Poznań University of Technology

On polynomial-time solvability and fixed code size algorithms

M.Drozdowski

Research Report RA-06/16

2016

Institute of Computing Science, Piotrowo 2, 60-965 Poznań, Poland

On polynomial-time solvability and fixed code size algorithms

M.Drozdowski
Institute of Computing Science, Poznań University of Technology,
Piotrowo 2, 60-965 Poznań, Poland

Abstract

In this paper it is argued that polynomial-time algorithms with fixed size code and limited size of data that can be integrated in one step have insufficient capability to solve hard combinatorial problems. Consequently, each polynomial-time algorithm running on a realistic model of computation has an instance which is not solved correctly or not in polynomial time.

Keywords: computational complexity, combinatorial optimization.

1 Introduction

In this paper we intend to demonstrate that polynomial-time algorithms with fixed size code running on a realistic model of a computer have insufficient information capacity to solve hard combinatorial problems and each such algorithm must have an instance on which it fails.

An idea that performance of algorithms is somehow related to the amount of information is haunting combinatorial optimization. For example, in [2] entropy of Markov chains representing behavior of simulated annealing algorithms with different configuration spaces is analytically computed for given instances of maximum satisfiability 3-SAT problem. A configuration space is defined as a space of solutions with their objective function values and their neighborhoods. A connection between entropy of the Markov chain and the convergence of the expected objective function value has been demonstrated: the higher the entropy, the better the objective values. In [6] histograms of the objective function values attainable in limited number of calls to objective function evaluations are examined. It is argued that there is a link between the fraction of problem instances achieving certain histogram of values and the entropy of the histogram. In evolutionary optimization populations of solutions are maintained. It is widely accepted rule of thumb that populations with more individuals and more diversified ones have bigger chances

Table 1: Summary of notations.

c_j	jth clause of SAT instance, for $j = 1, \ldots, m$
F	$F = \cap_{j=1}^{m} c_j$ conjunction of clauses c_j
I	instance of a problem
K	upper bound on integer values in uniform computation cost model
L	maximum number of bits used by any function computed in a step of the algorithm
$N(I)$	size of instance I
n	number of variables in SAT problem
m	number of clauses in SAT problem
$S_{\Pi}(I)$	set of solutions for instance I of search problem Π
x_i	ith variable in SAT problem, for $i = 1, \ldots, n$
XV	vector of n binary values, alternatively n-bit unsigned integer
$F(XV)$	the value of function F for some given value of vector XV
$\overline{x_i}$	negation of variable i in SAT problem, for $i = 1, \ldots, n$
$\widetilde{x_i}$	variable i with negation or without negation in SAT problem, for $i = 1, \ldots, n$
$x[i, j]$	the ith variable in clause c_j in SAT problem
Z	size of algorithm code in bits

of producing high quality solutions. Intuitively, such populations have more information.

The argument presented here is the following: On the one hand, hard combinatorial *problems* have number of unique possible solutions growing exponentially with the sizes of instances. On the other hand, polynomial time algorithms with fixed code size running on realistic models of computation can visit at most polynomial number of states. An important element of a realistic model of computation is that the size of data which can be integrated in one step is limited. Further organization of the paper is as follows. In Section 2 two examples provide evidence of the link between polynomial time computability and information size of the algorithm code. In Section 3 model of a realistic computer is introduced. In Section 4 all the earlier observations are put together to build a kind of explanation. Definitions and notations will be introduced in the text as needed. A summary of notations is collected in Table 1.

2 Evidence

In this section two examples are provided to demonstrate that if information source is given, then it is possible, at least in principle, to escape from some limitations of the algorithms solving hard combinatorial problems.

Example 1.
Let us consider the classic vertex coloring problem. This problem can be solved by a number of greedy coloring algorithms [4]. A greedy coloring algorithm proceeds as follows: (i) determine sequence σ of the vertices according to some rule, (ii) follow sequence σ and give each vertex the smallest feasible color. The quality of such greedy methods can be assessed by sizes of the graphs the algorithms fail to color optimally. More formally [4]:

Graph G is *slightly hard-to-color* for algorithm A if there exists at least one implementation of A which colors G in the suboptimal way.

