OPEN ACCESS

GRS Journal of Multidisciplinary Research and Studies
Abbriviate Tittle- GRS J Mul Res Stud
ISSN *(Online)- 3049-0561*
https://grspublisher.com/journal-details/GRSJ

Vol-2, Iss-3 (Mar- 2025)

Towards Solving NP-Complete and other Hard Problems Efficiently in Practice

Mircea-Adrian Digulescu
University of Bucharest

***Corresponding Author:** Mircea-Adrian Digulescu
"University of Bucharest"

Received: 28.02.2025 **Accepted: 10.03.2025** **Published: 14.03.2025**

Abstract: Until now, Computer Scientists have concerned themselves with identifying efficient algorithms for solving the general case of some problem – that is finding one which performs well when the size of the input tends to infinity. However, this is the precise opposite of what is actually needed in practice. Effectively solving some real-world problem entails identifying an algorithm which works well for all (or some) inputs up to some fixed upper bound dictated by the concrete practical application. Such an algorithm may be distinct from the one which solves the general case. Furthermore, a general case algorithm may not exist at all or finding it might prove painstakingly hard for the human mind. Fortunately, in practice all that is needed is one which works on the finite cases involved in the real world situations, not one which can, unaltered, solve any input correctly.

In this paper, we first introduce a theoretical framework for reasoning about finite algorithmics. It allows familiar concepts such as asymptotic complexity to be adapted to the case where the input size is bounded from above. We also present some elementary results within this theory. Secondly, we present a generic approach for automatically discovering an adequate algorithm for the finite case of some hard problem – if one exists. Thirdly, we argue why we expect the finite case of hard problems to be easier than the general case. Fourthly, we present some relevant ideas specific to three hard problems, namely 3CNF-SAT, String Compression and Integer Factorization. Fifthly, we discuss the significance of the theory and methods introduced in this paper – noting among other things that they can be used to automatically determine that either (i) P = NP, (ii) P <> NP or (iii) we don't really care about the distinction for practical purposes. Finally, we present four directions for immediate further research and formulate an open question which, when answered will, for all practical purposes, decide P=NP.

Enhancing the way Computer Scientists reason about hard problems is ultimately the single most important contribution we claim for this paper.

Keywords: *P equals NP, Finite Algorithmics, Theoretical Computer Science, Complexity Theory*.

1. Introduction

Until now, we as Computer Scientists have almost exclusively concerned ourselves with finding algorithms to solve interesting problems in the general case. That is to identify a single algorithm – some fixed finite sequence of lines of code – which solves said problem for any input. We then reason about upper time and space bounds for such an algorithm in terms of asymptotical complexity with regard to the input size in bits (or some general unbounded parameter which describes the difficulty of the input). Even when we are unable to find an algorithm suitable to our desires, we can reason about the constraints to which such – if it exists – must conform, in terms of lower-bounds. Furthermore, we can go on and analyze the relation between the relative hardness of problems – even those for which we do not yet have a satisfactory solution - by clustering them into complexity classes and then proceeding to examine the relationships between these. In 2005 there were about 417 complexity classes in the Complexity Zoo [1]. As of 2019, the number has grown to about 544 classes currently being investigated by humanity. However, they all pertain to solving some hard problem correctly and efficiently for all inputs, no matter how large.

The approach of solving a problem with which we are concerned in practice – of bounded input size - by reducing it to a potentially harder problem – of unbounded input size – and using our human creativity to find an asymptotically efficient algorithm for the latter has proven enormously successful over the past century. From pattern-matching, to bipartite (and then general) matching, from shortest paths in a graph to maximum flows, humanity has seen enormous success in solving practical problems via this approach. However, ever since its inception and increasingly pronounced during recent decades, the method has started to show its limitations.

A large number of problems, including SAT and all NP-Complete ones, integer factorization, solving stochastic games, all PSPACE-Complete problems and all problems in EXPTIME and above, as well as some mysterious ones like breaking AES encryption do not have any known efficient algorithm despite decades of research. These add to problems which are known not to admit any algorithm at all which solves them in the general case: like Kolmogorov complexity or solutions to Busy Beaver Game. Some of the others are suspected to not to admit such, but wheatear this is actually true or not is yet known.

Nevertheless, a large number of *instances* of many of such problems are actually solvable in practice. Modern SAT Solvers [2] can solve problems over up to millions of variables and a large number of those over tens of thousands [3][4]. There is no presently known method of deriving an instance of a SAT problem which is hard to solve by *any* heuristic (although finding easy cases of arbitrary size can be done). In fact, not even a theoretical framework exists to reason about such cases, despite numerous published empirical studies. The incomputable Busy Beaver problem itself has been solved for the two symbol game, up to 4

states inclusively [5].

There is an apparent discrepancy between how hard a problem seems to be in the general case (at least for the human mind) and how easy it is for *at least some* practical cases, which are the ones of actual concern to us. As such, it is time to turn our attention to studying actual instances of cases of hard problems – in particular those which might appear in practice (and are thus almost always of bounded input size). Investigating these might sometimes lead to efficient algorithms for all real-world needs, or even to the discovery of the general case solution.

Until now, computer science theory has paid little to no attention to finite algorithmics. The very foundational tool used to reason about algorithms – asymptotic complexity of a function, works by definition only for the limit to infinity. Under existing theory, all practical instances of a problem – which almost always entail some bounded input size – are trivially solvable in O(1). This includes computation of non-computable functions, without inclusion of proof of actual correctness.

In this paper we remedy this lacking of the current complexity theory by introducing finite algorithmics. We also present some noteworthy elementary results formulated under it.

As far as we know there is no theoretical prior work concerning finite algorithmics, as we are just now introducing the field.

Results exists which can, in retrospect, be regarded as part of field of finite algorithmics however. They can be found in the following domains: Artificial Intelligence and Machine Learning (there most problems solved have bounded input size by formulation), Heuristic Solvers for NP-Complete Problems (such as SAT Solvers) and to some extent Cryptography (since most cyphers have fixed key and block sizes). Nevertheless, even within these fields there has been to the best of our knowledge no systematic effort to date to introduce a theory which would allow formal reasoning about relative performance of various algorithms and relationships between various classes of problems with regard to a fixed upper bound on input size.

The study of the P/poly complexity class can be considered tangential to this work. We will discuss its relationship with some of the other complexity classes we introduce.

The rest of this paper is organized as follows.

In Section 3 we introduce the theory related to Finite Algorithmics as follows. Section 3.1 contains basic definitions pertaining to formulating computer science problems and solutions on the finite case. In Section 3.2 we introduce definitions which allow us to reason about natural functions restricted to a finite domain using concepts analogous to those employed in general case asymptotic theory. In Section 3.3 we present seven finite case complexity classes and define a few related concepts important to describing inherent difficulty within computer science problems. Section 3.4 formally introduces the problem of solving a computer science problem (i.e. producing the source code of an acceptable algorithm) and describes its inputs and outputs. Also in Section 3.4 we introduce a classification of existing general case computer science problems based on their known or apparent difficulty.

In Section 4 we present some elementary but very important results related to finite algorithmics, formulated within the theory we introduced in Section 3. Section 4.1 deals with relationships between finite complexity classes, both in relation to general case complexity (Sections 4.1.1 and 4.1.2) and among themselves (Section 4.1.3). In Section 4.2 we present a generic method for solving any computer science problem on the finite case (Section 4.2.1) and also introduce some very important elementary results pertaining to what performance guarantees can be attained for sure for certain types of problems (Section 4.2.2). In Section 4.4 we present further ideas which can be employed in the context of finite algorithmics to speed-up the quest for a solution to three well-known hard problems: 3CNF-SAT, String Compression (Kolmogorov Complexity) and Integer Factorization (hard only for a classical computer).

We use Section 4.3 to present 10 arguments which we consider overwhelmingly convincing to prove the existence of value in the study of finite algorithmics.

In Section 5 we discuss some clear implications of the results presented in this paper, including on the way we as computer science researchers ought to think about hard problems like P=NP.

In Section 6 we present four directions for immediate further research, and pose a crucial open question, within the realm of finite algorithmics. The answer to that open question can be used to decide (and for most practical purposes prove) P=NP. Furthermore, answering it can be done automatically (if but in a very long time frame). The mere existence of such a questions opens up new avenues in the quest for proofs in deciding P=NP.

Section 7 contains some brief Vitae of the author. Section 8 is dedicated to Acknowledgments and statement of interest (none).

Finally, In Annex 1 we include some estimated upper bounds for tractability for each finite complexity class, given existing hardware.

2. Materials and methods

This paper contains results of theoretical reasoning based on the author's current knowledge of advances in complexity theory, building of SAT Solvers and algorithms in general. Since it aims to introduce a new subfield of computer science, namely finite algorithmics, it stops short of providing experimental data as being out of the current scope Obtaining such experimental data, based on the methods presented here is of interest nevertheless and we, the author, encourage fellow scientists to try them out in practice and publish the findings Ultimately, the attractiveness of the field in general steams partially from the prospect of being able to enhance one's creativity using computers to automate trial-and-error. They can perform tasks such as eliminating obviously unpromising alternatives several orders of magnitude faster than a human.

3. Theory

We now proceed to introduce the required theory which enables formal reasoning about finite cases of general computer science problems.

Section 3.1 Introducing Finite Algorithmics

Definition 1 (Problem of restricted size)

Consider some problem **Prob** consisting of finding a proper algorithm **S** which, for any given an input **s** of length **|s|** from the universe possible inputs **U** <inclus in> **{0,1}*** produces some output **S(s)** which is among the set of valid outputs for input **s** for problem **Prob.**

We define **Prob[n0]** as the problem of finding such an algorithm

which produces desired output only when |s| <= k. Such an algorithm can have undefined behavior elsewhere.

Example: SUBSUM[1000] is the problem of finding an algorithm which computes correctly wheatear a particular sum is attainable by summing some or all of at most n0=1000 given integers.

Discussion: Note that the algorithm which is the answer to Prob[n0] can be different for different n0-s. SUBSUM[1000] might have a different algorithm than SUBSUM[1000000]. Also, solving the original problem Prob entails providing an algorithm which solves it for any input, regardless of the size – the same for all sizes. Thus, Prob[n0] can be regarded as a 1-parameter function [to include from N->{a,b,c,..}*], which, given some n0 outputs a string representing the desired algorithm in some chosen programming language. Prob itself can be regarded as a parameter-less function (or a constant) providing such.

Definition 2 (Problem of Exact Size)

We define Prob(n0) analogously to Prob[n0] to represent the problem of finding a proper algorithm when the input size is precisely n0.

We extend the notations of Definitions 1 and 2 to parameterized complexity accordingly. Namely, when we reason about the complexity of some algorithm not in terms of its input size, but in terms of some parameter **n** (for example number of variables in a 3CNF-SAT problem instance) – which only bounds the input size but is not exactly equal to it, the same notations apply replacing the length |s| of the input with the definition of this parameter.

The definition of what constitutes a proper algorithm for a given problem merits attention. For a particular family of computer science problems (e.g. boolean formula satisfiability) a myriad of constraints can be placed on either inputs (e.g. no more than 3 clauses per variable), outputs (e.g. should provide also a satisfiable assignment if one exists), algorithm itself (should be no longer than 10 Mbytes) or its runtime behavior (e.g. space and time complexity), in addition to the type of machine which will be running it (e.g. a probabilistic computer, quantum computer) in order to arrive at a particularization which is specific enough to allow us to reason about it formally. Some constraints are more interesting than others though.

Definition 3 (Full Problem Statement)

In order to specify the statement of a computer science problem fully, we require the following to be included:

- **Theoretical problem statement**. This is a formal description which specifies which particular outputs can be considered correct for a certain input. Example: For discrete logarithm we can consider an output correct if it represents the actual discrete logarithm of the input.
- **Type of machine used to solve it**. This can be Turing-equivalent, Probabilistic Turing-equivalent or Quantum Turing-equivalent. If humanity discovers other types of machines, this list can be expanded accordingly, without losing validity of most results within this paper.
- **Restriction on input size**. This can be specified directly, or via some parameter which constraints it. For the non-finite case, this limit is taken to be +INF. We require that this limit either be +INF or a natural number explicitly given (not merely constrained).
- **Restriction on output size**. This involves setting some constrains on the function which correlates the output of the algorithm to size of the corresponding input. *Example*: We require output be of polynomial size in the input size.
- **Restrictions on input universe**. In addition to size restrictions, we can require that the input satisfies some additional constraints, limiting generality (e.g. there are only 3 clauses per variable for a 3CNF-SAT instance, or that it represents a satisfiable formula). These can be included in 1. or not.
- **Accuracy requirements**. These specify how often and in what way is the algorithm allowed to stray from the strict correlation relationship between inputs and outputs defined in 1.. For a decision problem, these can be acceptable rates of false-positives and false-negatives over all valid input pairs. They can be specified in absolute terms (i.e. a natural number), or as a bound on the fraction of such to some other quantity – for example constraining Sensitivity and Specificity. For non-decision problems, constrains on absolute or relative error can be included here. Finally, sometimes different requirements for different subsets of the input universe can be formulated (e.g. in case a 3CNF-SAT formula has less than 2 clauses per variable, we require 100% Sensitivity and Specificity, but if it has more than we can settle for 99%).
- **Proof Requirements**. This specifies if the algorithm must provide some sort of additional output which can be used to construct a proof that it is indeed correct for the respective input. For a 3CNF-SAT formula this can be a satisfiable assignment, or a certificate of non-satisfiability (do note for this particular example that not all non-satisfiable 3CNF-SAT formulas may have non-satisfiability certificates of polynomial size). We call any such part of the output **a certificate** (of correctness). Accuracy requirements can be placed on this part of the output as well.
- **Completeness Requirements**. This specifies what kind of behavior the algorithm is guaranteed to have over the input universe. In particular, we say that it is **complete** if it terminates with the required guarantees for all inputs and **incomplete** if it does not do so for some of them (for which it may produce invalid outputs or simply never terminate).
- **Restrictions on size of algorithm**. For some fixed programming language considered, we require that the size of the algorithm produced to solve the problem be bounded from above by some function of the input size. For a general case algorithm, this size must be a constant (however it may be rather large). For a problem of restricted size, it can vary with the input size restriction. Nevertheless, for a particular input size it must have a definite upper bound.

