Why the Graph Isomorphism Problem Is Easier Than Propositional Satisfiability: A Possible Qualitative Explanation

Vladik Kreinovich and Olga Kosheleva
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
vladik@utep.edu, olgak@utep.edu

Abstract

A recent result has shown that the graph isomorphism problem can be solved in quasi-polynomial time, while the general belief is that only exponential time algorithms are possible for propositional satisfiability. This is somewhat counter-intuitive, since for propositional satisfiability, we need to look for one of 2^n options, while in graph isomorphism, we need to look for one of $n!$ options, and $n! \gg 2^n$. Our qualitative explanation for this counter-intuitive fact comes from the fact that, in general, a graph isomorphism problem has a unique solution – in contrast to propositional satisfiability which, in general, has many solutions – and it is known that problems with unique solutions are often easier to solve.

1 Complexity of Different NP-Problems and a Recent Result on Graph Isomorphism

To explain the result about graph isomorphisms, it is necessary to recall the corresponding definitions: what is a feasible algorithm, what is an NP-problem, etc. For detailed description of these notions, see, e.g., [7].

Feasible algorithms: reminder. Some theoretical algorithms require so much computation time that they are not practically useful. For example, if on inputs of bit size n, the algorithm requires exponentially many 2^n steps, then even for a moderate-size input, with $n = 1000$, this algorithm needs time which is larger than the lifetime of the Universe. It is therefore desirable to find out which algorithms are *feasible*.

Usually:

- algorithms that require time which is bounded by a polynomial $P(n)$ of the size of the input (e.g., n^2 or n^3 time) are practical, while
- algorithms whose running time increases faster than a polynomial (e.g., as 2^n) are not practically feasible.

Because of this fact, a *feasible algorithm* is usually defined as an algorithm A whose running time $t_A(x)$ on any input does not exceed some polynomial $P(n)$ of the length $\text{len}(x)$ of this input: $t_A(x) \le P(\text{len}(x))$. Formally, we can write this definition as

$$\exists P(n)\, \forall x\, (t_A(x) \le P(\text{len}(x))).$$

Comment. It is important to mention that the above definition does not fully capture the intuitive idea of when an algorithm is practically feasible. For example, an algorithm whose running time is $t_A(x) = 10^{1000} \cdot \text{len}(x)$ is clearly not practically feasible, but it is feasible according to the above definition: indeed, its running time is limited by a linear polynomial $P(n) = 10^{1000} \cdot n$.

On the other hand, for an algorithm whose running time is

$$t_A(x) = 2^{10^{-12} \cdot \text{len}(x)},$$

the computation time cannot be bounded by a polynomial, so this algorithm is not feasible according to the above definition. However, in practice, for all inputs whose size does not exceed 1 Terabyte, this algorithm works fast – so, for all practical purposes, it is feasible.

These are counter-examples, but, in general, this definition works well – and it is the best one we have :-(

What are NP-problems: reminder. Algorithms are used to solve problems. Computers are normally used to solve problems for which it is possible to check whether a proposed "solution" is indeed a solution or not. It may be difficult to find a solution, but it should not be that difficult to check that the solution is correct.

A good example is solving a system of nonlinear equations:

- it may be difficult to find a solution, but
- once a solution is computed, it is easy to check that it is indeed a correct solution: just
  - plug in the computed values into the corresponding equations and
  - check whether the left-hand side of each equation is equal to its right-hand side.

In precise terms, this means that there should be a feasible algorithm $C(x, y)$ that, given an input x and the proposed solution y, checks whether y is indeed a solution to the given problem. For this checking to be efficient, it is also important to require that simply copying y into a checking algorithm should also be feasible, i.e., that the length $\mathrm{len}(y)$ should be bounded by some polynomial $P_\ell(\mathrm{len}(x))$ of the length of x.

In line with this natural idea, a *generic problem* can be defined as a pair (C, P_ℓ), where $C(x, y)$ is a feasible algorithm producing "yes" or "no", and $P_\ell(n)$ is a polynomial. An *instance* of this generic problem is then defined as follows:

- *given:* an input x (a sequence of symbols),
- *find:* a string y for which $C(x, y)$ is true and $\mathrm{len}(y) \le P_\ell(\mathrm{len}(x))$.

Generic problems are also known as *NP-problems*, where NP is short of *Nondeterministic Polynomial*, meaning that once we have guessed y, we can check, in polynomial time, whether this guess is indeed a solution to our problem.