Graph G is *hard-to-color* for algorithm A if any implementation of A colors G in the suboptimal way.

The differences in algorithm implementations may arise as a result of ambiguity of the vertex sequencing rules. For example, if the vertices are sequenced according to the non-increasing order of vertex degrees (i.e. largest-first (LF) rule), then there can be may vertices of the same degree, and such vertices are alike for the LF rule. Any method of resolving the permutation of the vertices with equal degrees results in a different implementation of the greedy coloring algorithm with LF rule. In evaluating quality of different greedy algorithms we search for the (slightly) hard-to-color graphs G with the smallest number of vertices. In [4] a number of greedy algorithms defined by various sequencing rules are analyzed and for all of them graphs slightly hard-to-color and hard-to-color are provided. Except one. The random sequential (RS) rule sequences the vertices randomly. For the RS rule the slightly hard-to-color graph is a path of four vertices (P_4). Surprisingly, or maybe not, RS rule has no hard-to-color graph. Now this situation can be fought of as if all greedy coloring algorithms, except RS, had fixed size code in the sense of bits needed to encode them, while the size of RS expressed as the number of code bits were unbounded because RS has access to a source of unlimited amount of information by the use of perfectly random vertex sequences. In a sense the code of RS is incompressible and its amount of information is unlimited.

More formally, we will call algorithm A *fixed size code* (simply fixed code)

if the number of bits encoding the algorithm is constant.

Intuitively, it seems as if the fixed code greedy algorithms for vertex coloring carried an amount of information sufficient to solve to optimality only small instances. Conversely, RS cannot ultimately fail because it has enough information to deal with instances of any size.

Example 2.
In the following we consider 3-SAT problem defined as follows [3]:

3-SAT DECISION VERSION
INPUT: sums $c_j, j = 1, \ldots, m$, of triplets of binary variables chosen over a set of n binary variables $x_1, \ldots, x_n$ or their negations.
QUESTION: Is there any assignment of values 0/1 to binary variables $x_1, \ldots, x_n$ such that the conjunction of clauses $F = \cap_{j=1}^{m} c_j$ is 1?

3-SAT SEARCH VERSION
INPUT: The same as in the decision version.
REQUEST: Find the assignment of values 0/1 to binary variables $x_1, \ldots, x_n$, i.e. vector XV of n 0/1 values such that the conjunction of clauses $F(XV) = \cap_{j=1}^{m} c_j$ is 1. If such a vector does not exist then signal $\emptyset$.

Let us observe that the solution to search 3-SAT, the n-bit vector XV, can be interpreted as n-bit unsigned integer. We will denote by $F(XV)$ the value of function F for some given value of vector XV. The search 3-SAT can be solved by the following two algorithms:

Algorithm 1.
1: cin $>> I$;
2: **new**(XV, n);
3: **for**$(XV = 0; XV < 2^n; XV++)$
{
3.1: **if** $F(XV)$ { cout $<< XV$; **break**;}
}

In Algorithm 1 the data is read in step 1 in $N(I)$ time units. Then, $n < N(I)$ bits of memory are allocated in step 2 to hold vector XV. Values of vector XV are enumerated in loop 3. If an assignment of bits to XV satisfying F is found then the loop is terminated. Algorithm 1 enumerates all possible vectors XV. One evaluation of F can be performed in time $O(m \log n)$ which is $O(N(I))$ (log denotes base-2 logarithm). Algorithm 1 has complexity

$O(2^n m \log n)$, or equivalently, at most $O(2^{N(I)}N(I))$. Thus, computational complexity makes Algorithm 1 practically infeasible.