We have deliberately excluded running time and space complexity of the algorithm from the problem definition. This is because for a given problem we will reason about its difficulty in terms of the running-time required to solve it. As with classical complexity theory this can be taken for the Worst-Case, Average Case, Best Case or anything in-between (including "average case in practice"). The memory model we employ is generally the **RAM model**. We typically do not include any mention of space-complexity, since by employing a Perfect Hashing scheme on the accessed memory addresses, space can be bounded from above by the time

consumed, with only a small factor increase in the latter. We also generally but not always constrain the output to be of polynomial size in the input. We take space bounds to mean additional space besides that used by code of the program itself (which can be modified at runtime if needed!). Similarly we can exclude the time required to load the body of the program into memory (even if it may be extremely large – for example exponential in input size).

Other machine-specific runtime requirements (such as number of random bits used or number of qubits employed) can be applied accordingly as in the general case.

Proof requirements are specifically important when we are reasoning about algorithms we either do not know in advance, or about which we do not have sufficient insight to prove that they produce correct outputs for all inputs. For example, for determining the k-th bit of Chaitin's constant [6], for k between 10^9 and 10^9 + 10, an algorithm which simply outputs "1" for all inputs, might in fact be correct for all we know. However, without some insight into why it is correct, this may not be satisfactory enough.

Note that a proof need not always be a requirement. Many image recognition and other algorithms constructed via Machine Learning provide no proof of the correctness of their outputs. In fact, for such algorithms, we currently more or less have little-to-no idea both **why** they work so well in practice, and **when** they work this well (this latter failing has been shown to allow attacks for example against a road-sign recognition algorithm, which produce an image which to a human looks like a clear "STOP" sign, but to the algorithm it is seems a clear "Minimum speed 120 Km/h" sign). This has nevertheless not curtailed their adoption in practice.

Also note that the proof part of the output may be only what is required to complete or generate some larger proof (of potentially much larger size, e.g. exponentially larger) in some format which can convince either a human or, respectively, an automated proof verifier for the problem domain that the output is indeed correct. For a 3CNF-SAT instance for example, a proof of unsatisfiability could be just a small subset of the input variables – small enough to allow exhaustive trial of all possible assignments – which, when the input expression is reduced accordingly it generates empty (impossible) clauses.

The restrictions on the size of the algorithm itself are a novelty specific to finite algorithmics. For the general case, the implicit assumption made by humans in their quest for a solution is that there is a single algorithm (of some fixed size) which solves all inputs properly. The interestingness of our theory and of this paper in general rests on the assumption that some problems admit different algorithms (of potentially different sizes) for different input sizes – and that some may not even admit an algorithm for the general case.

In some cases it can be useful to "break" an algorithm (its source code) into a fixed part, which is the same for all inputs *in the problem space* (similarly to a fixed algorithm for the general case) and a variable part – the "hint" – which may vary with input size.

Definition 4 (Algorithms with Hints)

We define the solution to some problem **Prob** (of either general or restricted size), to consist of a fixed proper algorithm **S(instance, hint)** which takes as input both the instance of the problem and some hint data to produce its output, alongside a function **GEN(n)** which generates the hint for a particular input size **n**. The output for a particular problem instance, is thus **S(instance, GEN(|instance|))**.

We call S a **hinted algorithm.**

Discussion: The advantage of having the GEN(n) function split from the rest of the algorithm's body is that it could be precomputed (note that it takes as parameter the size of the input, not the input itself). Do note that by taking **S** to include a source-code interpreter (a machine simulator) and **GEN(n)** to include some source code, we can describe any algorithm in this fashion.

For general case problems, if we constrain GEN(**n**) to be polynomial in size to **n**, and **S** to run in polynomial time, the algorithms examined will all be contained within the complexity class P/poly. Do note that problems which do not admit P/poly algorithms in the general case (e.g. the hint would grow to super-polynomial size beyond a certain threshold) might very well be solvable efficiently for all sizes.

involved in practice – up to potentially very large ones. Also, P/poly solutions for the general case may be of no practical use for some problems. Determining the hint may take exponential time, may be no less hard than the original problem itself or the P/poly solution may imply no constructive method at all to generate the hint or even determine if a particular hint is adequate. Alternative algorithms requiring much shorter hints in practice might exist, but they might not behave well for arbitrary large inputs thus not making general case problem P/poly. Finite algorithmics can therefore be considered a field tangentially related to, but fully distinct from study of any general case complexity class, including P/poly.

Section 3.2 Finite complexity and its classes

In order to be able to reason easily about relative running times of various algorithms, on the finite case – where regular complexity theory will simply give O(1) – we would like to introduce some additional theory.

The easiest extension of definition of asymptotic approximation of some natural function (from N to N) is to simply introduce an upper bound on the constant hidden by the O, o, or Omega notations.

In the following we take a natural function to mean any monotonically non-decreasing function from natural numbers to natural numbers. Thus we include any function which might represent some running time or space complexity of some algorithms for any input up to a certain size (difficulty).

Definition 5 (Finite complexity with bounded constant and restricted domain)

For two natural functions f and g, some constant natural number c, and two other natural numbers n1 and n0, with n1<=n0, we say that $f(n)=O_{n1..n0}[c](g(n))$ **iff** f(n)<=c*g(n) for all n between n1 and n0 inclusively.

We extend the definition accordingly allow for n0 to be +INF.

Also, if n1 is the minimum possible value in the input universe, we can omit it and specify only n2.

The above definition allows us to describe relative performance of algorithms in some familiar way. For example, for the All-Pairs-Shortest-Path problem, we can say that the complexity of the Floyd-Warshall algorithm [7] is $T(n)=O_{+INF}[100](n^3)$. This essentially means that all of the operations performed by this very

short non-recursive algorithm (incrementing loop variables, dereferencing, comparisons and assignments) are no more than 100 *n^3. This is definitely the case for any n (there are probably less than 20 such operations per n^3).

The shortcomings of the above notation steam from the fact that for finite cases, we have that $f(n)=O_{n1..n0}[c](g(n))$ for any two non-zero functions f(n) and g(n), for some appropriate constant c. Thus, we need to introduce yet more theory for this approach to become useful.

The natural approach is to choose the constant as small as possible (introduce a tight bound).

Definition 6 (Finite complexity with minimal constant and restricted domain)

For two natural functions f and g, and two natural numbers n1 and n0, with n1<=n0, we say that $Const(f,g)_{n1..n0} = c0$ **iff** (i) $f(n)=_{n1..n0}O[c](g(n))$ and (ii) $f(n)!=_{n1..n2}O[c-1](g(n))$.

As before, we allow n0 to be +INF and n1 to be omitted where appropriate

We are now able to reason about an algorithm S in terms of "if it *were* to have complexity g(n), how large would the constant need to be?".

Definition 7 (Apparent relative finite complexity)

For three natural functions f, g, and h, and two natural numbers n1 and n0, with n1<=n0, we say that $f(n) = O_{n1..n0}^{h(n)}(g(n))$ **iff** $Const(f,g)_{n1..n0} <= h(n0)*Const(f,g)_{n1..(n1+n0)/2}$.

We say formally that for the interval n1..n0, the function f appears to have complexity g, within a factor of h.

When the function h is constant, we can write the constant directly.

Discussion: We have essentially constrained that the constant grows from the mid-point of the interval, to the endpoint of the interval with by a factor of at most h(n0).

For a general case algorithm of some complexity g(n), we have that there exists some n0, for which its apparent finite case complexity is also g(n) within a factor of h(n)=1. This follows directly from the fact in the general case, beyond a certain threshold, the constant remains fixed regardless of n.

Definition 8 (Certain finite complexity)

For two natural functions f, and g and two other natural numbers n1 and n0, with n1<=n0, we say that $f(n) = OC_{n1..n0}(g(n))$ **iff** $f(n) = O_{n1..n0}^{1+1/n^2}(g(n))$.

Discussion: We have chosen h(n)=1+1/n^2, such that Product(h(n)) when n tends to infinity is bounded (it is in fact ~3.68 - Wolframalfa was used to compute the limit). This allows us to reason that if $f(n) = OC_{n0}(g(n)) => f(n) = OC_{2*n0}(g(n))$ for all n0 beyond a certain threshold, then f(n) = O(g(n)). Any h(n) with bounded Product(h(n)) when n tends to infinity can be used to replace our choice.

Definition 9 (Polynomial rank of a finite complexity)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $PolyRank_{n1..n0}(f) = k$, **iff** (i) $f(n) = O_{n1..n0}^{2}(n^[k-1])$ and (ii) $f(n) != O_{n1..n0}^{2}(n^[k-2])$.

Discussion: We have chosen h(n)=2, such that Product(h(n)) after log(n) doublings of n0 is bounded from above by n (it is precisely n actually). This allows us to reason that if $Poly_{n0}(f(n)) = k =>$ $Poly_{2*n0}(f(n)) = k$ for all n0 beyond a certain threshold, then f(n) = O(n^k).

Definition 10 (Polylogarithmic rank of a finite complexity)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $LogRank_{n1..n0}(f) = k$, **iff** (i) $f(n) = OC_{n1..n0}(log(n)^k)$ and (ii) $f(n) != OC_{n1..n0}(log(n)^[k-1])$.

We consider only k>=1. If no such k exists, we say that $LogRank_{n1..n0}(f) = 0$.

Definition 11 (Linear finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $Linear_{n1..n0}$, **iff** $f(n) = OC_{n1..n0}(n)$.

Discussion: We have defined the linear complexity class such that it allows a very small growth factor for the constant, as n grows to infinity. So small actually these factors multiplied together are less than ~3.68.

Definition 12 (Polylogarithmic finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $PolyLog_{n1..n0}$, **iff** $LogRank_{n1..n0}(f) < log(n)/log(log(n))$.

Discussion: The value log(n)/log(log(n)) was chosen such that the resulting effective growth rate is linear or below

Definition 13 (Polynomial finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $Poly_{n1..n0}$, **iff** $PolyRank_{n1..n0}(f) < 1+log(log(n))$ and f NotApartine!= $PolyLog_{n1..n0}$.

Discussion: The value log(log(n)) was chosen such that any problem within this complexity class would most likely be tractable for almost all inputs which show up in practice. For example, if we take f(n) to represent the complexity of some algorithm based on its input size, for an input of size 2^64 (~16 Million Petabytes), the exponent in PolyRank(f) would be just 7. Also for an input of mere 1024 size, the maximum exponent can still be 5. This is very appropriate since some interesting problems, like for example Assignment Problem, have general case complexity around these thresholds. If the practical cases for the problem at hand involve n << 1024, the constant 1 in 1+log(log(n)) could be increased to something more suitable, like 2 or 5.

Essentially, if a problem belongs to the polynomial finite complexity class, we can expect that almost surely the associated algorithm will perform fast enough in practice, to make the problem tractable. Thus, the semantic meaning from the general-case Poly class is maintained.

Definition 14 (Semi-Polynomial finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $SemiPoly_{n1..n0}$, **iff** $PolyRank_{n1..n0}(f) < 1+log(n)$ and f NotApartine!= $Poly_{n1..n0}$.

Discussion: The value log(n) was chosen such that any problem within this complexity class would most likely be tractable for a significant number of inputs which show up in practice. For example, if we take f(n) to represent the complexity of some algorithm based on its input size, for an input of size 1024, the exponent in PolyRank(f) would be 11, placing the problem at the

threshold of tractability versus intractability given existing super-computers. Again, if in practice we except that n << 1024, the constant 1 in the 1+log(n) above can be adjusted to something more suitable.

Definition 15 (Exponential rank of a finite complexity)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $ExpRank_{n1..n0}(f) = 1/k$, **iff** (i) $f(n) = OC_{n1..n0}(2^{[n/k]})$ and (ii) $f(n) != OC_{n1..n0}(2^{[n/(k+1)]})$.

Discussion: We are thus describing for a certain n, how large the exponent of 2 needs to be, in order to tightly provide an upper bound for the function. We describe it as a fraction of n itself.

Definition 16 (Exponential finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $Exp_{n1..n0}$, **iff** $ExpRank_{n1..n0}(f) <= 8$ and furthermore PolyRank(f) > 1+log(n).

Discussion: We are taking the exponential finite complexity class to represent everything which is at most about simply exponential in n, which does not belong to any of the previous classes. This is a break from the general case EXPTIME complexity class, where the exponent is allowed to be polynomial in n, not just linear. We have chosen the value 8 instead of 1, to allow functions of the order of n! to fit into this class, up to n ~ 512 - which should be more than enough for anything beyond it to be considered intractable in practice.

Definition 17 (Intractable finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $Intr_{n1..n0}$, **iff** $ExpRank_{n1..n0}(f) > 8$ and furthermore PolyRank(f) > 1+log(n).

Discussion: We are basically naming everything above exponential finite complexity class to be Intractable. In practice, for some small n<110 (for example n<20), problems in this class may still be solvable. Nevertheless, if n is small enough, then the output for all possible inputs can be precomputed and given as a hint to an algorithm under Definition 4. It thus makes sense to expect that in practice anything super-exponential can either be precomputed or be considered intractable.

Definition 18 (Constant finite complexity class)

For a natural function f and two natural numbers n1 and n0, with n1<=n0, we say that $Complexity_{n1..n0}(f)$ Apartine= $Const_{n1..n0}$, **iff** (i) $f = OC_{n1..n0}(c0)$, for some fixed constant c0 and (ii) LogRank(f)<=1.

We also say in this context that c0 is the constant rank, or $ConstRank_{n1..n0}(f) = c0$.

Discussion: Constant finite complexity is quite similar to general case constant complexity. Do note however that the constant rank obtained in practice, might in fact be hiding some small growing non-constant function for the general case. Furthermore, when reasoning about complexity with regard to different upper bounds n0, the constant c0 must remain fixed – independent of n0.

Section 3.3 Finite complexity hierarchy

Given the definitions in Section 3.2, for any given natural interval n1..n0, with n1<=n0, we can classify all the natural functions f, into precisely one of the following classes.

1. $Const_{n1..n0}$
2. $PolyLog_{n1..n0}$
3. $Linear_{n1..n0}$
4. $Poly_{n1..n0}$
5. $SemiPoly_{n1..n0}$
6. $Exp_{n1..n0}$
7. $Intr_{n1..n0}$

The higher the level a function occupies in this hierarchy, the less tractable we expect a problem admitting an algorithm of this complexity to be.

