to prove this result for a specific set of polygons. Namely, we will prove it for polygons whose vertex coordinates are all integers. Let's give a formal definition.

**Definition 5.** We say that a polynomial has *integer-valued vertices* if for each vertex  $A_i$ , both of its coordinates are integers.

**Comment.** Now, we are ready to define computational complexity. Crudely speaking, a computational complexity  $t(n)$  of an algorithm is the maximal amount of time that this algorithm spends on inputs of length  $n$ . So, to define computational complexity, we must define the time that the algorithm spends on a given input, and the length of the input.

The *time* is usually defined as follows: every algorithm consists of several elementary computational steps. Even when we describe an algorithm as a program in some high-level programming language, with the possibility of using just one line (e.g., calling a subroutine) to describe complicated computations, the computer will translate this line (actually, a compiler will do that) into a sequence of hardware-supported elementary operations (like addition, multiplication, printing one number, etc). In general, the time of different operations is slightly different, but these times are usually of the same order. So, as a good approximation to the actual computation time, we can take the total number of elementary operations multiplied by the average time of an operation.

The average time of an elementary operation depends on what exactly brand of computer we are using. So, to compare algorithm irrespective on what processors they run on, people usually use just the total number of elementary operations. So, we arrive at the following definition:

**Definition 6.** By a *running time*  $t_U(x)$  of an algorithm  $U$  on input data  $x$ , we mean the total number of elementary computational steps that  $U$  goes through when applied to  $x$ .

*Comment.* This definition depends on what exactly we mean by an elementary step ([Garey et al 1979], [Lewis et al 1981], [Martin 1991]). There are several possible definitions, because what is hardware supported as a single operation in one computer may be implemented as a sequence of operations in another computer. The difference is often practically important,

but, as we will see from our proof, we will get the same negative result no matter which of the known definitions are used: the only condition is that an output of a number is considered as at least one computational step.

Now, let us describe the length of the input.

**Definition 7.** For each polygon  $P$  with integer-valued vertices, the computer representation consists of a list of pairs of coordinates  $(x_i, y_i)$  of its vertices, represented as their binary codes. Adding up the lengths of these codes, we get the *length* of an input.

**Definition 8.** By a (*worst-case*) *computational complexity*  $t_U^w(n)$  of an algorithm  $U$ , we mean a function that maps each integer  $n$  into the maximum of all the values  $t_U(x)$  for all inputs  $x$  of length  $n$ .

**MAIN THEOREM.** *If an algorithm  $U$  transforms one polygon with integer-valued vertices into another, then for  $n$  that are divisible by 8, its computational complexity is  $> 2^{n/8}$ .*

*Comment.* In other words, the computational complexity is growing exponentially, and thus, these algorithms are not always feasible.

**Proof.**

1. Let's fix an integer  $N$ , and consider the following two polygons with integer-valued vertices: a rectangle  $P(N)$  with vertices  $(0, 0)$ ,  $(1, 0)$ ,  $(0, N^2)$ , and  $(1, N^2)$ , and a square  $Q(N)$  with vertices  $(0, 0)$ ,  $(N, 0)$ ,  $(0, N)$  and  $(N, N)$ . These two polygons have equal area  $N^2$ , so they are equidecomposable. In other words, there exists polygons  $P_1, \dots, P_m$  that have no common interior points, and motions  $M_i$  such that  $P(N) = \cup P_i$  and  $Q(N) = \cup M_i(P_i)$ .

2. Let us prove that for every decomposition with this property,  $m \geq N/\sqrt{2}$ .

Indeed, since  $M_i(P_i)$  is a subset of a square  $Q(N)$ , the diameter  $diam(M_i(P_i))$  cannot exceed the diameter of  $Q(N)$  (that is equal to  $\sqrt{2}N$ ). The diameter does not change with motion, so  $diam(P_i) = diam(M_i(P_i)) \leq \sqrt{2}N$ . The length of the projection  $\pi(P_i)$  of  $P_i$  on the long side of  $P(N)$  cannot exceed the diameter of  $P_i$  and thus, cannot exceed  $\sqrt{2} \cdot N$ .

Since  $P(N) = \cup P_i$ , every point on the long side  $\pi(P(N))$  of  $P(N)$  belongs to at least one polygon  $P_i$ . Therefore,  $\pi(P(N)) = \cup \pi(P_i)$ . So, the total length  $N^2$  of this long side is less than or equal to the sum of the lengths of the projections  $\pi(P_i)$ . Since the length of each projection does not exceed  $\sqrt{2} \cdot N$ , the sum of these lengths does not exceed  $\sqrt{2} \cdot Nm$ . So,  $N^2 \leq \sqrt{2} \cdot Nm$ , hence,  $m \geq N/\sqrt{2}$ .

3. Generating each polygon means generating the coordinates of its vertices. Each polygon has at least 3 vertices, and each vertex has two coordinates. So, for each polygon, we generate at least 6 numbers. Totally, we generate at least  $6m$  numbers.