Algorithm 2.
1: cin >> I;
2: $XV = SolutionsTable[n][m][c_1] \dots [c_m]$;
3: cout << XV

Essentially, Algorithm 2 reads the solution vector XV (if the assignment satisfying $F(XV) = 1$ exists) or $\emptyset$ (if such XV does not exist) from table $SolutionsTable$ of precomputed solutions. $SolutionsTable[n][m][c_1] \dots [c_m]$ slightly extends the admissible syntax of the current programming languages because it has variable number of table dimensions. But still, transforming it to standard (e.g. C++) expressions is possible. For example, in the first step a pointer to a solution table for $[n][m]$ is selected from a 2D table of pointers and in the second step XV (or $\emptyset$) is found using $[c_1] \dots [c_m]$ to calculate the offset. Clause c_i is treated here as a $3 \log n$-bit index in a table. In uniform computation cost model [1] each operation (like the address calculation) can be done in constant time. The uniform model has practical sense if each number has value bounded by a constant and consequently constant size binary representation. If a uniform computation cost model [1] is assumed, then Algorithm 2 can fetch the right solution in $O(m)$ time. Assuming that each variable in a clause is different, the size of $SolutionsTable$ for search 3-SAT up to the required positions of n and m is at most

$$\sum_{i=1}^{n} i \sum_{j=1}^{m} (2^{i+1} \times 2^{i-1+1} \times 2^{i-2+1})^j \leq \sum_{i=1}^{n} n \sum_{j=1}^{m} (2^{3n})^m \leq n^2 m 2^{3nm}$$

Thus, for the uniform cost of computation, Algorithm 2 is polynomial although it uses exponential size storage of size $O(n^2 m 2^{3nm})$.

In the logarithmic cost model the cost of every operation is proportional to the number of involved bits. The number of different clauses c_i on n binary variables is at most $2^{n+1}2^{n-1+1}2^{n-2+1} = 2^{3n}$. Since any subset of clauses can be used, we have that

$$m \leq \sum_{i=1}^{2^{3n}} \binom{2^{3n}}{i} \leq 2^{2^{3n}}$$

and Algorithm 2 requires address pointers of length at least $\log(2^{2^{3n}}) = 2^{3n}$ bits. The storage size is $O(n2^{2^{3n}})$ bits for n alone. Thus, for a logarithmic

computation cost model Algorithm 2 is neither polynomial nor in polynomial space.

Let us return to Algorithm 2 with uniform cost model. On the one hand, Algorithm 2 with uniform cost model holds a promise of polynomial execution time. On the other hand, Algorithm 1 has clearly exponential runtime. What makes Algorithm 2 (with uniform cost model) so capable compared to Algorithm 1, is the amount of information carried in Algorithm 2 and embodied in $SolutionsTable$ of exponential size. Thus, the amount of information comprised in the algorithm determines its capability in solving hard combinatorial problems in limited time. It can be speculated that if the amount of information carried in the code of an algorithm is insufficient, then there will always be instances of hard combinatorial problems which *cannot* be solved in polynomial time and the algorithm *fails*. Note that algorithms solving problems in complexity classes are defined for the positive cases in [3] and we have to define what it means that an algorithm fails to solve an instance of a problem.

Algorithm A running in limited time T, *fails* on instance I of *decision* problem Π if $I \in Y_\Pi$ (i.e. the answer to I is positive) and A does not answer "yes" in time T. It also implies that A fails if it stops after T. In automaton representation of the algorithm, a failure means that the automaton does not reach the accepting and terminating state by time T.

In search problems an algorithm solves problem Π by providing for each instance I a solution $q \in S_\Pi(I)$ where $S_\Pi(I)$ is a non-empty set of solutions admissible for I. If $S_\Pi(I) = \emptyset$ then the algorithm signals that $S_\Pi(I)$ is empty, e.g., by producing $\emptyset$. We will be saying that algorithm A running in limited time T *fails* on instance I of *search* problem Π if: (i) if $S_\Pi(I) \neq \emptyset$ then A does not provide solution $q \in S_\Pi(I)$ by time T, (ii) if $S_\Pi(I) = \emptyset$ then A does not signal $\emptyset$ by time T. It follows that if algorithm A provides by time T a solution not in $S_\Pi(I)$, or A runs longer than T, then A fails on I.