We are now armed with the possibility to describe the variation in the classification of a particular natural function f, as we allow the input domain to expand.

Definition 19 (Threshold of complexity class explosion)

For a natural function f, a complexity hierarchy level l and a natural number n1, we say that $Explode_{n1}(f,l)$ = Min {z | f belongs to some complexity class of level at most l for any n1..n0, with n0<z, but does not for n1..z}. If the set is empty, we take the marker value +INF.

We can represent the hierarchy level by either its index above or the corresponding name (PolyLog,Linear,etc.).

Discussion: We are taking the explosion threshold for a function f, to be the minimum n0 beyond some small n1 value, where the function f will belong to a complexity class strictly above the respective level.

For practical considerations we can limit the smallest value in the input domain of f to some n1, large enough to be non-trivial. This value n1 can be fixed apriori (for example n=16 seems a promising candidate) or fixed in relation to a particular problem domain. For example, for parameterized complexity 3-CNF-SAT in n – the number of variables, anything below n=10 can be considered trivial. The bottom line in choosing n1 > 1 is to exclude some anomalous behavior of a function around the very start of its domain.

Do note that the function f might have different $Explode_{n1}(f,l)$ values, for different n1s. In fact, for some level l, there could be some n1 beyond which the explosion threshold is +INF.

Definition 20 (Threshold of complexity class collapse)

For a natural function f, a complexity hierarchy level l and a natural number n1, we say that $Collapse_{n1}(f,l)$ = Min {z | f belongs to some complexity class of level at most l for any 1..n0, with n0>=z and n0 <= n1}. If the set is empty, we take the marker value +INF.

We can also take n1 to be +INF.

Discussion: We are taking the collapse threshold for a function f, to be the minimum n0 beyond which f belongs to a certain complexity class or better, at least for up to another higher limit n1 (which may be +INF).

Section 3.4 Finite Algorithmics and Problems

When attempting to solve a computer science problem, we shall consider the following as input:

1. The Full Problem Statement according to Definition 3.
2. The interval n1..n0 of input size (or other difficulty constraining parameter) where practical instances of the problem lie.
3. The worst acceptable finite complexity class

for running time required by desired algorithm. We can reason in terms of worst-case/best-case/average-case either for the entire domain or simply for the instances which occur in practice. We can describe this by requiring that the running time belongs to a class up to a certain level of the finite complexity hierarchy described in Section 3.3. In practice this is results directly from point 2 above, the reasonable timeframe in which a solution to such a problem is useful and the speed of existing hardware. If the produced algorithm allows high degree of parallelism, then the intended cluster size can also be factored in. See Appendix 1 for approximations considering current state of the art hardware.

4. The amount of time which can be allotted to actually discovering the solution. When reasoning about a potentially variable upper input bound n0, this can also be expressed in terms of finite complexity class, with regard to the difficulty parameter n0.

5. Some collection of source-code for known algorithms and data structures.

The desired output for will consist of one of the following:

1. A Hinted Algorithm as per Definition 4, S and some fixed hint hintn0.

2. A Hinted Algorithm S and another algorithm GEN which can generate hintn for any n in n1..n0. We call the GEN algorithm, the Hint Genesis Algorithm.

3. A Hinted Algorithm An0 which generates the pair of algorithms from point 2 above, alongside its fixed hint - hintAn0. We call such an algorithm the Generator Algorithm.

When reasoning about the relative efficiency of finite case algorithms, we shall consider them both in terms of running time complexity and complexity of hint size, with relation to input size (difficulty). Thus, we can say that an algorithm is T(n)/G(n) efficient, where T(n) is running time and G(n) is hint size. The hint, as well as the program code, is assumed to be already loaded in memory. For example we can reason about a certain algorithm / problem saying it has finite case complexity **Poly**/**PolyLog** on the domain of interest.

The most straightforward formulation of the above is the following: "Given what we already know, find an efficient enough, potentially hinted, algorithm which solves the full problem statement on any input of size (difficulty) within the interval n1..n0, or show how one can be constructed."

It is simple to note that an output of type 2 above can be precomputed from one of type 3, by running the algorithm A. Furthermore an output of type 1 can be precomputed from one of type 2. It is useful however to reason about these options separately, since the precomputation step between the types may not always be polynomial.

Two more inputs might be useful for some problems for which there are known algorithms to solve them for some particular kinds of inputs (e.g. of small input size). These are:

*** The verifiability thresholds given existing algorithms**. Namely:

- v1: The answer for ALL instances of this input size (difficulty parameter) or below can be precomputed in feasible time.
- v2: The answer to ANY instance of this input size (difficulty) or below can be determined within feasible time.
- v3: The answer to MANY instances of this input size (difficulty) or below can be determined within feasible time.
- v4: The answer to SOME instances of this input size (difficulty) can be determined within feasible time. For instances beyond this input size, it is considered highly unlikely for then-existing state of the art to be able to solve any of them.

***** Golden **Data**. For instances of input size (difficulty) between v1..v4, some already existing correct input/output golden data might be offered. This can include:

- Tests with precise output.
- Tests with lower and/or upper bounds on the correct output.

Golden data might be useful to save running-time during testing, by avoiding the need to run the original algorithm which generated it (which might have consumed a lot of time or resources initially).

- **Efficiently solvable via a known algorithm (ES)**. Problems include string pattern-matching, shortest paths and many, many others. In fact most of the problems humanity has tackled are now included in this category. The state of the art algorithms known to the scientific community are sufficiently efficient to solve all practical instances of such problems.
- **Tractable but insufficiently so (TR)**. For some problems, like Assignment Problem, Multidimensional Range Queries we know sufficiently efficient algorithms to solve any instance of them relatively quickly, but for some practical applications we need even faster ones. We may not even know if such algorithms exist, as the gap between the lower-bounds and the upper-bounds, complexity wise can be quite large still.
- **Intractable for large input sizes, but tractable for small ones (PTR)**. For problems such as Prime Factorization, Discrete Logarithm, NP Complete problems like Boolean Formula Satisfiability, Knapsack problem and others, an algorithm for solving them precisely is known, but the best one is still very inefficient (largely in terms of running time), thus making it suitable only for small input sizes. Some problems in this category (especially some NP-Complete ones), might fall in the TR category for some practical applications, when a sufficiently accurate approximation algorithm is known, when the practical input sizes are small, or when the practical instances have some other trait (known or unknown) making them easier than the general case (like having a small target sum for the knapsack problem, or having a small number of clauses per variable for 3-CNF-SAT).
- **Intractable because of assumed hardness (ITRA)**. For problems in this category, no algorithm is known which solves any instance but those of trivial size and it is strongly suspected that none exists, because they belong

to a certain complexity class. Problems such as Quantified Boolean Satisfiability which belong to the complexity class PSPACE-Complete are believed not to be solvable in polynomial time, and be harder still than even NP-Complete problems. However it is not known if this is so or not. Furthermore, #P-Complete problems like #SAT are also believed to be hard. But this is yet again still unknown.

- **Intractable and mysterious (ITRM)**. There are problems - like determining the encryption key used to encrypt a known plain text using AES given the cypher output - which are not known to belong to a specific presumably hard complexity class. Nevertheless, they are generally regarded to be intractable by mere fact that a large number of researchers have spent time thinking about them and yet no efficient algorithm has been determined.
- **Truly Intractable (IT)**. Some problems, like Halting Problem, Busy Beaver, Kolmogorov Complexity (very useful in compression and encryption), word problem for semi-Thue systems, determining the bits of Chaitin's constant to non-trivial precision and many others have been proven to be intractable in the general case. That is, no algorithm exists which solves them. This nevertheless does not necessarily make them intractable for bounded-size input. Two-symbol Busy Beaver game for example has been solved precisely for up to 4 states [5]..

Finite Algorithms aims to provide efficient algorithms for problems in TR, PTR, ITRA, ITRM and IT classes but only for the finite cases which occur in practice, without necessarily solving or giving a definite negative answer with regard to a solution for the general case. Furthermore, known algorithms for general-case problems (in any tractability category) can be used in the automated or semi-automated quest for efficient ones for the finite case.

Definition 20a (Complexity of Solving a Problem)

Given a particular finite case computer science problem Prob, specified by the inputs 1,3-7 in this section (excluding the actual finite limits), we refer to complexity of solving this problem, as the complexity of some algorithm A(n) which given a natural number n, generates the source code of some hinted algorithm, along with its hint (output of type 1) for any n <= n0 (input 2).

In case we are interested in solving the problem for any upper input size (difficulty) bound, we can take the input 2 bound n0 to be +INF and allow A to take this special value for its single parameter.

We reason about the complexity of solving a problem in terms of complexity of the corresponding algorithm A.

For some problem Prob (potentially finite case) we denote **Complexity(Prob, n0)** the complexity of the most efficient algorithm which solves the problem for any n up to n0.

Discussion: The complexity of solving a problem can be thought of essentially as the running time of some algorithm which runs on some machine (e.g. a regular computer) which produces the source code required to solve any instance of such problem, up to some upper input (difficulty) bound which itself is below some n0. Note that solving a computer science problem is in itself a computer science problem, to which we can apply the entire theoretical framework presented.

4. Results

We now proceed to present some elementary results of high importance derived within the theoretical framework introduced in Section 3.

Section 4.1 Relationships between complexity classes, finite and general

In this subsection we present basic relationships between finite case and general case complexity classes for natural functions.

Section 4.1.1 From finite case complexity to general case

Theorem 21 (When finite case complexity implies general case complexity)

1. If f(n) = OC_{n0} (g(n)) for some n0, and also for any n'>=n0, we have that f(n) = $OC_{n'}$ (g(n)) => f(n) = $OC_{2*n'}$(g(n)), then f(n) = O(g(n)).
2. If PolyRankn0(f) = k, for some fixed natural numbers n0 and k, and also for any n'>=n0 we have that PolyRankn'(f) = k => PolyRank2*n'(f) = k, then f(n)=O(n^k).
   If such n0 and k exist, we can say that f belongs to the general case polynomial complexity class.
3. If $LogRank_{n0}$(f) = k, for some fixed natural number n0 and k, and also for any n'>=n0 we have that $LogRank_{n'}$(f) = k => $PolyRank_{2*n'}$(f) = k, then f(n)=O(log(n)^k).
4. If f Apartine=$PolyLog_{n0}$ for some n0 and also for any n'>=n0 we have that f Apartine=$PolyLog_{n'}$ => f Apartine= $PolyLog_{2*n'}$, then f(n)=O(n).
   If such n0 exists, we can say that f is grows at most Linearly. Depending on its exact PolyRank, it may in fact grow just polylogarithmically.
5. If f Apartine=$Linear_{n0}$ for some n0 and and also for any n'>=n0 we have that f Apartine=$Linear_{n'}$ => f Apartine=$Linear_{2*n'}$, then f(n)=O(n).
   Like above, if such n0 exists, we can say that f is grows at most Linearly.
6. If f Apartine=$Poly_{n0}$ for some n0 and and also for any n'>=n0 we have that f Apartine=$Poly_{n'}$ => f Apartine=$Poly_{2*n'}$(f), then f(n)=O(n^log(log(n))).
   If such n0 exists, we can say that f = O(n^log(log(n))). Note that this is strictly speaking superpolynomial, but barely so. Also, note that it is sub-exponential.
7. If f Apartine=SemiPolyn0 for some n0 and and also for any n'>=n0 we have that f Apartine=SemiPolyn' => f Apartine=SemiPoly2*n', then f(n)=O(n^log(n)).
   If such n0 exists, we can say that f = O(n^log(n)). Note that this is strictly speaking superpolynomial, however also sub-exponential.
8. If f Apartine=Expn0 for some n0 and also for any n'>=n0 we have that f Apartine=Expn' => f Apartine=Exp2*n', then f(n)=O(2^n).
   If such n0 exists, we can say that f = O(2^n), which is part of the EXP complexity class.
9. If f Apartine=$Const_{n0}$ for some n0 and and also for any n'>=n0 we have that f Apartine=$Const_{n'}$ => f Apartine=$Const_{2*n'}$, then f(n)=O(1).
   If such n0 exists, we can say that f is of constant growth rate.
10. If there exists an infinite number of natural numbers n0, such that Exploden0(f, Exp)<+INF => then

f(n)=Omega(2^n).
This means f belongs to EXP or worse.

Proof: Proving statements 1-9 involves straight forward induction and computation of the limit to infinity of the Product of the allowed constant growth rates under each corresponding definition. While their significance is crucial, the proof is trivial enough to be omitted from this paper. Proof of statement 10 is by contradiction, showing that there can exist no fixed constant hidden by an o(2^n) notation. Again, it is considered a simple exercise and is excluded from this paper.

Corollary: Statements 1-9, remain true if for some complexity level l, the hypothesis is replaced by $Explode_{+INF}(f,l) = +INF$. Also they remain true, if the induction hypothesis of the second part is extended to refer not only to n', but to all n0 <= n'' <= n'.

Discussion: The corollary above gives a direct criterion for converting between finite and general case complexity classes, when possible.

Theorem 22 (When finite case complexity excludes general case complexity)

For any natural function f and any finite complexity level l, if there exists an infinite number of n0 such that $Explode_{n0}(f,l) <+INF$ then f belongs to a complexity class *worse* than the corresponding general case complexity given for that level by Theorem 21.

Proof: Like for statement 10 for Theorem 21, the proof is by contraction, showing that no fixed constant can exists hidden by the O notation for the corresponding general case complexity class. We consider it rather trivial and omit it form this paper.

Discussion: The theorem gives a direct criterion for excluding a general case complexity for a function f, about which we know how it behaves in practice and we are also able to reason that there will be infinitely many lager values on which it continues to behave as such in terms of growth rate. Do note we require that an infinite number of n0 exist. It may be that the function f has smaller finite complexity for any small enough finite interval. However if there are an infinite number of suitable n0, once the bounds of that interval are allowed to grow sufficiently large, the complexity always explodes.

Theorem 23 (Precise determination of general case complexity)

For any natural function f and any finite complexity level l, f belongs to the corresponding general case complexity class under Theorem 21, **iff** there exists an n0, such that $Collapse_{n0..+INF}(f,l) = k$, for some fixed natural number k>=n0 and $Explode_{k..+INF}(f,l+1) = +INF$.