We assumed that generating one number requires at least 1 computational step. So, our algorithm requires at least  $6m \geq 6N/\sqrt{2}$  computational steps:  $t_U(x) \geq 6N/\sqrt{2}$ .

4. Let's now prove that our algorithm is exponential-time.

To prove that, let's take  $N = 2^k$ . In the binary code,  $N$  is 1 and  $k$  0's. So, the length of  $N$  is  $k + 1$ . Similarly, the length of  $N^2$  is  $2k + 1$ . For 0 and 1, the lengths are just 1 bit. By adding the corresponding numbers, one can easily check that the length  $n$  of the computer representation of the polynomials  $P(N)$  and  $Q(N)$  is equal to  $8k + 16$ .

In terms of  $n$ , we have  $k = (n - 16)/8 = n/8 - 2$ . Therefore,  $N = 2^k = 2^{n/8}/4$ , and  $t_U(x) \geq 6 \cdot 2^{n/8}/(4\sqrt{2}) > 2^{n/8}$  for some input  $x$  of length  $n$ . By definition of  $t_U^w(n)$ , this means that  $t_U^w(n) > 2^{n/8}$ . Q.E.D.

*Comments.*

1. From the practical viewpoint, generating the vertices of the partitioning polygons may not be the right thing to do. Another possibility is to describe the *locations* of the cuts that we need to produce these  $P_i$ , and the *motions* of each of the resulting parts. If we look for such algorithms, then for each of these pieces  $P_i$ , we still need to generate several parameters (that describe its motion), so the total number of computational steps will still be greater than  $C_1 m \geq C_2 2^{n/8}$ , i.e., still exponential-time.

2. Our formalization of the original recycling problem may be too idealized. In reality, if we lose a little bit when cutting, we can add glue to cover the resulting small gap. If we denote by  $\varepsilon$ , the distance that can be safely covered by glue only, then we end up with the following modified definitions.

Before we introduce them, we must explain how we formalize the restriction on the glue layer. The fact that the thickness of the glue does not exceed  $\varepsilon$ , means that when a set  $Q$  is composed out of the sets  $Q_i = M_i(P_i)$ , there can be points in  $Q$  that are not covered by any of the  $Q_i$ . However, each of these "glue" points must be no farther than  $\varepsilon$  from some "solid" point (i.e., a point from the union  $\cup Q_i$ ). This condition can be easily expressed if we use the well-known notion of a Hausdorff distance between the sets. For the reader's convenience, let's reproduce this definition:

**Definition 9.** Assume that  $A$  and  $B$  are bounded sets in a metric space  $X$  with a distance  $\rho$ . We say that the Hausdorff distance between  $A$  and  $B$  is  $\leq \varepsilon$  if the following two conditions hold:

- for every  $a \in A$ , there exists a  $b \in B$  that is  $\varepsilon$ -close to  $a$  (i.e., such that  $\rho(a, b) \leq \varepsilon$ );
- for every  $b \in B$ , there exists an  $a \in A$  that is  $\varepsilon$ -close to  $b$  (i.e., such that  $\rho(a, b) \leq \varepsilon$ ).

**Definition 4'.** Assume that a real number  $\varepsilon > 0$  is given. We say that an algorithm  $U$  enables to transform a polygon into another one with accuracy  $\varepsilon$ , if it takes two polygons  $P, Q$  as inputs, and in case they have the same area, returns as an output the two lists: a list of polygons  $P_1, \dots, P_m$ , and a list of motions  $M_1, \dots, M_n$  such that:

- different  $P_i$  have no common interior points;
- $P \supseteq \cup P_i$ , and the Hausdorff distance between  $P$  and  $\cup P_i$  does not exceed  $\varepsilon$ ;
- $Q \supseteq \cup M_i(P_i)$ , and the Hausdorff distance between  $Q$  and  $\cup M_i(P_i)$  does not exceed  $\varepsilon$ .

**THEOREM 2.** Assume that a real number  $\varepsilon > 0$  is given. If an algorithm  $U$  transforms one polygon with integer-valued vertices into another with accuracy  $\varepsilon$ , then for sufficiently large  $n$  that are divisible by 8, the computational complexity  $t_U^w(n)$  is  $> 2^{n/8}$ .

*Comment.* In other words, even if we allow small differences to be covered by glue only, the computational complexity is still growing exponentially, and thus, the algorithms are still not always feasible.

**The idea of the proof** is essentially the same as in the Main Theorem. The projection  $\pi(P_i)$  of  $P_i$  on the long side of  $P(N)$  is still  $\leq \sqrt{2}N$ . Since the distance between  $P$  and  $\cup P_i$  does not exceed  $\varepsilon$ , every point from  $\pi(P(N))$  either belongs to the one of the projections  $\pi(P_i)$ , or is at the distance  $\leq \varepsilon$  from one of these projections. In the second case, this point belongs to the  $\varepsilon$ -neighborhood of  $\pi(P_i)$ .