Informally, if algorithms for decision and search problems fail, then they do not provide a correct answer or they do not stop in polynomial time. The examples shown above inspire a thought that for any fixed code algorithm solving hard combinatorial problem an instance always exists on which a polynomial time algorithm fails.

Hypothesis 1 *Every fixed code polynomial-time algorithm solving a hard*

combinatorial problem has an instance[1] *on which the algorithm fails.*

Let us comment on the above hypothesis. If **P**≠**NP** then all fixed code algorithms for computationally hard decision problems (i.e. **NP**-complete problems) and consequently computationally hard search and optimization problems must have such an instance (the nemesis). Otherwise (if there were no nemesis instance for some algorithm solving an **NP**-complete problem), then the algorithm would solve each instance in polynomial time implying **P**=**NP**.

3 Computation Model

In this section a model of a computer is defined. We assume that the algorithm operates in discrete *steps* in a single thread of execution. These steps form a sequence in time and each step by convention takes a unit time. In each step of the execution, some *function* uses some *arguments* (input) to compute its *outcome* (output, result). The admissible functions must obey the following restrictions:

- The functions are executed in one step of the algorithm.
- The functions have fixed-size input. Size of the input is measured by the number of bits of the string encoding the function arguments. The size of the input any such function can process in a step is limited to L bits. This limit has practical motivation because only limited data sizes can be transfered (e.g. to/from memory) and processed in finite time.
- The function output is binary and of limited size (bounded by a constant). The binary representation is necessary to allow processing output of the function in the following steps.
- The functions computed in each step are deterministic in this sense that for a single input value only a single output value is obtained.
- It must be possible to encode the functions in bounded number of bits.

[1] Let's call it nemesis.

Informally, the arguments, the function and the outcome can be of any nature existing in practical models of computers. It means that we abstract away what the functions executed in one step actually calculate. For example, for function $f : x \to y$ the values of (x, y) are beyond the scope of our consideration. Functions defined above have a practical representation as CPU instructions. We assume RAM model of the computation and uniform cost of the computation [1] (though we make some references to the logarithmic computation cost model). In the RAM model it is assumed that memory is an array of addressable registers available to store intermediate results of the computation. The size of the memory is sufficiently big and is not limiting the algorithm. As a consequence of the above limits on the nature of the functions computed in one step (in particular limited input size L and limited output size), the maximum number of memory registers accessible in one step is limited. For instance, in indirect addressing the argument and/or result absolute addresses are calculated using content of the registers (base, offset, index). The address translation operation is part of the function executed in one step and the registers used for address calculation are considered part of the function input. An algorithm is represented as a sequence of functions. A practical equivalent is program code comprising a sequence of instructions. We assume existence of a program counter pointing to the next function (instruction) to be executed. By convention we consider program counter to be one of the registers, consequently an element of RAM.

The limit L on the number of bits which can be processed in a step means that the set of different values which can be presented as numbers in the registers has bounded cardinality. This limitation is practical because the size of registers which can be manipulated by existing computers in one step are limited (e.g. at most a few 64-bit registers in one clock tick).

Let us observe that Oracle Turing Machine (OTM) [3] which can be called to compute arbitrary functions in one step does not satisfy the above requirements because OTM can read arbitrary (i.e. unlimited) input size in one step. Here only a limited size input can be tackled in each computation step. Though it is possible to read arbitrary length input string by a fixed code algorithm (such as in a loop), but then it takes many read steps, not one. We exclude recursive functions from the set of functions computed in a step.