Proof: The proof consists of direct application of the definitions of Collapse and Explode to reduce to a proper application of Theorem 21. We consider it trivial enough to be omitted from this paper.

Note that we may not always know enough about a function to be able to apply either of the above theorems. Also, do note that even if a function belongs to a favorable general case complexity class, it does not mean it is solvable in practice, as its threshold $Collapse_{n0}(f,l)$ may be beyond the largest instances of practical applications.

Analogously, even if a function belongs to a certain unfavorable general case complexity class or worse, it may still be very solvable for all cases of practical importance. Its threshold $Explode_{+INF}(f,l)$ may be larger than some n0 which is the maximum size of practical instances and also $Collapse_{n0}(f,r)$ for some r<l may be small enough to make the problem practically tractable.

Section 4.1.2 From general case complexity to finite case

Theorem 24 (General case complexity generally implies finite case after some threshold)

For any natural function f which has general complexity O(g(n)) and Omega(h(n)), the following statements are true:

1. There exists some n0, such that for all n>n0, $f(n) = OC_n(g(n))$.
2. For any complexity class level l with growth rate no smaller than g(n), then $Collapse_{+INF}(f,l) < +INF$ and also there exists some n0, such that $Explode_{n0}(f,l) = +INF$.

If h(n) grows faster than the functions in complexity class l, then $Collapse_{+INF}(f,l) = +INF$ and also there exists some positive n0, such that $Explode_{n0}(f,l) < +INF$.

By growth rate of functions in a finite complexity class l, we refer to the asymptotic growth rate of the formula representative of such class, given by Definitions in Section 3.2.

Proof: The proof is by contradiction, following the direct application of the definitions of general asymptotic growth notations. We consider the proof trivial enough to be omitted.

Discussion: The theorem implies that having determined some bounds for the general case complexity allows us to say that after some threshold, those bounds will characterize the finite case as well.

Section 4.1.3 Relationships between finite case complexity classes

Relationships between finite complexity classes of different problems are trickier to characterize than in the general case. This is because in case of reduction from one problem to a number of applications of some others, it actually matters precisely how many applications there are of each of those other problems and also what the input size (difficulty) bounds for those are. As such, for example a polynomial number of applications of a solution of complexity class $Poly_{n0}$ might very well result in the PolyRank of the main algorithm to exceed the log(log(n0)) upper-threshold for the $Poly_{n0}$ class. Nevertheless, for a particular candidate algorithm we can definitely characterize its actual complexity with regard to the problems to which the solution is reduced, using the LogRank, PolyRank and ExpRank values.

Theorem 25 (Reductions between finite case problems)

The following statements are true:

1. Up to n0^k applications of an algorithm of $PolyRank_{n0} = j$ results in an algorithm of complexity $PolyRank_{n0} = k+j$.
2. Up to log(n0)^k applications of an algorithm of $LogRank_{n0} = j$, results in an algorithm of complexity $LogRank_{n0} = k+j$.
3. Up to 2^(n0/k) applications of an algorithm of $ExpRank_{n0} = 1/j$, results in an algorithm of complexity $ExpRank_{n0} = 1/k + 1/j$.

Proof: Again we omit proofs as they are mere algebraic symbolic

multiplications of the formulas corresponding to the definitions of the respective classes.

There are of course more interesting reduction theorems. The core aspect for reductions within some finite interval n1..n0 is for the resulting constant, expressed as function of n0 and (n1+n0)/2 respectively to allow the candidate algorithm to fit the definition of a certain complexity class. This generally involves computing the new LogRank, PolyRank or ExpRank values. But theorems describing relationships between lower and higher finite complexity classes can also be interesting. We leave such as open questions for further research.

Observation 26 (Variable finite complexity)

For some complexity class level l, and a (potentially infinite) sequence of increasing natural numbers a1,a2,… there exists a natural function f, such that for every i, we have that $Explode_{a[i]}(f,l)=a_{i+1}$.

Discussion: Essentially it can be that some function has infinitely many points where its finite complexity is small enough, however it never collapses permanently to such a favorable complexity. A function which grows extremely fast on successive powers of two, but very slowly in-between is one such example. In practice, it might be that a problem's optimal complexity varies wildly from one input size (difficulty) to another, within the bounds of its general-case complexity (if such a bound exists).

Finally, when solving a problem on the finite case, the data in Annex 1 which estimates the highest upper bounds for tractability for various complexity classes can prove of general interest.

Section 4.2 Automated and assisted solving of computer science problems on the finite case

In this subsection we present basic relationships between finite case and general case complexity classes for computer science problems. While in Section 3.2 we introduced the concept of finite complexity for natural functions, here we apply those concepts to refer to functions which describe the running times of algorithms and respectively of hint sizes.

Section 4.2.1 General Approach

Since Sections 4.1.1 and 4.1.2 refer to relationships between finite and general case complexity of natural functions in general, the results there apply both to ones representing the running time of an algorithm as well as those representing the size of its hint (under Definition 4).

We adopt the notation f(n)/g(n) from general complexity classes (e.g. P/Poly translates to $Poly_{n0}/Poly_{n0}$) to reason about finite complexities analogously. Under Definition 4 as well as the discussion in Section 3.4 regarding output, when solving a problem for the finite case interval n1..n0, the actual program itself, S, will be of constant size. In fact, a program of fixed size exists such that it solves all intervals n1..n0 for some potentially distinct and perhaps very large $Hint_{n0}$: namely one which includes an universal machine simulator (e.g. a registry machine simulator).

The manner in which we choose to consider splitting the actual algorithm for a finite case problem between the fixed part and the hint is as much art as it is science. Knowledge of the problem domain as well as trial and error may lead to various choices in this regard. Nevertheless, the value of finite algorithmics lies in the conjecture that some problems do not admit a sufficiently efficient algorithm for the general case (or that identifying such is not possible), but do in fact admit some (maybe distinct) algorithms for finite universes of inputs.

In our quest to identify a practical finite case solution to a problem on interval n1..n0, or determine that such does not exists (or that it is as hard as the general case), we can take the following approach.

Approach 25 (Automatic Solving of a Problem Instance)

Given some problem Prob, specified by inputs 1-7 described in section 3.4, we can construct a fixed algorithm to solve it which takes the following rough steps:

1. **It considers some enumerable (potentially finite) family F of fixed hinted algorithms S**. This family can be specific to the problem domain of Prob – it can essentially describe "what we can expect the source code of some algorithm which solves it to look like". Each algorithm in this family is hinted, as per Definition 4.
2. **It maintains some internal state s, describing the current status of the search for a solution**. This state can be of rather large size, so long as it fits the space complexity bounds imposed on the automatic solving algorithm itself. It may consist, for example, of the following:

a. Promising Algorithms and methods of Hint generation.

b. Algorithms and Hints which are adequate for some specific input sizes (difficulties).

c. Statistics on promising algorithms, hints and hint generation methods collected in step 3e below.

d. Information on negative results (inadequate algorithms / hints / hint generation methods), analogous to points a)-c) above.

3. **While a solution is not found, it explores some more, by doing the following**:
   a. Pick some algorithm S from the family F, based on the internal state s.
   b. Choose a potential hint hintS for S, from the finite set of potential hints, by some method GENS.
   c. Evaluate S preliminarily by running it on several inputs within the relevant domain, on which it has not been run before, using the hintS. For example, S can be run on Golden Data tests, for increasing input sizes (difficulties) from the ones in v1 up to v4.
   d. If S takes longer than the upper bound for the desired complexity on a particular input, or its course of execution seems unpromising, halt it (to be potentially resumed later). Note that given the theoretical framework concerning finite complexity classes, for a given input size (difficulty) any desired complexity translates to a precise upper bound on the running time (or number of operations of the algorithm). The constant is never "hidden" in finite algorithmics.
   e. Collect the following data with regard to the execution of algorithm S on hintS and the relevant test cases:

- Running Time and Space Consumed for some input.

Statistics regarding

- Input/Output correlations. These can include error rate, such as Specificity and Sensitivity and so on.
- Full or partial snapshots of its internal memory state at

various runtime moments. These are taken in the hope that correlations can be made between them and the desirability of the overall behavior of some algorithm

f. Use the data collected above to update the internal state describing the status of the search. Naturally, the data could be synthesized and aggregated before or after update of the internal state, in order to reduce its volume.

4. **Once a sufficiently adequate pair of algorithm S and hint generation method GEN_S are identified, it outputs them (i.e. their source code and for GEN_S its hint) as the solution.**

Discussion: The approach essentially considers algorithms and hints in some arbitrary order, tests them on available inputs and finally chooses the first one which is adequate enough. The art of producing an efficient implementation of this approach for some problem domain lies particularly in identifying some good way to choose this arbitrary order.

By introducing Approach 25 we are effectively shifting attention of researchers from focusing solely on understanding correlations between input/correct output pairs within a problem domain, to focusing on correlations between structure and hints of algorithms and their relative performance / adequacy with regard to that domain. This approach can be reminiscent of the domain of AI and Machine Learning.

Approach 26 (Family of algorithms for any problem domain)

For Step 1 of Approach 25 above, the following family of algorithms can be considered.

Given some object-oriented programming language grammar (e.g. C#,Typescript, etc.), consider only source codes (algorithms) which satisfy the following conditions:

1. They include as reference some or all types corresponding to data structures and algorithms included in part 5 of the input, as described in section 3.4. The algorithms which are not meant as general purpose data structures, are to be wrapped in a type called a Solver, which is essentially a data structure with a single Query() method producing the output. Part 5 of the input can be restricted to only what researchers believe to be relevant to the problem domain.
2. They define at most 20 new types (interfaces + source code implementation).
3. Each defined type contains exactly most 20 public methods.
4. Each defined type contains at most 20 private methods.
5. Each defined type includes at most 20 internal variables (which can be collections such as lists of dictionaries over other types).
6. Each defined type includes at most 20 “magic constants”, which are actual values of some type.
7. Each method takes at most 20 (typed) parameters.
8. Each method defines at most 20 local variables (for all levels of imbrication).
9. Each executable line of code, consists of one of the following:

a. A statement, which can be either:
   i. An assignment to some variable in scope from the result of the evaluation of an expression over variables in scope.
   ii. An evaluation of an expression over variables in scope without storing the resulting value.
   iii. A loop-related execution flow control directive, such as break or **continue.**

   By the definition of expression above we include public member methods invocations on some of the underlying parameter values, as well as parameters lists buildups using operators such as comma (,). Type properties or parameterless methods are represented as methods taking no parameters.

b. A conditional branching of the form IF(expression) then statement else statement.
c. A conditional loop of the form WHILE(true) then statement. This allows for both initial and final loop condition checking, via inclusion of an appropriate IF statement.
d. A return statement of the form RETURN(expression).

10. The expressions which occur throughout all methods are defined globally. Source code within methods uses them by specifying the corresponding IDs and the required assignment. The operators within an expression can be only invocations of methods on underlying types, or the parameter list builder (e.g. comma). All types are boxed. As such, for example a+b is represented as a.Invoke(“AddWith”, b).
11. There are at most 20 “heavy” expressions globally, which are defined on between 6 and 20 variables.
12. There ar at most 400 “light” expressions globally, which are defied on at most 5 variables.
13. Values are passed by reference to invoked methods. Thus, ther actual value can be changed by the method if it so choses or needs. Non-destruction of the input can be achieved via cloning.
14. No method has an imbrication level above 5 (e.g. IF contained within another IF contained within a WHILE, and so o). Algorithms which involve imbrication levels beyond this value can actually be rewritten to use private method calls.
15. There are at most 20 “heavy” methods globally, which have an imbrication level of either 4 or 5. All the others have imbrication level at most 3.
16. There are at most 400 lines of code for any single method body.
17. There are at most 20000 lines of code for all the methods’ bodies together.
18. There are at most 400 global magic constants, besides the ones allotted to each type.
19. There exists a single type, which is the actual Solver for the problem at hand which defines the following interface:

    a. Initialize(hint) – a method which initializes the solver with the corresponding hint, allowing any precomputations if required.
    b. Query(instance) – a method which returns the output for specified instance of the problem.

Discussion: The above family of fixed algorithms is understood by us, the author, to include essentially everything that a human researcher could reasonably discover on his own with regard to any problem domain. In fact, to the best of our knowledge, almost all (if not all) currently known algorithms for solving **any** general case problem can be effectively mapped to some member of this family. The value 20 which appears repeatedly was chosen arbitrarily to provide a rather generous upper bound. It is most likely that a lower value, of something like 5 would still allow sufficient coverage of everything which could be of interest to researchers. Furthermore, for specific problem domains, just a fixed algorithm of much smaller sophistication can be hypothesized to solve the problem on sufficiently large input sizes (difficulties), given the right hint. In that case, the quest for a solution simplifies to the quest for a proper Hint.

Do note that the family of algorithms described in Approach 26 is universal. Namely, it includes algorithms which simulate a registry machine. Given also that the algorithms are Hinted, it is possible to effectively circumvent any limits placed on imbrication depth, source code size or anything of the like, simply by moving the actual algorithm to the Hint. This however, runs contrary to the manner in which we suggest this approach has value. The Hint should be specific to the problem domain. For some problems (like 3CNF-SAT for example) it could include some bending of this rule – for example to allow for formulas specific to a particular input size to be evaluated in the context of conditional branching –. Nevertheless, the Hint should be very specific to the problem domain itself and in fact also to the particular finite bounds for which a solution is sought out.

For some very hard problems, it could be that allowing the program to modify itself essentially by incorporating parts of the hint or of the problem instance as part of its de-facto code might prove interesting avenues for exploration. Such problems might include simulation of human consciousness, resolving halting problem for finite cases and perhaps others not yet considered by humanity. Nevertheless, allowing a program to essentially alter itself should most likely not be the initial main focus of research within finite algorithmics.