Therefore,  $\pi(P(N))$  is equal to the union of  $\varepsilon$ -neighborhoods  $\tilde{P}_i$  of the intervals  $\pi(P_i)$ . Hence, the length  $N^2$  of  $\pi(P(N))$  does not exceed the sum of the lengths  $\tilde{l}_i$  of the sets  $\tilde{P}_i$ .

An  $\varepsilon$ -neighborhood of an interval is obtained by adding intervals of length  $\varepsilon$  to both of its ends. Therefore, the length of an  $\varepsilon$ -neighborhood is equal to the length of the original interval plus  $2\varepsilon$ . Since the length of  $\pi(P_i)$  is  $\leq \sqrt{2}N$ , we can conclude that  $\tilde{l}_i \leq \sqrt{2} \cdot N + 2\varepsilon$ . So, from  $N^2 \leq \tilde{l}_1 + \dots + \tilde{l}_m$ , we can conclude that  $N^2 \leq m(\sqrt{2} \cdot N + 2\varepsilon)$ , and hence, that  $m \geq N/(\sqrt{2} + 2\varepsilon N^{-1})$ .

From this inequality, similarly to the Main Theorem, we can conclude that  $t_U^w(n) \geq 6 \cdot 2^{n/8}/(4(\sqrt{2} + 2\varepsilon N^{-1}))$ . When  $n \rightarrow \infty$ , then  $N \rightarrow \infty$ , and hence  $6/(4(\sqrt{2} + 2\varepsilon N^{-1})) \rightarrow 6/(4\sqrt{2}) > 1$ . Therefore, for sufficiently large  $n$ , we have  $6/(4(\sqrt{2} + 2\varepsilon N^{-1})) > 1$  and hence  $t_U^w(n) > 2^{n/8}$ . Q.E.D.

### 3. COMMENTS AND OPEN PROBLEMS

Our main comment is as follows: from geometrical viewpoint, the above result is very simple. We decided to present it not because it is technically complicated (it is not), but because we want to attract the attention of researchers to the new potential area of applications of geombinatorics: geometrical recycling problems.

What are the open problems here? First of all, it would be nice to find feasible algorithms (or prove their non-existence) for different 3-D cases. Even in a 2-D case, we can think of an open problem that has clear application possibilities. This problem is related to the above-described one. Namely, if we do not need shape  $P$  any more, but need shape  $Q$  instead, then not only it would be nice to reshape the existing pieces of shape  $P$  into pieces of shape  $Q$ , but it would be nice (for both financial and environmental reasons) if instead of creating a new *mold* for making new pieces of shape  $Q$ , we would be able to cut and paste the mold that was used to shape  $P$  into a mold for  $Q$ .

In geometrical terms, we would like to find such (mutually disjoint) polygons (polyhedra)  $P_i$  and motions  $M_i$  that the union of  $P$  and all the  $P_i$  is congruent to the union of  $Q$  and all the  $Q_i = M_i(P_i)$ . In other words, we would like to find polygons that make  $P$  and  $Q$  *equicomplementable*. For a 2-D case, the existence of such  $P_i$  and the fact that they can be obtained by using an algorithm are well known: the standard existence proof (see, e.g., [Boltianskii 1978]) is already algorithmic. The open problems are:

- Is this algorithm exponential-time?
- If yes, then are there other algorithms that do not take exponential time?

**Acknowledgements.** This work was partially supported by an NSF grant No. CDA-9015006.

## REFERENCES

Boltianskii, Vladimir G. *Hilbert's Third Problem*, V. H. Winston & Sons, Washington, D.C., 1978.

Dehn, M. *Ueber raumgleiche Polyeder*, Nachr. Acad. Wiss. Gottingen Math.-Phys. Kl., 1900, pp. 345–354.

Garey, M. R.; Johnson, D. S. *Computers and intractability: a guide to the theory of NP-completeness*, W. F. Freeman, San Francisco, 1979.

Hilbert, David. *Mathematical Problems, lecture delivered before the International Congress of Mathematics in Paris in 1900*, translated in Bull. Amer. Math. Soc., 1902, Vol. 8, pp. 437–479.

Kosheleva, Olga M. *Axiomatization of volume in elementary geometry*, Siberian Mathematical Journal, 1980, Vol. 21, No. 1, pp. 78–85.

Kosheleva, Olga M.; Kreinovich, Vladik. *Algorithmic problems of combinatorial geometry*. Center for New Informational Technology “Informatika”, Technical Report, Leningrad, 1989 (in Russian).

Kosheleva, Olga M.; Kreinovich, Vladik, *An application of logic to combinatorial geometry: how many tetrahedra are equidecomposable with a cube?*, Quarterly of Mathematical Logic, 1994 (to appear).

Lewis, L. R.; Papadimitrou, C. H. *Elements of the theory of computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Martin, J. C. *Introduction to languages and the theory of computation*, McGraw-Hill, N.Y., 1991.

Sah, C.-H. *Hilbert's third problem: scissors congruence*, Pitman, London, 1979.