4 Kind of an explanation?

Theorem 2 *There exist instances of 3-SAT search problem of size $N(I)$ with $\Omega(2^{d_1 N(I)})$ unique solutions for uniform computation cost criterion and $\Omega(2^{N(I)/(d_2 \ln N(I))})$ for logarithmic computation cost criterion, where $d_1, d_2 > 0$ are constants.*

Proof. Assume there are n variables and $m = 4n$ clauses in 3-SAT. Let there be 4 clauses $c_{i1} = x_a + x_b + \widetilde{x_i}, c_{i2} = \overline{x_a} + x_b + \widetilde{x_i}, c_{i3} = x_a + \overline{x_b} + \widetilde{x_i}, c_{i4} = \overline{x_a} + \overline{x_b} + \widetilde{x_i}$ for each $i = 1, \ldots, n$. It can be verified that no valuing of x_a, x_b makes the four clauses simultaneously equal 1. The four clauses may simultaneously become equal 1 only if $\widetilde{x_i} = 1$. Satisfying formula $F = c_{11}c_{12}c_{13}c_{14} \ldots c_{n4}$ depends on valuing of variables $\widetilde{x_i}$ for $i = 1, \ldots, n$. Depending on whether x_i is negated or not there can be 2^n different ways of constructing formula F, thus leading to $2^n = 2^{m/4}$ different solutions. Variables x_a, x_b can be chosen arbitrarily. For example, a, b can be drawn randomly from $1, \ldots, n$ such that $a \neq b$ and $a, b \neq i$. Since there are $(n-1)(n-2)$ possible pairs a, b for each i, it is possible to generate pairs a, b satisfying the above conditions for $n \geq 3$.

Suppose the uniform cost is assumed, then each number has value limited from above by constant K. The length of the encoding of the instance data is $N(I) = 3m \log K + \log K = 12n \log K + \log K$ because it is necessary to record the indices of variables in $\log K$ bits, each binary variable induces 4 clauses of length $3 \log K$. Consequently, the number of possible unique solutions is $2^n = 2^{(N(I) - \log K)/(12 \log K)} = 2^{N(I)/(12 \log K)} 2^{-1/12}$, which is $\Omega(2^{d_1 N(I)})$, where $d_1 = 1/(12 \log K) > 0$ is constant.

Assume logarithmic cost, then the number of bits necessary to record n is $\lfloor \log n \rfloor + 1$. Length of the encoding string is $N(I) = 12n(\lfloor \log n \rfloor + 1) + \lfloor \log n \rfloor + 1 \leq 15n \log n = dn \ln n$, for $n > 2^{12}$ and $d = 15/\ln 2 \approx 21.6404$. An inverse function of $(cx \ln x)$, for some constant c, is $\frac{x}{c}/W(\frac{x}{c})$, where W is Lambert W-function [5]. Lambert W function for big x can be approximated by $W(x) = \ln x - \ln \ln x + O(1)$. Given instance size $N(I)$, we have $n \geq \frac{N(I)}{d}/W(\frac{N(I)}{d}) \approx \frac{N(I)}{d}/(\ln \frac{N(I)}{d} - \ln \ln \frac{N(I)}{d} + O(1)) \geq \frac{N(I)}{d}/(2 \ln \frac{N(I)}{d}) \geq \frac{N(I)}{d}/(2 \ln N(I) - 2 \ln d) \geq N(I)/(2d \ln N(I))$, for sufficiently big $N(I)$. Note that $N(I), dn \ln n, \frac{x}{c}/W(\frac{x}{c})$ are increasing in n, x. Thus, by approximating $N(I)$ from above we get a lower bound of n after calculating an inverse of the upper bound of $N(I)$. The number of possible unique solutions is $2^n \geq 2^{N(I)/(d_2 \ln N(I))}$ where $d_2 = 2d$. Observe that $2^{N(I)/(d_2 \ln N(I))}$ exceeds

any polynomial function of $N(I)$ for sufficiently big $N(I)$. It is because for a polynomial function $O(N(I)^k)$, $\ln(N(I)^k) < N(I)/(d_2 \ln N(I))$ with $N(I)$ tending to infinity.