Once a candidate fixed algorithm S which shows sufficient promise on small problem instances has been identified – for example one which works correctly and efficiently on all inputs up to the v1 threshold and on all other inputs on which it had been tested -, the only remaining issue is to determine a suitable hint for it for larger instances. The following approaches can be taken either alone or in combination:

- **Exhaustive Hint enumeration**. For problem instances of small size, this could potentially be done, especially if a convenient finite complexity class is suspected for the hint size with relation to input size. This could also be used as a starting point for problems where we are essentially clueless as to what a proper hint might be. For small enough instances, a starting point could be Deterministic Finite Cover Automata [8] which correctly recognizes the finite language of some decision problem – thus allowing all cases to be answered correctly and efficiently.
- **Inductive Hint construction**. Identify an algorithm which, given some hints for problem instances of smaller input size (difficulty), it constructs one for instances of larger size. This algorithm itself could be sought out using Approach 25 and 26. We would suggest however that it takes itself a hint of very small constant size (if any at all). The input on which it operates is the set of hints for smaller input sizes.
- **Adaptive Hint construction**. Use existing and future techniques to determine causal correlations between events – such as Deep Learning models -, to analyze the data collected in step 3e of Approach 25 (including correlations between parts of the memory state at runtime and ultimate behavior of the algorithm – correctness, timeout, etc.) to hypothesize, test and prioritize potential hints over others, as well as to eliminate obviously or apparently invalid ones. These techniques can also be used to alter and combine successful hints so that the search for an adequate one converges faster on an acceptable solution.
- Tapping Randomness. Include randomness in decision making with regard to which variation to try next or how to prioritize approaches. Many surprisingly efficient SAT Solvers today employ it.
- **Multiple Arm Bandits**. Ultimately, the quest for a proper hint, using some automated method, involves allotting a finite resource – running time – between several existing or new avenues of exploration: be it an existing hint is to be tried out on more cases for gathering further data, one is to be transformed by some rule or combined with another under some other method or some other random variation is to be introduced. Sometimes, the expected benefit of trying a particular method or transformation over another is unclear or cannot be known in advance. Taking such decisions, including with regard to how to balance exploration and exploitation pertains to a well-known computer science problem called Multiple Arm Bandits (see [9] for example).

Finally, at all stages of the approaches described above, a human researcher could intervene and make adjustments based on his own creative and rigorous judgment, potentially leading to further speed-ups in the search for a solution.

Note the that the approach described in this sub-section includes any currently known Machine-Learning algorithm, including Deep Learning with multiple number of layers in some neural network: The output of the learning is in our terminology the Hint to the algorithm, which, itself is merely a simulator of a neural network. The actual learning algorithm (e.g. Reinforced Learning) is just one potential method to be used in line 3.f) of Approach 25. Any such currently known learning algorithm is itself contained with the finite family of algorithms proposed by Approach 26.

Furthermore, Approach 25 could be refined to include the theory of Schmidhuber related to Gödel machines [10]. This can be applied either with regard to proving correctness or to simply speed up the search for an optimal algorithm.

Section 4.2.2 Elementary Results

In this sub-section we present some elementary results pertaining to finite complexity of computer science problems, considering the approaches described in Section 4.2.1.

In this sub-section we limit our attention to problems which have polynomial or smaller output sizes. This includes all decision problems (where the output is a single bit). The restriction that the

output size be polynomial is most of the time natural, since outputs of super-polynomial size would, in themselves, require a long running time to merely write out.

Observation 27 (Reduction to decision problems)

Any problem which has an output of polynomial size in the input can be reduced to a liner number of applications of a decision problem.

Proof sketch: Consider the decision problem asking "does there exist any correct output for this input instance, which is smaller than some natural number x?". By using binary search over the output space for a given input instance, one can, in a number of probes linear in the input size (logarithmic in the size of the output universe) determine some correct output using the solution to the decision problem above.

Theorem 28 (Every verifiable problem admits a $Poly_{n0}$/Exp_{n0} finite case algorithm)

For any problem **Prob**, which admits a general case verification algorithm **V** able to decide for any (input,output) pair it the output is correct for the given input, there exists a hinted algorithm **S**, such that for any finite case upper bound on input size **n0**, there exists an appropriate hint enabling **S** to solve **Prob[n0]** correctly for all inputs and in time $\mathbf{Poly_{n0}}$.

Proof sketch: Given a Verification algorithm, one can immediately construct an inefficient general case algorithm which produces a correct output for any given input: Simply enumerating all potential outputs for an input and using the verification algorithm to pick the correct one. Given this, the correct output can be precomputed for any input of size up to n0. There are 2^(n0+1) – 1 such potential inputs. The corresponding correct outputs could then be stored directly as Hint. An algorithm which, given an instance in this 2^(n0-1) universe, simply looks up position where the correct output is stored in the hint, using binary search, takes log(2^(n0-1)-1) < n0 steps to identify such. It can then merely output the corresponding output, which is $Poly_{n0}$, resulting in a total running time within the $Poly_{n0}$ finite complexity class. *Corollary*: Any verifiable decision problem admits a $\mathbf{Linear/Exp_{n0}}$ finite case algorithm for any **n0**.

Since a decision problem has constant output size (namely 1 bit) only the linear time taken to identify the correct index of the output determines complexity.

Discussion: Note that this method gives an algorithm to determine the correct output, not also to prove its correctness. Nevertheless the correctness for this particular construction can steam from the construction of the hint itself: The output is correct, because given the manner in which algorithm S and its Hint were constructed, it cannot be incorrect. However, given some arbitrary hint determining if it is indeed adequate is not implied by this theorem.

Complexity: For decision problems, the above approach involves merely 2*(2^(n0+1)-1) applications of the verification algorithm **V**. For problems of polynomial output size this is multiplied by the maximum size of the output universe, which is of the order 2^O(n0^c) for some constant c. In both cases, executing this approach directly for any n0 has general case complexity within EXPTIME, so long as the algorithm V is itself within this class (e.g. it is in P). While not generally considered tractable, EXPTIME is not the worst general case complexity class out there.

Theorem 29 (Solving verifiable problems optimally in the finite case is computable)

For any problem **Prob**, which admits a general case verification algorithm **V** able to decide for any (input,output) pair it the output is correct for the given input, for any natural number **n0**, there exists an unhinted algorithm which determines the optimal algorithm for solving Prob[**n0**].

Proof sketch: Any implementation of Approach 25, which exhaustively enumerates all algorithms and potential hints of joint size up to at most the size of the algorithm and hint resulting from the application of Theorem 28 for **Prob[n0]** will consider the optimal running time algorithm among them. Hint sizes outside Exp_{n0} are pointless, since an algorithm linear in input+output size (thus optimal) exists for such a hint. Considering algorithms which, together with their hints, are of size over Exp_{n0} is again pointless: the algorithm constructed in Theorem 28 is of very small constant size, thus allowing the joint size to remain within Exp_{n0}. As such, trial of algorithm/hint pairs only within these limits suffices.

Complexity: The complexity of employing this approach without further refinement is as follows. For every candidate algorithm/hint combination in the input universe, verifying its correctness can take at most **2^(n0-1)+1** applications of the verification algorithm V and similarly many applications of the candidate algorithm. For decision problems, the input universe itself is only slightly (by a very small constant) larger than **2^(n0-1)+1**. So this reduces to ~2^(2*n0) applications of V and the algorithm itself. These are within EXPTIME if V and the candidate within EXPTIME themselves. For problems with larger, but still polynomial sized outputs, this is multiplied by some factor 2^(O(n^c)) for some constant c, representing the increased input universe size. This keeps the complexity within EXPTIME, so long as V and the candidate are themselves within EXPTIME. The universe of potential algorithm/hint pair is doubly exponential in n0, making the total complexity no worse than 2EXP, which is within ELEMENTARY, thus computable.

Discussion: In practice, the input universe will be much smaller. Most likely only hints of $\mathbf{Poly_{n0}}$ or $\mathbf{SemiPoly_{n0}}$ size will be considered and the family of fixed algorithms for a problem domain will consist of just 1 or sometimes a very small subset of those described by Approach 26. This reduces complexity to at most EXPTIME. Also, most candidate algorithm/hint pairs will not be allowed to run beyond some desired complexity (most likely $Poly_{n0}$ or $SemiPoly_{n0}$) and will not be run on all possible inputs for verification purposes, resulting in further running time reductions.

Theorem 30 (Solving verifiable problems optimally in the general case is computable if they have a determinable collapse threshold)

For any problem **Prob**, which admits a general case verification algorithm **V** able to decide for any (input,output) pair it the output is correct for the given input, which has a known or determinable n1 such that $Explode_{n1}(Prob,l)$=+INF for some desired complexity hierarchy level l, there exists an unhinted algorithm which solves Prob in the general case complexity corresponding to finite complexity level l, so long as the verification algorithm V is belongs to this complexity level itself.

Proof sketch: One can apply the method in Theorem 29 for ever increasing n0's (for example taken under repeated doubling or repeated squaring) until it can be established that the n1 threshold has been reached. If an upper bound is known for it in advance, n0 can be taken to be directly n1. The method in Theorem 29 is

modified to output not just one suitable algorithm, but **all** of them. This multiplies the size of the output of the method by at most the size of the input universe, making it as large as 2^(2^n0). The correct algorithm, which solves the general case problem within the desired complexity, is necessarily amongst this outputted set. In terms of general complexity theory, this set has size O(1). As such, the general case algorithm constructed consists of running all such algorithms (O(1) of them) for any input instance given, and verifying each of their outputs using the algorithm V, picking the correct one. So long as the complexity of V is no larger than the desired class, this results in an algorithm of such general case complexity class.

Complexity: The complexity of employing this approach without further refinement is essentially within at most some log(n1) factor [for repeated doubling] of the complexity of a single application of the method under Theorem 29 for n1, modified to output any acceptable algorithm/hint pair (which does not modify the running time complexity of the brute force approach). As such, as argued for Theorem 29, this is within 2^(2^n1)), which, ironically enough is O(1) in terms of general case complexity.

Discussion: Note that by Theorem24 any problem which admits a general case algorithm of some corresponding complexity (e.g. P) also has such a fixed n1. In practice, n1 may or may not be knowable in advance. It can be guessed or some rule for its determination hypothesized. For example, it can be speculated that if the finite complexity class has not exploded for 5 successive repeated squaring applications, then this threshold has been met or exceeded. Or it could be speculated that its finite complexity class is monotonically non-increasing with increase in input-size (this is not always the case).

Note that for the output produced by the approach in Theorem 30 can be further trimmed down, both in practice (as some candidates are eliminated as more and more input instances are processed) and via theoretical reasoning, by a researcher which is able to prove that one such is actually always correct. A formal proof of this may itself be rather lengthy (e.g. consider the proof for classification of algebraic finite simple groups, which "has around 15.000 pages, spread through mathematics literature"). If it exists at all! Given Gödel's incompleteness theorem (see [11]), there are true statements expressed in first-order logic over natural numbers which cannot be proven. The desired proof might happen to be one of them. In the eventuality a proof exists, a researcher could again employ this theoretical framework and the approaches described in this paper to develop an algorithm to automatically find it. This is possible since verifying formal proofs is in fact a rather straight forward computer, thus a verification algorithm exists. Finding such a formal proof, or showing that one does not exists – that is the hard part.

The significance of Theorems 28-30 is rather major: It shows that finite algorithms can, at least theoretically, solve almost all problems currently considered hard – if as of now, only in 2EXP time – which may itself be a rather long wait. Nevertheless, their existence allows the problem of finding a solution to a computer science problem to be rephrased in terms of tradeoffs between the following three dimensions:

- Running Time of the solution algorithm S.
- Hint Size for algorithm S.
- Running time of computing a suitable algorithm/hint pair by some automated method, given existing knowledge.

There are undoubtedly many interesting questions and results pertaining to reductions and relationships between finite case problems and either general or other finite case problems. These include the "meta-problems" induced by some general problem, under some interesting or practical assumptions: the problem of finding a suitable algorithm for it, of finding a suitable hint for such and many more. Further interesting questions include applying the approaches here recursively upon themselves, to potentially produce faster than brute-force algorithms for solution finding. We leave such questions outside the scope of the current paper and propose them as avenues for further research.

One direction which seems particularly useful to us for priority examination is that of problems which fall in the following categories:

- They belong to the $Poly_{n0}/LogRank_{n0}=2$ finite complexity class. This entails the existence of a reasonably small number of potential hints (up to n0^log(n0)) – thus making exhaustive search quite feasible.
- They have known and relatively efficient output verification algorithms. All NP-Complete problems fall into this category.
- Preferably they admit natural formulations as decision problems. Determining satisfiable assignments for a Boolean formula is one such example (with the 3CNF-SAT decision problem). Computing Discreet Logarithm is not.
- There exists some non-trivial amount of solved hard test cases within existing body of research.

Section 4.3 Reasons for considering finite algorithmics valuable

The most important argument which needs to be made for acknowledging the importance of the study of finite algorithmics is why we should expect that finding a solution to the finite case of problems is easier than finding one for the general case.

We, the author, present the following arguments as indications that this is in fact the case:

1. **There exist problems which are incomputable in the general case, but computable in the finite case**. Consider the following problems.
    a. For classical computers: **Optimal String Compression** – determining the shortest program which outputs a given string. In the general case this is equivalent to computing the Kolmogorov complexity of the string and is incomputable. However, if we limit the problem to the practical application of considering only strings of length up to some n0, and limit the running time of the program to at most Exp_{n0}, the problem becomes computable: one can simply enumerate all programs of length no larger than n0, and run them until they either time out, produce a wrong string or produce the desired string. Afterwards the shortest correct one can be selected. In fact, one could even try to determine a suitable hinted algorithm using Approach 25 and some hint generation method which allows some, or most strings which appear in practice to be compressed efficiently (NB: With regard to any algorithm, there are strings which are

incompressible). The original proof by contradiction stating that Kolmogorov complexity is incomputable relies on the assumption that length of the source code of any such algorithm is shorter than the length of at least one target string. This however does not apply to the finite case, where strings have bounded size: A Kolmogorov complexity computation algorithm can indeed exists for this case (and does: for example one which includes precomputed output for each string), however it is necessarily longer than n0. Thus, solving the finite case, Exp_{n0} bound running-time Kolmogorov complexity is actually in EXPTIME.

b. For Finite Automata: Recognizing Prime Numbers – using a Finite Automata to determine if a string denoting the representation of an integer in some base (e.g. unary, binary, etc.) corresponds to a prime number. Finite Automata are unable to recognize this language. So for this computation model, PRIMES is incomputable. The proof is a relatively straight contradiction, using the Pumping Lemma [12]. The problem of recognizing all prime numbers up to some upper threshold n0 however is computable using Finite Automata: it is in fact a finite language and all finite languages are regular. Note however, that **different** automata are required for different n0-s.