NP-hard problems: a reminder. In principle, every NP-problem can be algorithmically solved: indeed, for every input x, we only need to consider possible strings y whose length does not exceed a given number $P_\ell(\mathrm{len}(x))$. There are finitely many such strings, so, in principle, we can try them all until we find a solution.

The problem with this algorithm is that it requires exponential time: e.g., even if we consider binary strings and $P_\ell(n) = n$, for an input of size n, we need to try all possible binary strings of length n, and there are 2^n of them. A natural question is: is it possible to solve all NP-problems by feasible algorithms? This is a known open question.

The class of all generic problems which can be solved by a feasible algorithm is usually denoted by P, short of *Polynomial-time*, so the above open question is whether the class NP of all NP-problems is equal to P: $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$.

Most computer scientists believe that $\mathrm{P} \neq \mathrm{NP}$, and moreover, that at least for some NP-problems, every algorithm for solving them requires – in the worst-case – exponential time.

While it is not known whether there are NP-problems which cannot be solved by a feasible algorithm, it is known that there are NP-problems which are harder than all other NP-problems: namely, for which every other NP-problem can be reduced to this problem. Such harder-than-all problems are known as *NP-hard*.

The above definition is based on the notion of reduction. Reduction is somewhat cumbersome to formally define, but it is intuitively clear. For example, each equation of the type $p + q \cdot x + \dfrac{r}{x} = 0$ can be feasible reduced to a quadratic equation $p \cdot x + q \cdot x^2 + r = 0$.

Historically the first example of an NP-hard problem is the *propositional satisfiability problem* (SAT), in which:

- we are given a propositional formula with k Boolean ("true"-"false") variables $v_1, \ldots, v_k$, and
- we need to find the values of these variables that make the given formula true.

For example, for a formula $(v_1 \vee \neg v_2 \vee v_3) \,\&\, (\neg v_1 \vee v_2)$, the values $v_1 = v_2 =$"false" make it true.

Graph Isomorphism problem: description and a recent result. For most other problems for which no polynomial-time algorithm is known, it was proven that these problems are also NP-hard. There are few problems, however, for which no one could come up with such a proof. The most well-known problem is the following *Graph Isomorphism problem:*

- *given:* two graphs x,
- *find* a mapping y that maps the first graph into the second one in such a way that they are isomorphic – i.e., that vertices connected by edges are mapped into vertices connected by edges, and vertices not connected by edges are mapped into vertices not connected by edges.

It was recently announced [1] that this problem has a *quasi-polynomial* algorithm, i.e., an algorithm whose computation time is bounded by $\exp(P(\ln(\text{len}(x))))$ for some polynomial $P(n)$.

If this polynomial was linear, then we would have a polynomial time. In general, $P(\ln(n)) \ll n$, so this computation time – while being somewhat longer than polynomial – is much smaller than exponential.

2 Why This New Result Is Somewhat Counter-Intuitive?

Intuitively, which problem should be more complex? As we have mentioned, while all NP-problems can be solved by simply testing all possible alternatives – i.e., all possible words y with length $\text{len}(y) \le P_\ell(\text{len}(x))$. The problem with this approach is that there are exponentially many such alternatives and thus, an algorithm that tests all these alternatives is not feasible.

From this viewpoint, it is reasonable to expect that the more alternatives we have to check, the more complex the resulting problem.

Propositional satisfiability vs. graph isomorphism: from the common sense viewpoint, which problem is more complex? Let us use the above argument to compare the complexity of propositional satisfiability and graph isomorphism problems.

To solve an instance of a propositional satisfiability problem by checking all possible alternatives, we need to check all possible tuples of n boolean variables

$v_1, \ldots, v_n$. Each of these variables has two possible values, so overall, we have 2^n possible tuples.

On the other hand, to solve an instance of a graph isomorphism problem by checking all possible alternatives, we need to check all possible ways to map, in a 1-1 manner, each vertex of the first graph to some vertex of the second graph. If we denote the number of vertices in both graphs by n, then for the first vertex, we have n possible options, for the second, $n-1$ possible options (since one vertex of the second graph is already taken), etc. Thus, overall, we need to check $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = n!$ alternatives.

Asymptotically, $n! \sim \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi \cdot n}$, so $n! \gg 2^n$. We therefore expect that the graph isomorphism problem be much harder than the propositional satisfiability problem.

The above result shows, however, that, vice versa, the graph isomorphism problem is much easier to solve – at least the best known algorithm for solving this problem is much faster than the best known algorithm for solving propositional satisfiability.