□

Let us observe that although the number of unique solutions in Theorem 2 is exponential, an adversary knowing the construction of $c_{i1}, \ldots, c_{i4}, F$ would easily figure out values of $\widetilde{x_i}$. However, this task may be made much more difficult if literals in c_{ij} and clauses c_{ij} in F are permuted truly randomly. In the following we show that an algorithm with a fixed code is able to compute in each step of its execution a fixed number of outcomes, i.e. make at most a fixed number of calculations with unique final values.

Theorem 3 *Any fixed code algorithm computes in one computation step at most a fixed number of outcomes.*

Proof. Let us remind that the algorithm operates in a single thread of execution. Fixed algorithm code, and fixed size of the input in each step imply that: i) the number of unique (i.e. different) functions computable in a step is fixed, ii) the number of unique outcomes is fixed. In the following we explain it in more detail.

Let Z denote size of the algorithm code in bits. The functions computed by the algorithm in each computation step are encoded in the Z bits. Thus, some subset of i bits from Z is used to encode each of the possible functions the algorithm may execute in any step. We abstract away here from how the function is constructed or what object such a function computes. We are only interested in counting the number of possible such functions. Suppose the set of i bits encoding such functions is given. A function which is not encoded ($i = 0$ bits) cannot be called. Even a default function, or `nop` (no operation) function, have to be distinguished from the other functions. Hence, the number of bits encoding a function must be $i > 0$. The number of functions which can be distinguished by referring to these i bits is at most 2^i. The set of i bits from Z bits can be selected in $\binom{Z}{i}$ ways. Thus, the number of different functions encoded on i bits which can be distinguished from each other is at most $2^i \binom{Z}{i}$. For the whole code of size Z, the number of different functions which can be discerned is at most:

$$\sum_{i=1}^{Z} 2^i \binom{Z}{i} \leq \sum_{i=1}^{Z} 2^Z \binom{Z}{i} \leq 2^Z \sum_{i=1}^{Z} \binom{Z}{i} \leq 2^Z 2^Z = 2^{2Z}.$$

Now we proceed to explaining that the number of outcomes a fixed code algorithm can produce in one step is limited. Each function fetches some number of argument values from RAM. The size of arguments used by any function executed in a step is limited to at most L bits. Hence, the number of possible input values is at most 2^L.

The limited number of functions which can be called in a step (a result of limited algorithm code size Z) implies a limit on the number of ways of choosing the addresses in RAM which can be referred to and used to fetch inputs to a function executed in a step of the algorithm. In other words, each function called in a step of the algorithm has its specific way of fetching its arguments. The number of ways of choosing arguments from memory cannot be greater than the number of functions that can be called in a step for otherwise the ways of choosing arguments should be also stored in the algorithm code. Thus, either the ways of choosing function arguments are encoded in the algorithm code and these are some of the functions we have already enumerated, or we get a contradiction because more code than Z bits are needed to additionally encode the ways of fetching function arguments. Let us remind that the functions computed in each step by the algorithm are deterministic in this sense that for a single input value only a single output value can be obtained. Thus, the number of possible outcomes in one step is at most:

$$2^L 2^{2Z}.$$

The above number is a product of the number of possible input values (2^L) and different ways of processing them, i.e. the number of different functions which can be encoded in Z bits (2^{2Z}). □

Let us comment on the above theorem. Note that the calculation of the number of possible positions $\binom{Z}{i}$ where particular set of i bits encoding some i-bit function resides, is capable enough to represent sequences of instructions existing in practical computer codes. For example, if some code has instruction `add` at addresses $x, y, z, \ldots$ then this is accounted for in our enumeration. Hence, complex directed graphs representing flows of control in real codes are represented in the above calculation. Again, let us observe that the functions considered above may fetch and store their results in RAM locations dynamically, i.e. they can be determined in the course of the computation. For instance, in indirect addressing the argument and/or result addresses are calculated using content of the registers (base, offset, index). In such a case the location(s) of the argument/result registers is translated

to an absolute register address(es). The address translation operation is part of some of the functions considered in the above theorem. Consequently, the data bits necessary to calculate the absolute address of the dynamic storage location are part of the L-bit input of the function. Note that program code may carry not only code but also data, as in Algorithm 2. The data from the code can be copied to the memory operated upon by the algorithm and even modified. Still, this does not change the upper bound on the number of outcomes in a step of the algorithm with a fixed code.