This illustrates that the finite case can be simpler than the general case. In fact, it is so for some very practical problems. Kolmogorov complexity features prominently within Information Theory and Cryptology.

2. **There exist problems with large thresholds of complexity explosion**. Consider the following problem, inspired by the Theorem of Classification of Finite Simple Groups: "Take some sort order for finite simple groups, such that groups of smaller order appear before groups of larger order. When tied, consider some other arbitrary criterion, like number of generators or anything else desired. Given an index k of a group in this sort order, and a series of pairs of numbers representing elements within this group, output the result of the group operation acting on the elements, under some fixed (but arbitrary) numbering for them. The length of this series is logarithmic in the group order. ". The problem asks essentially to compute group operations consistently within a specified finite simple group. We can name it GROUPOP. Here the difficulty parameter is not the input size, but the order n of the group which also bounds the input size. As per the Theorem of Classification of Finite Simple Groups, there exist actually only three infinite classes (cyclic groups of prime order, alternating groups of degree at least five and groups of Lie type). All of these have simple representations. However, there exist another 27 finite simple groups which do not belong to any of the infinite families. Out of these 27, the Monster Group, of order M = ~8*10^53, stands out as it does not have a simple representation. As such, performing group operations in any of the other finite simple groups is computationally much faster than in the Monster Group. For cyclic groups for example doing group operations is as simple as multiplication modulo the prime which is the order of the group. This is actually logarithmic in the value of the group order. For the Monster Group however, Wilson has described a method involving two 196882x196882 matrices [13]. Doing operations with these matrices is computationally very expensive, bringing GROUPOP outside Linear. Some other constructions have been proposed, however it still remains that the Monster Group is terribly difficult to work with. As such, one can say that Explode(GROUPOP,Linear) = M ~ 8*10^53. In fact, if operations within groups of the other two infinite families besides cyclic can be done in polylogarithmic time, we have that Explode(GROUPOP,PolyLog) = M ~ 8*10^53.

The value 8*10^53 is rather large – large enough to be considered non-trivial. The problem GROUPOP is rather simple up to this group order, and then it explodes drastically. Could it not be that something similar happens to other interesting problems, like Integer Factorization or 3CNF-SAT? Furthermore interestingly, given the fact there is a single Monster Group, the complexity of GROUPOP will ultimately collapse back: the super-linear complexity for M ~ 8*10^53 will ultimately be smaller than a single log factor of some larger group order. As such, we can state that $Collapse_{8*10^{\wedge}53}$(GROUPOP, PolyLog) < +INF. Essentially, the finite complexity of GROUPOP is bitonic – small at first for quite some values, then it grows drastically large (rather quickly) and then collapses back to being small. From the point of view of a general case complexity theory, the existence of the Monster Group is fully irrelevant. The extreme difficulty of doing operations there, given the fact it is a singular finite case, is in fact O(1).

Thus, finite case complexity theory offers a much better way to describe the structure of this problem than the general case one.

3. **There exist problems where precomputation specific to a certain input size is very useful**. Consider the problem of determining the Minimum Spanning Tree for a given graph with n vertexes and m edges. This problem is relatively easy and numerous general case algorithms with near-but-not-exactly optimal complexity exist: from Kruskal's O(m*log n) to Chazelle's near-linear O(m*alpha(m,n)) [14]. However, there exists one algorithm by Petite and Ramachandran [15] of optimal complexity – which, mysteriously enough is still unknown. Whatever it is, their solution is nevertheless bound by it. The approach involves precomputation of all optimal decision trees on log(log(log(n))) vertices. In this situation precomputation can be completed in O(n), which is no larger than the complexity of the outstanding part of the algorithm. As such, the precomputation step can be done on-the-fly for each instance of the problem, without any need to store it as a Hint to some hinted algorithm separately, for purposes of improving running time performance.

It could be that some very difficult problems (maybe even 3CNF-SAT) have solutions which involve precomputations for an input size (difficulty) of larger complexity class than the rest of the algorithm. If the result of these precomputations is short enough however,

we could simply store them as a Hint to some hinted algorithm, belonging to a favorable complexity class such as $Poly_n/Poly_n$. Note that computing the hint for some input size could belong to a much larger complexity class – such as $SemiPoly_n$ or Exp_n. However, this only needs be done once for all inputs of that size. Once computed, if it is short enough it could be used by a hinted algorithm to skip this potentially extremely time-consuming step. We, the author, strongly suspect that if practical solutions for finite or general case 3CNF-SAT problems exist, they will involve reasonably sized hints which require nevertheless large amounts of running time to compute.

4. **Other expressive computational models show significant drop in complexity from general case to finite case.** Consider potentially the closest relative of the Turing Machine – the Finite Automata. Consider a regular language with an infinite number of words. It can be succinctly described by a Deterministic Finite Automaton with some number of states. This number could then be reduced by computing the minimal automaton for the given language. So long as the language has infinitely many words, this is the best which can be achieved. However, when the attention is directed to a finite subset of this language – namely that of words which do not exceed some fixed finite length, it has been shown that the number of states could be reduced even further, using something called a Deterministic Finite Cover Automaton. This is basically an automaton which correctly recognizes the language up to words of at most the specified length, but it is allowed to error on anything longer. This is analogous to considering the finite case of some general case problem, where the sought-after solution is a specification for a Finite Automaton, not a registry machine. Deterministic finite cover automatons are expected to have a significantly smaller number of states than their counterparts for the unrestricted language. In fact, it has been shown that they have a smaller number of states than even their counterparts which recognize just the finite language precisely (are not allowed to error on longer words) [8] [16].

   It could be that classical computers exhibit a similar phenomenon for at least some languages - namely that complexity of recognizing on such up to some finite length is much smaller than that of recognizing it on the general case. While classical computers are a much stronger computational model than finite automata, the two are still closely related. For example, every bounded-space registry machine algorithm can be represented as an automaton which is initially fed the input and them some number occurrences of a special symbol, each corresponding to one clock tick of processing by the classical registry machine. The states of such an automaton are in fact the all the memory configurations the registry machine could encounter during its execution. While this representation is inefficient, it serves to illustrate the close relationship between the two computational models, for the finite case.

   Even for machines of larger or incomparable computational power (e.g. Quantum Computers, or the theoretical Blum-Shub-Smale machines [17]), the fact there exists sufficiently expressive computational models (the Finite Automatons) which experience complexity collapse for the finite case, serves as an indication the same could occur for these models also.

5. **There exist interesting problems which are outside 2EXP on the general case**. Consider any EXPSPACE-Hard problem for instance. Reachability in Petri Nets [18] is quite practically interesting and has recently been shown to be outside ELEMENTARY, thus outside 2EXP [19]. Deciding if two regular expressions which allow squaring (requiring exactly two adjacent copies of the operand) represent different languages is in EXPSPACE [20], as is the validity problem for extended linear temporal logic with times. Besides these, many problems within Game Theory are PSPACE-Complete (e.g. solving generalized Tic-Tac-Toe), while others still are actually incomputable.

   For these categories of problems, there is no hope of solving them in practice by discovering an efficient algorithm for the general case. The only hope to ever solve these is within finite algorithmics – solving not the problem in general, but some restriction of it to a finite case. Here, one can apply Theorem 28 to show that a Polyn0/Expn0 algorithm exists. Finding one however, may be outside 2EXP since the verification algorithm itself could be outside 2EXP. Nevertheless, the existence of a Polyn0 algorithm (if but of exponential size) shows that the finite case is indeed easier than the general case for interesting practical problems.

   A prominent result within finite algorithmics will be one which gives a solution to one of these practically important, but generally intractable problems for some non-trivial practical upper bound.

6. **Finite case problems are a particularization of the corresponding general case problems**. Essentially, we as researchers have reduced the practical finite-case problems we are interested to solve to some potentially harder ones – namely the general case. While sometimes the general case is easy enough, this is not always the case. As the easiness of 2-CNF-SAT relative to arbitrary Boolean formula satisfiability illustrates, sometimes the particularization is much easier than the general case. Further research should focus on relationships between finite case and general case for specific problems, to determine where this is the case and where not. The theoretical framework introduced in Section 3 serves as a tool.

7. **There exists an automated method for finding an optimal solution to verifiable problems in the finite case**. The proof sketch of Theorem 29 shows how an optimal algorithm for such problems can be constructed. There exists an analogous result from Jones [21] for general case verifiable decision problems. He essentially constructs an algorithm which runs, in a dove-tailing fashion all conceivable algorithms until one stops and produces the correct output. While asymptotically this is optimal for the general case, the hidden constant is astronomical – it is exponential in the index of the suitable algorithm in the enumeration. This makes it generally unusable in practice. Note that Jones' method does not actually identify the suitable algorithm. For each problem instance, there could be some different

algorithm which finishes first and outputs the correct answer (which is then verified by the verification algorithm). In the finite case, on the other hand, after spending some initial (potentially very large) amount of time, the optimal algorithm is determined (it's source code becomes available). Thus, it can thereafter be directly applied to any instance (up to the finite upper bound) where it performs efficiently enough. Using Jones' method for the finite case would entail dealing with the astronomical constant *on every run* of the algorithm – on every instance. Furthermore if a problem did not admit a general case efficient algorithm, his method no longer yields an algorithm of optimal complexity, since the index of the most efficient algorithm is no longer a constant.

8. There have been prior successful applications of the approach of automatically generating algorithms. The field of AI and Machine Learning, particularly Neural Networks is a perfect example where trying out and adjusting an algorithm, within a certain family results in something very useful. For most AI and Machine Learning applications (such as image recognition), the problem researchers were trying to surmount was the apparent lack of proper succinct description of a method to determine the correct output for a given input. For example, describing formally what "an image of a cat" was (or to go further, what "an image of a happy person" was) proved very difficult. Nevertheless, this was circumvented by employing an automated method of trial-and-error to essentially determine an algorithm which is good enough.

   The same could be applied to the situation where the difficulty lies not in identifying a formalism to describe the input/output relationship, but in finding an efficient algorithm to compute it, if but only in practice. As with AI and Machine Learning, we can now regard this process as the result of a combination of automated trials and researcher insight, not just of the latter.

The arguments above which serve to indicate that finite-case problems are indeed easier to solve (at least to us humans, potentially aided by computers) than their general case counterparts. However, there are two more arguments of a more abstract nature to indicate the existence of value in the approaches presented.

1. **There exist problems which admit rather simple and short efficient algorithms, but which require complex theory to prove their adequacy**. The clearest example can be considered the string matching algorithm due to Knuth-Morris-Pratt (KMP) [22]. It has less than 10 lines of code, a single method with no recursion, loop-nesting of at most 3 and all its expressions are over no more than 5 variables. Nevertheless, the theory behind it, especially with regard to proving its linear running time complexity, is the most likely cause why it has not been discovered earlier.
2. **Physical phenomena could exist which can be harnessed to allow rapid speedups for computations, but at some great cost**. Given current mainstream understanding of physics, concerning time-dilation, if we were able to send a computer with sufficient battery power to a place far away from any gravity wells (like planets, stars or black holes) and have it stand as still as possible relative to Earth, some important speedups can be attained. Other phenomena might exist which to allow for much greater speedups (quantum non-locality seem like a good place to start a search). These however, might entail travelling to distant regions of the Cosmos, or expending large amounts of resources, like battery power. However, this can be regarded as a one-time-cost. With the methods provided for by finite algorithmics, such a sped up computer could then rely back the optimal algorithm (e.g. via radio waves) for some problem. Thereafter, we could use it solve all practical instances, without the need to incur the one-time-cost ever again.

Section 4.4 Application of techniques to three well known problems

In this we present some directions for practical application of the theory and techniques presented within this paper to three hard problems. They are intended to be viewed as just an example of how the quest for adequate solutions can be altered with the introduction of finite-algorithmics. It is outside the scope of this paper to propose (much less test experimentally) a fully specified approach or method which can be employed to solve them. Nevertheless, we, the author, are confident that ideas formulated within the context of finite algorithmics – either based on the ones presented below or others – will eventually lead to an adequate solution to such, if one exists.

The main intended contribution of this section is to show, by way of example, how the change in reasoning due to finite algorithmics can lead to fundamentally different avenues of research for hard problems, from the ones currently pursued by computer scientists.

Section 4.4.1 3CNF-SAT

The following ideas can be applied to solving 3CNF-SAT, in the context of finite algorithmics:

1. **Consider only families of hard cases**. For example, for an n-variable formula, do not include in analysis clause configurations which are conjunctions of two or more formulas over less than n variables, as such could be solved recursively separately. Also ignore families of known easy cases. For current heuristics published in literature this means formulas with less than 2 or more than 5 clauses per variable.
2. **Discover specific problem structure incrementally**. Examine what makes some 20-variable 3CNF-SAT formulas harder to solve than others, for some algorithm or family of algorithms. It could be the existence of some tuple of clauses or some computable trait of a larger subset of clauses. Then look at 21-variable formulas and find which additional traits (besides those applicable from the 20-variable case) predict hardness. Then at 22-variable and so on. How many additional "hard" formulas specific to an n+1 – variable case are there (excluding those for formulas in up to n variables)? Do they belong to some finite number of families (we strongly expect a negative answer)? Is the number of such growing rapidly or slowly? How can they be described succinctly so as to potentially allow their storage as hint to some algorithm? Do same for 200- or 2000- variables randomly built 3CNF-SAT formulas. Such analysis could be aided by tools from AI and

Machine Learning, which may discover unexpected or counter-intuitive correlations.