How can we explain this counter-intuitive result?

3 Our Explanation

An important difference between graph isomorphism and propositional satisfiability: uniqueness of the solution. While both graph isomorphism and propositional satisfiability and NP-problems, there is an important difference between these problems in terms of number of solutions.

For graph isomorphism, the only possibility to have at least *two* different mapping under which two given graphs are isomorphic is when each of these graphs is isomorphic to itself under some *automorphism*. It is known that the vast majority of graphs do not have any non-trivial automorphisms. Thus, in the overwhelming majority of cases, the graph isomorphism problem has a unique solution.

In contrast, for propositional satisfiability, we often have many solutions. In the above example, $v_1 = v_2 =$"true" is also a solution, and each of these solutions is actually two different solutions, since in both cases, we can take $v_3 =$"true" and $v_3 =$"false". This can be easily explained for formulas in 3-CNF form, i.e., for formulas of the type $C_1 \,\&\, \ldots \,\&\, C_m$, where each "clause" C_j has the form $a \vee b \vee c$, and each "literal" a, b, or c is either a variable or its negation.

For a randomly selected tuple of Boolean variables, the probability that each literal is false is $1/2$, so the probability that all three literals are false is $(1/2)^3 = 1/8$, and the probability that one of the literals is true – and thus, that the clause is true – is $1 - 1/8 = 7/8$. Thus, a good estimate for the probability that all m clauses are true is $(7/8)^m$. So, out of 2^n tuples, the formula is true

for $2^n \cdot (7/8)^m$ tuples. This number is most probably not equal to 1. If it is smaller than 1, we expect that the formula has no satisfying vectors at all, if it is larger than 1, it probably has several satisfying vectors.

Uniqueness makes problems easier to solve. In general, it is known that uniqueness of a solution makes the problem easier to solve; see, e.g., [2, 3, 4, 5].

For example, in general, no algorithm is possible that, given a computable function on an interval (or, more generally, on a computable compact set), returns the point where this function attains its maximum. However, once we limit ourselves to functions that attains their maximum at a single point, then finding this location becomes an algorithmically solvable problem. On the other hand, if we allow the function to have two different locations where maximum is attained, the algorithm is no longer possible.

Similarly, in general, it is not possible to algorithmically find a solution to a computable system of equations, but, if we limit ourselves to systems that have exactly one solution, the problem becomes algorithmically solvable. And if we consider systems with two solutions, no general algorithms is possible.

Another example: in general, it is not possible to algorithmically find a fixed point of a computable mapping, but, if we only consider mappings with a unique fixed point, such an algorithm becomes possible. And no general algorithm is possible when we allow mapping with two fixed points.

Similar results are known for many other problems, so uniqueness indeed enhances computability.

This may be a possible qualitative explanation. This is probably how we can explain the counter-intuitive relative simplicity of the seemingly-harder-to-solve graph isomorphism problem: in this problems, solutions are almost always unique, and it is known that, in general, problems with unique solutions are easier to solve.

Acknowledgments.

This work was supported in part by the US National Science Foundation grants HRD-0734825, HRD-1242122, and DUE-0926721.

References

[1] L. Babai, "Graph isomorphism in quasipolynomial time", *Abstracts of the Combinatorics and Theoretical Computer Science Seminar*, University of Chicago, Chicago, Illinois, USA, November 10, 2015.

[2] U. Kohlenbach, "Effective moduli from ineffective uniqueness proofs. An unwinding of de La Vallee Poussin's proof for Chebycheff approximation", *Annals for Pure and Applied Logic*, 1993, Vol. 64, No. 1, pp. 27–94.

[3] U. Kohlenbach, *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*, Springer Verlag, Berlin-Heidelberg, 2008.

[4] V. Kreinovich, "Uniqueness implies algorithmic computability", *Proceedings of the 4th Student Mathematical Conference*, Leningrad University, Leningrad, 1975, pp. 19–21 (in Russian).

[5] V. Kreinovich, *Categories of space-time models*, Ph.D. dissertation, Novosibirsk, Soviet Academy of Sciences, Siberian Branch, Institute of Mathematics, 1979 (in Russian).

[6] V. A. Lifschitz, "Investigation of constructive functions by the method of filling", *J. Soviet Math.*, 1973, Vol. 1, pp. 41–47.

[7] C. Papadimitriou, *Computational Complexity*, Addison Welsey, Reading, Massachusetts, 1994.