Let us remind (cf. Section 3) that the algorithm operates in discrete steps executing some functions. The functions read and modify the registers. A *state* of the algorithm is a unique set of the register values. A *log* of an algorithm execution is a sequence of states it visits.

Theorem 4 *Any fixed code algorithm running in polynomial time visits at most a polynomial number of states.*

Proof. A current state of the algorithm is defined by the values of machine registers (the memory). The next state of the algorithm is determined by the code on the basis of the current state, including program counter. Assume that the algorithm code is given, fixed and encoded in Z bits. Only one transition is possible for a deterministic algorithm from the current state to the next one. Then, there is only one log possible, determined by the given input. In each step one function is executed. By Theorem 3 the number of possible functions and their input values is $2^L 2^{2Z}$ and each log has to cross in each step one of these (input value)$\times$function pairs. Consequently, the number of new states is also at most $2^L 2^{2Z}$.

Consider two consecutive steps of the algorithm. The number of unique symbols which can be written in two steps is at most $2 \times 2^L 2^{2Z}$. It is feasible because two different datasets of length at most L can be used each invoking a different function which can be expressed in the code. (In practice this number should be a bit smaller because we need some bits to distinguish sources of data and the functions called and these steering bits must be visible in the current state, but ignoring these bits does not reduce the upper limit on the number of symbols that can be written in each stage). It is also an upper limit because in order to write more than $2 \times 2^L 2^{2Z}$ symbols in two steps the algorithm would have to (i) write the extra symbols in any of the two steps, or (ii) the first step alters the second step behavior such that it can write more than $2^L 2^{2Z}$ symbols. (We exclude (iii) that the second step

modifies the first step by the unidirectional progress of time). Case (i) means that the algorithm would exceed its processing capability by processing more than L bits or executing a function which is not encoded in the Z bits of the algorithm code, violating Theorem 3. Case (ii) has analogous consequences, because it would violate the limit on the number of outcomes accessible in one state established in Theorem 3.

This reasoning can be extended to more than a pair of consecutive steps in a log. Suppose the algorithm runs in time $p(N(I))$, where p is a polynomial. The number of visited states is at most $p(N(I))2^L 2^{2Z}$. □

Let us comment on the above theorem and proof. The computational capability of a single step is limited to just $2^L 2^{2Z}$ different output values. This seemingly limits capability of any algorithm to obtaining at most $2^L 2^{2Z}$ different outputs. This limitation can be circumvented by extending algorithmically size of the data structures operated upon by the algorithm, while still obeying the limit of L bits for a function input size. However, even if in each step we apply a different dataset, then no more than $p(N(I))$ unique datasets can be applied and hence at most $p(N(I))2^L 2^{2Z}$ unique datasets can be visited. For example, if a L =64-bit register overflows in a 64-bit CPU as a result of the increment then the resolution of numbers can be extended by using more registers and updating them using carry flags and program code to choose update function for the extended set of registers. Then, in e.g. 2 steps of the algorithm operating on two registers at most 2×2^{64} different values can be obtained in these registers, not $(2^{64})^2$. This does not preclude visiting $(2^{64})^2$ unique values of the register pair, but not in two consecutive steps because the two registers have to be updated separately.

The argument that an algorithm can write at most $2 \times 2^L 2^{2Z}$ symbols in two steps in the proof of Theorem 4 has information-theoretic interpretation. Namely, the information does not arise from nothing, i.e. it can be called an information conservation rule.