3. **Discover what makes candidate algorithms actually solve hard instances when they eventually manage it**. For hard instances examine what actual choices (e.g. "lucky random assignments") allowed their eventual resolving? How do these choices correlate with the input instance (or part thereof) and between themselves? Is this knowledge (or at least part thereof) common to several hard instances? Can all such knowledge for some n-variable instance difficulty be represented succinctly and efficiently enough so as to allow a $Poly_{n0}/Poly_{n0}$ algorithm to use it to solve all much faster on subsequent runs? Is at least part of it common to many instances? Like with point 2 above, AI and Machine Learning tools might prove very valuable.
4. **Discover predictors for candidate algorithms non-performance**. Discover similar correlations as those in point 3 above, but for situations where a candidate algorithm performs very poorly. How can "really poor" choices be described formally and succinctly, so they can be avoided on subsequent runs?
5. **Consider a family of algorithm/hint pairs (or a fixed algorithm with a family of hints). Discover favorable and unfavorable correlations between parts of the content of the algorithm/hint themselves and performance/adequacy in trial runs**. What are good predictors for good/poor performance? Is it having a particular while loop in a certain place? Or doing random-restarts in some describable fashion? Researchers have been more or less attempting this step manually so far – leading to the discovery that random restarts are key to performance of advanced SAT Solvers [3]. However, formal methods from AI and Machine Learning and not only could be employed to deduce many more such correlations much faster.
6. **Consider correlations between *memory state* of candidate algorithm within a family and performance.** It could be that for some family of algorithms, a certain memory state (or part thereof), if encountered at runtime, is strongly correlated with very poor performance (for example a particular choice of random assignments, or set of impossible assignments deduced). It can be regarded as similar to steps 3, 4 and 5 above. Unlike steps 3 and 4, analysis aims not to learn something about the structure of instances themselves with regard to the candidate, but about the runtime behavior (which can be common to multiple candidates within a family) across some test battery, thus learning something about the desirability of having the memory state characterized in a particular fashion. Unlike step 5, here the analysis focuses not on the source code / hint contents used, but on the actually runtime memory state (which might be common at least partially to several algorithm/hint pairs). Like with points 2-5 above, tools from AI and Machine learning (and not only) could be employed.
7. **Perform the same analysis as step 6 above not for a single memory state but for a short sequence of such states.** Essentially, this calls for analysis to be expanded from examining single snapshots of memory to examining short "movies" of such snapshots (not necessarily sequentially chosen).
8. **Use some form of automatic recombination and selection method to generate new candidate algorithm/hint pairs and maintain the set under consideration within desired size limits**. This entails essentially using the information gathered in points 2-7 above to rank, modify and combine algorithms / hints such that one representing an adequate solution is found much quicker than by exhaustive enumeration. An example of a modification is to make an algorithm do a full or partial restart every time its runtime memory state can be characterized as "unfavorable" as per data obtained under methods 6-7 above. An example of a combination is to run two or more algorithms in some dove-tailing fashion for a number of steps and then decide how to continue based on their joint memory state. Other methodologies like those specific to genetic algorithms or again those employed in AI and Machine learning presently can be readily employed. Ranking or more specifically selection is required to keep the candidate set size within the space limits imposed by whatever hardware is attempting to find the solution.
9. Use ideas 1-9 above and others to incrementally generate algorithm/hint pairs for increasing difficulty. This way, the information collected with regard to solutions of instances of lesser difficulty (smaller number of variables) can be exploited to speed up and obtain similar information more difficult instances.

The ideas described above are meant to speed up some automated or semi-automated search for a suitable algorithm. However, one such may not exists. Even in that case, having a good choice of a heuristic algorithm, accompanied by a good choice of a hint can result in a huge drop in running time (if though perhaps not enough to make it fit some desired finite complexity class like SemiPolyn0). It could be that such drop is much higher than the time actually consumed to generate the pair. After all, as per Theorem 28, 3CNF-SAT admits a Polyn0/Expn0 algorithm. So the quest is actually for a more acceptable tradeoff between hint size (more specifically hint generation running time) and algorithm running time.

A final trick could be employed in practice. The universe of potentially hard formulas over n variables is rather large. It is of the order of Comb(4*Comb(n,3),4n)*2^(4n) for formulas with up to 4 clauses per variable, which is much larger than 2^n and even than 2^(4n). However, in practice we might be interested in solving just a very small subset of these – namely the ones which occurred as a reduction of some other practical problem. Sometimes, it could be that we are actually interested in solving a single very lengthy formula – one for example giving a winning strategy for a complex military game position, or an optimal design for a microchip. In such a case we can particularize further. When using ideas 1-9 above (and any others for that matter), we will consider only expressions which are formed by a subset of the clauses appearing in the original large instance. Thus, the universe of instances for all variable sizes is cut to ~2^4n which is a huge reduction. Furthermore, the ideas and methods described can now make use of structure specific to the original input instance to arrive at a solution much faster. The only draw-back is that the algorithm / hint pair can be expected to perform adequately only on the original input instance set (which may have a single element). This nevertheless, can be an acceptable and desirable tradeoff.

Section 4.4.2 Kolmogorov Complexity

Problems which in the general case are provably incomputable due to reduction from Kolmogorov complexity typically relate to string compression or have to do with entropy extraction (generating more randomness from less such). The two are not unrelated.

In this subsection we focus our attention on string compression. Formulating the problem for practical use in this case involves more than just restricting the input size. Formally we consider a string compression problem to have the following statement: "Given a set of strings of length no more than n0, determine some pair of (potentially hinted) algorithms $Compress_{n0}$ – which takes an input string and produces a digest – and $Decompress_{n0}$ – which takes a digest and produces the original string – such that the digest of maximum (or average) length is as short as possible (or simply "short enough") and both algorithms belong to $Poly_{n0}/Poly_{n0}$ (or some other acceptable finite complexity class)."

The above is an adaptation of the original formulation of Kolmogorov complexity, which referred to compressing a single string by using an algorithm of unbounded complexity (but which surely terminates) to produce it. The above formulation allows for the input set of strings to contain a single element as well. However, in practice it is more likely that a single solution is sought which can be used to compress several strings (potentially all the strings of length n0).

Under the above formulation, all ideas from Section 4.4.1 could be adapted here as well. The only difference will be in verification – as more and more algorithms are considered, performance entails not only examining running times but also lengths of generated digests.

Some ideas specific to string compression, formulated in the context of finite algorithmics are the following:

1. **Determine short incompressible strings which appear as substrings within the input set**. It is a well-known information theory result that for any length there exist incompressible strings. This can be shown via a simple counting argument for a binary alphabet. Furthermore, the density of incompressible strings is rather large. Using this information, one can attempt to "break down" the input strings into incompressible "atoms" which can then serve as part of a hint to an algorithm which only describes how to assemble them together to obtain the desired string. Incompressibility within this context does not need to be strict. A reduction of less than 3-4 characters for example could make a large string be considered just as well incompressible.
   Note that doing this is incomputable in the general case for sufficiently large strings. Nevertheless it is very computable within the finite-case formulation above.
2. **Given a list of short strings (atoms) determine a method which uses such to build a larger target string**. One straightforward such method is to break the target string into concatenations of atoms and then to store only the index of each such for each part. More sophisticated methods could involve exploiting correlations between contents at different positions (e.g. repeat adjacent occurrences of an atom).
3. **Apply ideas 1-2 above recursively, on the digest generated by the method in idea 2**. This allows further compression based on the non-randomness of the pattern in which atoms themselves occur within a target string. Note that the input for the recursive step is typically strictly shorter than the original input – which was already compressed by a prior application. The final output could then just include a number indicating how many times recursion was applied.
4. **Consider space-time tradeoffs in deciding which short strings to keep as hint to the solution algorithm and how to represent them**. Atoms themselves may not need to all be kept in their lengthy, full form. While a single atom is considered incompressible, a list of several may have a more succinct representation than simple enumeration of all such. For a binary alphabet, a suffix tree or even a simple trie may offer an efficient improvement. However, there may be other shorter representations which in turn require longer processing times to allow extraction of some "k-th atom".
5. **Exploit randomness**. Consider producing methods and algorithms which make random choices. In the context of decompression, such can produce the desired original string only with some probability (e.g. 1/2 or 2/3) – and in the other cases other produce something else or exceed desired running time. In the context of compression, such could produce valid digests only with a certain probability.
6. **Consider error-correction codes.** In the context of idea 5 above, consider padding some lengthy atoms using error correcting codes. While counterintuitive, this could potentially allow for shorter algorithms / digests to be generated – since one such need not output a precise string, but any of its correctable forms. Furthermore, simple detection of errors could be reason for rerunning said algorithm automatically for a different random seed, thus improving the probability of correct output under idea 5. Finally, given some input set of strings, all atoms within it might be sufficiently separated in terms of Hamming distance. Thus, there may be no need for additional padding. An algorithm which only very occasionally outputs the correct atom and the rest of the time something which is not an atom can be combined with an algorithm (like a Deterministic Cover Automaton) which simply recognizes the language of atoms for the given input string set.

Finally, all results pertaining to Kolmogorov extractors (entropy extractors), polynomial-time randomness (producing outputs which are indiscernible from random by any polynomial time algorithm) and related topics are relevant and can be further refined to apply to this context of finite case formulation. A prominent researcher in this field is Prof. Marius Zimand (see [23] or [24]).

As illustrated in Idea 6 briefly, a related problem to string compression is finite language recognition: "Given a set of strings, produce an algorithm which can determine if an input string is within this set or not." This related problem is extremely relevant to finite algorithmics. Firstly, any decision problem can be formulated in terms of determining if an input instance is within the set of instances for which the answer to the decision problem is "Yes". In any finite case of any problem, such a set is finite as well. A solution to efficiently deciding membership within this set solves the original problem.

In fact, compression of the set of outputs of some problem (e.g. 3CNF-SAT) on some small finite input universe, such that set

membership can be decided efficiently, can and should be employed in the course of running automated methods for finding its solution for larger input sizes (difficulties).

The starting point in the case of a decision problem for example, can be the Deterministic Finite Cover Automata for the set of strings which represent input instances with a "Yes" answer. Using such, group membership can be decided very quickly (linearly in instance size), however the size of the hint (the actual description of the DFCA) can grow too large. Nevertheless, we the author consider the relation between DFCAs and acceptable algorithms (in terms of running time / hint size / hint generation time) for set membership decision problems as a prime candidate for future research. We see such research as both general and specific to a particular problem domain (e.g. to the set of satisfiable 3CNF-SAT formulas over at most n0 variables).

Section 4.4.3 Integer Factorization

Factoring large integers can be solved efficiently by quantum computers, using Shor's algorithm [25]. Nevertheless, a similarly efficient algorithm for a classical computer is yet to be discovered. Integer factorization occurs mainly within the realm of cryptology and generally pertains to identifying a prime factor of a large semiprime number. Besides adaptation of the ideas from Section 4.4.1 which can prove useful, an idea specific to this problem is the following:

1. **Identify and store "hard" primes**. Given a target range for the integer to be factored (e.g. 512-bit or 1024-bit sized), and some state-of-the-art existent algorithm (e.g. Pollard's Rho algorithm or GNFS, or a combination of such), determine what constitutes "hard primes" for it. These are prime numbers which, when they appear in the composition of an integer to factor, cause the algorithms running time to increase drastically. If the number of such "hard primes" is relatively small in relation to maximum value of the integer to factor, they could all be stored. Even if there are relatively many such, ideas from Section 4.4.2 could be employed to get a more succinct representation of this set, allowing it to be enumerated.

The above idea, steams from the following anecdotal empirical experience of the author. Many years ago, he participated in an open factorization challenge (which was part of a larger computer science contest), which asked contestants to factor each of 10 large numbers within a week. The author encountered the following situation: The first 7 were relatively easy to factor and he managed to factor the 8^{th} and the 9^{th} as well using some more advanced techniques. However, the 10^{th} one seemed unbreakable. At that point we considered the following question: "How could the problem settlers have come up with such a hard case in such short a time [it was known to him that they themselves had only about one week to prepare the challenge]?". Given this, he tried the following: He searched on the internet for the primes which showed up as factors for the other two hard cases – namely the 8^{th} and the 9^{th}. He then identified a small number of short lists of primes which featured them. He then used a computer program to try out each of the primes on those lists against the hard 10^{th} challenge case. To his delight, this worked. The "hard prime" for the 10^{th} case was in fact taken from a list on the internet. This experience above serves to indicate that generating "hard primes" is no easy task. Like with 3CNF-SAT, most large instances of Integer Factorization are easy to solve. Those which remain may be hard due to the presence of some of these hard primes in the solution. Identifying all such and, if there are not that many, and including them as hint to some hinted algorithm, might make integer factorization easy for all practical sizes even for a classical computer.

Discussion

We have discussed the significance and implications of most results and theory throughout the paper, close to the place of their introduction. In this section we present a few ideas of more general significance.

The results in Section 4 serve to illustrate that analyzing a problem for the finite case, rather than on the sometimes more difficult general case holds value. Problems which are very hard (or even impossible) to solve in the general case may have acceptable finite case algorithms. Furthermore, the search for suitable algorithms in the finite case can be automated or sped up using computers.

The introduction of finite algorithms allows us, as humans to reason about hard problems differently. Ultimately, within the framework introduced in this paper one could ultimately prove that:

1. **P <> NP**. For example by proving that for any large enough finite input size upper bound n0, the length of the shortest hint for a $Poly_{n0}$ time algorithm which solves it is strictly larger than for n0-1. This does not necessarily entail that NP-Complete problems could not be solved in practice.
2. **P = NP**. For example by providing a polynomial time algorithm which constructs a hint for any finite input size upper bound n0 for an algorithm of bounded PolyRank time complexity. This could be further restricted to practical significance, by providing a $Poly_{n0}$ algorithm for hint construction for a $Poly_{n0}/Poly_{n0}$ algorithm.
3. **P = NP or P <> NP but we really do not care about the distinction for practical purposes**. This could be either because an efficient algorithm and hint have been identified for all practical bounds (favorable case) or because it has been proven that the shortest hint size for most practical cases is too large (unfavorable case). In the former situation, if P<>NP this essentially happens for input sizes outside of humanity's practical zone of interest, while in the latter, if P=NP this again happens for too large input sizes, such that the drop in complexity in the general case is in fact of no practical use.

The same discussion as above applies to the study of relationships between other complexity classes (such as between P and PSPACE).

The results and techniques presented in this paper can be applied not only to hard problems (PTR and above), but also to those which are relatively easy but for which we would like to identify even more efficient algorithms (TR). One such candidate is multidimensional range querying. An algorithm which breaks the "curse of dimensionality" – if such exists – could be sought and found using the same approaches.

Ultimately, we expect the change in mindset and in focus of research resulting from rephrasing a problem in terms finite algorithmics theory to lead, in the near future, to practical solutions for some of the hardest computer science problems which have been haunting humanity for many decades.

