MY ALGORITHM THAT SOLVES AN NP-COMPLETE PROBLEM IN POLYNOMIAL TIME, UNDER ASSUMPTION THAT P = NP

VICTOR PORTON

ABSTRACT. I present a particular polynomial-time algorithm that solves an NP-complete problem under the supposition that some such an algorithm exists (in other words, I present a constructive positive solution of P = NP under the assumption that P = NP).

1. INTRODUCTION

Fix any non-contradictory formal system, containing first-order predicate calculus (such as first-order predicate calculus or ZFC). Note that our formal system can be used to prove correctness of its own proofs (in polynomial time).

In this article I use the word “proof” exclusively either to denote proofs in our formal systems or to denote the proof presented in this article. I do not use it as a synonym of “certificate”. (However, certificates used are proofs.)

I present a particular polynomial-time algorithm that solves an NP-complete problem under the supposition that some such an algorithm exists (in other words, I present a constructive positive solution of P = NP under the assumption that P = NP).

The idea appeared as a synthesis from [3] and my own independently conceived [1]. A similar algorithm was first published in [2].

2. PROOF

I will call an NP-complete verifier an algorithm that verifies an NP-complete problem in polynomial time.

Obviously, if P = NP, then there exists some NP-complete verifier.

Let \( R(X) \) be the property, whether an arbitrary algorithm \( X \) (that takes any input data \( Y \)) produces a proof (in our formal system) of the statement (for every algorithm \( Y \))

\[
X(Y) = Z \Rightarrow \exists \text{algorithm } X' : X'(Z) = Y.
\]

I remind: \( X \) is in NP means that (for every \( Y \))

\[
X(Y) = Z \Rightarrow \exists \text{polynomial-time algorithm } X' : X'(Z) = Y.
\]
In the standard definition of NP we have the additional condition at the left side of the implication that \( Z = \text{true} \). But let us limit further consideration to such problems that \( X \) always halts; then we consider \( Z \in \{ \text{false}, \text{true} \} \).

So, \( X \) is an NP \( \Rightarrow R(X) \).

In the usual definition \( Z \) is taken to be one bit, but we could instead allow \( Z \) to be any polynomial amount of data, without changing concepts of \( R \) and of NP-complete.

**Proof of the main result.** Assume P = NP. Then my polynomial-time algorithm for the problem \( R \) for an input data \( X \) such that either \( R(X) \) or \( \neg R(X) \) is provable (This problem is NP-complete, for example, because finding proofs in our formal system is its special case.):

Let algorithm \( V(A) \) (of an input data \( X \)) be:

1. Run \( A \).
2. Check (by a known polynomial-time algorithm) that the step 1 produced either a proof of \( R(X) \) or a proof of \( \neg R(X) \). Return true, false, or unknown.

\( V(A) \) is polynomial-time if \( A \) is polynomial-time, because the output of step 1 is polynomial-size (dependently on \( X \)).

Our main decision algorithm \( M \):

Enumerating all algorithms \( A_n \) \((n \in \mathbb{N})\) run the algorithms \( V(A_n) \) on \( X \) in parallel (interleaving these algorithms, and before each step \( n \) of the loop adding \( A_n \) to our dynamic array (see below) of algorithms) until one of the “threads” \( F \) produces “true” or “false” (not “unknown”) (thus having a proof of \( R(X) \) or a proof of \( \neg R(X) \)).

The algorithm \( M \) halts, because there is a proof of \( R(X) \) or of \( \neg R(X) \) by the algorithm \( B = A_G \) for some fixed (assuming determinacy of our machine) \( n = G \) (by our supposition that there is an NP-complete verifier for provability of \( R(X) \) or what is the same for \( R(X) \)).

\( M \) solves \( R(X) \) because some algorithm \( V(A_n) \) solves \( R(X) \).

\( M \) is polynomial-time: \( G \) is fixed and \( B \) is polynomial-time (denote it \( C(B, C(X)) \) where \( C(X) \) is the size of \( X \)). So running time of our algorithm is bounded above (we can choose such a Turing-machine-equivalent computing environment that \( p(i) \) is non-decreasing) by

\[
\sum_{i=0}^{i=G+C(B,C(X))} p(i) \leq (G + C(B, C(X)) + 1)p(G + C(B, C(X)))
\]

where \( p(i) \) is the amount of steps in \( i \)-th main loop iteration (it polynomially depends on \( i \)).

Clarifications on parallelism:

By parallel, I mean interleaving algorithms with a fixed switch time. In terms of Turing machines, the term parallel can be defined through “virtual machines” (Turing machine simulating several Turing machines). The
algorithm is: create an algorithm that calculates a step of a ("virtual") Turing machine, use it to run the next step of \( v_0, \ldots, v_k, v_0, \ldots, v_k, \ldots \) (where \( v_0, \ldots, v_k \) is a dynamic array of Turing machines).

3. CONCLUSIONS

So, I’ve proved that a general solution of \( P = NP \) implies a specific solution that I’ve demonstrated.

The algorithm above is blatantly inefficient (at least it has a huge additive constant). So, it could not be used in modern practice.

It however can be useful as an argument to prove something other and/or as a “sketch” for an algorithm for some more particular problems. It could not be directly applied to a particular hard problem however, because the proof uses \( P = NP \) in an essential way.

My algorithm also can be optimized (however still remaining with a big additive constant) by somehow eliminating “bad” \( A_n \). This requires further research.

REFERENCES


Email address: porton@narod.ru