Corollary 5 *For 3-SAT search problem there are instance sizes for which a fixed size code algorithm cannot find a solution in polynomial time.*

Proof. The number of unique solutions for 3-SAT grows exponentially with instance size by Theorem 2 while the number of unique solutions which can be constructed by a polynomial-time algorithm is polynomially bounded by Theorem 4. Hence, there exist sufficiently big problem sizes $N(I)$ for which the number of unique solutions exceeds the number of solutions a

fixed code size algorithm is capable to construct. Thus, the algorithm fails by missing the polynomial run time limit, or fails by providing a wrong solution. □

The above corollary may be expressed in information-theoretic way. A hard combinatorial problem Π is a source of information. Π emits messages in the form of instances. Instances implicitly encode the solutions. The algorithm solving Π is a communication channel which decodes solutions from the instances. Input alphabet is the set of all symbols that the information source can emit. This input alphabet is equivalent to the set of all instances. The size of the input alphabet for the communication channel (the algorithm) is $\Omega(n2^n)$, or $\Omega(d_1 N(I) 2^{d_1 N(I)})$ or $\Omega(N(I)/(d_2 \ln N(I)) 2^{N(I)/(d_2 \ln N(I))})$ in units of instance size, by Theorem 2. Contrarily, the size of communication channel output alphabet is $O(p(N(I))2^L 2^{2^Z})$, where p is a polynomial. Hence, it can be said that the communication channel (the algorithm) has insufficient capacity to transmit a hard combinatorial problem.

In Corollary 5 we considered search version of 3-SAT. In the following we will show that 3-SAT is **NP**-equivalent [3] to demonstrate that the consequences of the earlier discussion extend to the decision version of the 3-SAT and consequently to all **NP**-complete problems.

Observation 6 *3-SAT search problem is* **NP***-equivalent.*

Proof. 3-SAT search problem is **NP**-hard because its decision version is **NP**-complete. 3-SAT search problem is **NP**-easy because it is possible to solve in $O(n)$ time the 3-SAT search problem by using an oracle solving the decision version with one of the variables x_i, $i = 1, \ldots, n$, set to test values 0 or 1. If one of the answers is positive, then component $XV[i]$ of the solution is set with the positive test value. Otherwise, the algorithm throws $\emptyset$ to signal that vector XV does not exist. □

5 Conclusions

In this paper we examined the number of solutions in hard combinatorial problems and the number of states that can be reached by a polynomial-time algorithm with fixed code on a realistic model of a computer. It appears that the first may have size growing exponentially with the size of the input and the second – size growing only proportionally with the time of computation.

Hence, for polynomial-time algorithms with fixed code running on realistic computer models, instances exist on which such algorithms fail.

References

[1] A.V.Aho, J.E.Hopcroft, J.D.Ullman, The design and analysis of computer algorithms, Addison-Wesley Publishing company, 1974 (I used Polish translation: Projektowanie i analiza algorytmow komputerowych, PWN, Warszawa, 1983).

[2] M.Fleischer, S.H.Jacobson, Information Theory and the Finite-Time Behavior of the Simulated Annealing Algorithm: Experimental Results, INFORMS Journal on Computing 11(1), Winter 1999.

[3] M.R.Garey, D.S.Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.

[4] M.Kubale (ed.), Graph Colorings, American Mathematical Society, Providence, Rhode Island, 2004. (I used Polish version: Optymalizacja dyskretna. Modele i metody kolorowania grafow, WNT, Warszawa, 2002. See also K.Manuszewski, Grafy algorytmicznie trudne do kolorowania, Ph.D. Thesis, Gdansk University of Technology, 1997).

[5] Eric W. Weisstein, Lambert W-Function, MathWorld–A Wolfram Web Resource. [accessed in September 2015]. `http://mathworld.wolfram.com/LambertW-Function.html`

[6] D.H.Wolpert, W.G. Macready, No Free Lunch Theorems for Optimization, IEEE Trans. on Evolutionary Computation 1(1), April 1997.