Conclusion and further research

Throughout this paper we have identified several avenues which we consider prime targets of future research. We briefly recap them here:

1. **Examining relationships between different finite complexity classes**. This can pertain to relationships between different finite complexity classes for the same problem domain (e.g. for different n0 upper bounds) or between different problem domains (e.g. resulting from reduction of one problem to another). Also, they could be unspecific pointing out interesting results for finite complexity in terms of natural functions in general without the need for them to represent something specific.
2. **Examining relationships between finite complexity classes and general case complexity**. Similarly this can occur within a problem domain, connect several problem domains or be unspecific, pertaining only to natural functions in general.

With regard to the these, we ask simply "What are interesting results which fall into these categories?". We presented a few elementary ones ourselves in this paper, in Section 4.1.

In addition to the above, we propose the following directions for future research, which seem to us important:

1. **Investigating relationships between Finite Automata and efficient Hinted Algorithms for the finite case**. Limiting input size, running time and usable space to some finite bound allows a problem to be solved within a computational model less powerful than a Turing machine. Namely, any algorithm on a classical computer which has bounded memory size and is limited to a maximum number of steps to perform (finite case complexity) can be accurately represented by a finite automaton over a ternary language: The states of the automaton represent the memory configurations which can be encountered during execution, transitions correspond to the small changes an algorithm can perform in one step leading from one memory configuration to another and the ternary language represents the clock ticks which the algorithm consumes. The first part of an input word is the binary representation of the input instance for the original problem, and all the rest are 3s. If the automaton accepts on such a constructed input, so does the corresponding classical computer algorithm. Ironically enough, not all automatons defined in this fashion have corresponding classical computer algorithms – a transition within an automaton can be from a corresponding memory state to any other, while for a classical computer a transition (one operation) only changes one word of memory (in the RAM model) at a time - thus it can point only to very similar states. While direct automaton construction and minimization based on the observation above may not lead to a time-wise feasible approach to solving a problem, conceptually it can offer deep insights. The relationship between the two computational models for the finite case warrants further research.
2. **For a specific problem domain investigate the growth of minimum hint size as the finite upper bound increases**. The fact a problem is limited to the finite case does mean the upper input size (difficulty) bound should remain fixed during analysis. While for practical applications existent at some moment such bound is a definite, effectively reaching it may entail examining correlations between solutions for smaller ones. One very interesting question is the following: "Given a problem Prob and some target finite complexity class for an efficient algorithm, how does the size of the shortest hint vary with the upper input size (difficulty) limit n0?" For general case solution, the answer is very simple: "It is 0 for all cases". Finite algorithmics however allows further nuance.

Finally we propose a specific, explicit question framed within the theory of finite algorithmics which, when answered, will give the strongest indication ever - if not a proof – for deciding the classical P=NP problem.

Consider some fixed, sufficiently expressive hinted algorithm. Such an algorithm can simply be one which receives, as part of the hint, the index of a more elaborate algorithm from the finite family described in Approach 26 and then runs such on the remaining hint and input instance. The family in Approach 26 can be considered to include as "predefined types" all popular data-structures and solvers for general case problems which are commonly known in literature as of November 2019.

Given the above fixed algorithm, answer the following question: **"What is the minimum length of some required hint, which allows it to decide satisfiability for any 3CNF-SAT formula over at most 2^20 variables within running time $Poly_{2^{20}}$?". Then answer the same question for 2^30 and 2^40.**

Firstly, if the hint sizes are small enough, answering these questions constructively will give the most efficient method for solving 3CNF-SAT in practice.

Secondly, by examining how the shortest hint size required grows for the 2^20, 2^30 and then for the 2^40 upper bounds on number of variables, one can get the strongest indication – if not even a sufficient proof – with regard to whether P=NP. If the hint sizes increase (at least significantly), this is a very strong indication that P<>NP. In fact, the only way this could happen and still have P=NP is if the additional sophistication in the structure of the 3CNF-SAT with an increase in the number of variables, drops to 0 beyond a certain finite bound (similarly to that of GROUPOP beyond the order of the Monster Group) above 2^40. We, the author, believe it to be extremely unlikely for 3CNF-SAT to behave so. Conversely, if hint sizes do not (at least no significantly) increase this would be a crushingly strong indication that P=NP.

Finally, if we were asked to take a guess, we would expect the answer to the above question to indicate a rather slow, but positive growth rate. Most likely on the order of $Linear_{n0}$ or $Poly_{n0}$. This would indicate that NP is outside P, however it would place it well within $Poly_{n0}$ or $SemiPoly_{n0}$ in practice. Furthermore, depending on how difficult computing such a hint proves to be in the general case, it place NP outside P but below EXPTIMP.

We conclude the paper here

Vitae

Mircea Digulescu is a computer scientist and software engineer. He

was awarded bronze medal at CEOI 2004 as well as 4th and 10th positions at ACM SEERC 2005 and 2006 respectively. He is still active in competitive programing on Codeforces where he had reached the first division. He has obtained Bachelors and Masters Degrees in Computer Science at from University of Bucharest – Faculty of Mathematics and Computer Science, where he had also been studying as a PHD Candidate in applied computer science. His main interests are within Complexity and Computability Theory, Game Theory, Algorithms and Data Structures and Cryptology.

Acknowledgments

No organizations funded the research presented in this paper. The author's last affiliation is PhD candidate at the University of Bucharest, Department of Computer Science of Faculty of Mathematics and Computer Science. The author is currently an independent researcher. Statement of interest: none.

I would like to thank late researcher Mihai Patrascu for his lecture held at an a training camp for competitive programming contestants many years ago, where amongst other things, he revealed the existence of a deterministic linear time algorithm for solving Minimum Spanning Tree problem, which worked only when the input size was greater than 10^80. His remark that anything below this size was solvable in O(1) served as an inspiration which ultimately contributed to the discovery of the ideas in this paper.

Warm thanks also to the few beautiful persons who inspired strong interest for solving hard computer science problems in practice.

They are, in this order, Anca, Tanya, Nicolet sinonim-obiectiv, Ctlina Ghiorghi, Syuzi Mrktcharyan, Florina Petre, Laura Pana, and, special thanks for being synonymous of the year 2024 to Eliza (mention to Yarina 22) and to exoplanet Anastheizia Anna2.

Many warms thanks to the rest of the beautiful persons, especially to all my beloved synonymouas (with the meaning of the definiton of love), named and unnamed, registed in RUCS or not, as well as to my friends.

This paper would not have existed without them.

A. Annex 1

The tables below detail the maximum estimated tractable difficulty for the finite complexity classes. It asserts 10 MFlop/s for single-core on commodity hardware (from empirical Codeforces.com experience), 83 TFlops/s for single-core on super-computer grade hardware, a number of 2 million cores for the fastest super-computer and 60 million for all the TOP500 super-computers combined (data compiled directly from https://www.top500.org/lists/2019/06/). The values for multicore architectures (supercomputers) assume the algorithm can be parallelized perfectly. Furthermore, these bounds are for a classical computer. Where random data is required, depending on its quality, generating one such word (or bit) may take longer than 1 Flop. Also, no similar bounds are provided for a quantum computer.

ExpRank = 1. For Exp in general, divide values by 8.	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
1 second	**16**	**32**	**46**	**50**
1 minute (60s)	**20**	**36**	**51**	**54**
1 hour (3600s)	**24**	**40**	**55**	**58**
1 month (2.6 MS)	**31**	**47**	**61**	**65**
1 year (31.5 MS)	**33**	**49**	**64**	**67**
10 years (315 MS)	**36**	**51**	**66**	**70**
100 years (3.15 TS)	**38**	**54**	**68**	**72**

SemiPoly	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
1 second	**35**	**179**	**568**	**725**
1 minute (60s)	**56**	**253**	**761**	**963**
1 hour (3600s)	**86**	**353**	**1010**	**1264**
1 month (2.6 MS)	**162**	**580**	**1553**	**1923**
1 year (31.5 MS)	**201**	**693**	**1818**	**2241**
10 years (315 MS)	**245**	**814**	**2096**	**2578**
100 years (3.15 TS)	**295**	**955**	**2410**	**2953**

Quadric	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
1 second	**3162**	**9*10^6**	**13*10^9**	**71*10^9**
1 minute (60s)	**24*10^3**	**70*10^6**	**100*10^9**	**547*10^9**
1 hour (3600s)	**190*10^3**	**0.55*10^9**	**0.77*10^12**	**4*10^12**
1 month (2.6 MS)	**5.1*10^6**	**15*10^9**	**21*10^12**	**114*10^12**
1 year (31.5 MS)	**18*10^6**	**51*10^9**	**72*10^12**	**396*10^12**
10 years (315 MS)	**56*10^6**	**162*10^9**	**229*10^12**	**1.3*10^15**
100 years (3.15 TS)	**177*10^6**	**511*10^9**	**723*10^12**	**3.9*10^15**

Poly	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
1 second	**342**	**18*10^3**	**0.5*10^6**	**1*10^6**
1 minute (60s)	**1.0*10^3**	**45*10^3**	**1.2*10^6**	**2.4*10^6**
1 hour (3600s)	**2.8*10^3**	**115*10^3**	**2.8*10^6**	**5.6*10^6**
1 month (2.6 MS)	**14*10^3**	**492*10^3**	**11*10^6**	**22*10^6**
1 year (31.5 MS)	**24*10^3**	**847*10^3**	**19*10^6**	**37*10^6**
10 years (315 MS)	**41*10^3**	**1.4*10^6**	**30*10^6**	**60*10^6**
100 years (3.15 TS)	**69*10^3**	**2.3*10^6**	**48*10^6**	**95*10^6**

Linear	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
1 second	**10^7**	**10^14**	**10^20**	**10^21**
1 minute (60s)	**10^8**	**10^15**	**10^22**	**10^23**
1 hour (3600s)	**10^10**	**10^17**	**10^23**	**10^25**
1 month (2.6 MS)	**10^13**	**10^20**	**10^26**	**10^28**
1 year (31.5 MS)	**10^14**	**10^21**	**10^27**	**10^29**
10 years (315 MS)	**10^15**	**10^22**	**10^28**	**10^30**
100 years (3.15 TS)	**10^15**	**10^23**	**10^29**	**10^31**

PolyLog	Single Core Commodity	Single Core Super Computer	Top Super Computer	Top 500 Super Computers combined
Same as Linear				

For LogRank <= 1+log(log(n)) and Const the growth rate allows inputs of almost any practical size to be solved in a very short amount of time, usually within much less than a second. Of course, sometimes in practice the exact LogRank mattes – for example when searching for a suitable value within an exponential universe of alternatives.

References

1. Bauer, D. (2018). *Automated design of tendon-driven soft foam hands using Markov-Chain-Monte-Carlo*

optimization methods (Doctoral dissertation, PhD thesis, Master's thesis, Karlsruhe Institute of Technology, 2018. 10, 32, 39, 40).
2. Goldberg, E., & Novikov, Y. (2007). BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, *155*(12), 1549-1561.
3. Alouneh, S., Abed, S. E., Al Shayeji, M. H., & Mesleh, R. (2019). A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. *Artificial Intelligence Review*, *52*, 2575-2601.
4. Sohanghpurwala, A. A., Hassan, M. W., & Athanas, P. (2017). Hardware accelerated SAT solvers—A survey. *Journal of Parallel and Distributed Computing*, *106*, 170-184.
5. Tiwana, H., & Singh, R. K. (2015). Analysis of Busy Beaver. *International Journal*, *5*(6).
6. Chaitin, G. J. (1975). A theory of program size formally identical to information theory. *Journal of the ACM (JACM)*, *22*(3), 329-340.
7. Floyd, R. W. (1962). Algorithm 97: shortest path. *Communications of the ACM*, *5*(6), 345-345.
8. Câmpeanu, C., Santean, N., & Yu, S. (2001). Minimal cover-automata for finite languages. *Theoretical Computer Science*, *267*(1-2), 3-16.
9. Manome, N., Shinohara, S., Suzuki, K., Tomonaga, K., & Mitsuyoshi, S. (2019). A multi-armed bandit algorithm available in stationary or non-stationary environments using self-organizing maps. In *Artificial Neural Networks and Machine Learning–ICANN 2019: Theoretical Neural Computation: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part I 28* (pp. 529-540). Springer International Publishing.
10. Schmidhuber, J. (2009). Ultimate cognition à la Gödel. *Cognitive Computation*, *1*, 177-193.
11. Chaitin, G. J. (1982). Gödel's theorem and information. *International Journal of Theoretical Physics*, *21*, 941-954.
12. Ehrenfeucht, A., Parikh, R., & Rozenberg, G. (1981). Pumping lemmas for regular sets. *SIAM Journal on Computing*, *10*(3), 536-541.
13. Holmes, P. E., & Wilson, R. A. (2003). A new computer construction of the Monster using 2-local subgroups. *Journal of the London Mathematical Society*, *67*(2), 349-364.
14. Chazelle, B. (1997, October). A faster deterministic algorithm for minimum spanning trees. In *Proceedings 38th Annual Symposium on Foundations of Computer Science* (pp. 22-31). IEEE.
15. Pettie, S., & Ramachandran, V. (2002). An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, *49*(1), 16-34.
16. Cadilhac, M. (2005). Cover Automata for Finite Languages.
17. Blum, L., Shub, M., & Smale, S. (1989). On a theory of computation and complexity over the real numbers: *NP*-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, *21*(1), 1-46.
18. Araki, T., & Kasami, T. (1976). Some decision problems related to the reachability problem for Petri nets. *Theoretical Computer Science*, *3*(1), 85-104.
19. Czerwiński, W., Lasota, S., Lazić, R., Leroux, J., & Mazowiecki, F. (2020). The reachability problem for Petri nets is not elementary. *Journal of the ACM (JACM)*, *68*(1), 1-28.
20. Meyer, A. R., & Stockmeyer, L. J. (1972, October). The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT* (Vol. 72, pp. 125-129).
21. Jones, N. D. (1997). *Computability and complexity: from a programming perspective*. MIT press.
22. Knuth, D. E., Morris, Jr, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM journal on computing*, *6*(2), 323-350.
23. Zimand, M. (1986). On the topological size of sets of random strings. *Mathematical Logic Quarterly*, *32*(6), 81-88.
24. Zimand, M. (2009). Extracting the Kolmogorov complexity of strings and sequences from sources with limited independence. *arXiv preprint arXiv:0902.2141*.
25. Shor, P. W. (1994, November). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (pp. 124-134). Ieee.